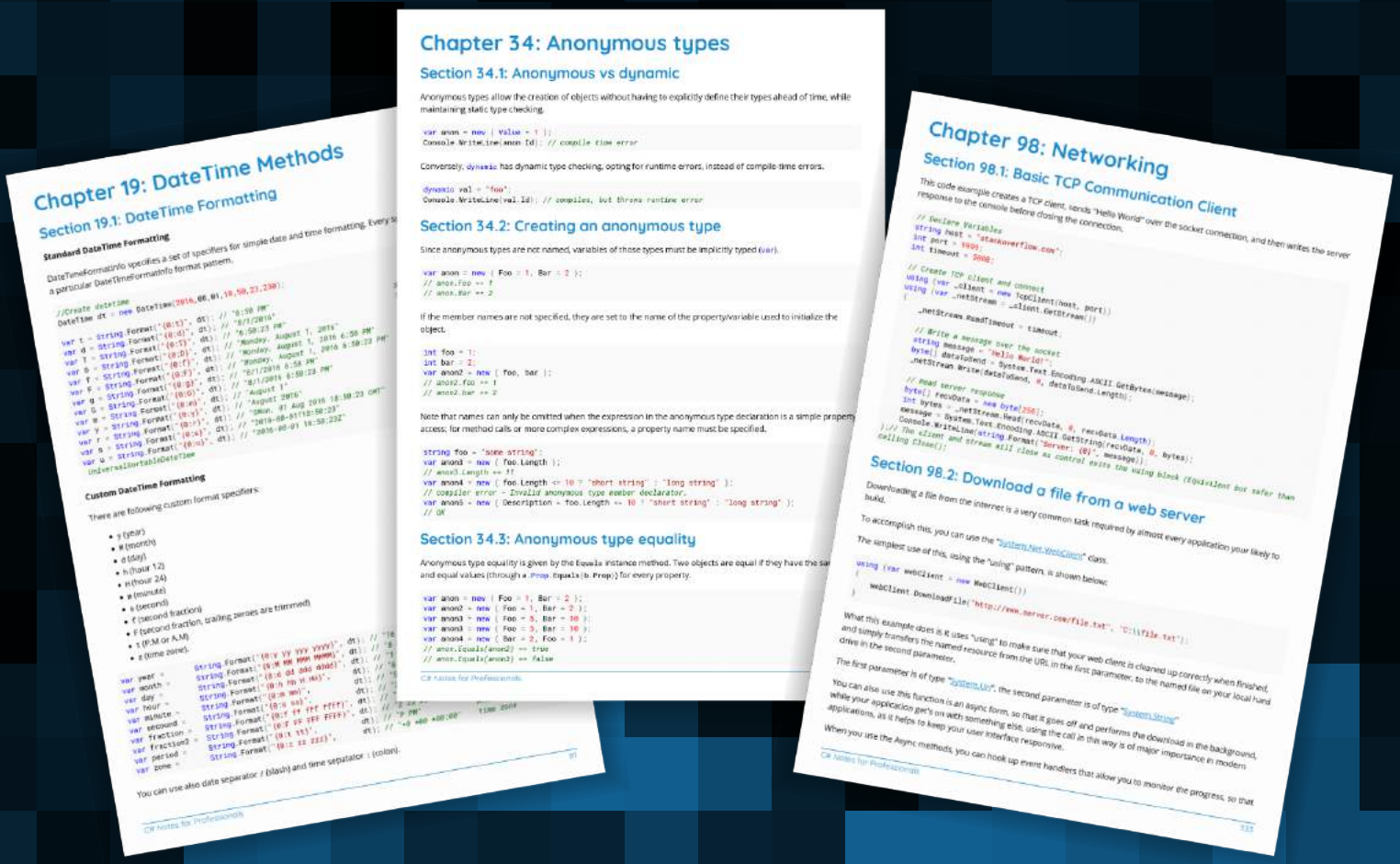
C#

Notes for Professionals



700+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Disclaimer
This is an unofficial free book created for educational purposes and is not affiliated with official C# group(s) or company(s). All trademarks and registered trademarks are the property of their respective owners

Contents

About 1

Chapter 1: Getting started with C# Language 2

Section 1.1: Creating a new console application (Visual Studio) 2

Section 1.2: Creating a new project in Visual Studio (console application) and Running it in Debug mode 4

Section 1.3: Creating a new program using .NET Core 7

Section 1.4: Creating a new program using Mono 9

Section 1.5: Creating a new query using LinqPad 9

Section 1.6: Creating a new project using Xamarin Studio 12

Chapter 2: Literals 18

Section 2.1: uint literals 18

Section 2.2: int literals 18

Section 2.3: sbyte literals 18

Section 2.4: decimal literals 18

Section 2.5: double literals 18

Section 2.6: float literals 18

Section 2.7: long literals 18

Section 2.8: ulong literal 18

Section 2.9: string literals 19

Section 2.10: char literals 19

Section 2.11: byte literals 19

Section 2.12: short literal 19

Section 2.13: ushort literal 19

Section 2.14: bool literals 19

Chapter 3: Operators 20

Section 3.1: Overloadable Operators 20

Section 3.2: Overloading equality operators 21

Section 3.3: Relational Operators 22

Section 3.4: Implicit Cast and Explicit Cast Operators 24

Section 3.5: Short-circuiting Operators 25

Section 3.6: ? : Ternary Operator 26

Section 3.7: ?. (Null Conditional Operator) 27

Section 3.8: "Exclusive or" Operator 27

Section 3.9: default Operator 28

Section 3.10: Assignment operator '=' 28

Section 3.11: sizeof 28

Section 3.12: ?? Null-Coalescing Operator 29

Section 3.13: Bit-Shifting Operators 29

Section 3.14: => Lambda operator 29

Section 3.15: Class Member Operators: Null Conditional Member Access 31

Section 3.16: Class Member Operators: Null Conditional Indexing 31

Section 3.17: Postfix and Prefix increment and decrement 31

Section 3.18: typeof 32

Section 3.19: Binary operators with assignment 32

Section 3.20: nameof Operator 32

Section 3.21: Class Member Operators: Member Access 33

Section 3.22: Class Member Operators: Function Invocation 33

Section 3.23: Class Member Operators: Aggregate Object Indexing 33
Chapter 4: Conditional Statements 34
Section 4.1: If-Else Statement 34
Section 4.2: If statement conditions are standard boolean expressions and values 34
Section 4.3: If-Else If-Else Statement 35
Chapter 5: Equality Operator 36
Section 5.1: Equality kinds in c# and equality operator 36
Chapter 6: Equals and GetHashCode 37
Section 6.1: Writing a good GetHashCode override 37
Section 6.2: Default Equals behavior 37
Section 6.3: Override Equals and GetHashCode on custom types 38
Section 6.4: Equals and GetHashCode in IEqualityComparator 39
Chapter 7: Null-Coalescing Operator 41
Section 7.1: Basic usage 41
Section 7.2: Null fall-through and chaining 41
Section 7.3: Null coalescing operator with method calls 42
Section 7.4: Use existing or create new 43
Section 7.5: Lazy properties initialization with null coalescing operator 43
Chapter 8: Null-conditional Operators 44
Section 8.1: Null-Conditional Operator 44
Section 8.2: The Null-Conditional Index 44
Section 8.3: Avoiding NullReferenceExceptions 45
Section 8.4: Null-conditional Operator can be used with Extension Method 45
Chapter 9: nameof Operator 47
Section 9.1: Basic usage: Printing a variable name 47
Section 9.2: Raising PropertyChanged event 47
Section 9.3: Argument Checking and Guard Clauses 48
Section 9.4: Strongly typed MVC action links 48
Section 9.5: Handling PropertyChanged events 49
Section 9.6: Applied to a generic type parameter 49
Section 9.7: Printing a parameter name 50
Section 9.8: Applied to qualified identifiers 50
Chapter 10: Verbatim Strings 51
Section 10.1: Interpolated Verbatim Strings 51
Section 10.2: Escaping Double Quotes 51
Section 10.3: Verbatim strings instruct the compiler to not use character escapes 51
Section 10.4: Multiline Strings 52
Chapter 11: Common String Operations 53
Section 11.1: Formatting a string 53
Section 11.2: Correctly reversing a string 53
Section 11.3: Padding a string to a fixed length 54
Section 11.4: Getting x characters from the right side of a string 55
Section 11.5: Checking for empty String using String.IsNullOrEmpty() and String.IsNullOrWhiteSpace() 56
Section 11.6: Trimming Unwanted Characters Off the Start and/or End of Strings 57
Section 11.7: Convert Decimal Number to Binary,Octal and Hexadecimal Format 57
Section 11.8: Construct a string from Array 57
Section 11.9: Formatting using ToString 58
Section 11.10: Splitting a String by another string 59

Section 11.11: Splitting a String by specific character ... 59
Section 11.12: Getting Substrings of a given string ... 59
Section 11.13: Determine whether a string begins with a given sequence ... 59
Section 11.14: Getting a char at specific index and enumerating the string ... 59
Section 11.15: Joining an array of strings into a new one ... 60
Section 11.16: Replacing a string within a string ... 60
Section 11.17: Changing the case of characters within a String ... 60
Section 11.18: Concatenate an array of strings into a single string ... 61
Section 11.19: String Concatenation ... 61
Chapter 12: String.Format ... 62
Section 12.1: Since C# 6.0 ... 62
Section 12.2: Places where String.Format is 'embedded' in the framework ... 62
Section 12.3: Create a custom format provider ... 62
Section 12.4: Date Formatting ... 63
Section 12.5: Currency Formatting ... 64
Section 12.6: Using custom number format ... 65
Section 12.7: Align left/ right, pad with spaces ... 65
Section 12.8: Numeric formats ... 66
Section 12.9: ToString() ... 66
Section 12.10: Escaping curly brackets inside a String.Format() expression ... 67
Section 12.11: Relationship with ToString() ... 67
Chapter 13: String Concatenate ... 68
Section 13.1: + Operator ... 68
Section 13.2: Concatenate strings using System.Text.StringBuilder ... 68
Section 13.3: Concat string array elements using String.Join ... 68
Section 13.4: Concatenation of two strings using $ ... 69
Chapter 14: String Manipulation ... 70
Section 14.1: Replacing a string within a string ... 70
Section 14.2: Finding a string within a string ... 70
Section 14.3: Removing (Trimming) white-space from a string ... 70
Section 14.4: Splitting a string using a delimiter ... 71
Section 14.5: Concatenate an array of strings into a single string ... 71
Section 14.6: String Concatenation ... 71
Section 14.7: Changing the case of characters within a String ... 71
Chapter 15: String Interpolation ... 73
Section 15.1: Format dates in strings ... 73
Section 15.2: Padding the output ... 73
Section 15.3: Expressions ... 74
Section 15.4: Formatting numbers in strings ... 74
Section 15.5: Simple Usage ... 75
Chapter 16: String Escape Sequences ... 76
Section 16.1: Escaping special symbols in string literals ... 76
Section 16.2: Unicode character escape sequences ... 76
Section 16.3: Escaping special symbols in character literals ... 76
Section 16.4: Using escape sequences in identifiers ... 76
Section 16.5: Unrecognized escape sequences produce compile-time errors ... 77
Chapter 17: StringBuilder ... 78
Section 17.1: What a StringBuilder is and when to use one ... 78
Section 17.2: Use StringBuilder to create string from a large number of records ... 79

Chapter 18: Regex Parsing 80
Section 18.1: Single match 80
Section 18.2: Multiple matches 80
Chapter 19: DateTime Methods 81
Section 19.1: DateTime Formatting 81
Section 19.2: DateTime.AddDays(Double) 82
Section 19.3: DateTime.AddHours(Double) 82
Section 19.4: DateTime.Parse(String) 82
Section 19.5: DateTime.TryParse(String, DateTime) 82
Section 19.6: DateTime.AddMilliseconds(Double) 83
Section 19.7: DateTime.Compare(DateTime t1, DateTime t2) 83
Section 19.8: DateTime.DaysInMonth(Int32,Int32) 83
Section 19.9: DateTime.AddYears(Int32) 84
Section 19.10: Pure functions warning when dealing with DateTime 84
Section 19.11: DateTime.TryParseExact(String,String,IFormatProvider,DateTimeStyles,DateTime) 84
Section 19.12: DateTime.Add(TimeSpan) 86
Section 19.13: Parse and TryParse with culture info 86
Section 19.14: DateTime as initializer in for-loop 87
Section 19.15: DateTime.ParseExact(String,String,IFormatProvider) 87
Section 19.16: DateTime ToString, ToShortDateString, ToLongDateString and ToString formatted 88
Section 19.17: Current Date 88
Chapter 20: Arrays 89
Section 20.1: Declaring an array 89
Section 20.2: Initializing an array filled with a repeated non-default value 89
Section 20.3: Copying arrays 90
Section 20.4: Comparing arrays for equality 90
Section 20.5: Multi-dimensional arrays 91
Section 20.6: Getting and setting array values 91
Section 20.7: Iterate over an array 91
Section 20.8: Creating an array of sequential numbers 92
Section 20.9: Jagged arrays 92
Section 20.10: Array covariance 94
Section 20.11: Arrays as IEnumerable<> instances 94
Section 20.12: Checking if one array contains another array 94
Chapter 21: O(n) Algorithm for circular rotation of an array 96
Section 21.1: Example of a generic method that rotates an array by a given shift 96
Chapter 22: Enum 98
Section 22.1: Enum basics 98
Section 22.2: Enum as flags 99
Section 22.3: Using << notation for flags 101
Section 22.4: Test flags-style enum values with bitwise logic 101
Section 22.5: Add and remove values from flagged enum 102
Section 22.6: Enum to string and back 102
Section 22.7: Enums can have unexpected values 103
Section 22.8: Default value for enum == ZERO 103
Section 22.9: Adding additional description information to an enum value 104
Section 22.10: Get all the members values of an enum 105
Section 22.11: Bitwise Manipulation using enums 105
Chapter 23: Tuples 106

Section 23.1: Accessing tuple elements 106
Section 23.2: Creating tuples 106
Section 23.3: Comparing and sorting Tuples 106
Section 23.4: Return multiple values from a method 107
Chapter 24: Guid 108
Section 24.1: Getting the string representation of a Guid 108
Section 24.2: Creating a Guid 108
Section 24.3: Declaring a nullable GUID 108
Chapter 25: BigInteger 110
Section 25.1: Calculate the First 1,000-Digit Fibonacci Number 110
Chapter 26: Collection Initializers 111
Section 26.1: Collection initializers 111
Section 26.2: C# 6 Index Initializers 111
Section 26.3: Collection initializers in custom classes 112
Section 26.4: Using collection initializer inside object initializer 113
Section 26.5: Collection Initializers with Parameter Arrays 114
Chapter 27: An overview of C# collections 115
Section 27.1: HashSet<T> 115
Section 27.2: Dictionary<TKey, TValue> 115
Section 27.3: SortedSet<T> 116
Section 27.4: T[ ] (Array of T) 116
Section 27.5: List<T> 117
Section 27.6: Stack<T> 117
Section 27.7: LinkedList<T> 117
Section 27.8: Queue 118
Chapter 28: Looping 119
Section 28.1: For Loop 119
Section 28.2: Do - While Loop 120
Section 28.3: Foreach Loop 120
Section 28.4: Looping styles 121
Section 28.5: Nested loops 122
Section 28.6: continue 122
Section 28.7: While loop 123
Section 28.8: break 123
Chapter 29: Iterators 125
Section 29.1: Creating Iterators Using Yield 125
Section 29.2: Simple Numeric Iterator Example 126
Chapter 30: IEnumerable 127
Section 30.1: IEnumerable with custom Enumerator 127
Section 30.2: IEnumerable<int> 128
Chapter 31: Value type vs Reference type 129
Section 31.1: Passing by reference using ref keyword 129
Section 31.2: Changing values elsewhere 130
Section 31.3: ref vs out parameters 131
Section 31.4: Assignment 132
Section 31.5: Difference with method parameters ref and out 132
Section 31.6: Passing by reference 133
Chapter 32: Built-in Types 134
Section 32.1: Conversion of boxed value types 134

Section 32.2: Comparisons with boxed value types ... 134
Section 32.3: Immutable reference type - string ... 134
Section 32.4: Value type - char ... 135
Section 32.5: Value type - short, int, long (signed 16 bit, 32 bit, 64 bit integers) ... 135
Section 32.6: Value type - ushort, uint, ulong (unsigned 16 bit, 32 bit, 64 bit integers) ... 135
Section 32.7: Value type - bool ... 136
Chapter 33: Aliases of built-in types ... 137
Section 33.1: Built-In Types Table ... 137
Chapter 34: Anonymous types ... 138
Section 34.1: Anonymous vs dynamic ... 138
Section 34.2: Creating an anonymous type ... 138
Section 34.3: Anonymous type equality ... 138
Section 34.4: Generic methods with anonymous types ... 139
Section 34.5: Instantiating generic types with anonymous types ... 139
Section 34.6: Implicitly typed arrays ... 139
Chapter 35: Dynamic type ... 141
Section 35.1: Creating a dynamic object with properties ... 141
Section 35.2: Creating a dynamic variable ... 141
Section 35.3: Returning dynamic ... 141
Section 35.4: Handling Specific Types Unknown at Compile Time ... 141
Chapter 36: Type Conversion ... 143
Section 36.1: Explicit Type Conversion ... 143
Section 36.2: MSDN implicit operator example ... 143
Chapter 37: Casting ... 145
Section 37.1: Checking compatibility without casting ... 145
Section 37.2: Cast an object to a base type ... 145
Section 37.3: Conversion Operators ... 145
Section 37.4: LINQ Casting operations ... 147
Section 37.5: Explicit Casting ... 147
Section 37.6: Safe Explicit Casting (`as` operator) ... 147
Section 37.7: Implicit Casting ... 148
Section 37.8: Explicit Numeric Conversions ... 148
Chapter 38: Nullable types ... 149
Section 38.1: Initialising a nullable ... 149
Section 38.2: Check if a Nullable has a value ... 149
Section 38.3: Get the value of a nullable type ... 149
Section 38.4: Getting a default value from a nullable ... 150
Section 38.5: Default value of nullable types is null ... 150
Section 38.6: Effective usage of underlying Nullable<T> argument ... 151
Section 38.7: Check if a generic type parameter is a nullable type ... 152
Chapter 39: Constructors and Finalizers ... 153
Section 39.1: Static constructor ... 153
Section 39.2: Singleton constructor pattern ... 154
Section 39.3: Default Constructor ... 154
Section 39.4: Forcing a static constructor to be called ... 155
Section 39.5: Calling a constructor from another constructor ... 156
Section 39.6: Calling the base class constructor ... 156
Section 39.7: Finalizers on derived classes ... 157
Section 39.8: Exceptions in static constructors ... 157

Section 39.9: Constructor and Property Initialization 158
Section 39.10: Generic Static Constructors 160
Section 39.11: Calling virtual methods in constructor 160
Chapter 40: Access Modifiers 162
Section 40.1: Access Modifiers Diagrams 162
Section 40.2: public 163
Section 40.3: private 163
Section 40.4: protected internal 164
Section 40.5: internal 165
Section 40.6: protected 166
Chapter 41: Interfaces 167
Section 41.1: Implementing an interface 167
Section 41.2: Explicit interface implementation 167
Section 41.3: Interface Basics 169
Section 41.4: IComparable<T> as an Example of Implementing an Interface 171
Section 41.5: Implementing multiple interfaces 172
Section 41.6: Why we use interfaces 172
Section 41.7: "Hiding" members with Explicit Implementation 173
Chapter 42: Static Classes 175
Section 42.1: Static Classes 175
Section 42.2: Static class lifetime 175
Section 42.3: Static keyword 176
Chapter 43: Singleton Implementation 177
Section 43.1: Statically Initialized Singleton 177
Section 43.2: Lazy, thread-safe Singleton (using Lazy<T>) 177
Section 43.3: Lazy, thread-safe Singleton (using Double Checked Locking) 177
Section 43.4: Lazy, thread safe singleton (for .NET 3.5 or older, alternate implementation) 178
Chapter 44: Dependency Injection 180
Section 44.1: Dependency Injection C# and ASP.NET with Unity 180
Section 44.2: Dependency injection using MEF 182
Chapter 45: Partial class and methods 185
Section 45.1: Partial classes 185
Section 45.2: Partial classes inheriting from a base class 185
Section 45.3: Partial methods 186
Chapter 46: Object initializers 187
Section 46.1: Simple usage 187
Section 46.2: Usage with non-default constructors 187
Section 46.3: Usage with anonymous types 187
Chapter 47: Methods 189
Section 47.1: Calling a Method 189
Section 47.2: Anonymous method 189
Section 47.3: Declaring a Method 190
Section 47.4: Parameters and Arguments 190
Section 47.5: Return Types 190
Section 47.6: Default Parameters 191
Section 47.7: Method overloading 192
Section 47.8: Access rights 193
Chapter 48: Extension Methods 194
Section 48.1: Extension methods - overview 194

Section 48.2: Null checking ... 196
Section 48.3: Explicitly using an extension method ... 197
Section 48.4: Generic Extension Methods ... 197
Section 48.5: Extension methods can only see public (or internal) members of the extended class ... 199
Section 48.6: Extension methods for chaining ... 199
Section 48.7: Extension methods with Enumeration ... 200
Section 48.8: Extension methods dispatch based on static type ... 201
Section 48.9: Extension methods on Interfaces ... 202
Section 48.10: Extension methods in combination with interfaces ... 203
Section 48.11: Extension methods aren't supported by dynamic code ... 203
Section 48.12: Extensions and interfaces together enable DRY code and mixin-like functionality ... 204
Section 48.13: IList<T> Extension Method Example: Comparing 2 Lists ... 205
Section 48.14: Extension methods as strongly typed wrappers ... 206
Section 48.15: Using Extension methods to create beautiful mapper classes ... 206
Section 48.16: Using Extension methods to build new collection types (e.g. DictList) ... 207
Section 48.17: Extension methods for handling special cases ... 208
Section 48.18: Using Extension methods with Static methods and Callbacks ... 209
Chapter 49: Named Arguments ... 211
Section 49.1: Argument order is not necessary ... 211
Section 49.2: Named arguments and optional parameters ... 211
Section 49.3: Named Arguments can make your code more clear ... 211
Chapter 50: Named and Optional Arguments ... 213
Section 50.1: Optional Arguments ... 213
Section 50.2: Named Arguments ... 214
Chapter 51: Data Annotation ... 217
Section 51.1: Data Annotation Basics ... 217
Section 51.2: Creating a custom validation attribute ... 217
Section 51.3: Manually Execute Validation Attributes ... 218
Section 51.4: Validation Attributes ... 218
Section 51.5: EditableAttribute (data modeling attribute) ... 220
Chapter 52: Keywords ... 222
Section 52.1: as ... 222
Section 52.2: goto ... 223
Section 52.3: volatile ... 224
Section 52.4: checked, unchecked ... 225
Section 52.5: virtual, override, new ... 226
Section 52.6: stackalloc ... 229
Section 52.7: break ... 230
Section 52.8: const ... 232
Section 52.9: async, await ... 233
Section 52.10: for ... 234
Section 52.11: abstract ... 235
Section 52.12: fixed ... 236
Section 52.13: default ... 237
Section 52.14: sealed ... 238
Section 52.15: is ... 238
Section 52.16: this ... 239
Section 52.17: readonly ... 240
Section 52.18: typeof ... 241
Section 52.19: foreach ... 241

Section 52.20: dynamic 242
Section 52.21: try, catch, finally, throw 243
Section 52.22: void 244
Section 52.23: namespace 244
Section 52.24: ref, out 245
Section 52.25: base 246
Section 52.26: float, double, decimal 248
Section 52.27: operator 249
Section 52.28: char 250
Section 52.29: params 250
Section 52.30: while 251
Section 52.31: null 253
Section 52.32: continue 254
Section 52.33: string 254
Section 52.34: return 255
Section 52.35: unsafe 255
Section 52.36: switch 257
Section 52.37: var 258
Section 52.38: when 259
Section 52.39: lock 260
Section 52.40: uint 261
Section 52.41: if, if...else, if... else if 261
Section 52.42: static 262
Section 52.43: internal 264
Section 52.44: using 265
Section 52.45: where 265
Section 52.46: int 267
Section 52.47: ulong 268
Section 52.48: true, false 268
Section 52.49: struct 268
Section 52.50: extern 269
Section 52.51: bool 270
Section 52.52: interface 270
Section 52.53: delegate 271
Section 52.54: unchecked 271
Section 52.55: ushort 272
Section 52.56: sizeof 272
Section 52.57: in 272
Section 52.58: implicit 273
Section 52.59: do 273
Section 52.60: long 274
Section 52.61: enum 274
Section 52.62: partial 275
Section 52.63: event 276
Section 52.64: sbyte 277
Chapter 53: Object Oriented Programming In C# 278
Section 53.1: Classes: 278
Chapter 54: Recursion 279
Section 54.1: Recursion in plain English 279
Section 54.2: Fibonacci Sequence 279

Section 54.3: PowerOf calculation 280
Section 54.4: Recursively describe an object structure 280
Section 54.5: Using Recursion to Get Directory Tree 281
Section 54.6: Factorial calculation 284
Chapter 55: Naming Conventions 285
Section 55.1: Capitalization conventions 285
Section 55.2: Enums 286
Section 55.3: Interfaces 286
Section 55.4: Exceptions 286
Section 55.5: Private fields 287
Section 55.6: Namespaces 287
Chapter 56: XML Documentation Comments 288
Section 56.1: Simple method annotation 288
Section 56.2: Generating XML from documentation comments 288
Section 56.3: Method documentation comment with param and returns elements 290
Section 56.4: Interface and class documentation comments 290
Section 56.5: Referencing another class in documentation 291
Chapter 57: Comments and regions 292
Section 57.1: Comments 292
Section 57.2: Regions 292
Section 57.3: Documentation comments 293
Chapter 58: Inheritance 295
Section 58.1: Inheritance. Constructors' calls sequence 295
Section 58.2: Inheriting from a base class 297
Section 58.3: Inheriting from a class and implementing an interface 298
Section 58.4: Inheriting from a class and implementing multiple interfaces 298
Section 58.5: Constructors In A Subclass 299
Section 58.6: Inheritance Anti-patterns 299
Section 58.7: Extending an abstract base class 300
Section 58.8: Testing and navigating inheritance 301
Section 58.9: Inheriting methods 301
Section 58.10: Base class with recursive type specification 302
Chapter 59: Generics 305
Section 59.1: Implicit type inference (methods) 305
Section 59.2: Type inference (classes) 306
Section 59.3: Using generic method with an interface as a constraint type 306
Section 59.4: Type constraints (new-keyword) 307
Section 59.5: Type constraints (classes and interfaces) 308
Section 59.6: Checking equality of generic values 309
Section 59.7: Reflecting on type parameters 310
Section 59.8: Covariance 310
Section 59.9: Contravariance 311
Section 59.10: Invariance 312
Section 59.11: Variant interfaces 312
Section 59.12: Variant delegates 313
Section 59.13: Variant types as parameters and return values 314
Section 59.14: Type Parameters (Interfaces) 314
Section 59.15: Type constraints (class and struct) 315
Section 59.16: Explicit type parameters 315
Section 59.17: Type Parameters (Classes) 315

Section 59.18: Type Parameters (Methods) 316
Section 59.19: Generic type casting 316
Section 59.20: Configuration reader with generic type casting 317
Chapter 60: Using Statement 319
Section 60.1: Using Statement Basics 319
Section 60.2: Gotcha: returning the resource which you are disposing 320
Section 60.3: Multiple using statements with one block 321
Section 60.4: Gotcha: Exception in Dispose method masking other errors in Using blocks 322
Section 60.5: Using statements are null-safe 322
Section 60.6: Using Dispose Syntax to define custom scope 322
Section 60.7: Using Statements and Database Connections 323
Section 60.8: Executing code in constraint context 325
Chapter 61: Using Directive 326
Section 61.1: Associate an Alias to Resolve Conflicts 326
Section 61.2: Using alias directives 326
Section 61.3: Access Static Members of a Class 326
Section 61.4: Basic Usage 327
Section 61.5: Reference a Namespace 327
Section 61.6: Associate an Alias with a Namespace 327
Chapter 62: IDisposable interface 329
Section 62.1: In a class that contains only managed resources 329
Section 62.2: In a class with managed and unmanaged resources 329
Section 62.3: IDisposable, Dispose 330
Section 62.4: using keyword 330
Section 62.5: In an inherited class with managed resources 331
Chapter 63: Reflection 332
Section 63.1: Get the members of a type 332
Section 63.2: Get a method and invoke it 332
Section 63.3: Creating an instance of a Type 333
Section 63.4: Get a Strongly-Typed Delegate to a Method or Property via Reflection 336
Section 63.5: Get a generic method and invoke it 337
Section 63.6: Get a System.Type 338
Section 63.7: Getting and setting properties 338
Section 63.8: Create an instance of a Generic Type and invoke it's method 338
Section 63.9: Custom Attributes 339
Section 63.10: Instantiating classes that implement an interface (e.g. plugin activation) 340
Section 63.11: Get a Type by name with namespace 340
Section 63.12: Determining generic arguments of instances of generic types 341
Section 63.13: Looping through all the properties of a class 342
Chapter 64: IQueryable interface 343
Section 64.1: Translating a LINQ query to a SQL query 343
Chapter 65: Linq to Objects 344
Section 65.1: Using LINQ to Objects in C# 344
Chapter 66: LINQ Queries 348
Section 66.1: Chaining methods 348
Section 66.2: First, FirstOrDefault, Last, LastOrDefault, Single, and SingleOrDefault 349
Section 66.3: Except 352
Section 66.4: SelectMany 354
Section 66.5: Any 355

Section 66.6: JOINS 355
Section 66.7: Skip and Take 358
Section 66.8: Defining a variable inside a Linq query (let keyword) 358
Section 66.9: Zip 359
Section 66.10: Range and Repeat 359
Section 66.11: Basics 360
Section 66.12: All 360
Section 66.13: Aggregate 361
Section 66.14: Distinct 362
Section 66.15: SelectMany: Flattening a sequence of sequences 362
Section 66.16: GroupBy 364
Section 66.17: Query collection by type / cast elements to type 365
Section 66.18: Enumerating the Enumerable 366
Section 66.19: Using Range with various Linq methods 367
Section 66.20: Where 368
Section 66.21: Using SelectMany instead of nested loops 368
Section 66.22: Contains 368
Section 66.23: Joining multiple sequences 370
Section 66.24: Joining on multiple keys 372
Section 66.25: ToLookup 372
Section 66.26: SkipWhile 372
Section 66.27: Query Ordering - OrderBy() ThenBy() OrderByDescending() ThenByDescending() 373
Section 66.28: Sum 374
Section 66.29: GroupBy one or multiple fields 374
Section 66.30: OrderBy 375
Section 66.31: Any and First(OrDefault) - best practice 376
Section 66.32: GroupBy Sum and Count 376
Section 66.33: SequenceEqual 377
Section 66.34: ElementAt and ElementAtOrDefault 377
Section 66.35: DefaultIfEmpty 378
Section 66.36: ToDictionary 379
Section 66.37: Concat 380
Section 66.38: Build your own Linq operators for IEnumerable<T> 380
Section 66.39: Select - Transforming elements 381
Section 66.40: OrderByDescending 382
Section 66.41: Union 382
Section 66.42: GroupJoin with outer range variable 383
Section 66.43: Linq Quantifiers 383
Section 66.44: TakeWhile 383
Section 66.45: Reverse 384
Section 66.46: Count and LongCount 385
Section 66.47: Incrementally building a query 385
Section 66.48: Select with Func<TSource, int, TResult> selector - Use to get ranking of elements 387
Chapter 67: LINQ to XML 389
Section 67.1: Read XML using LINQ to XML 389
Chapter 68: Parallel LINQ (PLINQ) 391
Section 68.1: Simple example 391
Section 68.2: WithDegreeOfParallelism 391
Section 68.3: AsOrdered 391
Section 68.4: AsUnordered 391

Chapter 69: XmlDocument and the System.Xml namespace 392
Section 69.1: XmlDocument vs XDocument (Example and comparison) 392
Section 69.2: Reading from XML document 394
Section 69.3: Basic XML document interaction 395
Chapter 70: XDocument and the System.Xml.Linq namespace 396
Section 70.1: Generate an XML document 396
Section 70.2: Generate an XML document using fluent syntax 396
Section 70.3: Modify XML File 397
Chapter 71: C# 7.0 Features 399
Section 71.1: Language support for Tuples 399
Section 71.2: Local functions 403
Section 71.3: out var declaration 404
Section 71.4: Pattern Matching 405
Section 71.5: Digit separators 407
Section 71.6: Binary literals 407
Section 71.7: throw expressions 408
Section 71.8: Extended expression bodied members list 409
Section 71.9: ref return and ref local 410
Section 71.10: ValueTask<T> 411
Chapter 72: C# 6.0 Features 413
Section 72.1: Exception filters 413
Section 72.2: String interpolation 417
Section 72.3: Auto-property initializers 423
Section 72.4: Null propagation 426
Section 72.5: Expression-bodied function members 429
Section 72.6: Operator nameof 431
Section 72.7: Using static type 433
Section 72.8: Index initializers 433
Section 72.9: Improved overload resolution 435
Section 72.10: Await in catch and finally 436
Section 72.11: Minor changes and bugfixes 437
Section 72.12: Using an extension method for collection initialization 437
Section 72.13: Disable Warnings Enhancements 438
Chapter 73: C# 5.0 Features 440
Section 73.1: Async & Await 440
Section 73.2: Caller Information Attributes 441
Chapter 74: C# 4.0 Features 442
Section 74.1: Optional parameters and named arguments 442
Section 74.2: Variance 443
Section 74.3: Dynamic member lookup 443
Section 74.4: Optional ref keyword when using COM 444
Chapter 75: C# 3.0 Features 445
Section 75.1: Implicitly typed variables (var) 445
Section 75.2: Language Integrated Queries (LINQ) 445
Section 75.3: Lambda expresions 446
Section 75.4: Anonymous types 446
Chapter 76: Exception Handling 448
Section 76.1: Creating Custom Exceptions 448
Section 76.2: Finally block 450

Section 76.3: Best Practices ... 451
Section 76.4: Exception Anti-patterns ... 453
Section 76.5: Basic Exception Handling ... 455
Section 76.6: Handling specific exception types ... 455
Section 76.7: Aggregate exceptions / multiple exceptions from one method ... 456
Section 76.8: Throwing an exception ... 457
Section 76.9: Unhandled and Thread Exception ... 457
Section 76.10: Implementing IErrorHandler for WCF Services ... 458
Section 76.11: Using the exception object ... 460
Section 76.12: Nesting of Exceptions & try catch blocks ... 462
Chapter 77: NullReferenceException ... 463
Section 77.1: NullReferenceException explained ... 463
Chapter 78: Handling FormatException when converting string to other types ... 464
Section 78.1: Converting string to integer ... 464
Chapter 79: Read & Understand Stacktraces ... 466
Section 79.1: Stack trace for a simple NullReferenceException in Windows Forms ... 466
Chapter 80: Diagnostics ... 468
Section 80.1: Redirecting log output with TraceListeners ... 468
Section 80.2: Debug.WriteLine ... 468
Chapter 81: Overflow ... 469
Section 81.1: Integer overflow ... 469
Section 81.2: Overflow during operation ... 469
Section 81.3: Ordering matters ... 469
Chapter 82: Getting Started: Json with C# ... 470
Section 82.1: Simple Json Example ... 470
Section 82.2: First things First: Library to work with Json ... 470
Section 82.3: C# Implementation ... 470
Section 82.4: Serialization ... 471
Section 82.5: Deserialization ... 471
Section 82.6: Serialization & De-Serialization Common Utilities function ... 471
Chapter 83: Using json.net ... 473
Section 83.1: Using JsonConverter on simple values ... 473
Section 83.2: Collect all fields of JSON object ... 475
Chapter 84: Lambda expressions ... 477
Section 84.1: Lambda Expressions as Shorthand for Delegate Initialization ... 477
Section 84.2: Lambda Expression as an Event Handler ... 477
Section 84.3: Lambda Expressions with Multiple Parameters or No Parameters ... 478
Section 84.4: Lambdas can be emitted both as `Func` and `Expression` ... 478
Section 84.5: Put Multiple Statements in a Statement Lambda ... 478
Section 84.6: Lambdas for both `Func` and `Action` ... 479
Section 84.7: Using lambda syntax to create a closure ... 479
Section 84.8: Passing a Lambda Expression as a Parameter to a Method ... 479
Section 84.9: Basic lambda expressions ... 479
Section 84.10: Basic lambda expressions with LINQ ... 480
Section 84.11: Lambda syntax with statement block body ... 480
Section 84.12: Lambda expressions with System.Linq.Expressions ... 480
Chapter 85: Generic Lambda Query Builder ... 481
Section 85.1: QueryFilter class ... 481
Section 85.2: GetExpression Method ... 481

Section 85.3: GetExpression Private overload 482
Section 85.4: ConstantExpression Method 483
Section 85.5: Usage 484
Chapter 86: Properties 485
Section 86.1: Auto-implemented properties 485
Section 86.2: Default Values for Properties 485
Section 86.3: Public Get 486
Section 86.4: Public Set 486
Section 86.5: Accessing Properties 486
Section 86.6: Read-only properties 488
Section 86.7: Various Properties in Context 488
Chapter 87: Initializing Properties 490
Section 87.1: C# 6.0: Initialize an Auto-Implemented Property 490
Section 87.2: Initializing Property with a Backing Field 490
Section 87.3: Property Initialization during object instantiation 490
Section 87.4: Initializing Property in Constructor 490
Chapter 88: INotifyPropertyChanged interface 491
Section 88.1: Implementing INotifyPropertyChanged in C# 6 491
Section 88.2: INotifyPropertyChanged With Generic Set Method 492
Chapter 89: Events 494
Section 89.1: Declaring and Raising Events 494
Section 89.2: Event Properties 495
Section 89.3: Creating cancelable event 496
Section 89.4: Standard Event Declaration 497
Section 89.5: Anonymous Event Handler Declaration 498
Section 89.6: Non-Standard Event Declaration 498
Section 89.7: Creating custom EventArgs containing additional data 499
Chapter 90: Expression Trees 501
Section 90.1: Create Expression Trees with a lambda expression 501
Section 90.2: Creating Expression Trees by Using the API 501
Section 90.3: Compiling Expression Trees 501
Section 90.4: Parsing Expression Trees 502
Section 90.5: Expression Tree Basic 502
Section 90.6: Examining the Structure of an Expression using Visitor 503
Section 90.7: Understanding the expressions API 503
Chapter 91: Overload Resolution 505
Section 91.1: Basic Overloading Example 505
Section 91.2: "params" is not expanded, unless necessary 505
Section 91.3: Passing null as one of the arguments 506
Chapter 92: BindingList<T> 507
Section 92.1: Add item to list 507
Section 92.2: Avoiding N*2 iteration 507
Chapter 93: Preprocessor directives 508
Section 93.1: Conditional Expressions 508
Section 93.2: Other Compiler Instructions 508
Section 93.3: Defining and Undefining Symbols 509
Section 93.4: Region Blocks 510
Section 93.5: Disabling and Restoring Compiler Warnings 510
Section 93.6: Generating Compiler Warnings and Errors 510

Section 93.7: Custom Preprocessors at project level 511
Section 93.8: Using the Conditional attribute 511
Chapter 94: Structs 513
Section 94.1: Declaring a struct 513
Section 94.2: Struct usage 514
Section 94.3: Structs are copied on assignment 515
Section 94.4: Struct implementing interface 515
Chapter 95: Attributes 516
Section 95.1: Creating a custom attribute 516
Section 95.2: Reading an attribute 516
Section 95.3: Using an attribute 517
Section 95.4: DebuggerDisplay Attribute 517
Section 95.5: Caller info attributes 518
Section 95.6: Obsolete Attribute 519
Section 95.7: Reading an attribute from interface 519
Chapter 96: Delegates 521
Section 96.1: Declaring a delegate type 521
Section 96.2: The Func<T, TResult>, Action<T> and Predicate<T> delegate types 522
Section 96.3: Combine Delegates (Multicast Delegates) 523
Section 96.4: Safe invoke multicast delegate 524
Section 96.5: Delegate Equality 525
Section 96.6: Underlying references of named method delegates 525
Section 96.7: Assigning a named method to a delegate 526
Section 96.8: Assigning to a delegate by lambda 527
Section 96.9: Encapsulating transformations in funcs 527
Section 96.10: Passing delegates as parameters 527
Section 96.11: Closure inside a delegate 528
Chapter 97: File and Stream I/O 529
Section 97.1: Reading from a file using the System.IO.File class 529
Section 97.2: Lazily reading a file line-by-line via an IEnumerable 529
Section 97.3: Async write text to a file using StreamWriter 529
Section 97.4: Copy File 529
Section 97.5: Writing lines to a file using the System.IO.StreamWriter class 530
Section 97.6: Writing to a file using the System.IO.File class 530
Section 97.7: Create File 531
Section 97.8: Move File 531
Section 97.9: Delete File 532
Section 97.10: Files and Directories 532
Chapter 98: Networking 533
Section 98.1: Basic TCP Communication Client 533
Section 98.2: Download a file from a web server 533
Section 98.3: Async TCP Client 534
Section 98.4: Basic UDP Client 535
Chapter 99: Performing HTTP requests 536
Section 99.1: Creating and sending an HTTP POST request 536
Section 99.2: Creating and sending an HTTP GET request 536
Section 99.3: Error handling of specific HTTP response codes (such as 404 Not Found) 537
Section 99.4: Retrieve HTML for Web Page (Simple) 537
Section 99.5: Sending asynchronous HTTP POST request with JSON body 537

Chapter 100: Reading and writing .zip files 539
Section 100.1: Writing to a zip file 539
Section 100.2: Writing Zip Files in-memory 539
Section 100.3: Get files from a Zip file 539
Section 100.4: The following example shows how to open a zip archive and extract all .txt files to a folder 540
Chapter 101: FileSystemWatcher 541
Section 101.1: IsFileReady 541
Section 101.2: Basic FileWatcher 541
Chapter 102: Access network shared folder with username and password 543
Section 102.1: Code to access network shared file 543
Chapter 103: Asynchronous Socket 545
Section 103.1: Asynchronous Socket (Client / Server) example 545
Chapter 104: Action Filters 552
Section 104.1: Custom Action Filters 552
Chapter 105: Polymorphism 553
Section 105.1: Types of Polymorphism 553
Section 105.2: Another Polymorphism Example 554
Chapter 106: Immutability 557
Section 106.1: System.String class 557
Section 106.2: Strings and immutability 557
Chapter 107: Indexer 558
Section 107.1: A simple indexer 558
Section 107.2: Overloading the indexer to create a SparseArray 558
Section 107.3: Indexer with 2 arguments and interface 559
Chapter 108: Checked and Unchecked 560
Section 108.1: Checked and Unchecked 560
Section 108.2: Checked and Unchecked as a scope 560
Chapter 109: Stream 561
Section 109.1: Using Streams 561
Chapter 110: Timers 563
Section 110.1: Multithreaded Timers 563
Section 110.2: Creating an Instance of a Timer 564
Section 110.3: Assigning the "Tick" event handler to a Timer 565
Section 110.4: Example: Using a Timer to perform a simple countdown 565
Chapter 111: Stopwatches 567
Section 111.1: IsHighResolution 567
Section 111.2: Creating an Instance of a Stopwatch 567
Chapter 112: Threading 569
Section 112.1: Avoiding Reading and Writing Data Simultaneously 569
Section 112.2: Creating and Starting a Second Thread 570
Section 112.3: Parallel.ForEach Loop 570
Section 112.4: Deadlocks (hold resource and wait) 571
Section 112.5: Simple Complete Threading Demo 573
Section 112.6: Creating One Thread Per Processor 574
Section 112.7: Simple Complete Threading Demo using Tasks 574
Section 112.8: Deadlocks (two threads waiting on each other) 575
Section 112.9: Explicit Task Parallism 576

Section 112.10: Implicit Task Parallelism 576
Section 112.11: Starting a thread with parameters 577
Chapter 113: Async/await, Backgroundworker, Task and Thread Examples 578
Section 113.1: ASP.NET Configure Await 578
Section 113.2: Task "run and forget" extension 580
Section 113.3: Async/await 580
Section 113.4: BackgroundWorker 581
Section 113.5: Task 582
Section 113.6: Thread 583
Chapter 114: Async-Await 584
Section 114.1: Await operator and async keyword 584
Section 114.2: Concurrent calls 585
Section 114.3: Try/Catch/Finally 586
Section 114.4: Returning a Task without await 587
Section 114.5: Async/await will only improve performance if it allows the machine to do additional work 587
Section 114.6: Web.config setup to target 4.5 for correct async behaviour 588
Section 114.7: Simple consecutive calls 588
Section 114.8: Blocking on async code can cause deadlocks 589
Chapter 115: Synchronization Context in Async-Await 591
Section 115.1: Pseudocode for async/await keywords 591
Section 115.2: Disabling synchronization context 591
Section 115.3: Why SynchronizationContext is so important? 592
Chapter 116: BackgroundWorker 593
Section 116.1: Using a BackgroundWorker to complete a task 593
Section 116.2: Assigning Event Handlers to a BackgroundWorker 594
Section 116.3: Creating a new BackgroundWorker instance 595
Section 116.4: Assigning Properties to a BackgroundWorker 595
Chapter 117: Task Parallel Library 596
Section 117.1: Parallel.ForEach 596
Section 117.2: Parallel.For 596
Section 117.3: Parallel.Invoke 597
Chapter 118: Making a variable thread safe 598
Section 118.1: Controlling access to a variable in a Parallel.For loop 598
Chapter 119: Lock Statement 599
Section 119.1: Throwing exception in a lock statement 599
Section 119.2: Simple usage 599
Section 119.3: Return in a lock statement 600
Section 119.4: Anti-Patterns and gotchas 600
Section 119.5: Using instances of Object for lock 604
Chapter 120: Yield Keyword 605
Section 120.1: Simple Usage 605
Section 120.2: Correctly checking arguments 605
Section 120.3: Early Termination 606
Section 120.4: More Pertinent Usage 607
Section 120.5: Lazy Evaluation 608
Section 120.6: Try...finally 609
Section 120.7: Eager evaluation 610
Section 120.8: Using yield to create an IEnumerator<T> when implementing IEnumerable<T> 610

Section 120.9: Lazy Evaluation Example: Fibonacci Numbers 611
Section 120.10: The difference between break and yield break 612
Section 120.11: Return another Enumerable within a method returning Enumerable 613
Chapter 121: Task Parallel Library (TPL) Dataflow Constructs 614
Section 121.1: ActionBlock<T> 614
Section 121.2: BroadcastBlock<T> 614
Section 121.3: BufferBlock<T> 615
Section 121.4: JoinBlock<T1, T2,...> 616
Section 121.5: WriteOnceBlock<T> 617
Section 121.6: BatchedJoinBlock<T1, T2,...> 618
Section 121.7: TransformBlock<TInput, TOutput> 619
Section 121.8: TransformManyBlock<TInput, TOutput> 619
Section 121.9: BatchBlock<T> 620
Chapter 122: Functional Programming 622
Section 122.1: Func and Action 622
Section 122.2: Higher-Order Functions 622
Section 122.3: Avoid Null References 622
Section 122.4: Immutability 624
Section 122.5: Immutable collections 625
Chapter 123: Func delegates 626
Section 123.1: Without parameters 626
Section 123.2: With multiple variables 626
Section 123.3: Lambda & anonymous methods 627
Section 123.4: Covariant & Contravariant Type Parameters 627
Chapter 124: Function with multiple return values 629
Section 124.1: "anonymous object" + "dynamic keyword" solution 629
Section 124.2: Tuple solution 629
Section 124.3: Ref and Out Parameters 629
Chapter 125: Binary Serialization 631
Section 125.1: Controlling serialization behavior with attributes 631
Section 125.2: Serialization Binder 631
Section 125.3: Some gotchas in backward compatibility 633
Section 125.4: Making an object serializable 635
Section 125.5: Serialization surrogates (Implementing ISerializationSurrogate) 636
Section 125.6: Adding more control by implementing ISerializable 638
Chapter 126: ICloneable 640
Section 126.1: Implementing ICloneable in a class 640
Section 126.2: Implementing ICloneable in a struct 640
Chapter 127: IComparable 642
Section 127.1: Sort versions 642
Chapter 128: Accessing Databases 644
Section 128.1: Connection Strings 644
Section 128.2: Entity Framework Connections 644
Section 128.3: ADO.NET Connections 645
Chapter 129: Using SQLite in C# 648
Section 129.1: Creating simple CRUD using SQLite in C# 648
Section 129.2: Executing Query 651
Chapter 130: Caching 653
Section 130.1: MemoryCache 653

Chapter 131: Code Contracts 654
Section 131.1: Postconditions 654
Section 131.2: Invariants 654
Section 131.3: Defining Contracts on Interface 655
Section 131.4: Preconditions 656
Chapter 132: Code Contracts and Assertions 658
Section 132.1: Assertions to check logic should always be true 658
Chapter 133: Structural Design Patterns 659
Section 133.1: Adapter Design Pattern 659
Chapter 134: Creational Design Patterns 663
Section 134.1: Singleton Pattern 663
Section 134.2: Factory Method pattern 664
Section 134.3: Abstract Factory Pattern 667
Section 134.4: Builder Pattern 669
Section 134.5: Prototype Pattern 673
Chapter 135: Implementing Decorator Design Pattern 676
Section 135.1: Simulating cafeteria 676
Chapter 136: Implementing Flyweight Design Pattern 678
Section 136.1: Implementing map in RPG game 678
Chapter 137: System.Management.Automation 681
Section 137.1: Invoke simple synchronous pipeline 681
Chapter 138: System.DirectoryServices.Protocols.LdapConnection 682
Section 138.1: Authenticated SSL LDAP connection, SSL cert does not match reverse DNS 682
Section 138.2: Super Simple anonymous LDAP 683
Chapter 139: C# Authentication handler 684
Section 139.1: Authentication handler 684
Chapter 140: Pointers 686
Section 140.1: Pointers for array access 686
Section 140.2: Pointer arithmetic 686
Section 140.3: The asterisk is part of the type 687
Section 140.4: void* 687
Section 140.5: Member access using -> 687
Section 140.6: Generic pointers 688
Chapter 141: Pointers & Unsafe Code 689
Section 141.1: Introduction to unsafe code 689
Section 141.2: Accessing Array Elements Using a Pointer 690
Section 141.3: Compiling Unsafe Code 690
Section 141.4: Retrieving the Data Value Using a Pointer 691
Section 141.5: Passing Pointers as Parameters to Methods 692
Chapter 142: How to use C# Structs to create a Union type (Similar to C Unions) 693
Section 142.1: C-Style Unions in C# 693
Section 142.2: Union Types in C# can also contain Struct fields 694
Chapter 143: Reactive Extensions (Rx) 696
Section 143.1: Observing TextChanged event on a TextBox 696
Section 143.2: Streaming Data from Database with Observable 696
Chapter 144: AssemblyInfo.cs Examples 697
Section 144.1: Global and local AssemblyInfo 697
Section 144.2: [AssemblyVersion] 697

Section 144.3: Automated versioning 698
Section 144.4: Common fields 698
Section 144.5: [AssemblyTitle] 698
Section 144.6: [AssemblyProduct] 698
Section 144.7: [InternalsVisibleTo] 698
Section 144.8: [AssemblyConfiguration] 699
Section 144.9: [AssemblyKeyFile] 699
Section 144.10: Reading Assembly Attributes 699
Chapter 145: Creating a Console Application using a Plain-Text Editor and the C# Compiler (csc.exe) 701
Section 145.1: Creating a Console application using a Plain-Text Editor and the C# Compiler 701
Chapter 146: CLSCompliantAttribute 703
Section 146.1: Access Modifier to which CLS rules apply 703
Section 146.2: Violation of CLS rule: Unsigned types / sbyte 703
Section 146.3: Violation of CLS rule: Same naming 704
Section 146.4: Violation of CLS rule: Identifier __ 705
Section 146.5: Violation of CLS rule: Inherit from non CLSComplaint class 705
Chapter 147: ObservableCollection<T> 706
Section 147.1: Initialize ObservableCollection<T> 706
Chapter 148: Hash Functions 707
Section 148.1: PBKDF2 for Password Hashing 707
Section 148.2: Complete Password Hashing Solution using Pbkdf2 707
Section 148.3: MD5 711
Section 148.4: SHA1 711
Section 148.5: SHA256 712
Section 148.6: SHA384 712
Section 148.7: SHA512 713
Chapter 149: Generating Random Numbers in C# 714
Section 149.1: Generate a random int 714
Section 149.2: Generate a random int in a given range 714
Section 149.3: Generating the same sequence of random numbers over and over again 714
Section 149.4: Create multiple random class with different seeds simultaneously 715
Section 149.5: Generate a Random double 715
Section 149.6: Generate a random character 715
Section 149.7: Generate a number that is a percentage of a max value 715
Chapter 150: Cryptography (System.Security.Cryptography) 717
Section 150.1: Modern Examples of Symmetric Authenticated Encryption of a string 717
Section 150.2: Introduction to Symmetric and Asymmetric Encryption 728
Section 150.3: Simple Symmetric File Encryption 729
Section 150.4: Cryptographically Secure Random Data 730
Section 150.5: Password Hashing 731
Section 150.6: Fast Asymmetric File Encryption 731
Chapter 151: ASP.NET Identity 736
Section 151.1: How to implement password reset token in asp.net identity using user manager 736
Chapter 152: Unsafe Code in .NET 739
Section 152.1: Using unsafe with arrays 739
Section 152.2: Using unsafe with strings 739
Section 152.3: Unsafe Array Index 740
Chapter 153: C# Script 741

Section 153.1: Simple code evaluation ... 741
Chapter 154: Runtime Compile ... 742
Section 154.1: RoslynScript ... 742
Section 154.2: CSharpCodeProvider ... 742
Chapter 155: Interoperability ... 743
Section 155.1: Import function from unmanaged C++ DLL ... 743
Section 155.2: Calling conventions ... 743
Section 155.3: C++ name mangling ... 744
Section 155.4: Dynamic loading and unloading of unmanaged DLLs ... 744
Section 155.5: Reading structures with Marshal ... 745
Section 155.6: Dealing with Win32 Errors ... 746
Section 155.7: Pinned Object ... 747
Section 155.8: Simple code to expose class for com ... 748
Chapter 156: .NET Compiler Platform (Roslyn) ... 750
Section 156.1: Semantic model ... 750
Section 156.2: Syntax tree ... 750
Section 156.3: Create workspace from MSBuild project ... 751
Chapter 157: ILGenerator ... 752
Section 157.1: Creates a DynamicAssembly that contains a UnixTimestamp helper method ... 752
Section 157.2: Create method override ... 753
Chapter 158: T4 Code Generation ... 755
Section 158.1: Runtime Code Generation ... 755
Chapter 159: Creating Own MessageBox in Windows Form Application ... 756
Section 159.1: How to use own created MessageBox control in another Windows Form application ... 756
Section 159.2: Creating Own MessageBox Control ... 756
Chapter 160: Including Font Resources ... 759
Section 160.1: Instantiate 'Fontfamily' from Resources ... 759
Section 160.2: Integration method ... 759
Section 160.3: Usage with a 'Button' ... 759
Chapter 161: Import Google Contacts ... 761
Section 161.1: Requirements ... 761
Section 161.2: Source code in the controller ... 761
Section 161.3: Source code in the view ... 764
Chapter 162: Garbage Collector in .Net ... 765
Section 162.1: Weak References ... 765
Section 162.2: Large Object Heap compaction ... 766
Chapter 163: Microsoft.Exchange.WebServices ... 767
Section 163.1: Retrieve Specified User's Out of Office Settings ... 767
Section 163.2: Update Specific User's Out of Office Settings ... 767
Chapter 164: Windows Communication Foundation ... 769
Section 164.1: Getting started sample ... 769
Credits ... 771
You may also like ... 785

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/CSharpBook

This *C# Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official C# group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with C# Language

Version Release Date

1.0 2002-01-01
1.2 2003-04-01
2.0 2005-09-01
3.0 2007-08-01
4.0 2010-04-01
5.0 2013-06-01
6.0 2015-07-01
7.0 2017-03-07

Section 1.1: Creating a new console application (Visual Studio)

1. Open Visual Studio
2. In the toolbar, go to **File** → **New Project**
3. Select the **Console Application** project type
4. Open the file `Program.cs` in the Solution Explorer
5. Add the following code to `Main()`:

```
public class Program
{
    public static void Main()
    {
        // Prints a message to the console.
        System.Console.WriteLine("Hello, World!");

        /* Wait for the user to press a key. This is a common
           way to prevent the console window from terminating
           and disappearing before the programmer can see the contents
           of the window, when the application is run via Start from within VS. */
        System.Console.ReadKey();
    }
}
```

6. In the toolbar, click **Debug** -> **Start Debugging** or hit **F5** or **ctrl + F5** (running without debugger) to run the program.

Live Demo on ideone

Explanation

- `class Program` is a class declaration. The class `Program` contains the data and method definitions that your program uses. Classes generally contain multiple methods. Methods define the behavior of the class. However, the `Program` class has only one method: `Main`.
- `static void Main()` defines the `Main` method, which is the entry point for all C# programs. The `Main` method states what the class does when executed. Only one `Main` method is allowed per class.
- `System.Console.WriteLine("Hello, world!");` method prints a given data (in this example, `Hello, world!`) as an output in the console window.

- `System.Console.ReadKey()`, ensures that the program won't close immediately after displaying the message. It does this by waiting for the user to press a key on the keyboard. Any key press from the user will terminate the program. The program terminates when it has finished the last line of code in the `main()` method.

Using the command line

To compile via command line use either `MSBuild` or `csc.exe` *(the C# compiler)*, both part of the Microsoft Build Tools package.

To compile this example, run the following command in the same directory where `HelloWorld.cs` is located:

```
%WINDIR%\\Microsoft.NET\\Framework64\\v4.0.30319\\csc.exe HelloWorld.cs
```

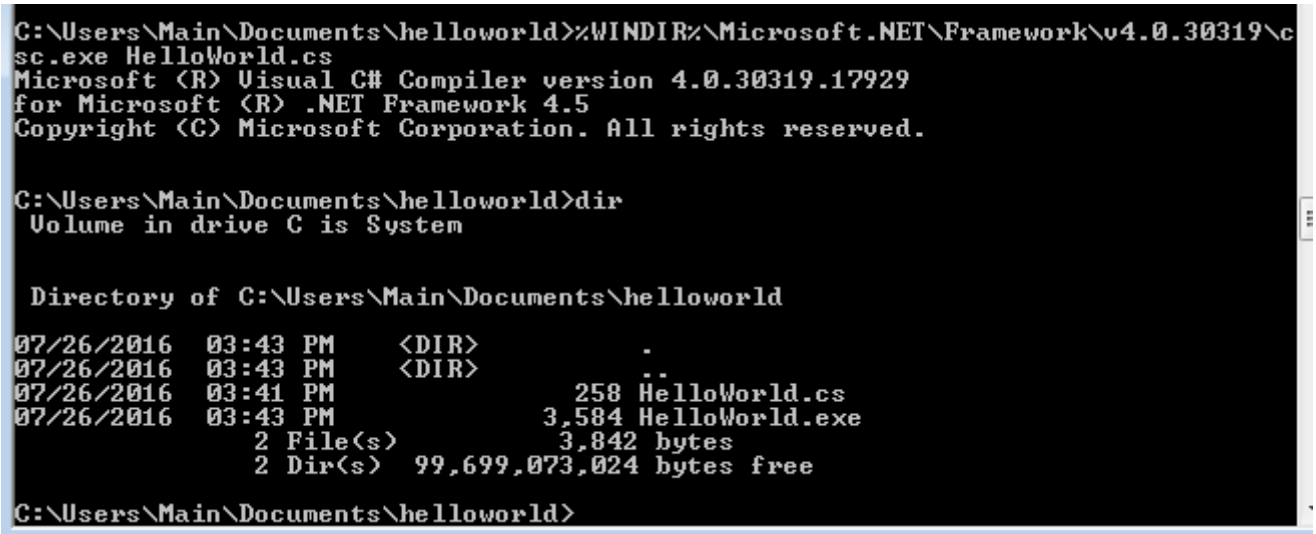
It can also be possible that you have two main methods inside one application. In this case, you have to tell the compiler which main method to execute by typing the following command in the **console**.(suppose Class `ClassA` also has a main method in the same `HelloWorld.cs` file in HelloWorld namespace)

```
%WINDIR%\\Microsoft.NET\\Framework64\\v4.0.30319\\csc.exe HelloWorld.cs /main:HelloWorld.ClassA
```

where HelloWorld is namespace

***Note**: This is the path where **.NET framework v4.0** is located in general. Change the path according to your .NET version. In addition, the directory might be **framework** instead of **framework64** if you're using the 32-bit .NET Framework. From the Windows Command Prompt, you can list all the csc.exe Framework paths by running the following commands (the first for 32-bit Frameworks):*

```
dir %WINDIR%\\Microsoft.NET\\Framework\\csc.exe /s/b
dir %WINDIR%\\Microsoft.NET\\Framework64\\csc.exe /s/b
```



There should now be an executable file named `HelloWorld.exe` in the same directory. To execute the program from the command prompt, simply type the executable's name and hit `Enter` as follows:

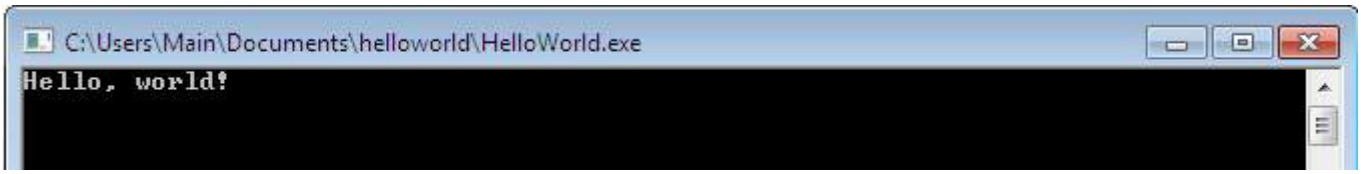
```
HelloWorld.exe
```

This will produce:

> Hello, world!

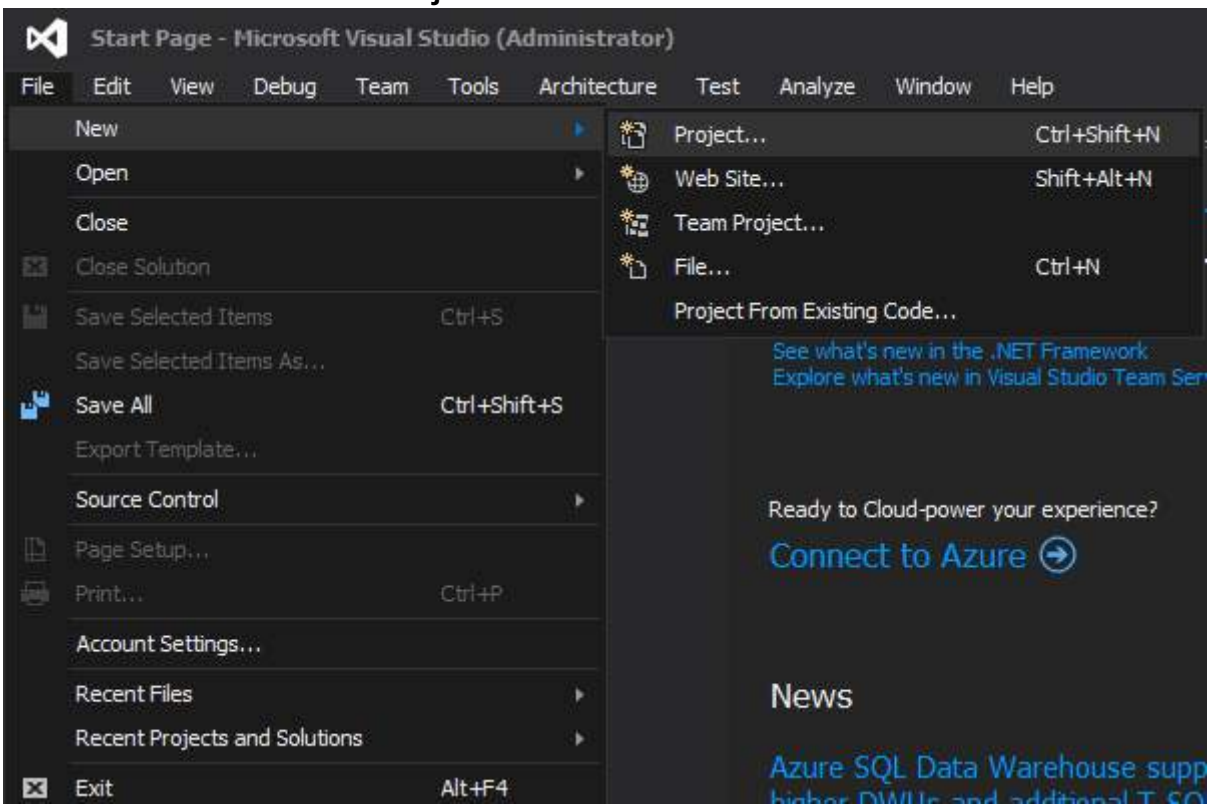
```
C:\Users\Main\Documents\helloworld>HelloWorld
Hello, world!
```

You may also double click the executable and launch a new console window with the message "**Hello, world!**"

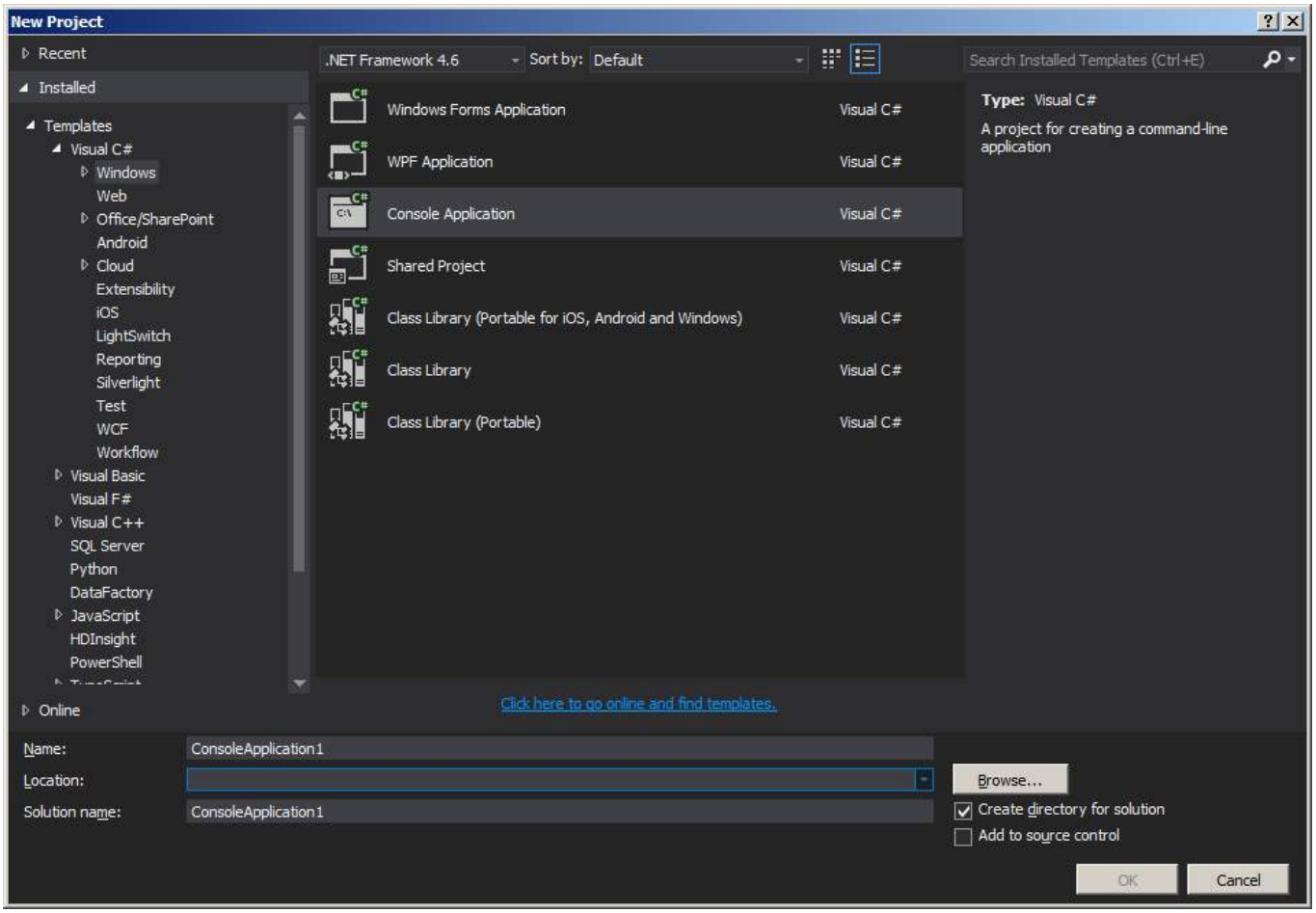


Section 1.2: Creating a new project in Visual Studio (console application) and Running it in Debug mode

1. **Download and install Visual Studio**. Visual Studio can be downloaded from VisualStudio.com. The Community edition is suggested, first because it is free, and second because it involves all the general features and can be extended further.
2. **Open Visual Studio.**
3. **Welcome.** Go to **File** → **New** → **Project**.

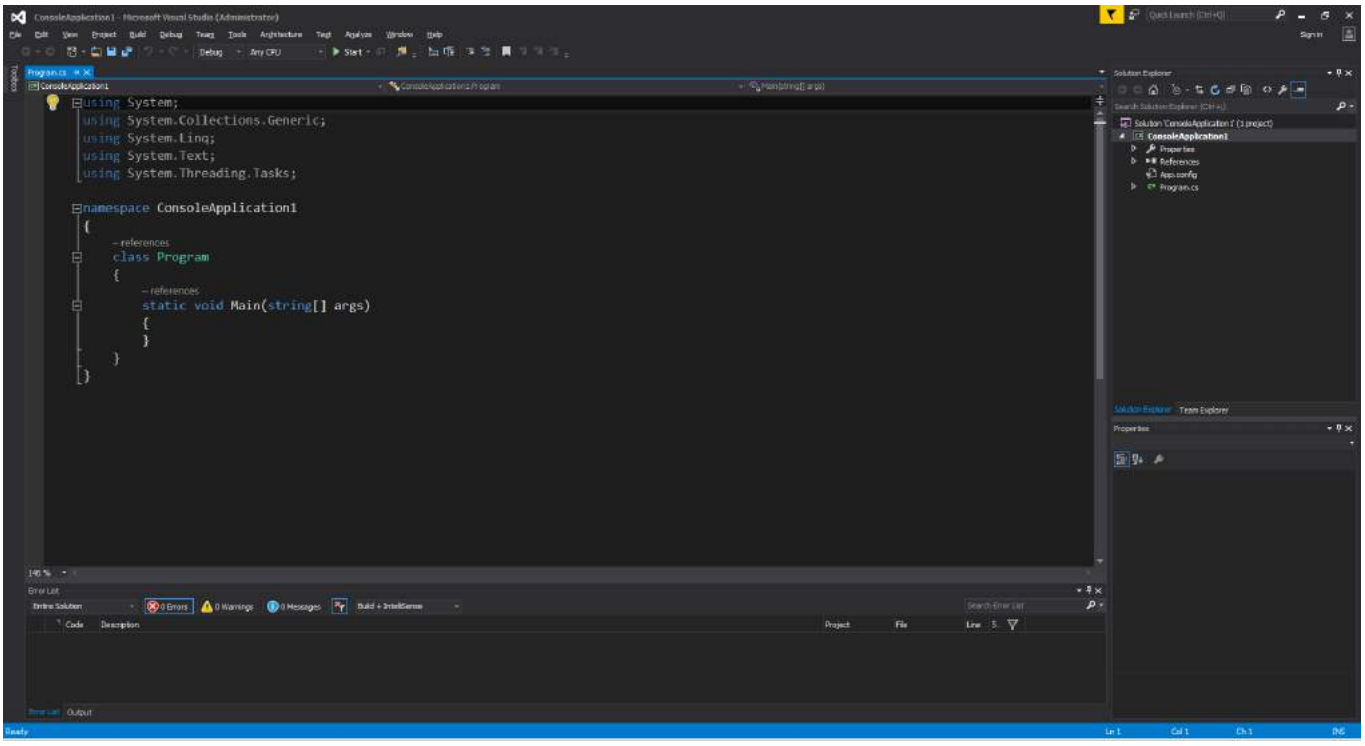


4. Click **Templates** → **Visual C#** → **Console Application**



5. **After selecting Console Application,** Enter a name for your project, and a location to save and press `OK`. Don't worry about the Solution name.

6. **Project created**. The newly created project will look similar to:



(Always use descriptive names for projects so that they can easily be distinguished from other projects. It is recommended not to use spaces in project or class name.)

7. **Write code.** You can now update your `Program.cs` to present "Hello world!" to the user.

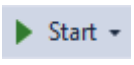
```
using System;

namespace ConsoleApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

Add the following two lines to the `public static void Main(string[] args)` object in `Program.cs`: (make sure it's inside the braces)

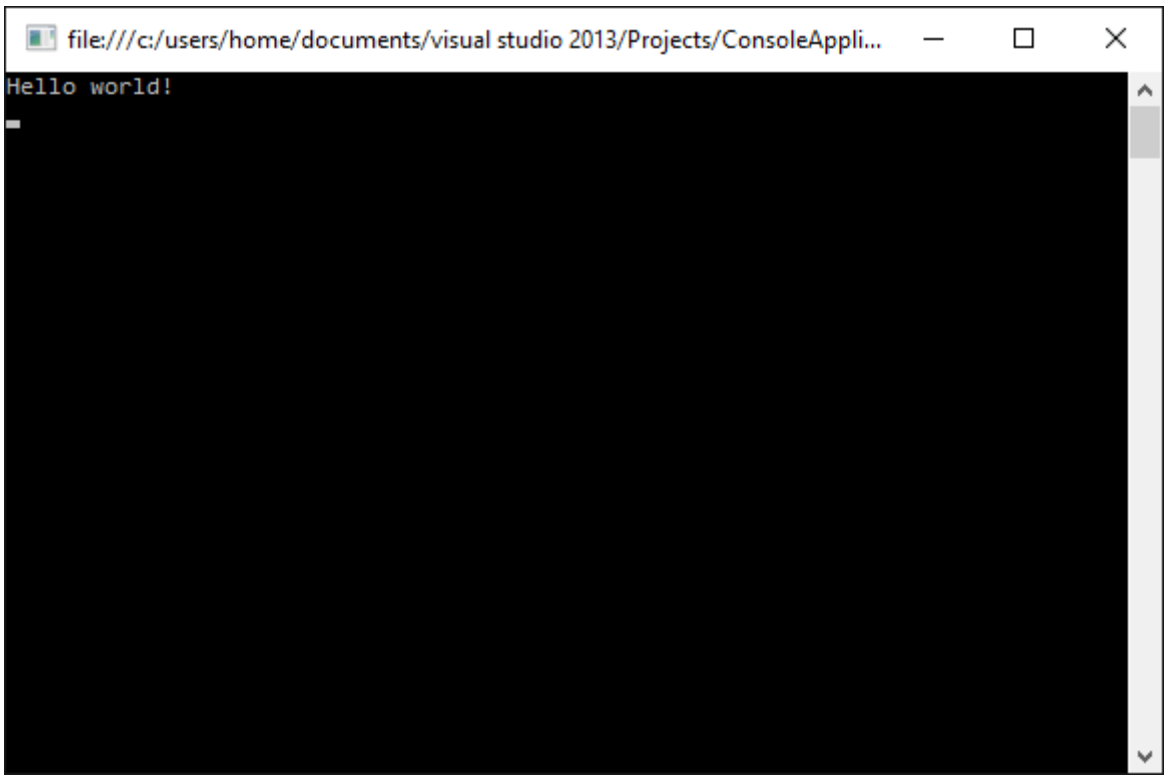
```
Console.WriteLine("Hello world!");
Console.Read();
```

Why `Console.Read()`? The first line prints out the text "Hello world!" to the console, and the second line waits for a single character to be entered; in effect, this causes the program to pause execution so that you're able to see the output while debugging. Without `Console.Read();`, when you start debugging the application it will just print "Hello world!" to the console and then immediately close. Your code window should now look like the following:

```
using System;

namespace ConsoleApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
            Console.Read();
        }
    }
}
```

8. **Debug your program.** Press the Start Button on the toolbar near the top of the window 

or press F5 on your keyboard to run your application. If the button is not present, you can run the program from the top menu: **Debug** → **Start Debugging**. The program will compile and then open a console window. It should look similar to the following screenshot:



9. **Stop the program.** To close the program, just press any key on your keyboard. The `Console.Read()` we added was for this same purpose. Another way to close the program is by going to the menu where the `Start` button was, and clicking on the `Stop` button.

Section 1.3: Creating a new program using .NET Core

First install the **.NET Core SDK** by going through the installation instructions for the platform of your choice:

- Windows
- OSX
- Linux
- Docker

After the installation has completed, open a command prompt, or terminal window.

1. Create a new directory with `mkdir hello_world` and change into the newly created directory with `cd hello_world`.
2. Create a new console application with `dotnet new console`.
   This will produce two files:

   - **hello_world.csproj**

     ```
     <Project Sdk="Microsoft.NET.Sdk">

       <PropertyGroup>
         <OutputType>Exe</OutputType>
         <TargetFramework>netcoreapp1.1</TargetFramework>
       </PropertyGroup>

     </Project>
     ```

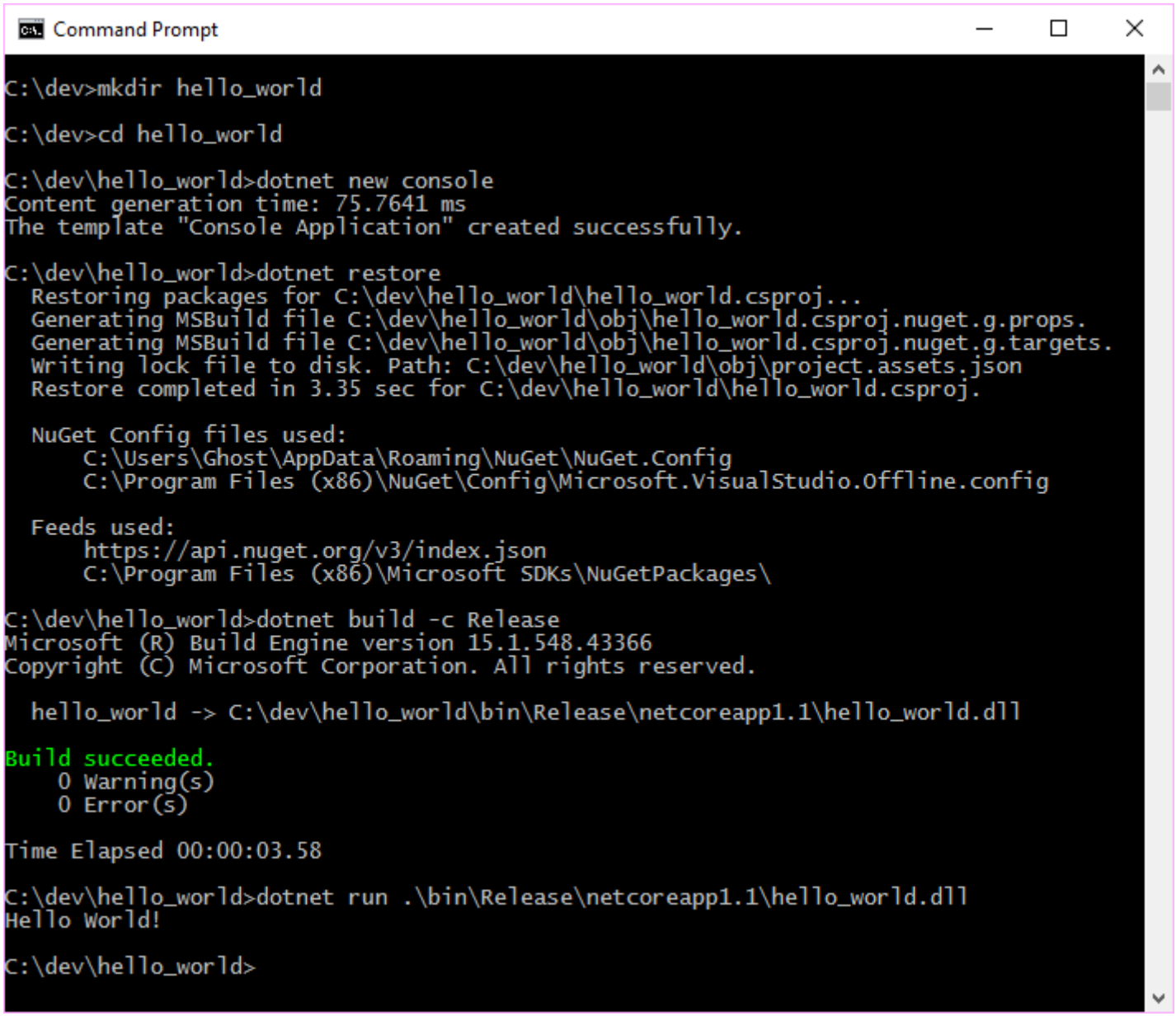
   - **Program.cs**

     ```
     using System;
     ```

```
namespace hello_world
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

3. Restore the needed packages with `dotnet restore`.
4. *Optional* Build the application with `dotnet build` for Debug or `dotnet build -c Release` for Release. `dotnet run` will also run the compiler and throw build errors, if any are found.
5. Run the application with `dotnet run` for Debug or `dotnet run .\bin\Release\netcoreapp1.1\hello_world.dll` for Release.

Command Prompt output



Section 1.4: Creating a new program using Mono

First install Mono by going through the install instructions for the platform of your choice as described in their installation section.

Mono is available for Mac OS X, Windows and Linux.

After installation is done, create a text file, name it `HelloWorld.cs` and copy the following content into it:

```
public class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, world!");
        System.Console.WriteLine("Press any key to exit..");
        System.Console.Read();
    }
}
```

If you are using Windows, run the Mono Command Prompt which is included in the Mono installation and ensures that the necessary environment variables are set. If on Mac or Linux, open a new terminal.

To compile the newly created file, run the following command in the directory containing `HelloWorld.cs`:

```
mcs -out:HelloWorld.exe HelloWorld.cs
```

The resulting `HelloWorld.exe` can then be executed with:

```
mono HelloWorld.exe
```

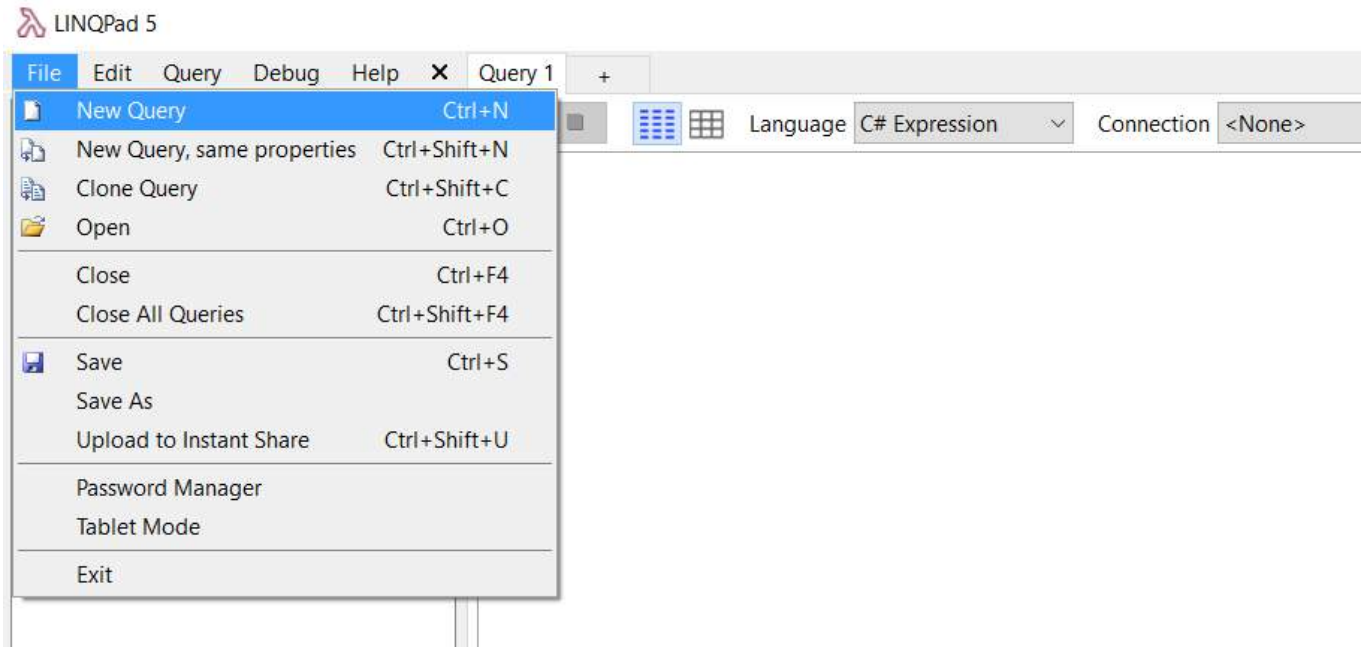
which will produce the output:

```
Hello, world!
Press any key to exit..
```

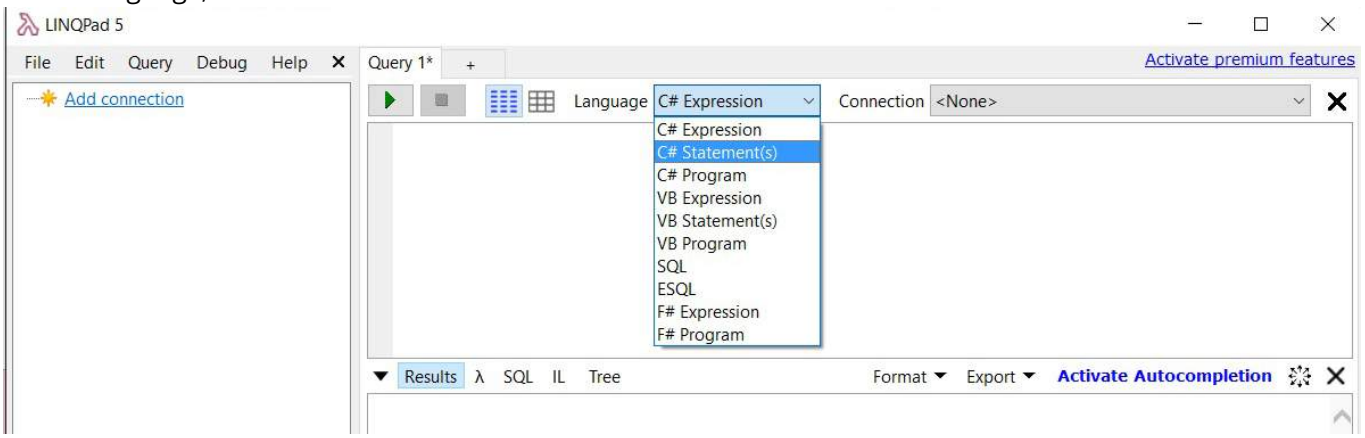
Section 1.5: Creating a new query using LinqPad

LinqPad is a great tool that allows you to learn and test features of .Net languages (C#, F# and VB.Net.)

1. Install LinqPad
2. Create a new Query (Ctrl + N)

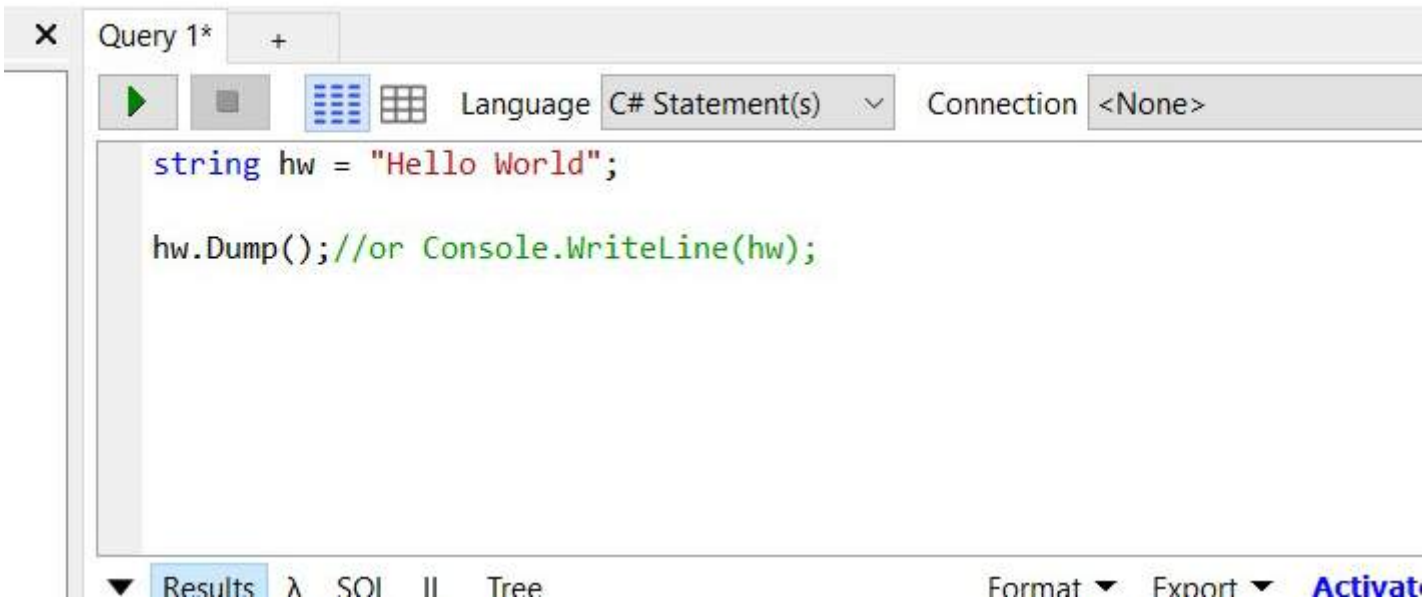


3. Under language, select "C# statements"

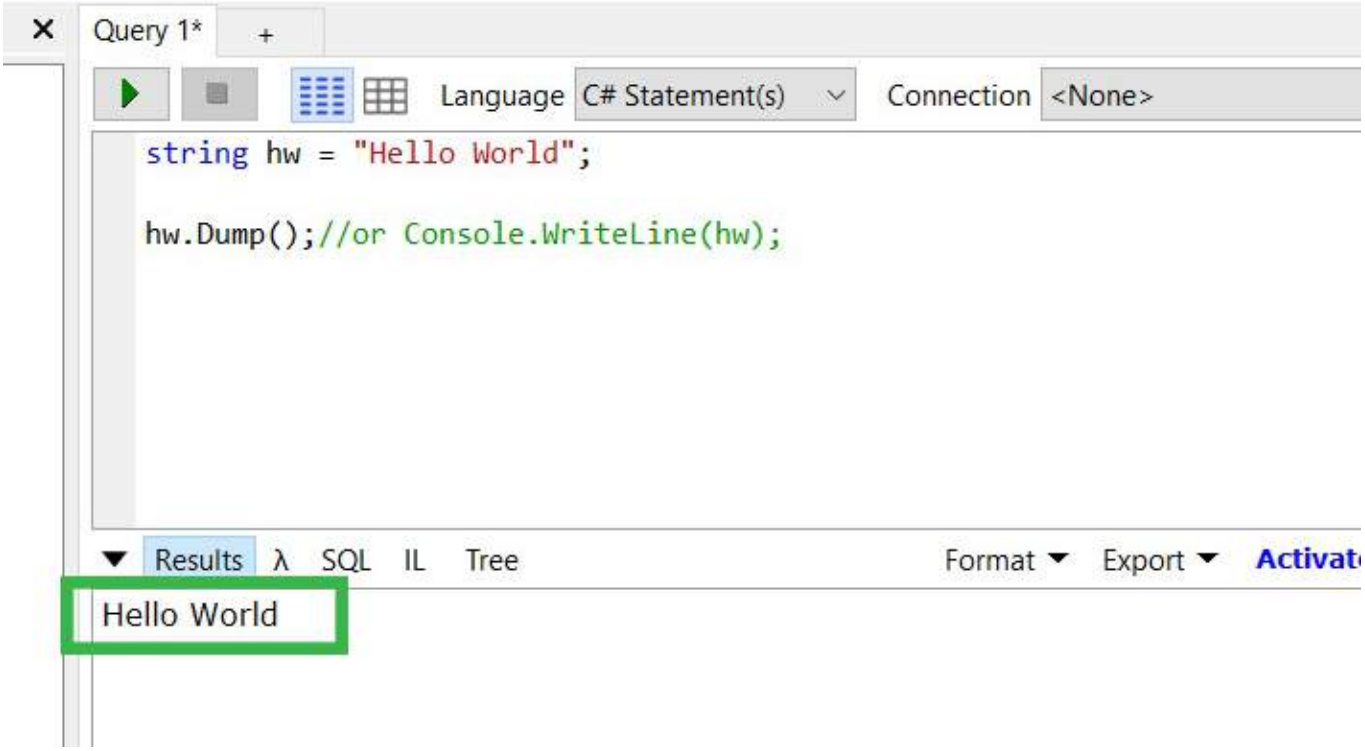


4. Type the following code and hit run (F5)

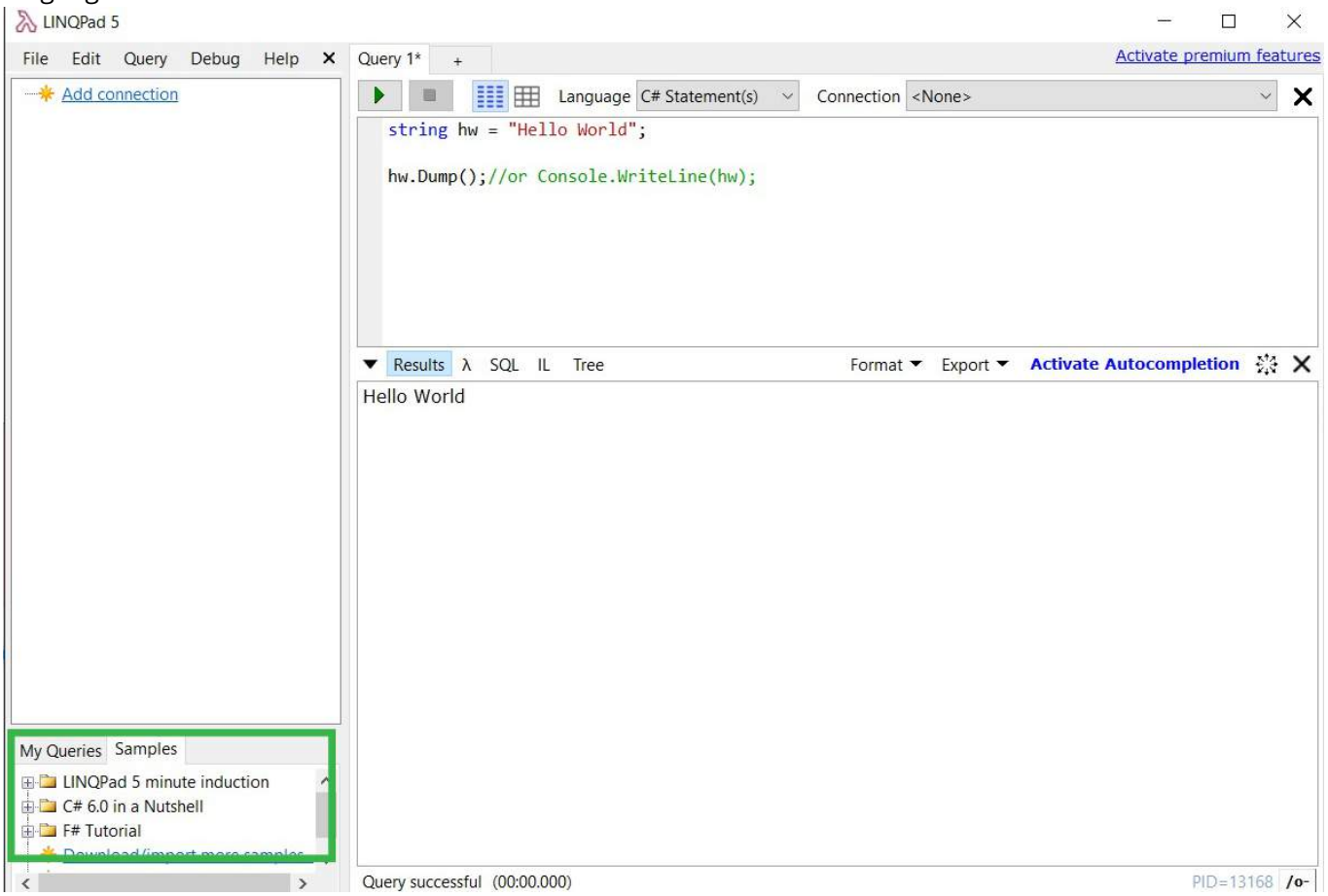
```
string hw = "Hello World";

hw.Dump(); //or Console.WriteLine(hw);
```



5. You should see "Hello World" printed out in the results screen.

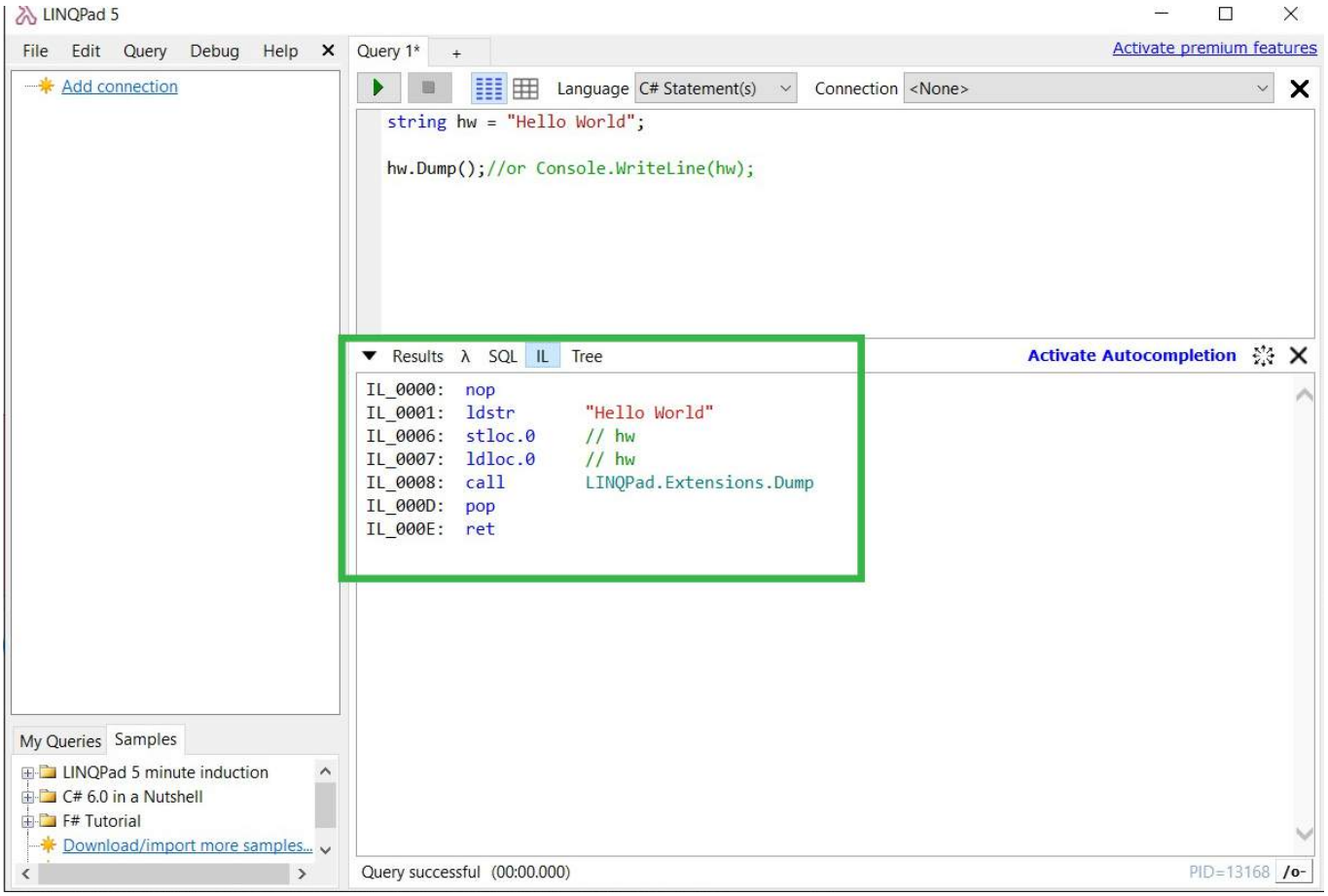


6. Now that you have created your first .Net program, go and check out the samples included in LinqPad via the "Samples" browser. There are many great examples that will show you many different features of the .Net languages.



Notes:

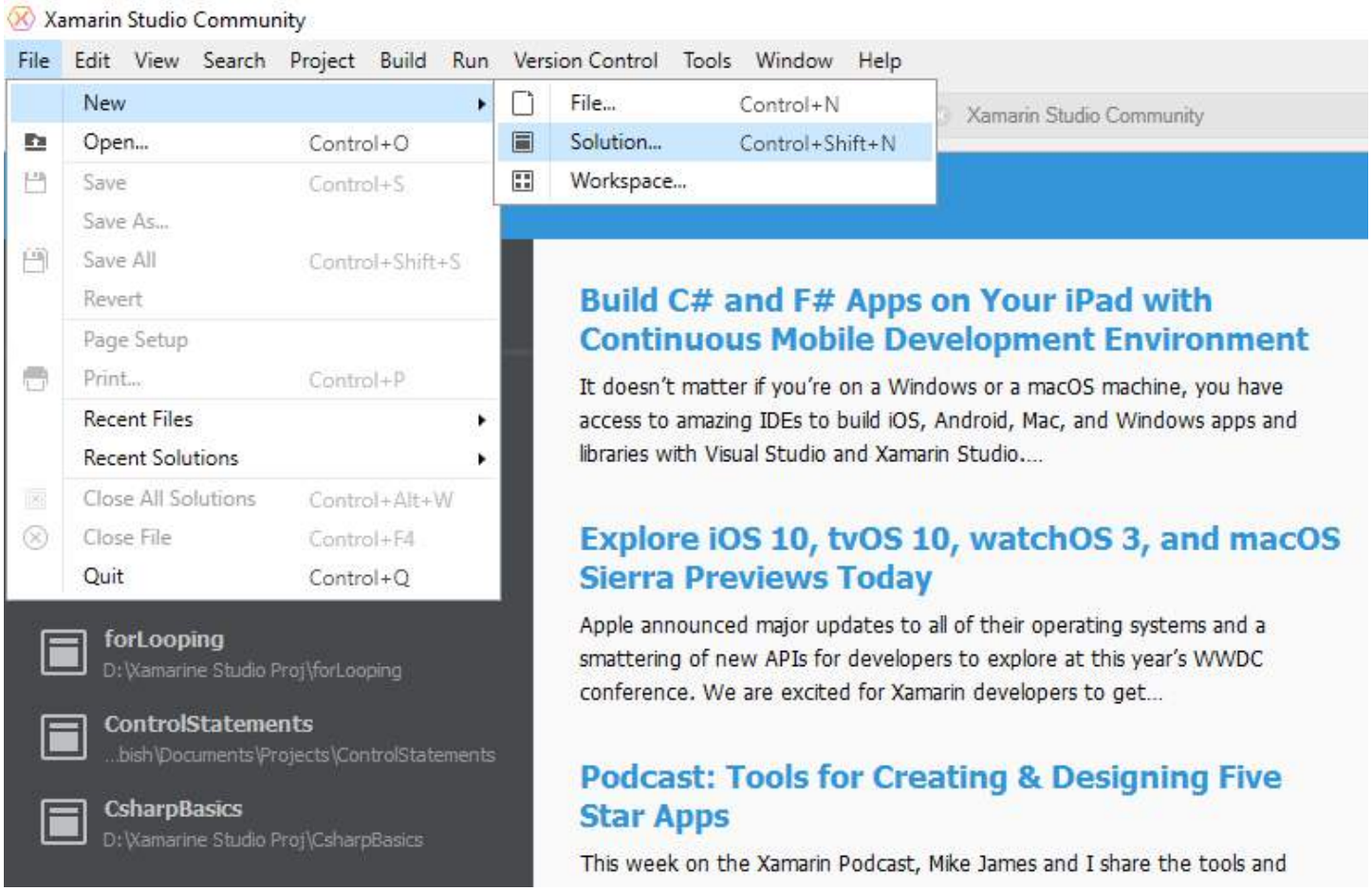
1. If you click on "IL", you can inspect the IL code that your .net code generates. This is a great learning tool.



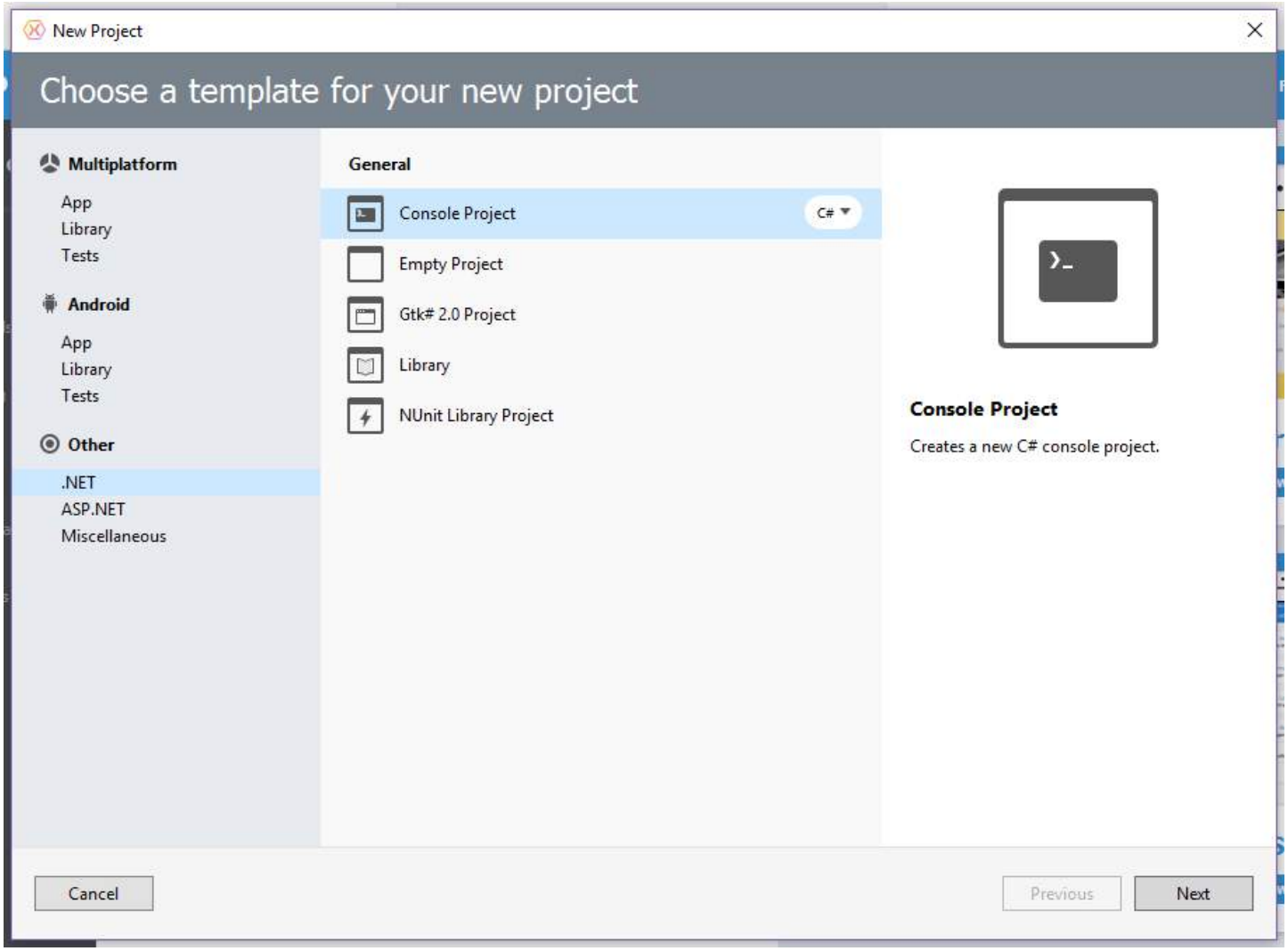
2. When using `LINQ to SQL` or `Linq to Entities` you can inspect the SQL that's being generated which is another great way to learn about LINQ.

Section 1.6: Creating a new project using Xamarin Studio

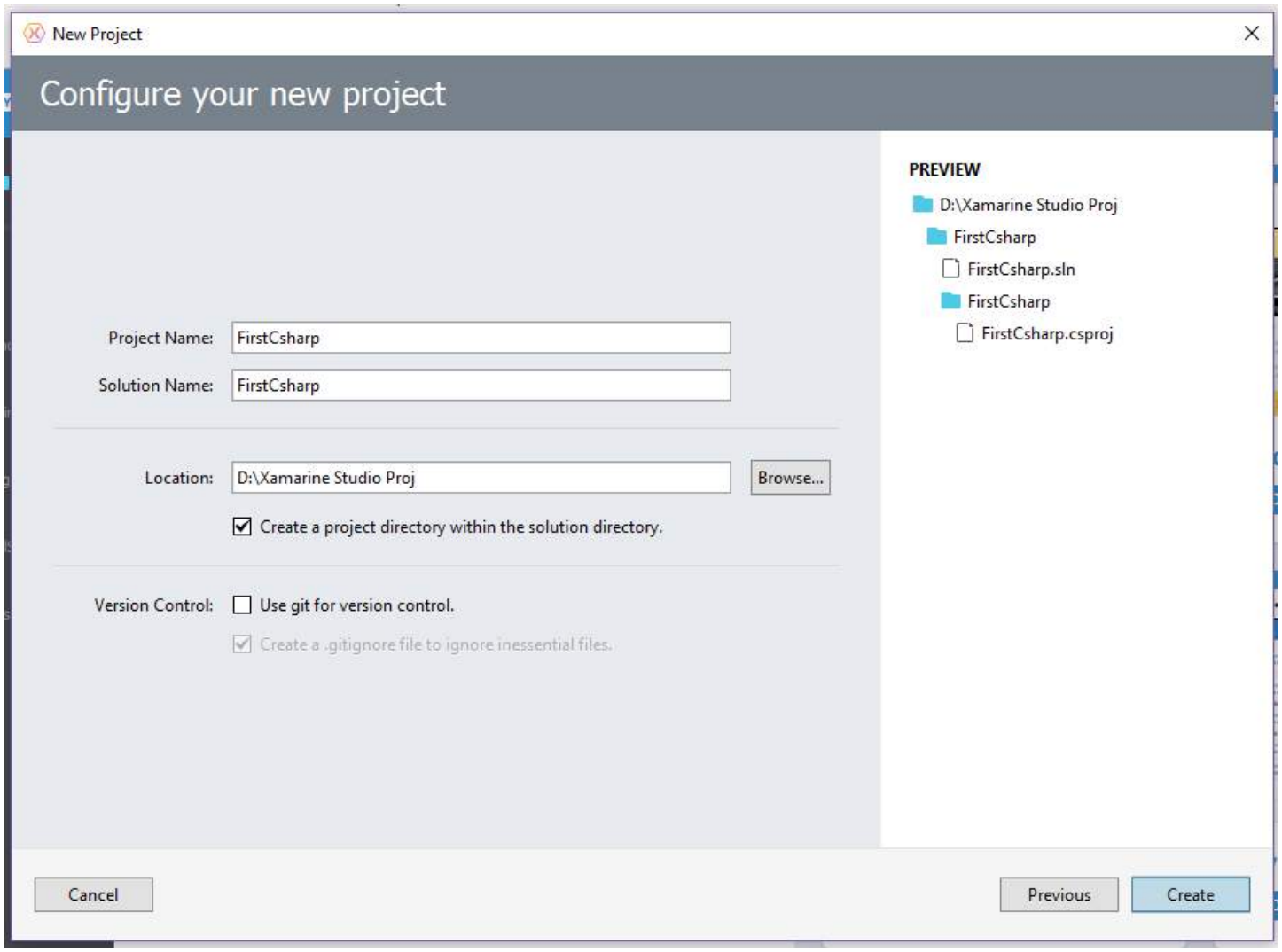
1. Download and install Xamarin Studio Community.
2. Open Xamarin Studio.
3. Click **File** → **New** → **Solution**.



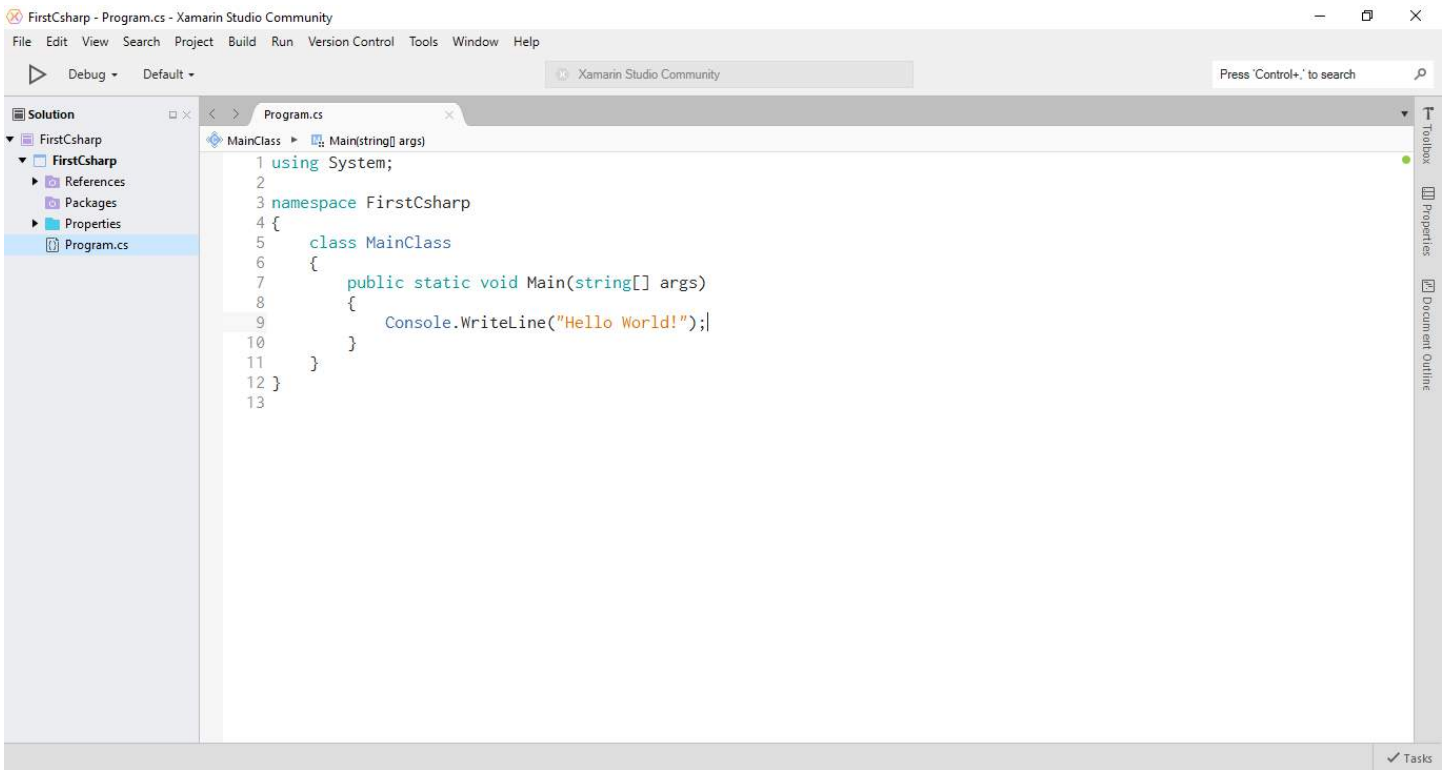
4. Click **.NET** → **Console Project** and choose **C#**.
5. Click `Next` to proceed.



6. Enter the **Project Name** and `Browse...` for a **Location** to Save and then click `Create`.



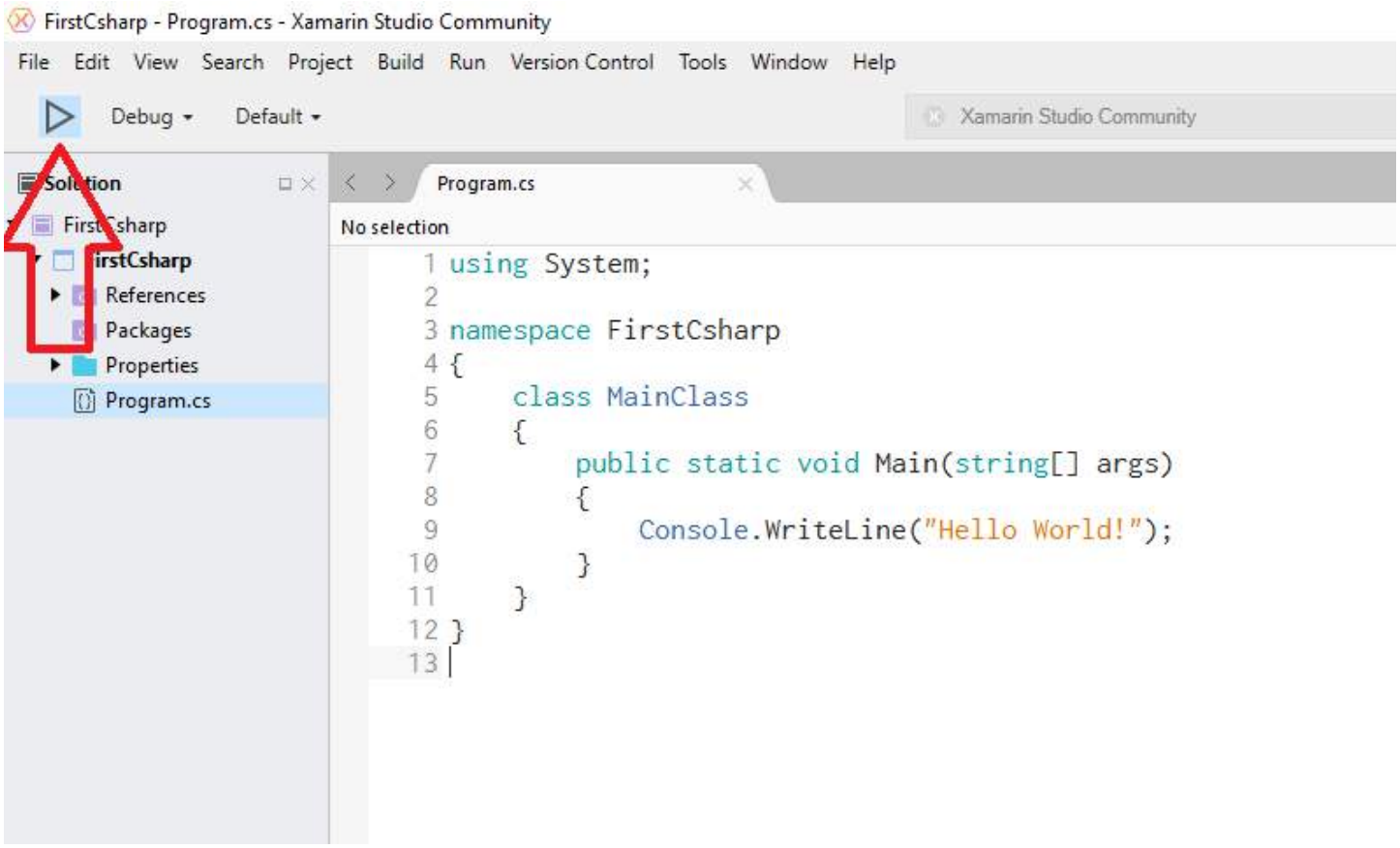
7. The newly created project will look similar to:



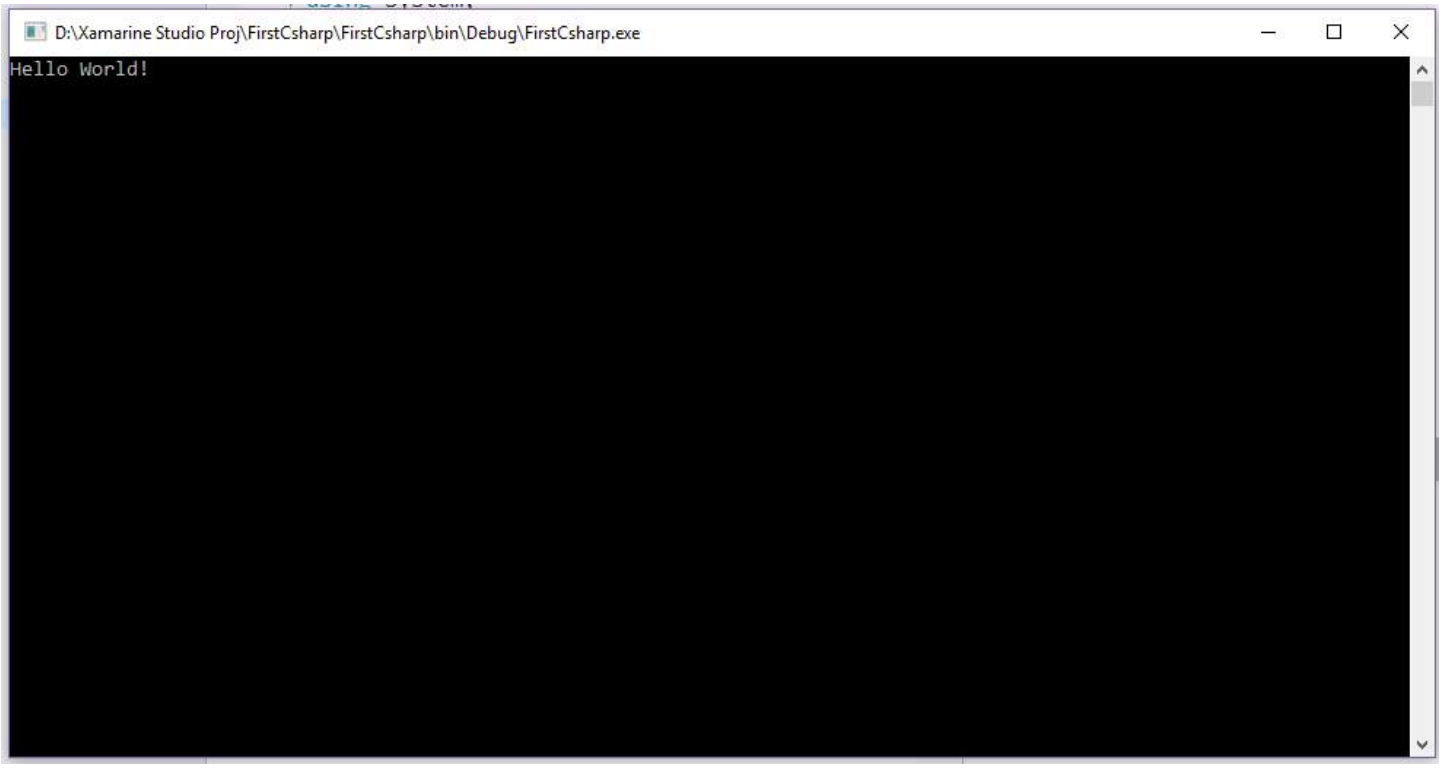
8. This is the code in the Text Editor:

```
using System;

namespace FirstCsharp
{
    public class MainClass
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.ReadLine();
        }
    }
}
```

9. To run the code, press F5 or click the **Play Button** as shown below:



10. Following is the Output:


D:\Xamarine Studio Proj\FirstCsharp\FirstCsharp\bin\Debug\FirstCsharp.exe
Hello World!

Chapter 2: Literals

Section 2.1: uint literals

`uint` literals are defined by using the suffix `U` or `u`, or by using an integral values within the range of `uint`:

```
uint ui = 5U;
```

Section 2.2: int literals

`int` literals are defined by simply using integral values within the range of `int`:

```
int i = 5;
```

Section 2.3: sbyte literals

`sbyte` type has no literal suffix. Integer literals are implicitly converted from `int`:

```
sbyte sb = 127;
```

Section 2.4: decimal literals

`decimal` literals are defined by using the suffix M or m on a real number:

```
decimal m = 30.5M;
```

Section 2.5: double literals

`double` literals are defined by using the suffix D or d, or by using a real number:

```
double d = 30.5D;
```

Section 2.6: float literals

`float` literals are defined by using the suffix F or f, or by using a real number:

```
float f = 30.5F;
```

Section 2.7: long literals

`long` literals are defined by using the suffix `L` or `l`, or by using an integral values within the range of `long`:

```
long l = 5L;
```

Section 2.8: ulong literal

`ulong` literals are defined by using the suffix `UL`, `ul`, `Ul`, `uL`, `LU`, `lu`, `Lu`, or `lU`, or by using an integral values within the range of `ulong`:

```
ulong ul = 5UL;
```

Section 2.9: string literals

`string` literals are defined by wrapping the value with double-quotes `"`:

```
string s = "hello, this is a string literal";
```

String literals may contain escape sequences. See String Escape Sequences

Additionally, C# supports verbatim string literals (See Verbatim Strings). These are defined by wrapping the value with double-quotes `"`, and prepending it with `@`. Escape sequences are ignored in verbatim string literals, and all whitespace characters are included:

```
string s = @"The path is:
C:\Windows\System32";
//The backslashes and newline are included in the string
```

Section 2.10: char literals

`char` literals are defined by wrapping the value with single-quotes `'`:

```
char c = 'h';
```

Character literals may contain escape sequences. See String Escape Sequences

A character literal must be exactly one character long (after all escape sequences have been evaluated). Empty character literals are not valid. The default character (returned by `default(char)` or `new char()`) is `'\0'`, or the NULL character (not to be confused with the `null` literal and null references).

Section 2.11: byte literals

`byte` type has no literal suffix. Integer literals are implicitly converted from `int`:

```
byte b = 127;
```

Section 2.12: short literal

`short` type has no literal. Integer literals are implicitly converted from `int`:

```
short s = 127;
```

Section 2.13: ushort literal

`ushort` type has no literal suffix. Integer literals are implicitly converted from `int`:

```
ushort us = 127;
```

Section 2.14: bool literals

`bool` literals are either `true` or `false`;

```
bool b = true;
```

Chapter 3: Operators

Parameter	Details
operatorSymbol	The operator being overloaded, e.g. +, -, /, *
OperandType	The type that will be returned by the overloaded operator.
operand1	The first operand to be used in performing the operation.
operand2	The second operand to be used in performing the operation, when doing binary operations.
statements	Optional code needed to perform the operation before returning the result.

In C#, an operator is a program element that is applied to one or more operands in an expression or statement. Operators that take one operand, such as the increment operator (++) or new, are referred to as unary operators. Operators that take two operands, such as arithmetic operators (+,-,*,/), are referred to as binary operators. One operator, the conditional operator (?:), takes three operands and is the sole ternary operator in C#.

Section 3.1: Overloadable Operators

C# allows user-defined types to overload operators by defining static member functions using the `operator` keyword.
The following example illustrates an implementation of the + operator.

If we have a `Complex` class which represents a complex number:

```
public struct Complex
{
    public double Real { get; set; }
    public double Imaginary { get; set; }
}
```

And we want to add the option to use the `+` operator for this class. i.e.:

```
Complex a = new Complex() { Real = 1, Imaginary = 2 };
Complex b = new Complex() { Real = 4, Imaginary = 8 };
Complex c = a + b;
```

We will need to overload the `+` operator for the class. This is done using a static function and the `operator` keyword:

```
public static Complex operator +(Complex c1, Complex c2)
{
   return new Complex
   {
       Real = c1.Real + c2.Real,
       Imaginary = c1.Imaginary + c2.Imaginary
   };
}
```

Operators such as `+`, `-`, `*`, `/` can all be overloaded. This also includes Operators that don't return the same type (for example, `==` and `!=` can be overloaded, despite returning booleans) The rule below relating to pairs is also enforced here.

Comparison operators have to be overloaded in pairs (e.g. if `<` is overloaded, `>` also needs to be overloaded).

A full list of overloadable operators (as well as non-overloadable operators and the restrictions placed on some overloadable operators) can be seen at MSDN - Overloadable Operators (C# Programming Guide).

Version ≥ 7.0

overloading of `operator is` was introduced with the pattern matching mechanism of C# 7.0. For details see Pattern Matching

Given a type `Cartesian` defined as follows

```
public class Cartesian
{
    public int X { get; }
    public int Y { get; }
}
```

An overloadable `operator is` could e.g. be defined for `Polar` coordinates

```
public static class Polar
{
    public static bool operator is(Cartesian c, out double R, out double Theta)
    {
        R = Math.Sqrt(c.X*c.X + c.Y*c.Y);
        Theta = Math.Atan2(c.Y, c.X);
        return c.X != 0 || c.Y != 0;
    }
}
```

which can be used like this

```
var c = Cartesian(3, 4);
if (c is Polar(var R, *))
{
    Console.WriteLine(R);
}
```

(The example is taken from the Roslyn Pattern Matching Documentation)

Section 3.2: Overloading equality operators

Overloading just equality operators is not enough. Under different circumstances, all of the following can be called:

1. `object.Equals` and `object.GetHashCode`
2. `IEquatable<T>.Equals` (optional, allows avoiding boxing)
3. `operator ==` and `operator !=` (optional, allows using operators)

When overriding `Equals`, `GetHashCode` must also be overridden. When implementing `Equals`, there are many special cases: comparing to objects of a different type, comparing to self etc.

When NOT overridden `Equals` method and `==` operator behave differently for classes and structs. For classes just references are compared, and for structs values of properties are compared via reflection what can negatively affect performance. `==` can not be used for comparing structs unless it is overridden.

Generally equality operation must obey the following rules:

- Must not *throw exceptions*.
- Reflexivity: `A` always equals `A` (may not be true for `NULL` values in some systems).
- Transitivity: if `A` equals `B`, and `B` equals `C`, then `A` equals `C`.
- If `A` equals `B`, then `A` and `B` have equal hash codes.

- Inheritance tree independence: if `B` and `C` are instances of `Class2` inherited from `Class1`: `Class1.Equals(A,B)` must always return the same value as the call to `Class2.Equals(A,B)`.

```
class Student : IEquatable<Student>
{
    public string Name { get; set; } = "";

    public bool Equals(Student other)
    {
        if (ReferenceEquals(other, null)) return false;
        if (ReferenceEquals(other, this)) return true;
        return string.Equals(Name, other.Name);
    }

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;

        return Equals(obj as Student);
    }

    public override int GetHashCode()
    {
        return Name?.GetHashCode() ?? 0;
    }

    public static bool operator ==(Student left, Student right)
    {
        return Equals(left, right);
    }

    public static bool operator !=(Student left, Student right)
    {
        return !Equals(left, right);
    }
}
```

Section 3.3: Relational Operators

Equals

Checks whether the supplied operands (arguments) are equal

```
"a" == "b"      // Returns false.
"a" == "a"      // Returns true.
1 == 0          // Returns false.
1 == 1          // Returns true.
false == true   // Returns false.
false == false // Returns true.
```

Unlike Java, the equality comparison operator works natively with strings.

The equality comparison operator will work with operands of differing types if an implicit cast exists from one to the other. If no suitable implicit cast exists, you may call an explicit cast or use a method to convert to a compatible type.

```
1 == 1.0              // Returns true because there is an implicit cast from int to double.
new Object() == 1.0   // Will not compile.
```

```
MyStruct.AsInt() == 1 // Calls AsInt() on MyStruct and compares the resulting int with 1.
```

Unlike Visual Basic.NET, the equality comparison operator is not the same as the equality assignment operator.

```
var x = new Object();
var y = new Object();
x == y // Returns false, the operands (objects in this case) have different references.
x == x // Returns true, both operands have the same reference.
```

Not to be confused with the assignment operator (=).

For value types, the operator returns `true` if both operands are equal in value.
For reference types, the operator returns `true` if both operands are equal in *reference* (not value). An exception is that string objects will be compared with value equality.

Not Equals

Checks whether the supplied operands are *not* equal.

```
"a" != "b"     // Returns true.
"a" != "a"     // Returns false.
1 != 0         // Returns true.
1 != 1         // Returns false.
false != true  // Returns true.
false != false // Returns false.

var x = new Object();
var y = new Object();
x != y // Returns true, the operands have different references.
x != x // Returns false, both operands have the same reference.
```

This operator effectively returns the opposite result to that of the equals (`==`) operator

Greater Than

Checks whether the first operand is greater than the second operand.

```
3 > 5    //Returns false.
1 > 0    //Returns true.
2 > 2    //Return false.

var x = 10;
var y = 15;
x > y    //Returns false.
y > x    //Returns true.
```

Less Than

Checks whether the first operand is less than the second operand.

```
2 < 4     //Returns true.
1 < -3    //Returns false.
2 < 2     //Return false.

var x = 12;
var y = 22;
x < y    //Returns true.
y < x    //Returns false.
```

Greater Than Equal To

Checks whether the first operand is greater than equal to the second operand.

```
7 >= 8    //Returns false.
0 >= 0    //Returns true.
```

Less Than Equal To

Checks whether the first operand is less than equal to the second operand.

```
2 <= 4    //Returns true.
1 <= -3    //Returns false.
1 <= 1     //Returns true.
```

Section 3.4: Implicit Cast and Explicit Cast Operators

C# allows user-defined types to control assignment and casting through the use of the `explicit` and `implicit` keywords. The signature of the method takes the form:

```
public static <implicit/explicit> operator <ResultingType>(<SourceType> myType)
```

The method cannot take any more arguments, nor can it be an instance method. It can, however, access any private members of type it is defined within.

An example of both an `implicit` and `explicit` cast:

```
public class BinaryImage
{
    private bool[] _pixels;

    public static implicit operator ColorImage(BinaryImage im)
    {
        return new ColorImage(im);
    }

    public static explicit operator bool[](BinaryImage im)
    {
        return im._pixels;
    }
}
```

Allowing the following cast syntax:

```
var binaryImage = new BinaryImage();
ColorImage colorImage = binaryImage; // implicit cast, note the lack of type
bool[] pixels = (bool[])binaryImage; // explicit cast, defining the type
```

The cast operators can work both ways, going *from* your type and going *to* your type:

```
public class BinaryImage
{
    public static explicit operator ColorImage(BinaryImage im)
    {
        return new ColorImage(im);
    }
```

```
    public static explicit operator BinaryImage(ColorImage cm)
    {
        return new BinaryImage(cm);
    }
}
```

Finally, the `as` keyword, which can be involved in casting within a type hierarchy, is **not** valid in this situation. Even after defining either an `explicit` or `implicit` cast, you cannot do:

```
ColorImage cm = myBinaryImage as ColorImage;
```

It will generate a compilation error.

Section 3.5: Short-circuiting Operators

By definition, the short-circuiting boolean operators will only evaluate the second operand if the first operand can not determine the overall result of the expression.

It means that, if you are using && operator as *firstCondition && secondCondition* it will evaluate *secondCondition* only when *firstCondition* is true and ofcource the overall result will be true only if both of *firstOperand* and *secondOperand* are evaluated to true. This is useful in many scenarios, for example imagine that you want to check whereas your list has more than three elements but you also have to check if list has been initialized to not run into *NullReferenceException*. You can achieve this as below:

```
bool hasMoreThanThreeElements = myList != null && mList.Count > 3;
```

mList.Count > 3 will not be checked untill myList != null is met.

Logical AND

`&&` is the short-circuiting counterpart of the standard boolean AND (`&`) operator.

```
var x = true;
var y = false;

x && x // Returns true.
x && y // Returns false (y is evaluated).
y && x // Returns false (x is not evaluated).
y && y // Returns false (right y is not evaluated).
```

Logical OR

`||` is the short-circuiting counterpart of the standard boolean OR (`|`) operator.

```
var x = true;
var y = false;

x || x // Returns true (right x is not evaluated).
x || y // Returns true (y is not evaluated).
y || x // Returns true (x and y are evaluated).
y || y // Returns false (y and y are evaluated).
```

Example usage

```
if(object != null && object.Property)
// object.Property is never accessed if object is null, because of the short circuit.
```

```
    Action1();
else
    Action2();
```

Section 3.6: ? : Ternary Operator

Returns one of two values depending on the value of a Boolean expression.

Syntax:

```
condition ? expression_if_true : expression_if_false;
```

Example:

```
string name = "Frank";
Console.WriteLine(name == "Frank" ? "The name is Frank" : "The name is not Frank");
```

The ternary operator is right-associative which allows for compound ternary expressions to be used. This is done by adding additional ternary equations in either the true or false position of a parent ternary equation. Care should be taken to ensure readability, but this can be useful shorthand in some circumstances.

In this example, a compound ternary operation evaluates a `clamp` function and returns the current value if it's within the range, the `min` value if it's below the range, or the `max` value if it's above the range.

```
light.intensity = Clamp(light.intensity, minLight, maxLight);

public static float Clamp(float val, float min, float max)
{
    return (val < min) ? min : (val > max) ? max : val;
}
```

Ternary operators can also be nested, such as:

```
a ? b ? "a is true, b is true" : "a is true, b is false" : "a is false"

// This is evaluated from left to right and can be more easily seen with parenthesis:

a ? (b ? x : y) : z

// Where the result is x if a && b, y if a && !b, and z if !a
```

When writing compound ternary statements, it's common to use parenthesis or indentation to improve readability.

The types of *expression_if_true* and *expression_if_false* must be identical or there must be an implicit conversion from one to the other.

```
condition ? 3 : "Not three"; // Doesn't compile because `int` and `string` lack an implicit
conversion.

condition ? 3.ToString() : "Not three"; // OK because both possible outputs are strings.

condition ? 3 : 3.5; // OK because there is an implicit conversion from `int` to `double`. The
ternary operator will return a `double`.

condition ? 3.5 : 3; // OK because there is an implicit conversion from `int` to `double`. The
ternary operator will return a `double`.
```

The type and conversion requirements apply to your own classes too.

```
public class Car
{}

public class SportsCar : Car
{}

public class SUV : Car
{}

condition ? new SportsCar() : new Car(); // OK because there is an implicit conversion from
`SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new Car() : new SportsCar(); // OK because there is an implicit conversion from
`SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new SportsCar() : new SUV(); // Doesn't compile because there is no implicit conversion
from `SportsCar` to SUV or `SUV` to `SportsCar`. The compiler is not smart enough to realize that
both of them have an implicit conversion to `Car`.

condition ? new SportsCar() as Car : new SUV() as Car; // OK because both expressions evaluate to a
reference of type `Car`. The ternary operator will return a reference of type `Car`.
```

Section 3.7: ?. (Null Conditional Operator)

Version ≥ 6.0

Introduced in C# 6.0, the Null Conditional Operator `?.` will immediately return `null` if the expression on its left-hand side evaluates to `null`, instead of throwing a `NullReferenceException`. If its left-hand side evaluates to a non-`null` value, it is treated just like a normal `.` operator. Note that because it might return `null`, its return type is always a nullable type. That means that for a struct or primitive type, it is wrapped into a `Nullable<T>`.

```
var bar = Foo.GetBar()?.Value; // will return null if GetBar() returns null
var baz = Foo.GetBar()?.IntegerValue; // baz will be of type Nullable<int>, i.e. int?
```

This comes handy when firing events. Normally you would have to wrap the event call in an if statement checking for `null` and raise the event afterwards, which introduces the possibility of a race condition. Using the Null conditional operator this can be fixed in the following way:

```
event EventHandler<string> RaiseMe;
RaiseMe?.Invoke("Event raised");
```

Section 3.8: "Exclusive or" Operator

The operator for an "exclusive or" (for short XOR) is: ^

This operator returns true when one, but only one, of the supplied bools are true.

```
true ^ false   // Returns true
false ^ true   // Returns true
false ^ false  // Returns false
true ^ true    // Returns false
```

Section 3.9: default Operator

Value Type (where T : struct)

The built-in primitive data types, such as `char`, `int`, and `float`, as well as user-defined types declared with `struct`, or `enum`. Their default value is `new T()` :

```
default(int)            // 0
default(DateTime)       // 0001-01-01 12:00:00 AM
default(char)           // '\0' This is the "null character", not a zero or a line break.
default(Guid)           // 00000000-0000-0000-0000-000000000000
default(MyStruct)       // new MyStruct()

// Note: default of an enum is 0, and not the first *key* in that enum
// so it could potentially fail the Enum.IsDefined test
default(MyEnum)         // (MyEnum)0
```

Reference Type (where T : class)

Any `class`, `interface`, array or delegate type. Their default value is `null` :

```
default(object)         // null
default(string)         // null
default(MyClass)        // null
default(IDisposable)    // null
default(dynamic)        // null
```

Section 3.10: Assignment operator '='

The assignment operator `=` sets thr left hand operand's value to the value of right hand operand, and return that value:

```
int a = 3;     // assigns value 3 to variable a
int b = a = 5; // first assigns value 5 to variable a, then does the same for variable b
Console.WriteLine(a = 3 + 4); // prints 7
```

Section 3.11: sizeof

Returns an `int` holding the size of a type* in bytes.

```
sizeof(bool)    // Returns 1.
sizeof(byte)    // Returns 1.
sizeof(sbyte)   // Returns 1.
sizeof(char)    // Returns 2.
sizeof(short)   // Returns 2.
sizeof(ushort)  // Returns 2.
sizeof(int)     // Returns 4.
sizeof(uint)    // Returns 4.
sizeof(float)   // Returns 4.
sizeof(long)    // Returns 8.
sizeof(ulong)   // Returns 8.
sizeof(double)  // Returns 8.
sizeof(decimal) // Returns 16.
```

**Only supports certain primitive types in safe context.*

In an unsafe context, `sizeof` can be used to return the size of other primitive types and structs.

```
public struct CustomType
{
    public int value;
}

static void Main()
{
    unsafe
    {
        Console.WriteLine(sizeof(CustomType)); // outputs: 4
    }
}
```

Section 3.12: ?? Null-Coalescing Operator

The Null-Coalescing operator `??` will return the left-hand side when not null. If it is null, it will return the right-hand side.

```
object foo = null;
object bar = new object();

var c = foo ?? bar;
//c will be bar since foo was null
```

The `??` operator can be chained which allows the removal of `if` checks.

```
//config will be the first non-null returned.
var config = RetrieveConfigOnMachine() ??
             RetrieveConfigFromService() ??
             new DefaultConfiguration();
```

Section 3.13: Bit-Shifting Operators

The shift operators allow programmers to adjust an integer by shifting all of its bits to the left or the right. The following diagram shows the affect of shifting a value to the left by one digit.

Left-Shift

```
uint value = 15;              // 00001111

uint doubled = value << 1;    // Result = 00011110 = 30
uint shiftFour = value << 4;  // Result = 11110000 = 240
```

Right-Shift

```
uint value = 240;             // 11110000

uint halved = value >> 1;     // Result = 01111000 = 120
uint shiftFour = value >> 4;  // Result = 00001111 = 15
```

Section 3.14: => Lambda operator

Version ≥ 3.0

The `=>` operator has the same precedence as the assignment operator `=` and is right-associative.

It is used to declare lambda expressions and also it is widely used with LINQ Queries:

```
string[] words = { "cherry", "apple", "blueberry" };

int shortestWordLength = words.Min((string w) => w.Length); //5
```

When used in LINQ extensions or queries the type of the objects can usually be skipped as it is inferred by the compiler:

```
int shortestWordLength = words.Min(w => w.Length); //also compiles with the same result
```

The general form of lambda operator is the following:

```
(input parameters) => expression
```

The parameters of the lambda expression are specified before `=>` operator, and the actual expression/statement/block to be executed is to the right of the operator:

```
// expression
(int x, string s) => s.Length > x

// expression
(int x, int y) => x + y

// statement
(string x) => Console.WriteLine(x)

// block
(string x) => {
        x += " says Hello!";
        Console.WriteLine(x);
    }
```

This operator can be used to easily define delegates, without writing an explicit method:

```
delegate void TestDelegate(string s);

TestDelegate myDelegate = s => Console.WriteLine(s + " World");

myDelegate("Hello");
```

instead of

```
void MyMethod(string s)
{
    Console.WriteLine(s + " World");
}

delegate void TestDelegate(string s);

TestDelegate myDelegate = MyMethod;

myDelegate("Hello");
```

Section 3.15: Class Member Operators: Null Conditional Member Access

```
var zipcode = myEmployee?.Address?.ZipCode;
//returns null if the left operand is null.
//the above is the equivalent of:
var zipcode = (string)null;
if (myEmployee != null && myEmployee.Address != null)
    zipcode = myEmployee.Address.ZipCode;
```

Section 3.16: Class Member Operators: Null Conditional Indexing

```
var letters = null;
char? letter = letters?[1];
Console.WriteLine("Second Letter is {0}",letter);
//in the above example  rather than throwing an error because letters is null
//letter is assigned the value null
```

Section 3.17: Postfix and Prefix increment and decrement

Postfix increment `X++` will add 1 to x

```
var x = 42;
x++;
Console.WriteLine(x); // 43
```

Postfix decrement

`X--`

will subtract one

```
var x = 42
x--;
Console.WriteLine(x); // 41
```

`++x` is called prefix increment it increments the value of x and then returns x while `x++` returns the value of x and then increments

```
var x = 42;
Console.WriteLine(++x); // 43
System.out.println(x); // 43
```

while

```
var x = 42;
Console.WriteLine(x++); // 42
System.out.println(x); // 43
```

both are commonly used in for loop

```
for(int i = 0; i < 10; i++)
{
}
```

Section 3.18: typeof

Gets `System.Type` object for a type.

```
System.Type type = typeof(Point)        //System.Drawing.Point
System.Type type = typeof(IDisposable)  //System.IDisposable
System.Type type = typeof(Colors)       //System.Drawing.Color
System.Type type = typeof(List<>)       //System.Collections.Generic.List`1[T]
```

To get the run-time type, use `GetType` method to obtain the `System.Type` of the current instance.

Operator `typeof` takes a type name as parameter, which is specified at compile time.

```
public class Animal {}
public class Dog : Animal {}

var animal = new Dog();

Assert.IsTrue(animal.GetType() == typeof(Animal)); // fail, animal is typeof(Dog)
Assert.IsTrue(animal.GetType() == typeof(Dog));    // pass, animal is typeof(Dog)
Assert.IsTrue(animal is Animal);                   // pass, animal implements Animal
```

Section 3.19: Binary operators with assignment

C# has several operators that can be combined with an `=` sign to evaluate the result of the operator and then assign the result to the original variable.

Example:

```
x += y
```

is the same as

```
x = x + y
```

Assignment operators:

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`
- `&=`
- `|=`
- `^=`
- `<<=`
- `>>=`

Section 3.20: nameof Operator

Returns a string that represents the unqualified name of a `variable`, `type`, or `member`.

```
int counter = 10;
nameof(counter); // Returns "counter"
Client client = new Client();
```

```
nameof(client.Address.PostalCode)); // Returns "PostalCode"
```

The `nameof` operator was introduced in C# 6.0. It is evaluated at compile-time and the returned string value is inserted inline by the compiler, so it can be used in most cases where the constant string can be used (e.g., the `case` labels in a `switch` statement, attributes, etc...). It can be useful in cases like raising & logging exceptions, attributes, MVC Action links, etc...

Section 3.21: Class Member Operators: Member Access

```
var now = DateTime.UtcNow;
//accesses member of a class.  In this case the UtcNow property.
```

Section 3.22: Class Member Operators: Function Invocation

```
var age = GetAge(dateOfBirth);
//the above calls the function GetAge passing parameter dateOfBirth.
```

Section 3.23: Class Member Operators: Aggregate Object Indexing

```
var letters = "letters".ToCharArray();
char letter = letters[1];
Console.WriteLine("Second Letter is {0}",letter);
//in the above example we take the second character from the array
//by calling letters[1]
//NB: Array Indexing starts at 0; i.e. the first letter would be given by letters[0].
```

Chapter 4: Conditional Statements

Section 4.1: If-Else Statement

Programming in general often requires a `decision` or a `branch` within the code to account for how the code operates under different inputs or conditions. Within the C# programming language (and most programming languages for this matter), the simplest and sometimes the most useful way of creating a branch within your program is through an `If-Else` statement.

Lets assume we have method (a.k.a. a function) which takes an int parameter which will represent a score up to 100, and the method will print out a message saying whether we pass or fail.

```
static void PrintPassOrFail(int score)
{
    if (score >= 50) // If score is greater or equal to 50
    {
        Console.WriteLine("Pass!");
    }
    else // If score is not greater or equal to 50
    {
        Console.WriteLine("Fail!");
    }
}
```

When looking at this method, you may notice this line of code (`score >= 50`) inside the `If` statement. This can be seen as a `boolean` condition, where if the condition is evaluated to equal `true`, then the code that is in between the `if { }` is ran.

For example, if this method was called like this: `PrintPassOrFail(60);`, the output of the method would be a Console Print saying ***Pass!*** since the parameter value of 60 is greater or equal to 50.

However, if the method was called like: `PrintPassOrFail(30);`, the output of the method would print out saying ***Fail!***. This is because the value 30 is not greater or equal to 50, thus the code in between the `else { }` is ran instead of the `If` statement.

In this example, we've said that *score* should go up to 100, which hasn't been accounted for at all. To account for *score* not going past 100 or possibly dropping below 0, see the **If-Else If-Else Statement** example.

Section 4.2: If statement conditions are standard boolean expressions and values

The following statement

```
if (conditionA && conditionB && conditionC) //...
```

is exactly equivalent to

```
bool conditions = conditionA && conditionB && conditionC;
if (conditions) // ...
```

in other words, the conditions inside the "if" statement just form an ordinary Boolean expression.

A common mistake when writing conditional statements is to explicitly compare to `true` and `false`:

```
if (conditionA == true && conditionB == false && conditionC == true) // ...
```

This can be rewritten as

```
if (conditionA && !conditionB && conditionC)
```

Section 4.3: If-Else If-Else Statement

Following on from the **If-Else Statement** example, it is now time to introduce the `Else If` statement. The `Else If` statement follows directly after the `If` statement in the **If-Else If-Else** structure, but intrinsically has has a similar syntax as the `If` statement. It is used to add more branches to the code than what a simple **If-Else** statement can.

In the example from **If-Else Statement**, the example specified that the score goes up to 100; however there were never any checks against this. To fix this, lets modify the method from **If-Else Statement** to look like this:

```
static void PrintPassOrFail(int score)
{
    if (score > 100) // If score is greater than 100
    {
        Console.WriteLine("Error: score is greater than 100!");
    }
    else if (score < 0) // Else If score is less than 0
    {
        Console.WriteLine("Error: score is less than 0!");
    }
    else if (score >= 50) // Else if score is greater or equal to 50
    {
        Console.WriteLine("Pass!");
    }
    else // If none above, then score must be between 0 and 49
    {
        Console.WriteLine("Fail!");
    }
}
```

All these statements will run in order from the top all the way to the bottom until a condition has been met. In this new update of the method, we've added two new branches to now accommodate for the score going *out of bounds*.

For example, if we now called the method in our code as `PrintPassOFail(110);`, the output would be a Console Print saying ***Error: score is greater than 100!***; and if we called the method in our code like `PrintPassOrFail(-20);`, the output would say ***Error: score is less than 0!***.

Chapter 5: Equality Operator

Section 5.1: Equality kinds in c# and equality operator

In C#, there are two different kinds of equality: reference equality and value equality. Value equality is the commonly understood meaning of equality: it means that two objects contain the same values. For example, two integers with the value of 2 have value equality. Reference equality means that there are not two objects to compare. Instead, there are two object references, both of which refer to the same object.

```
object a = new object();
object b = a;
System.Object.ReferenceEquals(a, b);  //returns true
```

For predefined value types, the equality operator (==) returns true if the values of its operands are equal, false otherwise. For reference types other than string, == returns true if its two operands refer to the same object. For the string type, == compares the values of the strings.

```
// Numeric equality: True
Console.WriteLine((2 + 2) == 4);

// Reference equality: different objects,
// same boxed value: False.
object s = 1;
object t = 1;
Console.WriteLine(s == t);

// Define some strings:
string a = "hello";
string b = String.Copy(a);
string c = "hello";

// Compare string values of a constant and an instance: True
Console.WriteLine(a == b);

// Compare string references;
// a is a constant but b is an instance: False.
Console.WriteLine((object)a == (object)b);

// Compare string references, both constants
// have the same value, so string interning
// points to same reference: True.
Console.WriteLine((object)a == (object)c);
```

Chapter 6: Equals and GetHashCode

Section 6.1: Writing a good GetHashCode override

`GetHashCode` has major performance effects on Dictionary<> and HashTable.

Good `GetHashCode` Methods

- should have an even distribution
  - every integer should have a roughly equal chance of returning for a random instance
  - if your method returns the same integer (e.g. the constant '999') for each instance, you'll have bad performance
- should be quick
  - These are NOT cryptographic hashes, where slowness is a feature
  - the slower your hash function, the slower your dictionary
- must return the same HashCode on two instances that `Equals` evaluates to true
  - if they do not (e.g. because `GetHashCode` returns a random number), items may not be found in a `List`, `Dictionary`, or similar.

A good method to implement `GetHashCode` is to use one prime number as a starting value, and add the hashcodes of the fields of the type multiplied by other prime numbers to that:

```
public override int GetHashCode()
{
    unchecked // Overflow is fine, just wrap
    {
        int hash = 3049; // Start value (prime number).

        // Suitable nullity checks etc, of course :)
        hash = hash * 5039 + field1.GetHashCode();
        hash = hash * 883 + field2.GetHashCode();
        hash = hash * 9719 + field3.GetHashCode();
        return hash;
    }
}
```

Only the fields which are used in the `Equals`-method should be used for the hash function.

If you have a need to treat the same type in different ways for Dictionary/HashTables, you can use IEqualityComparer.

Section 6.2: Default Equals behavior

`Equals` is declared in the `Object` class itself.

```
public virtual bool Equals(Object obj);
```

By default, `Equals` has the following behavior:

- If the instance is a reference type, then `Equals` will return true only if the references are the same.
- If the instance is a value type, then `Equals` will return true only if the type and value are the same.
- `string` is a special case. It behaves like a value type.

```
namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
        {
            //areFooClassEqual: False
            Foo fooClass1 = new Foo("42");
            Foo fooClass2 = new Foo("42");
            bool areFooClassEqual = fooClass1.Equals(fooClass2);
            Console.WriteLine("fooClass1 and fooClass2 are equal: {0}", areFooClassEqual);
            //False

            //areFooIntEqual: True
            int fooInt1 = 42;
            int fooInt2 = 42;
            bool areFooIntEqual = fooInt1.Equals(fooInt2);
            Console.WriteLine("fooInt1 and fooInt2 are equal: {0}", areFooIntEqual);

            //areFooStringEqual: True
            string fooString1 = "42";
            string fooString2 = "42";
            bool areFooStringEqual = fooString1.Equals(fooString2);
            Console.WriteLine("fooString1 and fooString2 are equal: {0}", areFooStringEqual);
        }
    }

    public class Foo
    {
        public string Bar { get; }

        public Foo(string bar)
        {
            Bar = bar;
        }
    }
}
```

Section 6.3: Override Equals and GetHashCode on custom types

For a class `Person` like:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); //false because it's reference Equals
```

But defining `Equals` and `GetHashCode` as follows:

```
public class Person
{
    public string Name { get; set; }
```

```
    public int Age { get; set; }
    public string Clothes { get; set; }

    public override bool Equals(object obj)
    {
        var person = obj as Person;
        if(person == null) return false;
        return Name == person.Name && Age == person.Age; //the clothes are not important when
comparing two persons
    }

    public override int GetHashCode()
    {
        return Name.GetHashCode()*Age;
    }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); // result is true
```

Also using LINQ to make different queries on persons will check both `Equals` and `GetHashCode`:

```
var persons = new List<Person>
{
     new Person{ Name = "Jon", Age = 20, Clothes = "some clothes"},
     new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes"},
     new Person{ Name = "Jon", Age = 20, Clothes = ""}
};

var distinctPersons = persons.Distinct().ToList();//distinctPersons has Count = 2
```

Section 6.4: Equals and GetHashCode in IEqualityComparator

For given type `Person`:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

List<Person> persons = new List<Person>
{
    new Person{ Name = "Jon", Age = 20, Clothes = "some clothes"},
    new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes"},
    new Person{ Name = "Jon", Age = 20, Clothes = ""}
};

var distinctPersons = persons.Distinct().ToList();// distinctPersons has Count = 3
```

But defining `Equals` and `GetHashCode` into an `IEqualityComparator` :

```
public class PersonComparator : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name && x.Age == y.Age; //the clothes are not important when comparing
```

```
two persons;
    }

    public int GetHashCode(Person obj) { return obj.Name.GetHashCode() * obj.Age; }
}

var distinctPersons = persons.Distinct(new PersonComparator()).ToList();// distinctPersons has
Count = 2
```

Note that for this query, two objects have been considered equal if both the `Equals` returned true and the `GetHashCode` have returned the same hash code for the two persons.

Chapter 7: Null-Coalescing Operator

Parameter	Details
`possibleNullObject`	The value to test for null value. If non null, this value is returned. Must be a nullable type.
`defaultValue`	The value returned if `possibleNullObject` is null. Must be the same type as `possibleNullObject`.

Section 7.1: Basic usage

Using the `null-coalescing operator (??)` allows you to specify a default value for a nullable type if the left-hand operand is `null`.

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString ?? "not provided"));
```

Live Demo on .NET Fiddle

This is logically equivalent to:

```
string testString = null;
if (testString == null)
{
    Console.WriteLine("The specified string is - not provided");
}
else
{
    Console.WriteLine("The specified string is - " + testString);
}
```

or using the ternary operator (?:) operator:

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString == null ? "not provided" :
testString));
```

Section 7.2: Null fall-through and chaining

The left-hand operand must be nullable, while the right-hand operand may or may not be. The result will be typed accordingly.

Non-nullable

```
int? a = null;
int b = 3;
var output = a ?? b;
var type = output.GetType();

Console.WriteLine($"Output Type :{type}");
Console.WriteLine($"Output value :{output}");
```

Output:

> Type :System.Int32

> value :3

View Demo

Nullable

```
int? a = null;
int? b = null;
var output = a ?? b;
```

`output` will be of type `int?` and equal to `b`, or `null`.

Multiple Coalescing

Coalescing can also be done in chains:

```
int? a = null;
int? b = null;
int c = 3;
var output = a ?? b ?? c;

var type = output.GetType();
Console.WriteLine($"Type :{type}");
Console.WriteLine($"value :{output}");
```

Output:

> Type :System.Int32
> value :3

View Demo

Null Conditional Chaining

The null coalescing operator can be used in tandem with the null propagation operator to provide safer access to properties of objects.

```
object o = null;
var output = o?.ToString() ?? "Default Value";
```

Output:

> Type :System.String
> value :Default Value

View Demo

Section 7.3: Null coalescing operator with method calls

The null coalescing operator makes it easy to ensure that a method that may return `null` will fall back to a default value.

Without the null coalescing operator:

```
string name = GetName();

if (name == null)
    name = "Unknown!";
```

With the null coalescing operator:

```
string name = GetName() ?? "Unknown!";
```

Section 7.4: Use existing or create new

A common usage scenario that this feature really helps with is when you are looking for an object in a collection and need to create a new one if it does not already exist.

```
IEnumerable<MyClass> myList = GetMyList();
var item = myList.SingleOrDefault(x => x.Id == 2) ?? new MyClass { Id = 2 };
```

Section 7.5: Lazy properties initialization with null coalescing operator

```
private List<FooBar> _fooBars;

public List<FooBar> FooBars
{
    get { return _fooBars ?? (_fooBars = new List<FooBar>()); }
}
```

The first time the property `.FooBars` is accessed the `_fooBars` variable will evaluate as `null`, thus falling through to the assignment statement assigns and evaluates to the resulting value.

Thread safety

This is **not thread-safe** way of implementing lazy properties. For thread-safe laziness, use the `Lazy<T>` class built into the .NET Framework.

C# 6 Syntactic Sugar using expression bodies

Note that since C# 6, this syntax can be simplified using expression body for the property:

```
private List<FooBar> _fooBars;

public List<FooBar> FooBars => _fooBars ?? ( _fooBars = new List<FooBar>() );
```

Subsequent accesses to the property will yield the value stored in the `_fooBars` variable.

Example in the MVVM pattern

This is often used when implementing commands in the MVVM pattern. Instead of initializing the commands eagerly with the construction of a viewmodel, commands are lazily initialized using this pattern as follows:

```
private ICommand _actionCommand = null;
public ICommand ActionCommand =>
   _actionCommand ?? ( _actionCommand = new DelegateCommand( DoAction ) );
```

Chapter 8: Null-conditional Operators

Section 8.1: Null-Conditional Operator

The `?.` operator is syntactic sugar to avoid verbose null checks. It's also known as the Safe navigation operator.

Class used in the following example:

```
public class Person
{
    public int Age { get; set; }
    public string Name { get; set; }
    public Person Spouse { get; set; }
}
```

If an object is potentially null (such as a function that returns a reference type) the object must first be checked for null to prevent a possible `NullReferenceException`. Without the null-conditional operator, this would look like:

```
Person person = GetPerson();

int? age = null;
if (person != null)
    age = person.Age;
```

The same example using the null-conditional operator:

```
Person person = GetPerson();

var age = person?.Age;    // 'age' will be of type 'int?', even if 'person' is not null
```

Chaining the Operator

The null-conditional operator can be combined on the members and sub-members of an object.

```
// Will be null if either `person` or `person.Spouse` are null
int? spouseAge = person?.Spouse?.Age;
```

Combining with the Null-Coalescing Operator

The null-conditional operator can be combined with the null-coalescing operator to provide a default value:

```
// spouseDisplayName will be "N/A" if person, Spouse, or Name is null
var spouseDisplayName = person?.Spouse?.Name ?? "N/A";
```

Section 8.2: The Null-Conditional Index

Similarly to the `?.` operator, the null-conditional index operator checks for null values when indexing into a collection that may be null.

```
string item = collection?[index];
```

is syntactic sugar for

```
string item = null;
```

```
if(collection != null)
{
    item = collection[index];
}
```

Section 8.3: Avoiding NullReferenceExceptions

```
var person = new Person
{
    Address = null;
};

var city = person.Address.City; //throws a NullReferenceException
var nullableCity = person.Address?.City; //returns the value of null
```

This effect can be chained together:

```
var person = new Person
{
    Address = new Address
    {
        State = new State
        {
            Country = null
        }
    }
};

// this will always return a value of at least "null" to be stored instead
// of throwing a NullReferenceException
var countryName = person?.Address?.State?.Country?.Name;
```

Section 8.4: Null-conditional Operator can be used with Extension Method

Extension Method can work on null references, but you can use `?.` to null-check anyway.

```
public class Person
{
    public string Name {get; set;}
}

public static class PersonExtensions
{
    public static int GetNameLength(this Person person)
    {
        return person == null ? -1 : person.Name.Length;
    }
}
```

Normally, the method will be triggered for `null` references, and return -1:

```
Person person = null;
int nameLength = person.GetNameLength(); // returns -1
```

Using `?.` the method will not be triggered for `null` references, and the type is `int?`:

```
Person person = null;
int? nameLength = person?.GetNameLength(); // nameLength is null.
```

This behavior is actually expected from the way in which the `?.` operator works: it will avoid making instance method calls for null instances, in order to avoid `NullReferenceExceptions`. However, the same logic applies to the extension method, despite the difference on how the method is declared.

For more information on why the extension method is called in the first example, please see the Extension methods - null checking documentation.

Chapter 9: nameof Operator

The `nameof` operator allows you to get the name of a **variable**, **type** or **member** in string form without hard-coding it as a literal.

The operation is evaluated at compile-time, which means that you can rename a referenced identifier, using an IDE's rename feature, and the name string will update with it.

Section 9.1: Basic usage: Printing a variable name

The `nameof` operator allows you to get the name of a variable, type or member in string form without hard-coding it as a literal. The operation is evaluated at compile-time, which means that you can rename, using an IDE's rename feature, a referenced identifier and the name string will update with it.

```
var myString = "String Contents";
Console.WriteLine(nameof(myString));
```

Would output

> myString

because the name of the variable is "myString". Refactoring the variable name would change the string.

If called on a reference type, the `nameof` operator returns the name of the current reference, *not* the name or type name of the underlying object. For example:

```
string greeting = "Hello!";
Object mailMessageBody = greeting;

Console.WriteLine(nameof(greeting)); // Returns "greeting"
Console.WriteLine(nameof(mailMessageBody)); // Returns "mailMessageBody", NOT "greeting"!
```

Section 9.2: Raising PropertyChanged event

Snippet

```
public class Person : INotifyPropertyChanged
{
    private string _address;

    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    public string Address
    {
        get { return _address; }
        set
        {
            if (_address == value)
            {
```

```
                return;
            }

            _address = value;
            OnPropertyChanged(nameof(Address));
        }
    }
}

...

var person = new Person();
person.PropertyChanged += (s,e) => Console.WriteLine(e.PropertyName);

person.Address = "123 Fake Street";
```

Console Output

> Address

Section 9.3: Argument Checking and Guard Clauses

Prefer

```
public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException(nameof(orderLine));
        ...
    }
}
```

Over

```
public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException("orderLine");
        ...
    }
}
```

Using the `nameof` feature makes it easier to refactor method parameters.

Section 9.4: Strongly typed MVC action links

Instead of the usual loosely typed:

```
@Html.ActionLink("Log in", "UserController", "LogIn")
```

You can now make action links strongly typed:

```
@Html.ActionLink("Log in", @typeof(UserController), @nameof(UserController.LogIn))
```

Now if you want to refactor your code and rename the `UserController.LogIn` method to `UserController.SignIn`, you don't need to worry about searching for all string occurrences. The compiler will do the job.

Section 9.5: Handling PropertyChanged events

Snippet

```
public class BugReport : INotifyPropertyChanged
{
    public string Title { ... }
    public BugStatus Status { ... }
}

...

private void BugReport_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    var bugReport = (BugReport)sender;

    switch (e.PropertyName)
    {
        case nameof(bugReport.Title):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Title);
            break;

        case nameof(bugReport.Status):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Status);
            break;
    }
}

...

var report = new BugReport();
report.PropertyChanged += BugReport_PropertyChanged;

report.Title = "Everything is on fire and broken";
report.Status = BugStatus.ShowStopper;
```

Console Output

> Title changed to Everything is on fire and broken
>
> Status changed to ShowStopper

Section 9.6: Applied to a generic type parameter

Snippet

```
public class SomeClass<TItem>
{
    public void PrintTypeName()
    {
        Console.WriteLine(nameof(TItem));
    }
}
```

```
...

var myClass = new SomeClass<int>();
myClass.PrintTypeName();

Console.WriteLine(nameof(SomeClass<int>));
```

Console Output

> TItem
>
> SomeClass

Section 9.7: Printing a parameter name

Snippet

```
public void DoSomething(int paramValue)
{
    Console.WriteLine(nameof(paramValue));
}

...

int myValue = 10;
DoSomething(myValue);
```

Console Output

> paramValue

Section 9.8: Applied to qualified identifiers

Snippet

```
Console.WriteLine(nameof(CompanyNamespace.MyNamespace));
Console.WriteLine(nameof(MyClass));
Console.WriteLine(nameof(MyClass.MyNestedClass));
Console.WriteLine(nameof(MyNamespace.MyClass.MyNestedClass.MyStaticProperty));
```

Console Output

> MyNamespace
>
> MyClass
>
> MyNestedClass
>
> MyStaticProperty

Chapter 10: Verbatim Strings

Section 10.1: Interpolated Verbatim Strings

Verbatim strings can be combined with the new String interpolation features found in C#6.

```
Console.WriteLine($@"Testing \n 1 2 {5 - 2}
New line");
```

Output:

> Testing \n 1 2 3
> New line

Live Demo on .NET Fiddle

As expected from a verbatim string, the backslashes are ignored as escape characters. And as expected from an interpolated string, any expression inside curly braces is evaluated before being inserted into the string at that position.

Section 10.2: Escaping Double Quotes

Double Quotes inside verbatim strings can be escaped by using 2 sequential double quotes `""` to represent one double quote `"` in the resulting string.

```
var str = @""I don't think so,"" he said.";
Console.WriteLine(str);
```

Output:

> "I don't think so," he said.

Live Demo on .NET Fiddle

Section 10.3: Verbatim strings instruct the compiler to not use character escapes

In a normal string, the backslash character is the escape character, which instructs the compiler to look at the next character(s) to determine the actual character in the string. (Full list of character escapes)

In verbatim strings, there are no character escapes (except for `""` which is turned into a `"`). To use a verbatim string, just prepend a `@` before the starting quotes.

This verbatim string

```
var filename = @"c:\temp\newfile.txt"
```

Output:

> c:\temp\newfile.txt

As opposed to using an ordinary (non-verbatim) string:

```
var filename = "c:\temp\newfile.txt"
```

that will output:

```
c:    emp
ewfile.txt
```

using character escaping. (The `\t` is replaced with a tab character and the `\n` is replace with a newline.)

Live Demo on .NET Fiddle

Section 10.4: Multiline Strings

```
var multiLine = @"This is a

multiline paragraph";
```

Output:

> This is a
>
> multiline paragraph

Live Demo on .NET Fiddle

Multi-line strings that contain double-quotes can also be escaped just as they were on a single line, because they are verbatim strings.

```
var multilineWithDoubleQuotes = @"I went to a city named

                        ""San Diego""

                    during summer vacation.";
```

Live Demo on .NET Fiddle

It should be noted that the spaces/tabulations at the start of lines 2 and 3 here are actually present in the value of the variable; check this question for possible solutions.

Chapter 11: Common String Operations

Section 11.1: Formatting a string

Use the `String.Format()` method to replace one or more items in the string with the string representation of a specified object:

```
String.Format("Hello {0} Foo {1}", "World", "Bar") //Hello World Foo Bar
```

Section 11.2: Correctly reversing a string

Most times when people have to reverse a string, they do it more or less like this:

```
char[] a = s.ToCharArray();
System.Array.Reverse(a);
string r = new string(a);
```

However, what these people don't realize is that this is actually wrong.
And I don't mean because of the missing NULL check.

It is actually wrong because a Glyph/GraphemeCluster can consist out of several codepoints (aka. characters).

To see why this is so, we first have to be aware of the fact what the term "character" actually means.

Reference:

> Character is an overloaded term than can mean many things.
>
> A code point is the atomic unit of information. Text is a sequence of code points. Each code point is a number which is given meaning by the Unicode standard.
>
> A grapheme is a sequence of one or more code points that are displayed as a single, graphical unit that a reader recognizes as a single element of the writing system. For example, both a and ä are graphemes, but they may consist of multiple code points (e.g. ä may be two code points, one for the base character a followed by one for the diaresis; but there's also an alternative, legacy, single code point representing this grapheme). Some code points are never part of any grapheme (e.g. the zero-width non-joiner, or directional overrides).
>
> A glyph is an image, usually stored in a font (which is a collection of glyphs), used to represent graphemes or parts thereof. Fonts may compose multiple glyphs into a single representation, for example, if the above ä is a single code point, a font may chose to render that as two separate, spatially overlaid glyphs. For OTF, the font's GSUB and GPOS tables contain substitution and positioning information to make this work. A font may contain multiple alternative glyphs for the same grapheme, too.

So in C#, a character is actually a CodePoint.

Which means, if you just reverse a valid string like `Les Misérables`, which can look like this

```
string s = "Les Mise\u0301rables";
```

as a sequence of characters, you will get:

> selbaŕesiM seL

As you can see, the accent is on the R character, instead of the e character.
Although string.reverse.reverse will yield the original string if you both times reverse the char array, this kind of reversal is definitely NOT the reverse of the original string.

You'll need to reverse each GraphemeCluster only.
So, if done correctly, you reverse a string like this:

```
    private static System.Collections.Generic.List<string> GraphemeClusters(string s)
    {
        System.Collections.Generic.List<string> ls = new System.Collections.Generic.List<string>();

        System.Globalization.TextElementEnumerator enumerator =
System.Globalization.StringInfo.GetTextElementEnumerator(s);
        while (enumerator.MoveNext())
        {
            ls.Add((string)enumerator.Current);
        }

        return ls;
    }


    // this
    private static string ReverseGraphemeClusters(string s)
    {
        if(string.IsNullOrEmpty(s) || s.Length == 1)
             return s;

        System.Collections.Generic.List<string> ls = GraphemeClusters(s);
        ls.Reverse();

        return string.Join("", ls.ToArray());
    }

    public static void TestMe()
    {
        string s = "Les Mise\u0301rables";
        // s = "noël";
        string r = ReverseGraphemeClusters(s);

        // This would be wrong:
        // char[] a = s.ToCharArray();
        // System.Array.Reverse(a);
        // string r = new string(a);

        System.Console.WriteLine(r);
    }
```

And - oh joy - you'll realize if you do it correctly like this, it will also work for Asian/South-Asian/East-Asian languages (and French/Swedish/Norwegian, etc.)...

Section 11.3: Padding a string to a fixed length

```
string s = "Foo";
string paddedLeft = s.PadLeft(5);        // paddedLeft = "  Foo" (pads with spaces by default)
string paddedRight = s.PadRight(6, '+'); // paddedRight = "Foo+++"
```

```
string noPadded = s.PadLeft(2);          // noPadded = "Foo" (original string is never shortened)
```

Section 11.4: Getting x characters from the right side of a string

Visual Basic has Left, Right, and Mid functions that returns characters from the Left, Right, and Middle of a string. These methods does not exist in C#, but can be implemented with `Substring()`. They can be implemented as an extension methods like the following:

```
public static class StringExtensions
{
    /// <summary>
    /// VB Left function
    /// </summary>
    /// <param name="stringparam"></param>
    /// <param name="numchars"></param>
    /// <returns>Left-most numchars characters</returns>
    public static string Left( this string stringparam, int numchars )
    {
        // Handle possible Null or numeric stringparam being passed
        stringparam += string.Empty;

        // Handle possible negative numchars being passed
        numchars = Math.Abs( numchars );

        // Validate numchars parameter
        if (numchars > stringparam.Length)
            numchars = stringparam.Length;

        return stringparam.Substring( 0, numchars );
    }

    /// <summary>
    /// VB Right function
    /// </summary>
    /// <param name="stringparam"></param>
    /// <param name="numchars"></param>
    /// <returns>Right-most numchars characters</returns>
    public static string Right( this string stringparam, int numchars )
    {
        // Handle possible Null or numeric stringparam being passed
        stringparam += string.Empty;

        // Handle possible negative numchars being passed
        numchars = Math.Abs( numchars );

        // Validate numchars parameter
        if (numchars > stringparam.Length)
            numchars = stringparam.Length;

        return stringparam.Substring( stringparam.Length - numchars );
    }

    /// <summary>
    /// VB Mid function - to end of string
    /// </summary>
    /// <param name="stringparam"></param>
    /// <param name="startIndex">VB-Style startindex, 1st char startindex = 1</param>
    /// <returns>Balance of string beginning at startindex character</returns>
    public static string Mid( this string stringparam, int startindex )
```

```
    {
        // Handle possible Null or numeric stringparam being passed
        stringparam += string.Empty;

        // Handle possible negative startindex being passed
        startindex = Math.Abs( startindex );

        // Validate numchars parameter
        if (startindex > stringparam.Length)
            startindex = stringparam.Length;

        // C# strings are zero-based, convert passed startindex
        return stringparam.Substring( startindex - 1 );
    }

    /// <summary>
    /// VB Mid function - for number of characters
    /// </summary>
    /// <param name="stringparam"></param>
    /// <param name="startIndex">VB-Style startindex, 1st char startindex = 1</param>
    /// <param name="numchars">number of characters to return</param>
    /// <returns>Balance of string beginning at startindex character</returns>
    public static string Mid( this string stringparam, int startindex, int numchars)
    {
        // Handle possible Null or numeric stringparam being passed
        stringparam += string.Empty;

        // Handle possible negative startindex being passed
        startindex = Math.Abs( startindex );

        // Handle possible negative numchars being passed
        numchars = Math.Abs( numchars );

        // Validate numchars parameter
        if (startindex > stringparam.Length)
            startindex = stringparam.Length;

        // C# strings are zero-based, convert passed startindex
        return stringparam.Substring( startindex - 1, numchars );

    }
}
```

This extension method can be used as follows:

```
string myLongString = "Hello World!";
string myShortString = myLongString.Right(6);  // "World!"
string myLeftString = myLongString.Left(5);    // "Hello"
string myMidString1 = myLongString.Left(4);    // "lo World"
string myMidString2 = myLongString.Left(2,3);    // "ell"
```

Section 11.5: Checking for empty String using String.IsNullOrEmpty() and String.IsNullOrWhiteSpace()

```
string nullString = null;
string emptyString = "";
string whitespaceString = "    ";
string tabString = "\t";
string newlineString = "\n";
string nonEmptyString = "abc123";
```

```
bool result;

result = String.IsNullOrEmpty(nullString);            // true
result = String.IsNullOrEmpty(emptyString);           // true
result = String.IsNullOrEmpty(whitespaceString);      // false
result = String.IsNullOrEmpty(tabString);             // false
result = String.IsNullOrEmpty(newlineString);         // false
result = String.IsNullOrEmpty(nonEmptyString);        // false

result = String.IsNullOrWhiteSpace(nullString);       // true
result = String.IsNullOrWhiteSpace(emptyString);      // true
result = String.IsNullOrWhiteSpace(tabString);        // true
result = String.IsNullOrWhiteSpace(newlineString);    // true
result = String.IsNullOrWhiteSpace(whitespaceString); // true
result = String.IsNullOrWhiteSpace(nonEmptyString);   // false
```

Section 11.6: Trimming Unwanted Characters Off the Start and/or End of Strings

`String.Trim()`

```
string x = "   Hello World!    ";
string y = x.Trim(); // "Hello World!"

string q = "{(Hi!*";
string r = q.Trim( '(', '*', '{' ); // "Hi!"
```

`String.TrimStart()` and `String.TrimEnd()`

```
string q = "{(Hi*";
string r = q.TrimStart( '{' ); // "(Hi*"
string s = q.TrimEnd( '*' );   // "{(Hi"
```

Section 11.7: Convert Decimal Number to Binary,Octal and Hexadecimal Format

1. To convert decimal number to binary format use **base 2**

```
Int32 Number = 15;
Console.WriteLine(Convert.ToString(Number, 2));  //OUTPUT : 1111
```

2. To convert decimal number to octal format use **base 8**

```
int Number = 15;
Console.WriteLine(Convert.ToString(Number, 8));  //OUTPUT : 17
```

3. To convert decimal number to hexadecimal format use **base 16**

```
var Number = 15;
Console.WriteLine(Convert.ToString(Number, 16));  //OUTPUT : f
```

Section 11.8: Construct a string from Array

The `String.Join` method will help us to construct a string From array/list of characters or string. This method accepts two parameters. The first one is the delimiter or the separator which will help you to separate each element in the array. And the second parameter is the Array itself.

String from `char array`:

```
string delimiter=",";
char[] charArray = new[] { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charArray);
```

Output : `a,b,c` if we change the `delimiter` as `""` then the output will become `abc`.

String from `List of char`:

```
string delimiter = "|";
List<char> charList = new List<char>() { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charList);
```

Output : `a|b|c`

String from `List of Strings`:

```
string delimiter = " ";
List<string> stringList = new List<string>() { "Ram", "is", "a","boy" };
string inputString = String.Join(delimiter, stringList);
```

Output : `Ram is a boy`

String from `array of strings`:

```
string delimiter = "_";
string[] stringArray = new [] { "Ram", "is", "a","boy" };
string inputString = String.Join(delimiter, stringArray);
```

Output : `Ram_is_a_boy`

Section 11.9: Formatting using ToString

Usually we are using `String.Format` method for formatting purpose, the `.ToString` is usually used for converting other types to string. We can specify the format along with the ToString method while conversion is taking place, So we can avoid an additional Formatting. Let Me Explain how it works with different types;

Integer to formatted string:

```
int intValue = 10;
string zeroPaddedInteger = intValue.ToString("000"); // Output will be "010"
string customFormat = intValue.ToString("Input value is 0"); // output will be   "Input value is
10"
```

double to formatted string:

```
double doubleValue = 10.456;
string roundedDouble = doubleValue.ToString("0.00"); // output 10.46
string integerPart = doubleValue.ToString("00");    // output 10
string customFormat = doubleValue.ToString("Input value is 0.0");  // Input value is 10.5
```

Formatting DateTime using ToString

```
DateTime currentDate = DateTime.Now; //  {7/21/2016 7:23:15 PM}
string dateTimeString = currentDate.ToString("dd-MM-yyyy HH:mm:ss"); // "21-07-2016 19:23:15"
```

```
string dateOnlyString = currentDate.ToString("dd-MM-yyyy"); // "21-07-2016"
string dateWithMonthInWords = currentDate.ToString("dd-MMMM-yyyy HH:mm:ss"); // "21-July-2016
19:23:15"
```

Section 11.10: Splitting a String by another string

```
string str = "this--is--a--complete--sentence";
string[] tokens = str.Split(new[] { "--" }, StringSplitOptions.None);
```

Result:

> ["this", "is", "a", "complete", "sentence"]

Section 11.11: Splitting a String by specific character

```
string helloWorld = "hello world, how is it going?";
string[] parts1 = helloWorld.Split(',');

//parts1: ["hello world", " how is it going?"]

string[] parts2 = helloWorld.Split(' ');

//parts2: ["hello", "world,", "how", "is", "it", "going?"]
```

Section 11.12: Getting Substrings of a given string

```
string helloWorld = "Hello World!";
string world = helloWorld.Substring(6); //world = "World!"
string hello = helloWorld.Substring(0,5); // hello = "Hello"
```

`Substring` returns the string up from a given index, or between two indexes (both inclusive).

Section 11.13: Determine whether a string begins with a given sequence

```
string HelloWorld = "Hello World";
HelloWorld.StartsWith("Hello"); // true
HelloWorld.StartsWith("Foo"); // false
```

Finding a string within a string

Using the `System.String.Contains` you can find out if a particular string exists within a string. The method returns a boolean, true if the string exists else false.

```
string s = "Hello World";
bool stringExists = s.Contains("ello");  //stringExists =true as the string contains the substring
```

Section 11.14: Getting a char at specific index and enumerating the string

You can use the `Substring` method to get any number of characters from a string at any given location. However, if you only want a single character, you can use the string indexer to get a single character at any given index like you

do with an array:

```
string s = "hello";
char c = s[1]; //Returns 'e'
```

Notice that the return type is `char`, unlike the `Substring` method which returns a `string` type.

You can also use the indexer to iterate through the characters of the string:

```
string s = "hello";
foreach (char c in s)
    Console.WriteLine(c);
/********* This will print each character on a new line:
h
e
l
l
o
**********/
```

Section 11.15: Joining an array of strings into a new one

```
var parts = new[] { "Foo", "Bar", "Fizz", "Buzz"};
var joined = string.Join(", ", parts);

//joined = "Foo, Bar, Fizz, Buzz"
```

Section 11.16: Replacing a string within a string

Using the `System.String.Replace` method, you can replace part of a string with another string.

```
string s = "Hello World";
 s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

All the occurrences of the search string are replaced.

This method can also be used to remove part of a string, using the `String.Empty` field:

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

Section 11.17: Changing the case of characters within a String

The `System.String` class supports a number of methods to convert between uppercase and lowercase characters in a string.

- `System.String.ToLowerInvariant` is used to return a String object converted to lowercase.
- `System.String.ToUpperInvariant` is used to return a String object converted to uppercase.

Note: The reason to use the *invariant* versions of these methods is to prevent producing unexpected culture-specific letters. This is explained here in detail.

Example:

```
string s = "My String";
```

```
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

Note that you *can* choose to specify a specific **Culture** when converting to lowercase and uppercase by using the String.ToLower(CultureInfo) and String.ToUpper(CultureInfo) methods accordingly.

Section 11.18: Concatenate an array of strings into a single string

The `System.String.Join` method allows to concatenate all elements in a string array, using a specified separator between each element:

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(",", words); // singleString = "One,Two,Three,Four"
```

Section 11.19: String Concatenation

String Concatenation can be done by using the `System.String.Concat` method, or (much easier) using the `+` operator:

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

In C# 6 this can be done as follows:

```
string concat = $"{first},{second}";
```

Chapter 12: String.Format

Parameter	Details
format	A composite format string, which defines the way *args* should be combined into a string.
args	A sequence of objects to be combined into a string. Since this uses a `params` argument, you can either use a comma-separated list of arguments or an actual object array.
provider	A collection of ways of formatting objects to strings. Typical values include CultureInfo.InvariantCulture and CultureInfo.CurrentCulture.

The `Format` methods are a set of overloads in the `System.String` class used to create strings that combine objects into specific string representations. This information can be applied to `String.Format`, various `WriteLine` methods as well as other methods in the .NET framework.

Section 12.1: Since C# 6.0

`Version ≥ 6.0`

Since C# 6.0 it is possible to use string interpolation in place of `String.Format`.

```
string name = "John";
string lastname = "Doe";
Console.WriteLine($"Hello {name} {lastname}!");
```

> Hello John Doe!

More examples for this under the topic C# 6.0 features: String interpolation.

Section 12.2: Places where String.Format is 'embedded' in the framework

There are several places where you can use `String.Format` *indirectly*: The secret is to look for the overload with the signature `string format, params object[] args`, e.g.:

```
Console.WriteLine(String.Format("{0} - {1}", name, value));
```

Can be replaced with shorter version:

```
Console.WriteLine("{0} - {1}", name, value);
```

There are other methods which also use `String.Format`e.g.:

```
Debug.WriteLine(); // and Print()
StringBuilder.AppendFormat();
```

Section 12.3: Create a custom format provider

```
public class CustomFormat : IFormatProvider, ICustomFormatter
{
    public string Format(string format, object arg, IFormatProvider formatProvider)
    {
        if (!this.Equals(formatProvider))
```

```
        {
            return null;
        }

        if (format == "Reverse")
        {
            return String.Join("", arg.ToString().Reverse());
        }

        return arg.ToString();
    }

    public object GetFormat(Type formatType)
    {
        return formatType==typeof(ICustomFormatter) ? this:null;
    }
}
```

Usage:

```
String.Format(new CustomFormat(), "-> {0:Reverse} <-", "Hello World");
```

Output:

```
-> dlroW olleH <-
```

Section 12.4: Date Formatting

```
DateTime date = new DateTime(2016, 07, 06, 18, 30, 14);
// Format: year, month, day hours, minutes, seconds

Console.Write(String.Format("{0:dd}",date));

//Format by Culture info
String.Format(new System.Globalization.CultureInfo("mn-MN"),"{0:dddd}",date);
```

Version ≥ 6.0

```
Console.Write($"{date:ddd}");
```

output :

```
06
Лхагва
06
```

Specifier	Meaning	Sample	Result
d	Date	{0:d}	7/6/2016
dd	Day, zero-padded	{0:dd}	06
ddd	Short day name	{0:ddd}	Wed
dddd	Full day name	{0:dddd}	Wednesday
D	Long date	{0:D}	Wednesday, July 6, 2016
f	Full date and time, short	{0:f}	Wednesday, July 6, 2016 6:30 PM
ff	Second fractions, 2 digits	{0:ff}	20
fff	Second fractions, 3 digits	{0:fff}	201
ffff	Second fractions, 4 digits	{0:ffff}	2016
F	Full date and time, long	{0:F}	Wednesday, July 6, 2016 6:30:14 PM

g	Default date and time	{0:g}	7/6/2016 6:30 PM
gg	Era	{0:gg}	A.D
hh	Hour (2 digits, 12H)	{0:hh}	06
HH	Hour (2 digits, 24H)	{0:HH}	18
M	Month and day	{0:M}	July 6
mm	Minutes, zero-padded	{0:mm}	30
MM	Month, zero-padded	{0:MM}	07
MMM	3-letter month name	{0:MMM}	Jul
MMMM	Full month name	{0:MMMM}	July
ss	Seconds	{0:ss}	14
r	RFC1123 date	{0:r}	Wed, 06 Jul 2016 18:30:14 GMT
s	Sortable date string	{0:s}	2016-07-06T18:30:14
t	Short time	{0:t}	6:30 PM
T	Long time	{0:T}	6:30:14 PM
tt	AM/PM	{0:tt}	PM
u	Universal sortable local time	{0:u}	2016-07-06 18:30:14Z
U	Universal GMT	{0:U}	Wednesday, July 6, 2016 9:30:14 AM
Y	Month and year	{0:Y}	July 2016
yy	2 digit year	{0:yy}	16
yyyy	4 digit year	{0:yyyy}	2016
zz	2 digit timezone offset	{0:zz}	+09
zzz	full time zone offset	{0:zzz}	+09:00

Section 12.5: Currency Formatting

The "c" (or currency) format specifier converts a number to a string that represents a currency amount.

```
string.Format("{0:c}", 112.236677) // $112.23 - defaults to system
```

Precision

Default is 2. Use c1, c2, c3 and so on to control precision.

```
string.Format("{0:C1}", 112.236677) //$112.2
string.Format("{0:C3}", 112.236677) //$112.237
string.Format("{0:C4}", 112.236677) //$112.2367
string.Format("{0:C9}", 112.236677) //$112.236677000
```

Currency Symbol

1. Pass `CultureInfo` instance to use custom culture symbol.

```
string.Format(new CultureInfo("en-US"), "{0:c}", 112.236677); //$112.24
string.Format(new CultureInfo("de-DE"), "{0:c}", 112.236677); //112,24 €
string.Format(new CultureInfo("hi-IN"), "{0:c}", 112.236677); //□ 112.24
```

2. Use any string as currency symbol. Use `NumberFormatInfo` as to customize currency symbol.

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
nfi = (NumberFormatInfo) nfi.Clone();
nfi.CurrencySymbol = "?";
```

```
string.Format(nfi, "{0:C}", 112.236677); //?112.24
nfi.CurrencySymbol = "?%^&";
string.Format(nfi, "{0:C}", 112.236677); //?%^&112.24
```

Position of Currency Symbol

Use CurrencyPositivePattern for positive values and CurrencyNegativePattern for negative values.

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
nfi.CurrencyPositivePattern = 0;
string.Format(nfi, "{0:C}", 112.236677); //$112.24 - default
nfi.CurrencyPositivePattern = 1;
string.Format(nfi, "{0:C}", 112.236677); //112.24$
nfi.CurrencyPositivePattern = 2;
string.Format(nfi, "{0:C}", 112.236677); //$ 112.24
nfi.CurrencyPositivePattern = 3;
string.Format(nfi, "{0:C}", 112.236677); //112.24 $
```

Negative pattern usage is the same as positive pattern. A lot more use cases please refer to original link.

Custom Decimal Separator

```
NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;
nfi.CurrencyPositivePattern = 0;
nfi.CurrencyDecimalSeparator = "..";
string.Format(nfi, "{0:C}", 112.236677); //$112..24
```

Section 12.6: Using custom number format

`NumberFormatInfo` can be used for formatting both integer and float numbers.

```
// invariantResult is "1,234,567.89"
var invarianResult = string.Format(CultureInfo.InvariantCulture, "{0:#,###,##}", 1234567.89);

// NumberFormatInfo is one of classes that implement IFormatProvider
var customProvider = new NumberFormatInfo
{
    NumberDecimalSeparator = "_NS_", // will be used instead of ','
    NumberGroupSeparator = "_GS_", // will be used instead of '.'
};

// customResult is "1_GS_234_GS_567_NS_89"
var customResult = string.Format(customProvider, "{0:#,###.##}", 1234567.89);
```

Section 12.7: Align left/ right, pad with spaces

The second value in the curly braces dictates the length of the replacement string. By adjusting the second value to be positive or negative, the alignment of the string can be changed.

```
string.Format("LEFT:  string: ->{0,-5}<- int: ->{1,-5}<-", "abc", 123);
string.Format("RIGHT: string: ->{0,5}<- int: ->{1,5}<-", "abc", 123);
```

Output:

```
LEFT:  string: ->abc  <- int: ->123  <-
RIGHT: string: ->  abc<- int: ->  123<-
```

Section 12.8: Numeric formats

```
// Integral types as hex
string.Format("Hexadecimal: byte2: {0:x2}; byte4: {0:X4}; char: {1:x2}", 123, (int)'A');

// Integers with thousand separators
string.Format("Integer, thousand sep.: {0:#,#}; fixed length: >{0,10:#,#}<", 1234567);

// Integer with leading zeroes
string.Format("Integer, leading zeroes: {0:00}; ", 1);

// Decimals
string.Format("Decimal, fixed precision: {0:0.000}; as percents: {0:0.00%}", 0.12);
```

Output:

```
Hexadecimal: byte2: 7b; byte4: 007B; char: 41
Integer, thousand sep.: 1,234,567; fixed length: > 1,234,567<
Integer, leading zeroes: 01;
Decimal, fixed precision: 0.120; as percents: 12.00%
```

Section 12.9: ToString()

The ToString() method is present on all reference object types. This is due to all reference types being derived from Object which has the ToString() method on it. The ToString() method on the object base class returns the type name. The fragment below will print out "User" to the console.

```
public class User
{
    public string Name { get; set; }
    public int Id { get; set; }
}

...

var user = new User {Name = "User1", Id = 5};
Console.WriteLine(user.ToString());
```

However, the class User can also override ToString() in order to alter the string it returns. The code fragment below prints out "Id: 5, Name: User1" to the console.

```
public class User
{
    public string Name { get; set; }
    public int Id { get; set; }
    public override ToString()
    {
        return string.Format("Id: {0}, Name: {1}", Id, Name);
    }
}

...

var user = new User {Name = "User1", Id = 5};
Console.WriteLine(user.ToString());
```

Section 12.10: Escaping curly brackets inside a String.Format() expression

```
string outsidetext = "I am outside of bracket";
string.Format("{{I am in brackets!}} {0}", outsidetext);

//Outputs "{I am in brackets!} I am outside of bracket"
```

Section 12.11: Relationship with ToString()

While the `String.Format()` method is certainly useful in formatting data as strings, it may often be a bit overkill, especially when dealing with a single object as seen below :

```
String.Format("{0:C}", money);  // yields "$42.00"
```

An easier approach might be to simply use the `ToString()` method available on all objects within C#. It supports all of the same standard and custom formatting strings, but doesn't require the necessary parameter mapping as there will only be a single argument :

```
money.ToString("C");  // yields "$42.00"
```

Caveats & Formatting Restrictions

While this approach may be simpler in some scenarios, the `ToString()` approach is limited with regards to adding left or right padding like you might do within the `String.Format()` method :

```
String.Format("{0,10:C}", money);  // yields "    $42.00"
```

In order to accomplish this same behavior with the `ToString()` method, you would need to use another method like `PadLeft()` or `PadRight()` respectively :

```
money.ToString("C").PadLeft(10);  // yields "    $42.00"
```

Chapter 13: String Concatenate

Section 13.1: + Operator

```
string s1 = "string1";
string s2 = "string2";

string s3 = s1 + s2; // "string1string2"
```

Section 13.2: Concatenate strings using System.Text.StringBuilder

Concatenating strings using a StringBuilder can offer performance advantages over simple string concatenation using +. This is due to the way memory is allocated. Strings are reallocated with each concatenation, StringBuilders allocate memory in blocks only reallocating when the current block is exhausted. This can make a huge difference when doing a lot of small concatenations.

```
StringBuilder sb = new StringBuilder();
for (int i = 1; i <= 5; i++)
{
    sb.Append(i);
    sb.Append(" ");
}
Console.WriteLine(sb.ToString()); // "1 2 3 4 5 "
```

Calls to `Append()` can be daisy chained, because it returns a reference to the `StringBuilder`:

```
StringBuilder sb = new StringBuilder();
sb.Append("some string ")
  .Append("another string");
```

Section 13.3: Concat string array elements using String.Join

The `String.Join` method can be used to concatenate multiple elements from a string array.

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";

string result = String.Join(separator, value, 1, 2);
Console.WriteLine(result);
```

> Produces the following output: "orange, grape"

This example uses the `String.Join(String, String[], Int32, Int32)` overload, which specifies the start index and count on top of the separator and value.

If you do not wish to use the startIndex and count overloads, you can join all string given. Like this:

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";
string result = String.Join(separator, value);
Console.WriteLine(result);
```

which will produce;

> apple, orange, grape, pear

Section 13.4: Concatenation of two strings using $

$ provides an easy and a concise method to concatenate multiple strings.

```
var str1 = "text1";
var str2 = " ";
var str3 = "text3";
string result2 = $"{str1}{str2}{str3}"; //"text1 text3"
```

Chapter 14: String Manipulation

Section 14.1: Replacing a string within a string

Using the `System.String.Replace` method, you can replace part of a string with another string.

```
string s = "Hello World";
s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

All the occurrences of the search string are replaced:

```
string s = "Hello World";
s = s.Replace("l", "L"); // s = "HeLLo WorLD"
```

`String.Replace` can also be used to *remove* part of a string, by specifying an empty string as the replacement value:

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

Section 14.2: Finding a string within a string

Using the `System.String.Contains` you can find out if a particular string exists within a string. The method returns a boolean, true if the string exists else false.

```
string s = "Hello World";
bool stringExists = s.Contains("ello");  //stringExists =true as the string contains the substring
```

Using the `System.String.IndexOf` method, you can locate the starting position of a substring within an existing string.
Note the returned position is zero-based, a value of -1 is returned if the substring is not found.

```
string s = "Hello World";
int location = s.IndexOf("ello"); // location = 1
```

To find the first location from the ***end*** of a string, use the `System.String.LastIndexOf` method:

```
string s = "Hello World";
int location = s.LastIndexOf("l"); // location = 9
```

Section 14.3: Removing (Trimming) white-space from a string

The `System.String.Trim` method can be used to remove all leading and trailing white-space characters from a string:

```
string s = "     String with spaces at both ends      ";
s = s.Trim(); // s = "String with spaces at both ends"
```

In addition:

- To remove white-space only from the *beginning* of a string use: `System.String.TrimStart`
- To remove white-space only from the *end* of a string use: `System.String.TrimEnd`

Substring to extract part of a string.

The `System.String.Substring` method can be used to extract a portion of the string.

```
string s ="A portion of word that is retained";
s=str.Substring(26);  //s="retained"

s1 = s.Substring(0,5);  //s="A por"
```

Section 14.4: Splitting a string using a delimiter

Use the `System.String.Split` method to return a string array that contains substrings of the original string, split based on a specified delimiter:

```
string sentence = "One Two Three Four";
string[] stringArray = sentence.Split(' ');

foreach (string word in stringArray)
{
    Console.WriteLine(word);
}
```

Output:

> One
> Two
> Three
> Four

Section 14.5: Concatenate an array of strings into a single string

The `System.String.Join` method allows to concatenate all elements in a string array, using a specified separator between each element:

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(",", words); // singleString = "One,Two,Three,Four"
```

Section 14.6: String Concatenation

String Concatenation can be done by using the `System.String.Concat` method, or (much easier) using the `+` operator:

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

Section 14.7: Changing the case of characters within a String

The `System.String` class supports a number of methods to convert between uppercase and lowercase characters in a string.

- `System.String.ToLowerInvariant` is used to return a String object converted to lowercase.
- `System.String.ToUpperInvariant` is used to return a String object converted to uppercase.

Note: The reason to use the *invariant* versions of these methods is to prevent producing unexpected culture-specific letters. This is explained here in detail.

Example:

```
string s = "My String";
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

Note that you *can* choose to specify a specific **Culture** when converting to lowercase and uppercase by using the String.ToLower(CultureInfo) and String.ToUpper(CultureInfo) methods accordingly.

Chapter 15: String Interpolation

Section 15.1: Format dates in strings

```
var date = new DateTime(2015, 11, 11);
var str = $"It's {date:MMMM d, yyyy}, make a wish!";
System.Console.WriteLine(str);
```

You can also use the [DateTime.ToString] method to format the `DateTime` object. This will produce the same output as the code above.

```
var date = new DateTime(2015, 11, 11);
var str = date.ToString("MMMM d, yyyy");
str = "It's " + str + ", make a wish!";
Console.WriteLine(str);
```

Output:

> It's November 11, 2015, make a wish!

Live Demo on .NET Fiddle

Live Demo using DateTime.ToString

> **Note:** `MM` stands for months and `mm` for minutes. Be very careful when using these as mistakes can introduce bugs that may be difficult to discover.

Section 15.2: Padding the output

String can be formatted to accept a padding parameter that will specify how many character positions the inserted string will use :

```
${value, padding}
```

> **NOTE:** Positive padding values indicate left padding and negative padding values indicate right padding.

Left Padding

A left padding of 5 (adds 3 spaces before the value of number, so it takes up a total of 5 character positions in the resulting string.)

```
var number = 42;
var str = $"The answer to life, the universe and everything is {number, 5}.";
//str is "The answer to life, the universe and everything is    42.";
//                                                          ^^^^^
System.Console.WriteLine(str);
```

Output:

```
The answer to life, the universe and everything is    42.
```

Live Demo on .NET Fiddle

Right Padding

Right padding, which uses a negative padding value, will add spaces to the end of the current value.

```
var number = 42;
var str = $"The answer to life, the universe and everything is ${number, -5}.";
//str is "The answer to life, the universe and everything is 42    .";
//                                                          ^^^^^
System.Console.WriteLine(str);
```

Output:

```
The answer to life, the universe and everything is 42    .
```

Live Demo on .NET Fiddle

Padding with Format Specifiers

You can also use existing formatting specifiers in conjunction with padding.

```
var number = 42;
var str = $"The answer to life, the universe and everything is ${number, 5:f1}";
//str is "The answer to life, the universe and everything is 42.1 ";
//                                                          ^^^^^
```

Live Demo on .NET Fiddle

Section 15.3: Expressions

Full expressions can also be used in interpolated strings.

```
var StrWithMathExpression = $"1 + 2 = {1 + 2}"; // -> "1 + 2 = 3"

string world = "world";
var StrWithFunctionCall = $"Hello, {world.ToUpper()}!"; // -> "Hello, WORLD!"
```

Live Demo on .NET Fiddle

Section 15.4: Formatting numbers in strings

You can use a colon and the standard numeric format syntax to control how numbers are formatted.

```
var decimalValue = 120.5;

var asCurrency = $"It costs {decimalValue:C}";
// String value is "It costs $120.50" (depending on your local currency settings)

var withThreeDecimalPlaces = $"Exactly {decimalValue:F3}";
// String value is "Exactly 120.500"

var integerValue = 57;
```

```
var prefixedIfNecessary = $"{integerValue:D5}";
// String value is "00057"
```

Live Demo on .NET Fiddle

Section 15.5: Simple Usage

```
var name = "World";
var str = $"Hello, {name}!";
//str now contains: "Hello, World!";
```

Behind the scenes

Internally this

```
$"Hello, {name}!"
```

Will be compiled to something like this:

```
string.Format("Hello, {0}!", name);
```

Chapter 16: String Escape Sequences

Section 16.1: Escaping special symbols in string literals

Backslash

```
// The filename will be c:\myfile.txt in both cases
string filename = "c:\\myfile.txt";
string filename = @"c:\myfile.txt";
```

The second example uses a verbatim string literal, which doesn't treat the backslash as an escape character.

Quotes

```
string text = "\"Hello World!\", said the quick brown fox.";
string verbatimText = @""""Hello World!"", said the quick brown fox.";
```

Both variables will contain the same text.

> "Hello World!", said the quick brown fox.

Newlines

Verbatim string literals can contain newlines:

```
string text = "Hello\r\nWorld!";
string verbatimText = @"Hello
World!";
```

Both variables will contain the same text.

Section 16.2: Unicode character escape sequences

```
string sqrt = "\\u221A";      // √
string emoji = "\\U0001F601"; // ?
string text = "\\u0022Hello World\\u0022"; // "Hello World"
string variableWidth = "\\x22Hello World\\x22"; // "Hello World"
```

Section 16.3: Escaping special symbols in character literals

Apostrophes

```
char apostrophe = '\'';
```

Backslash

```
char oneBackslash = '\\';
```

Section 16.4: Using escape sequences in identifiers

Escape sequences are not restricted to `string` and `char` literals.

Suppose you need to override a third-party method:

```
protected abstract IEnumerable<Texte> ObtenirŒuvres();
```

and suppose the character `Œ` is not available in the character encoding you use for your C# source files. You are lucky, it is permitted to use escapes of the type `\u####` or `\U########` in ***identifiers*** in the code. So it is legal to write:

```
protected override IEnumerable<Texte> Obtenir\u0152uvres()
{
    // ...
}
```

and the C# compiler will know `Œ` and `\u0152` are the same character.

(However, it might be a good idea to switch to UTF-8 or a similar encoding that can handle all characters.)

Section 16.5: Unrecognized escape sequences produce compile-time errors

The following examples will not compile:

```
string s = "\c";
char c = '\c';
```

Instead, they will produce the error `Unrecognized escape sequence` at compile time.

Chapter 17: StringBuilder

Section 17.1: What a StringBuilder is and when to use one

A `StringBuilder` represents a series of characters, which unlike a normal string, are mutable. Often times there is a need to modify strings that we've already made, but the standard string object is not mutable. This means that each time a string is modified, a new string object needs to be created, copied to, and then reassigned.

```
string myString = "Apples";
mystring += " are my favorite fruit";
```

In the above example, `myString` initially only has the value `"Apples"`. However, when we concatenate `" are my favorite fruit"`, what the string class does internally needs to do involves:

- Creating a new array of characters equal to the length of `myString` and the new string we are appending.
- Copying all of the characters of `myString` into the beginning of our new array and copying the new string into the end of the array.
- Create a new string object in memory and reassign it to `myString`.

For a single concatenation, this is relatively trivial. However, what if needed to perform many append operations, say, in a loop?

```
String myString = "";
for (int i = 0; i < 10000; i++)
    myString += " "; // puts 10,000 spaces into our string
```

Due to the repeated copying and object creation, this will bring significantly degrade the performance of our program. We can avoid this by instead using a `StringBuilder`.

```
StringBuilder myStringBuilder = new StringBuilder();
for (int i = 0; i < 10000; i++)
    myStringBuilder.Append(' ');
```

Now when the same loop is run, the performance and speed of the execution time of the program will be significantly faster than using a normal string. To make the `StringBuilder` back into a normal string, we can simply call the `ToString()` method of `StringBuilder`.

However, this isn't the only optimization `StringBuilder` has. In order to further optimize functions, we can take advantage of other properties that help improve performance.

```
StringBuilder sb = new StringBuilder(10000); // initializes the capacity to 10000
```

If we know in advance how long our `StringBuilder` needs to be, we can specify its size ahead of time, which will prevent it from needing to resize the character array it has internally.

```
sb.Append('k', 2000);
```

Though using `StringBuilder` for appending is much faster than a string, it can run even faster if you only need to add a single character many times.

Once you have completed building your string, you may use the `ToString()` method on the `StringBuilder` to convert it to a basic `string`. This is often necessary because the `StringBuilder` class does not inherit from `string`.

For example, here is how you can use a `StringBuilder` to create a `string`:

```
string RepeatCharacterTimes(char character, int times)
{
    StringBuilder builder = new StringBuilder("");
    for (int counter = 0; counter < times; counter++)
    {
        //Append one instance of the character to the StringBuilder.
        builder.Append(character);
    }
    //Convert the result to string and return it.
    return builder.ToString();
}
```

In conclusion, `StringBuilder` should be used in place of string when many modifications to a string need to be made with performance in mind.

Section 17.2: Use StringBuilder to create string from a large number of records

```
public string GetCustomerNamesCsv()
{
    List<CustomerData> customerDataRecords = GetCustomerData(); // Returns a large number of
records, say, 10000+

    StringBuilder customerNamesCsv = new StringBuilder();
    foreach (CustomerData record in customerDataRecords)
    {
       customerNamesCsv
            .Append(record.LastName)
            .Append(',')
            .Append(record.FirstName)
            .Append(Environment.NewLine);
    }

    return customerNamesCsv.ToString();
}
```

Chapter 18: Regex Parsing

Name	Details
Pattern	The `string` pattern that has to be used for the lookup. For more information: msdn
RegexOptions *[Optional]*	The common options in here are `Singleline` and `Multiline`. They are changing the behaviour of pattern-elements like the dot (.) which won't cover a `NewLine` (\n) in `Multiline-Mode` but in `SingleLine-Mode`. Default behaviour: msdn
Timeout *[Optional]*	Where patterns are getting more complex the lookup can consume more time. This is the passed timeout for the lookup just as known from network-programming.

Section 18.1: Single match

```
using System.Text.RegularExpressions;
```

```
string pattern = ":(.*?):";
string lookup = "--:text in here:--";

// Instanciate your regex object and pass a pattern to it
Regex rgxLookup = new Regex(pattern, RegexOptions.Singleline, TimeSpan.FromSeconds(1));
// Get the match from your regex-object
Match mLookup = rgxLookup.Match(lookup);

// The group-index 0 always covers the full pattern.
// Matches inside parentheses will be accessed through the index 1 and above.
string found = mLookup.Groups[1].Value;
```

Result:

```
found = "text in here"
```

Section 18.2: Multiple matches

```
using System.Text.RegularExpressions;
```

```
List<string> found = new List<string>();
string pattern = ":(.*?):";
string lookup = "--:text in here:--:another one:-:third one:---!123:fourth:";

// Instanciate your regex object and pass a pattern to it
Regex rgxLookup = new Regex(pattern, RegexOptions.Singleline, TimeSpan.FromSeconds(1));
MatchCollection mLookup = rgxLookup.Matches(lookup);

foreach(Match match in mLookup)
{
    found.Add(match.Groups[1].Value);
}
```

Result:

```
found = new List<string>() { "text in here", "another one", "third one", "fourth" }
```

Chapter 19: DateTime Methods

Section 19.1: DateTime Formatting

Standard DateTime Formatting

DateTimeFormatInfo specifies a set of specifiers for simple date and time formatting. Every specifier correspond to a particular DateTimeFormatInfo format pattern.

```
//Create datetime
DateTime dt = new DateTime(2016,08,01,18,50,23,230);

var t = String.Format("{0:t}", dt); // "6:50 PM"                             ShortTime
var d = String.Format("{0:d}", dt); // "8/1/2016"                           ShortDate
var T = String.Format("{0:T}", dt); // "6:50:23 PM"                         LongTime
var D = String.Format("{0:D}", dt); // "Monday, August 1, 2016"             LongDate
var f = String.Format("{0:f}", dt); // "Monday, August 1, 2016 6:50 PM"     LongDate+ShortTime
var F = String.Format("{0:F}", dt); // "Monday, August 1, 2016 6:50:23 PM"  FullDateTime
var g = String.Format("{0:g}", dt); // "8/1/2016 6:50 PM"                   ShortDate+ShortTime
var G = String.Format("{0:G}", dt); // "8/1/2016 6:50:23 PM"                ShortDate+LongTime
var m = String.Format("{0:m}", dt); // "August 1"                           MonthDay
var y = String.Format("{0:y}", dt); // "August 2016"                        YearMonth
var r = String.Format("{0:r}", dt); // "SMon, 01 Aug 2016 18:50:23 GMT"     RFC1123
var s = String.Format("{0:s}", dt); // "2016-08-01T18:50:23"                SortableDateTime
var u = String.Format("{0:u}", dt); // "2016-08-01 18:50:23Z"
 UniversalSortableDateTime
```

Custom DateTime Formatting

There are following custom format specifiers:

- `y` (year)
- `M` (month)
- `d` (day)
- `h` (hour 12)
- `H` (hour 24)
- `m` (minute)
- `s` (second)
- `f` (second fraction)
- `F` (second fraction, trailing zeroes are trimmed)
- `t` (P.M or A.M)
- `z` (time zone).

```
var year =       String.Format("{0:y yy yyy yyyy}", dt); // "16 16 2016 2016"  year
var month =      String.Format("{0:M MM MMM MMMM}", dt); // "8 08 Aug August"  month
var day =        String.Format("{0:d dd ddd dddd}", dt); // "1 01 Mon Monday"  day
var hour =       String.Format("{0:h hh H HH}",     dt); // "6 06 18 18"       hour 12/24
var minute =     String.Format("{0:m mm}",          dt); // "50 50"            minute
var secound =    String.Format("{0:s ss}",          dt); // "23 23"            second
var fraction =   String.Format("{0:f ff fff ffff}", dt); // "2 23 230 2300"    sec.fraction
var fraction2 =  String.Format("{0:F FF FFF FFFF}", dt); // "2 23 23 23"       without zeroes
var period =     String.Format("{0:t tt}",          dt); // "P PM"             A.M. or P.M.
var zone =       String.Format("{0:z zz zzz}",      dt); // "+0 +00 +00:00"    time zone
```

You can use also date separator `/` (slash) and time sepatator `:` (colon).

For code example

For more information MSDN.

Section 19.2: DateTime.AddDays(Double)

Add days into a dateTime object.

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(36);
Console.WriteLine("Today: {0:dddd}", today);
Console.WriteLine("36 days from today: {0:dddd}", answer);
```

You also can subtract days passing a negative value:

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(-3);
Console.WriteLine("Today: {0:dddd}", today);
Console.WriteLine("-3 days from today: {0:dddd}", answer);
```

Section 19.3: DateTime.AddHours(Double)

```
double[] hours = {.08333, .16667, .25, .33333, .5, .66667, 1, 2,
                  29, 30, 31, 90, 365};
DateTime dateValue = new DateTime(2009, 3, 1, 12, 0, 0);

foreach (double hour in hours)
  Console.WriteLine("{0} + {1} hour(s) = {2}", dateValue, hour,
                    dateValue.AddHours(hour));
```

Section 19.4: DateTime.Parse(String)

```
// Converts the string representation of a date and time to its DateTime equivalent

var dateTime = DateTime.Parse("14:23 22 Jul 2016");

Console.WriteLine(dateTime.ToString());
```

Section 19.5: DateTime.TryParse(String, DateTime)

```
// Converts the specified string representation of a date and time to its DateTime equivalent and
returns a value that indicates whether the conversion succeeded

string[] dateTimeStrings = new []{
    "14:23 22 Jul 2016",
    "99:23 2x Jul 2016",
    "22/7/2016 14:23:00"
};

foreach(var dateTimeString in dateTimeStrings){

    DateTime dateTime;

    bool wasParsed = DateTime.TryParse(dateTimeString, out dateTime);

    string result = dateTimeString +
        (wasParsed
```

```
            ? $"was parsed to {dateTime}"
            : "can't be parsed to DateTime");

    Console.WriteLine(result);
}
```

Section 19.6: DateTime.AddMilliseconds(Double)

```
string dateFormat = "MM/dd/yyyy hh:mm:ss.fffffff";
DateTime date1 = new DateTime(2010, 9, 8, 16, 0, 0);
Console.WriteLine("Original date: {0} ({1:N0} ticks)\n",
                  date1.ToString(dateFormat), date1.Ticks);

DateTime date2 = date1.AddMilliseconds(1);
Console.WriteLine("Second date:   {0} ({1:N0} ticks)",
                  date2.ToString(dateFormat), date2.Ticks);
Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)\n",
                  date2 - date1, date2.Ticks - date1.Ticks);

DateTime date3 = date1.AddMilliseconds(1.5);
Console.WriteLine("Third date:    {0} ({1:N0} ticks)",
                  date3.ToString(dateFormat), date3.Ticks);
Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)",
                  date3 - date1, date3.Ticks - date1.Ticks);
```

Section 19.7: DateTime.Compare(DateTime t1, DateTime t2)

```
DateTime date1 = new DateTime(2009, 8, 1, 0, 0, 0);
DateTime date2 = new DateTime(2009, 8, 1, 12, 0, 0);
int result = DateTime.Compare(date1, date2);
string relationship;

if (result < 0)
    relationship = "is earlier than";
else if (result == 0)
    relationship = "is the same time as";
else relationship = "is later than";

Console.WriteLine("{0} {1} {2}", date1, relationship, date2);
```

Section 19.8: DateTime.DaysInMonth(Int32,Int32)

```
const int July = 7;
const int Feb = 2;

int daysInJuly = System.DateTime.DaysInMonth(2001, July);
Console.WriteLine(daysInJuly);

// daysInFeb gets 28 because the year 1998 was not a leap year.
int daysInFeb = System.DateTime.DaysInMonth(1998, Feb);
Console.WriteLine(daysInFeb);

// daysInFebLeap gets 29 because the year 1996 was a leap year.
int daysInFebLeap = System.DateTime.DaysInMonth(1996, Feb);
Console.WriteLine(daysInFebLeap);
```

Section 19.9: DateTime.AddYears(Int32)

Add years on the dateTime object:

```
DateTime baseDate = new DateTime(2000, 2, 29);
Console.WriteLine("Base Date: {0:d}\n", baseDate);

// Show dates of previous fifteen years.
for (int ctr = -1; ctr >= -15; ctr--)
   Console.WriteLine("{0,2} year(s) ago:{1:d}",
                     Math.Abs(ctr), baseDate.AddYears(ctr));

Console.WriteLine();

// Show dates of next fifteen years.
for (int ctr = 1; ctr <= 15; ctr++)
   Console.WriteLine("{0,2} year(s) from now: {1:d}",
                     ctr, baseDate.AddYears(ctr));
```

Section 19.10: Pure functions warning when dealing with DateTime

Wikipedia currently defines a pure function as follows:

1. The function always evaluates the same result value given the same argument value(s). The function result value cannot depend on any hidden information or state that may change while program execution proceeds or between different executions of the program, nor can it depend on any external input from I/O devices .
2. Evaluation of the result does not cause any semantically observable side effect or output, such as mutation of mutable objects or output to I/O devices

As a developer you need to be aware of pure methods and you will stumble upon these a lot in many areas. One I have seen that bites many junior developers is working with DateTime class methods. A lot of these are pure and if you are unaware of these you can be in for a suprise. An example:

```
        DateTime sample = new DateTime(2016, 12, 25);
        sample.AddDays(1);
        Console.WriteLine(sample.ToShortDateString());
```

Given the example above one may expect the result printed to console to be '26/12/2016' but in reality you end up with the same date. This is because AddDays is a pure method and does not affect the original date. To get the expected output you would have to modify the AddDays call to the following:

```
        sample = sample.AddDays(1);
```

Section 19.11: DateTime.TryParseExact(String,␣String,␣IFormatProvider,␣DateTimeStyles,␣DateTime)

Converts the specified string representation of a date and time to its DateTime equivalent using the specified format, culture-specific format information, and style. The format of the string representation must match the specified format exactly. The method returns a value that indicates whether the conversion succeeded.

For Example

```
CultureInfo enUS = new CultureInfo("en-US");
```

```
string dateString;
System.DateTime dateValue;
```

Parse date with no style flags.

```
dateString = " 5/01/2009 8:30 AM";
if (DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.None, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}


// Allow a leading space in the date string.
if(DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.AllowLeadingWhite, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Use custom formats with M and MM.

```
dateString = "5/01/2009 09:00";
if(DateTime.TryParseExact(dateString, "M/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}

// Allow a leading space in the date string.
if(DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out
dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Parse a string with time zone information.

```
dateString = "05/01/2009 01:30:42 PM -05:00";
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS, DateTimeStyles.None, out
dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

```
// Allow a leading space in the date string.
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS,
DateTimeStyles.AdjustToUniversal, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Parse a string represengting UTC.

```
dateString = "2008-06-11T16:11:20.0904778Z";
if(DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture, DateTimeStyles.None, out
dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
  Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}

if (DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture,
DateTimeStyles.RoundtripKind, out dateValue))
{
   Console.WriteLine("Converted '{0}' to {1} ({2}).", dateString, dateValue, dateValue.Kind);
}
else
{
   Console.WriteLine("'{0}' is not in an acceptable format.", dateString);
}
```

Outputs

```
' 5/01/2009 8:30 AM' is not in an acceptable format.
Converted ' 5/01/2009 8:30 AM' to 5/1/2009 8:30:00 AM (Unspecified).
Converted '5/01/2009 09:00' to 5/1/2009 9:00:00 AM (Unspecified).
'5/01/2009 09:00' is not in an acceptable format.
Converted '05/01/2009 01:30:42 PM -05:00' to 5/1/2009 11:30:42 AM (Local).
Converted '05/01/2009 01:30:42 PM -05:00' to 5/1/2009 6:30:42 PM (Utc).
Converted '2008-06-11T16:11:20.0904778Z' to 6/11/2008 9:11:20 AM (Local).
Converted '2008-06-11T16:11:20.0904778Z' to 6/11/2008 4:11:20 PM (Utc).
```

Section 19.12: DateTime.Add(TimeSpan)

```
// Calculate what day of the week is 36 days from this instant.
System.DateTime today = System.DateTime.Now;
System.TimeSpan duration = new System.TimeSpan(36, 0, 0, 0);
System.DateTime answer = today.Add(duration);
System.Console.WriteLine("{0:dddd}", answer);
```

Section 19.13: Parse and TryParse with culture info

You might want to use it when parsing DateTimes from different cultures (languages), following example parses Dutch date.

```
DateTime dateResult;
var dutchDateString = "31 oktober 1999 04:20";
var dutchCulture = CultureInfo.CreateSpecificCulture("nl-NL");
DateTime.TryParse(dutchDateString, dutchCulture, styles, out dateResult);
// output {31/10/1999 04:20:00}
```

Example of Parse:

```
DateTime.Parse(dutchDateString, dutchCulture)
// output {31/10/1999 04:20:00}
```

Section 19.14: DateTime as initializer in for-loop

```
// This iterates through a range between two DateTimes
// with the given iterator (any of the Add methods)

DateTime start = new DateTime(2016, 01, 01);
DateTime until = new DateTime(2016, 02, 01);

// NOTICE: As the add methods return a new DateTime you have
// to overwrite dt in the iterator like dt = dt.Add()
for (DateTime dt = start; dt < until; dt = dt.AddDays(1))
{
    Console.WriteLine("Added {0} days. Resulting DateTime: {1}",
                      (dt - start).Days, dt.ToString());
}
```

Iterating on a `TimeSpan` works the same way.

Section 19.15: DateTime.ParseExact(String, String, IFormatProvider)

Converts the specified string representation of a date and time to its DateTime equivalent using the specified format and culture-specific format information. The format of the string representation must match the specified format exactly.

Convert a specific format string to equivalent DateTime

Let's say we have a culture-specific DateTime string `08-07-2016 11:30:12 PM` as `MM-dd-yyyy hh:mm:ss tt` format and we want it to convert to equivalent `DateTime` object

```
string str = "08-07-2016 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "MM-dd-yyyy hh:mm:ss tt", CultureInfo.CurrentCulture);
```

Convert a date time string to equivalent `DateTime` object without any specific culture format

Let's say we have a DateTime string in `dd-MM-yy hh:mm:ss tt` format and we want it to convert to equivalent `DateTime` object, without any specific culture information

```
string str = "17-06-16 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "dd-MM-yy hh:mm:ss tt", CultureInfo.InvariantCulture);
```

Convert a date time string to equivalent DateTime object without any specific culture format with different format

Let's say we have a Date string , example like '23-12-2016' or '12/23/2016' and we want it to convert to equivalent `DateTime` object, without any specific culture information

```
   string date =  '23-12-2016' or date = 12/23/2016';
   string[] formats = new string[] {"dd-MM-yyyy","MM/dd/yyyy"}; // even can add more possible
formats.
   DateTime date = DateTime.ParseExact(date,formats,
CultureInfo.InvariantCulture,DateTimeStyles.None);
```

NOTE : `System.Globalization` needs to be added for CultureInfo Class

Section 19.16: DateTime ToString, ToShortDateString, ToLongDateString and ToString formatted

```
using System;

public class Program
{
    public static void Main()
    {
        var date = new DateTime(2016,12,31);

        Console.WriteLine(date.ToString());        //Outputs: 12/31/2016 12:00:00 AM
        Console.WriteLine(date.ToShortDateString()); //Outputs: 12/31/2016
        Console.WriteLine(date.ToLongDateString()); //Outputs: Saturday, December 31, 2016
        Console.WriteLine(date.ToString("dd/MM/yyyy"));    //Outputs: 31/12/2016
    }
}
```

Section 19.17: Current Date

To get the current date you use the `DateTime.Today` property. This returns a `DateTime` object with today's date. When this is then converted `.ToString()` it is done so in your system's locality by default.

For example:

```
Console.WriteLine(DateTime.Today);
```

Writes today's date, in your local format to the console.

Chapter 20: Arrays

Section 20.1: Declaring an array

An array can be declared and filled with the default value using square bracket ([ ]) initialization syntax. For example, creating an array of 10 integers:

```
int[] arr = new int[10];
```

Indices in C# are zero-based. The indices of the array above will be 0-9. For example:

```
int[] arr = new int[3] {7,9,4};
Console.WriteLine(arr[0]); //outputs 7
Console.WriteLine(arr[1]); //outputs 9
```

Which means the system starts counting the element index from 0. Moreover, accesses to elements of arrays are done in **constant time**. That means accessing to the first element of the array has the same cost (in time) of accessing the second element, the third element and so on.

You may also declare a bare reference to an array without instantiating an array.

```
int[] arr = null;   // OK, declares a null reference to an array.
int first = arr[0]; // Throws System.NullReferenceException because there is no actual array.
```

An array can also be created and initialized with custom values using collection initialization syntax:

```
int[] arr = new int[] { 24, 2, 13, 47, 45 };
```

The `new int[]` portion can be omitted when declaring an array variable. This is not a self-contained *expression*, so using it as part of a different call does not work (for that, use the version with `new`):

```
int[] arr = { 24, 2, 13, 47, 45 };  // OK
int[] arr1;
arr1 = { 24, 2, 13, 47, 45 };       // Won't compile
```

Implicitly typed arrays

Alternatively, in combination with the `var` keyword, the specific type may be omitted so that the type of the array is inferred:

```
// same as int[]
var arr = new [] { 1, 2, 3 };
// same as string[]
var arr = new [] { "one", "two", "three" };
// same as double[]
var arr = new [] { 1.0, 2.0, 3.0 };
```

Section 20.2: Initializing an array filled with a repeated non-default value

As we know we can declare an array with default values:

```
int[] arr = new int[10];
```

This will create an array of 10 integers with each element of the array having value `0` (the default value of type `int`).

To create an array initialized with a non-default value, we can use `Enumerable.Repeat` from the `System.Linq` Namespace:

1. To create a `bool` array of size 10 filled with **"true"**

```
bool[] booleanArray = Enumerable.Repeat(true, 10).ToArray();
```

2. To create an `int` array of size 5 filled with **"100"**

```
int[] intArray = Enumerable.Repeat(100, 5).ToArray();
```

3. To create a `string` array of size 5 filled with **"C#"**

```
string[] strArray = Enumerable.Repeat("C#", 5).ToArray();
```

Section 20.3: Copying arrays

Copying a partial array with the static `Array.Copy()` method, beginning at index 0 in both, source and destination:

```
var sourceArray = new int[] { 11, 12, 3, 5, 2, 9, 28, 17 };
var destinationArray= new int[3];
Array.Copy(sourceArray, destinationArray, 3);

// destinationArray will have 11,12 and 3
```

Copying the whole array with the `CopyTo()` instance method, beginning at index 0 of the source and the specified index in the destination:

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = new int[6];
sourceArray.CopyTo(destinationArray, 2);

// destinationArray will have 0, 0, 11, 12, 7 and 0
```

`Clone` is used to create a copy of an array object.

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = (int)sourceArray.Clone();

//destinationArray will be created and will have 11,12,17.
```

Both `CopyTo` and `Clone` perform shallow copy which means the contents contains references to the same object as the elements in the original array.

Section 20.4: Comparing arrays for equality

LINQ provides a built-in function for checking the equality of two `IEnumerables`, and that function can be used on arrays.

The `SequenceEqual` function will return `true` if the arrays have the same length and the values in corresponding indices are equal, and `false` otherwise.

```
int[] arr1 = { 3, 5, 7 };
int[] arr2 = { 3, 5, 7 };
bool result = arr1.SequenceEqual(arr2);
Console.WriteLine("Arrays equal? {0}", result);
```

This will print:

```
Arrays equal? True
```

Section 20.5: Multi-dimensional arrays

Arrays can have more than one dimension. The following example creates a two-dimensional array of ten rows and ten columns:

```
int[,] arr = new int[10, 10];
```

An array of three dimensions:

```
int[,,] arr = new int[10, 10, 10];
```

You can also initialize the array upon declaration:

```
int[,] arr = new int[4, 2] { {1, 1}, {2, 2}, {3, 3}, {4, 4} };

// Access a member of the multi-dimensional array:
Console.Out.WriteLine(arr[3, 1]);  // 4
```

Section 20.6: Getting and setting array values

```
int[] arr = new int[] { 0, 10, 20, 30};

// Get
Console.WriteLine(arr[2]); // 20

// Set
arr[2] = 100;

// Get the updated value
Console.WriteLine(arr[2]); // 100
```

Section 20.7: Iterate over an array

```
int[] arr = new int[] {1, 6, 3, 3, 9};

for (int i = 0; i < arr.Length; i++)
{
    Console.WriteLine(arr[i]);
}
```

using foreach:

```
foreach (int element in arr)
{
    Console.WriteLine(element);
}
```

using unsafe access with pointers https://msdn.microsoft.com/en-ca/library/y31yhkeb.aspx

```
unsafe
{
    int length = arr.Length;
    fixed (int* p = arr)
    {
        int* pInt = p;
        while (length-- > 0)
        {
            Console.WriteLine(*pInt);
            pInt++;// move pointer to next element
        }
    }
}
```

Output:

```
1
6
3
3
9
```

Section 20.8: Creating an array of sequential numbers

LINQ provides a method that makes it easy to create a collection filled with sequential numbers. For example, you can declare an array which contains the integers between 1 and 100.

The `Enumerable.Range` method allows us to create sequence of integer numbers from a specified start position and a number of elements.

The method takes two arguments: the starting value and the number of elements to generate.

```
Enumerable.Range(int start, int count)
```

Note that `count` cannot be negative.

Usage:

```
int[] sequence = Enumerable.Range(1, 100).ToArray();
```

This will generate an array containing the numbers 1 through 100 (`[1, 2, 3, ..., 98, 99, 100]`).

Because the `Range` method returns an `IEnumerable<int>`, we can use other LINQ methods on it:

```
int[] squares = Enumerable.Range(2, 10).Select(x => x * x).ToArray();
```

This will generate an array that contains 10 integer squares starting at 4: `[4, 9, 16, ..., 100, 121]`.

Section 20.9: Jagged arrays

Jagged arrays are arrays that instead of primitive types, contain arrays (or other collections). It's like an array of arrays - each array element contains another array.

They are similar to multidimensional arrays, but have a slight difference - as multidimensional arrays are limited to a fixed number of rows and columns, with jagged arrays, every row can have a different number of columns.

Declaring a jagged array

For example, declaring a jagged array with 8 columns:

```
int[][] a = new int[8][];
```

The second [] is initialized without a number. To initialize the sub arrays, you would need to do that separately:

```
for (int i = 0; i < a.length; i++)
{
    a[i] = new int[10];
}
```

Getting/Setting values

Now, getting one of the subarrays is easy. Let's print all the numbers of the 3rd column of `a`:

```
for (int i = 0; i < a[2].length; i++)
{
    Console.WriteLine(a[2][i]);
}
```

Getting a specific value:

```
a[<row_number>][<column_number>]
```

Setting a specific value:

```
a[<row_number>][<column_number>] = <value>
```

Remember: It's always recommended to use jagged arrays (arrays of arrays) rather than multidimensional arrays (matrixes). It's faster and safer to use.

Note on the order of the brackets

Consider a three-dimensional array of five-dimensional arrays of one-dimensional arrays of `int`. This is written in C# as:

```
int[,,][,,,,][] arr = new int[8, 10, 12][,,,,][];
```

In the CLR type system, the convention for the ordering of the brackets is reversed, so with the above `arr` instance we have:

```
    arr.GetType().ToString() == "System.Int32[][,,,,][,,]"
```

and likewise:

```
    typeof(int[,,][,,,,][]).ToString() == "System.Int32[][,,,,][,,]"
```

Section 20.10: Array covariance

```
string[] strings = new[] {"foo", "bar"};
object[] objects = strings; // implicit conversion from string[] to object[]
```

This conversion is not type-safe. The following code will raise a runtime exception:

```
string[] strings = new[] {"Foo"};
object[] objects = strings;

objects[0] = new object(); // runtime exception, object is not string
string str = strings[0];   // would have been bad if above assignment had succeeded
```

Section 20.11: Arrays as IEnumerable<> instances

All arrays implement the non-generic `IList` interface (and hence non-generic `ICollection` and `IEnumerable` base interfaces).

More importantly, one-dimensional arrays implement the `IList<>` and `IReadOnlyList<>` generic interfaces (and their base interfaces) for the type of data that they contain. This means that they can be treated as generic enumerable types and passed in to a variety of methods without needing to first convert them to a non-array form.

```
int[] arr1 = { 3, 5, 7 };
IEnumerable<int> enumerableIntegers = arr1; //Allowed because arrays implement IEnumerable<T>
List<int> listOfIntegers = new List<int>();
listOfIntegers.AddRange(arr1); //You can pass in a reference to an array to populate a List.
```

After running this code, the list `listOfIntegers` will contain a `List<int>` containing the values 3, 5, and 7.

The `IEnumerable<>` support means arrays can be queried with LINQ, for example `arr1.Select(i => 10 * i)`.

Section 20.12: Checking if one array contains another array

```
public static class ArrayHelpers
{
    public static bool Contains<T>(this T[] array, T[] candidate)
    {
        if (IsEmptyLocate(array, candidate))
            return false;

        if (candidate.Length > array.Length)
            return false;

        for (int a = 0; a <= array.Length - candidate.Length; a++)
        {
            if (array[a].Equals(candidate[0]))
            {
                int i = 0;
                for (; i < candidate.Length; i++)
                {
                    if (false == array[a + i].Equals(candidate[i]))
                        break;
                }
                if (i == candidate.Length)
                    return true;
            }
        }
```

```
        return false;
    }

    static bool IsEmptyLocate<T>(T[] array, T[] candidate)
    {
        return array == null
            || candidate == null
            || array.Length == 0
            || candidate.Length == 0
            || candidate.Length > array.Length;
    }
}
```

/// Sample

```
byte[] EndOfStream = Encoding.ASCII.GetBytes("---3141592---");
byte[] FakeReceivedFromStream = Encoding.ASCII.GetBytes("Hello, world!!!---3141592---");
if (FakeReceivedFromStream.Contains(EndOfStream))
{
    Console.WriteLine("Message received");
}
```

Chapter 21: O(n) Algorithm for circular rotation of an array

In my path to studying programming there have been simple, but interesting problems to solve as exercises. One of those problems was to rotate an array(or another collection) by a certain value. Here I will share with you a simple formula to do it.

Section 21.1: Example of a generic method that rotates an array by a given shift

I would like to point out that we rotate left when the shifting value is negative and we rotate right when the value is positive.

```
    public static void Main()
    {
        int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int shiftCount = 1;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]

        array = new []{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        shiftCount = 15;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]

        array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        shiftCount = -1;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [2, 3, 4, 5, 6, 7, 8, 9, 10, 1]

        array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        shiftCount = -35;
        Rotate(ref array, shiftCount);
        Console.WriteLine(string.Join(", ", array));
        // Output: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
    }

    private static void Rotate<T>(ref T[] array, int shiftCount)
    {
        T[] backupArray= new T[array.Length];

        for (int index = 0; index < array.Length; index++)
        {
            backupArray[(index + array.Length + shiftCount % array.Length) % array.Length] =
array[index];
        }

        array = backupArray;
    }
```

The thing that is important in this code is the formula with which we find the new index value after the rotation.

(index + array.Length + shiftCount % array.Length) % array.Length

Here is a little more information about it:

(shiftCount % array.Length) -> we normalize the shifting value to be in the length of the array (since in an array with length 10, shifting 1 or 11 is the same thing, the same goes for -1 and -11).

array.Length + (shiftCount % array.Length) -> this is done due to left rotations to make sure we do not go into a negative index, but rotate it to the end of the array. Without it for an array with length 10 for index 0 and a rotation -1 we would go into a negative number (-1) and not get the real rotation index value, which is 9. (10 + (-1 % 10) = 9)

index + array.Length + (shiftCount % array.Length) -> not much to say here as we apply the rotation to the index to get the new index. (0 + 10 + (-1 % 10) = 9)

index + array.Length + (shiftCount % array.Length) % array.Length -> the second normalization is making sure that the new index value does not go outside of the array, but rotates the value in the beginning of the array. It is for right rotations, since in an array with length 10 without it for index 9 and a rotation 1 we would go into index 10, which is outside of the array, and not get the real rotation index value is 0. ((9 + 10 + (1 % 10)) % 10 = 0)

Chapter 22: Enum

An enum can derive from any of the following types: byte, sbyte, short, ushort, int, uint, long, ulong. The default is int, and can be changed by specifying the type in the enum definition:

public enum Weekday : byte { Monday = 1, Tuesday = 2, Wednesday = 3, Thursday = 4, Friday = 5 }

This is useful when P/Invoking to native code, mapping to data sources, and similar circumstances. In general, the default int should be used, because most developers expect an enum to be an int.

Section 22.1: Enum basics

From MSDN:

> An enumeration type (also named an enumeration or an enum) provides an efficient way to define a set of named **integral constants** that may be **assigned to a variable**.

Essentially, an enum is a type that only allows a set of finite options, and each option corresponds to a number. By default, those numbers are increasing in the order the values are declared, starting from zero. For example, one could declare an enum for the days of the week:

```
public enum Day
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

That enum could be used like this:

```
// Define variables with values corresponding to specific days
Day myFavoriteDay = Day.Friday;
Day myLeastFavoriteDay = Day.Monday;

// Get the int that corresponds to myFavoriteDay
// Friday is number 4
int myFavoriteDayIndex = (int)myFavoriteDay;

// Get the day that represents number 5
Day dayFive = (Day)5;
```

By default the underlying type of each element in the `enum` is `int`, but `byte`, `sbyte`, `short`, `ushort`, `uint`, `long` and `ulong` can be used as well. If you use a type other than `int`, you must specify the type using a colon after the enum name:

```
public enum Day : byte
{
    // same as before
}
```

The numbers after the name are now bytes instead of integers. You could get the underlying type of the enum as follows:

```
Enum.GetUnderlyingType(typeof(Days));
```

Output:

```
System.Byte
```

Demo: .NET fiddle

Section 22.2: Enum as flags

The `FlagsAttribute` can be applied to an enum changing the behaviour of the `ToString()` to match the nature of the enum:

```
[Flags]
enum MyEnum
{
    //None = 0, can be used but not combined in bitwise operations
    FlagA = 1,
    FlagB = 2,
    FlagC = 4,
    FlagD = 8
    //you must use powers of two or combinations of powers of two
    //for bitwise operations to work
}

var twoFlags = MyEnum.FlagA | MyEnum.FlagB;

// This will enumerate all the flags in the variable: "FlagA, FlagB".
Console.WriteLine(twoFlags);
```

Because `FlagsAttribute` relies on the enumeration constants to be powers of two (or their combinations) and enum values are ultimately numeric values, you are limited by the size of the underlying numeric type. The largest available numeric type that you can use is `UInt64`, which allows you to specify 64 distinct (non-combined) flag enum constants. The `enum` keyword defaults to the underlying type `int`, which is `Int32`. The compiler will allow the declaration of values wider than 32 bit. Those will wrap around without a warning and result in two or more enum members of the same value. Therefore, if an enum is meant to accomodate a bitset of more than 32 flags, you need to specify a bigger type explicitely:

```
public enum BigEnum : ulong
{
    BigValue = 1 << 63
}
```

Although flags are often only a single bit, they can be combined into named "sets" for easier use.

```
[Flags]
enum FlagsEnum
{
    None = 0,
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,

    Default = Option1 | Option3,
```

```
    All = Option1 | Option2 | Option3,
}
```

To avoid spelling out the decimal values of powers of two, the left-shift operator (<<) can also be used to declare the same enum

```
[Flags]
enum FlagsEnum
{
    None = 0,
    Option1 = 1 << 0,
    Option2 = 1 << 1,
    Option3 = 1 << 2,

    Default = Option1 | Option3,
    All = Option1 | Option2 | Option3,
}
```

Starting with C# 7.0, binary literals can be used too.

To check if the value of enum variable has a certain flag set, the `HasFlag` method can be used. Let's say we have

```
[Flags]
enum MyEnum
{
    One = 1,
    Two = 2,
    Three = 4
}
```

And a `value`

```
var value = MyEnum.One | MyEnum.Two;
```

With `HasFlag` we can check if any of the flags is set

```
if(value.HasFlag(MyEnum.One))
    Console.WriteLine("Enum has One");

if(value.HasFlag(MyEnum.Two))
    Console.WriteLine("Enum has Two");

if(value.HasFlag(MyEnum.Three))
    Console.WriteLine("Enum has Three");
```

Also we can iterate through all values of enum to get all flags that are set

```
var type = typeof(MyEnum);
var names = Enum.GetNames(type);

foreach (var name in names)
{
    var item = (MyEnum)Enum.Parse(type, name);

    if (value.HasFlag(item))
        Console.WriteLine("Enum has " + name);
}
```

Or

```
foreach(MyEnum flagToCheck in Enum.GetValues(typeof(MyEnum)))
{
    if(value.HasFlag(flagToCheck))
    {
         Console.WriteLine("Enum has " + flagToCheck);
    }
}
```

All three examples will print:

```
Enum has One
Enum has Two
```

Section 22.3: Using << notation for flags

The left-shift operator (`<<`) can be used in flag enum declarations to ensure that each flag has exactly one `1` in binary representation, as flags should.

This also helps to improve readability of large enums with plenty of flags in them.

```
[Flags]
public enum MyEnum
{
    None  = 0,
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2,
    Flag4 = 1 << 3,
    Flag5 = 1 << 4,
    ...
    Flag31 = 1 << 30
}
```

It is obvious now that `MyEnum` contains proper flags only and not any messy stuff like `Flag30 = 1073741822` (or 11111111111111111111111111110 in binary) which is inappropriate.

Section 22.4: Test flags-style enum values with bitwise logic

A flags-style enum value needs to be tested with bitwise logic because it may not match any single value.

```
[Flags]
enum FlagsEnum
{
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,
    Option2And3 = Option2 | Option3;

    Default = Option1 | Option3,
}
```

The `Default` value is actually a combination of two others *merged* with a bitwise OR. Therefore to test for the presence of a flag we need to use a bitwise AND.

```
var value = FlagsEnum.Default;
```

```
bool isOption2And3Set = (value & FlagsEnum.Option2And3) == FlagsEnum.Option2And3;

Assert.True(isOption2And3Set);
```

Section 22.5: Add and remove values from flagged enum

This code is to add and remove a value from a flagged enum-instance:

```
[Flags]
public enum MyEnum
{
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2
}

var value = MyEnum.Flag1;

// set additional value
value |= MyEnum.Flag2;  //value is now Flag1, Flag2
value |= MyEnum.Flag3;  //value is now Flag1, Flag2, Flag3

// remove flag
value &= ~MyEnum.Flag2; //value is now Flag1, Flag3
```

Section 22.6: Enum to string and back

```
public enum DayOfWeek
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}


// Enum to string
string thursday = DayOfWeek.Thursday.ToString(); // "Thursday"

string seventhDay = Enum.GetName(typeof(DayOfWeek), 6); // "Saturday"

string monday = Enum.GetName(typeof(DayOfWeek), DayOfWeek.Monday); // "Monday"


// String to enum (.NET 4.0+ only - see below for alternative syntax for earlier .NET versions)
DayOfWeek tuesday;
Enum.TryParse("Tuesday", out tuesday); // DayOfWeek.Tuesday

DayOfWeek sunday;
bool matchFound1 = Enum.TryParse("SUNDAY", out sunday); // Returns false (case-sensitive match)

DayOfWeek wednesday;
bool matchFound2 = Enum.TryParse("WEDNESDAY", true, out wednesday); // Returns true;
DayOfWeek.Wednesday (case-insensitive match)


// String to enum (all .NET versions)
```

```
DayOfWeek friday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Friday"); // DayOfWeek.Friday

DayOfWeek caturday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Caturady"); // Thows
ArgumentException

// All names of an enum type as strings
string[] weekdays = Enum.GetNames(typeof(DayOfWeek));
```

Section 22.7: Enums can have unexpected values

Since an enum can be cast to and from its underlying integral type, the value may fall outside the range of values given in the definition of the enum type.

Although the below enum type `DaysOfWeek` only has 7 defined values, it can still hold any `int` value.

```
public enum DaysOfWeek
{
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
    Sunday = 7
}

DaysOfWeek d = (DaysOfWeek)31;
Console.WriteLine(d); // prints 31

DaysOFWeek s = DaysOfWeek.Sunday;
s++; // No error
```

There is currently no way to define an enum which does not have this behavior.

However, undefined enum values can be detected by using the method `Enum.IsDefined`. For example,

```
DaysOfWeek d = (DaysOfWeek)31;
Console.WriteLine(Enum.IsDefined(typeof(DaysOfWeek),d)); // prints False
```

Section 22.8: Default value for enum == ZERO

The default value for an enum is zero. If an enum does not define an item with a value of zero, its default value will be zero.

```
public class Program
{
    enum EnumExample
    {
        one = 1,
        two = 2
    }

    public void Main()
    {
        var e = default(EnumExample);

        if (e == EnumExample.one)
            Console.WriteLine("defaults to one");
```

```
        else
            Console.WriteLine("Unknown");
    }
}
```

Example: https://dotnetfiddle.net/l5Rwie

Section 22.9: Adding additional description information to an enum value

In some cases you might want to add an additional description to an enum value, for instance when the enum value itself is less readable than what you might want to display to the user. In such cases you can use the `System.ComponentModel.DescriptionAttribute` class.

For example:

```
public enum PossibleResults
{
    [Description("Success")]
    OK = 1,
    [Description("File not found")]
    FileNotFound = 2,
    [Description("Access denied")]
    AccessDenied = 3
}
```

Now, if you would like to return the description of a specific enum value you can do the following:

```
public static string GetDescriptionAttribute(PossibleResults result)
{
        return
((DescriptionAttribute)Attribute.GetCustomAttribute((result.GetType().GetField(result.ToString())),
typeof(DescriptionAttribute))).Description;
}

static void Main(string[] args)
{
    PossibleResults result = PossibleResults.FileNotFound;
    Console.WriteLine(result); // Prints "FileNotFound"
    Console.WriteLine(GetDescriptionAttribute(result)); // Prints "File not found"
}
```

This can also be easily transformed to an extension method for all enums:

```
static class EnumExtensions
{
    public static string GetDescription(this Enum enumValue)
    {
        return
((DescriptionAttribute)Attribute.GetCustomAttribute((enumValue.GetType().GetField(enumValue.ToStrin
g())), typeof(DescriptionAttribute))).Description;
    }
}
```

And then easily used like this: `Console.WriteLine(result.GetDescription());`

Section 22.10: Get all the members values of an enum

```
enum MyEnum
{
    One,
    Two,
    Three
}

foreach(MyEnum e in Enum.GetValues(typeof(MyEnum)))
    Console.WriteLine(e);
```

This will print:

```
One
Two
Three
```

Section 22.11: Bitwise Manipulation using enums

The FlagsAttribute should be used whenever the enumerable represents a collection of flags, rather than a single value. The numeric value assigned to each enum value helps when manipulating enums using bitwise operators.

Example 1 : With [Flags]

```
[Flags]
enum Colors
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}

var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

> prints Red,Blue

Example 2 : Without [Flags]

```
enum Colors
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}
var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

> prints 3

Chapter 23: Tuples

Section 23.1: Accessing tuple elements

To access tuple elements use `Item1-Item8` properties. Only the properties with index number less or equal to tuple size are going to be available (i.e. one cannot access `Item3` property in `Tuple<T1,T2>`).

```
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
var item1 = tuple.Item1; // "foo"
var item2 = tuple.Item2; // 123
var item3 = tuple.Item3; // true
var item4 = tuple.Item4; // new My Class()
```

Section 23.2: Creating tuples

Tuples are created using generic types `Tuple<T1>-Tuple<T1,T2,T3,T4,T5,T6,T7,T8>`. Each of the types represents a tuple containing 1 to 8 elements. Elements can be of different types.

```
// tuple with 4 elements
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
```

Tuples can also be created using static `Tuple.Create` methods. In this case, the types of the elements are inferred by the C# Compiler.

```
// tuple with 4 elements
var tuple = Tuple.Create("foo", 123, true, new MyClass());
```

Version ≥ 7.0

Since C# 7.0, Tuples can be easily created using ValueTuple.

```
var tuple = ("foo", 123, true, new MyClass());
```

Elements can be named for easier decomposition.

```
(int number, bool flag, MyClass instance) tuple = (123, true, new MyClass());
```

Section 23.3: Comparing and sorting Tuples

Tuples can be compared based on their elements.

As an example, an enumerable whose elements are of type `Tuple` can be sorted based on comparisons operators defined on a specified element:

```
List<Tuple<int, string>> list = new List<Tuple<int, string>>();
list.Add(new Tuple<int, string>(2, "foo"));
list.Add(new Tuple<int, string>(1, "bar"));
list.Add(new Tuple<int, string>(3, "qux"));

list.Sort((a, b) => a.Item2.CompareTo(b.Item2)); //sort based on the string element

foreach (var element in list) {
    Console.WriteLine(element);
}

// Output:
```

```
// (1, bar)
// (2, foo)
// (3, qux)
```

Or to reverse the sort use:

```
list.Sort((a, b) => b.Item2.CompareTo(a.Item2));
```

Section 23.4: Return multiple values from a method

Tuples can be used to return multiple values from a method without using out parameters. In the following example `AddMultiply` is used to return two values (sum, product).

```
void Write()
{
    var result = AddMultiply(25, 28);
    Console.WriteLine(result.Item1);
    Console.WriteLine(result.Item2);
}

Tuple<int, int> AddMultiply(int a, int b)
{
    return new Tuple<int, int>(a + b, a * b);
}
```

Output:

```
53
700
```

Now C# 7.0 offers an alternative way to return multiple values from methods using value tuples More info about `ValueTuple` struct.

Chapter 24: Guid

GUID (or UUID) is an acronym for 'Globally Unique Identifier' (or 'Universally Unique Identifier'). It is a 128-bit integer number used to identify resources.

Section 24.1: Getting the string representation of a Guid

A string representation of a Guid can be obtained by using the built in `ToString` method

```
string myGuidString = myGuid.ToString();
```

Depending on your needs you can also format the Guid, by adding a format type argument to the `ToString` call.

```
var guid = new Guid("7febf16f-651b-43b0-a5e3-0da8da49e90d");

// None           "7febf16f651b43b0a5e30da8da49e90d"
Console.WriteLine(guid.ToString("N"));

// Hyphens        "7febf16f-651b-43b0-a5e3-0da8da49e90d"
Console.WriteLine(guid.ToString("D"));

// Braces         "{7febf16f-651b-43b0-a5e3-0da8da49e90d}"
Console.WriteLine(guid.ToString("B"));

// Parentheses    "(7febf16f-651b-43b0-a5e3-0da8da49e90d)"
Console.WriteLine(guid.ToString("P"));

// Hex            "{0x7febf16f,0x651b,0x43b0{0xa5,0xe3,0x0d,0xa8,0xda,0x49,0xe9,0x0d}}"
Console.WriteLine(guid.ToString("X"));
```

Section 24.2: Creating a Guid

These are the most common ways to create an instance of Guid:

- Creating an empty guid (`00000000-0000-0000-0000-000000000000`):

```
Guid g = Guid.Empty;
Guid g2 = new Guid();
```

- Creating a new (pseudorandom) Guid:

```
Guid g = Guid.NewGuid();
```

- Creating Guids with a specific value:

```
Guid g = new Guid("0b214de7-8958-4956-8eed-28f9ba2c47c6");
Guid g2 = new Guid("0b214de7895849568eed28f9ba2c47c6");
Guid g3 = Guid.Parse("0b214de7-8958-4956-8eed-28f9ba2c47c6");
```

Section 24.3: Declaring a nullable GUID

Like other value types, GUID also has a nullable type which can take null value.

Declaration :

```
Guid? myGuidVar = null;
```

This is particularly useful when retrieving data from the data base when there is a possibility that value from a table is NULL.

Chapter 25: BigInteger

Section 25.1: Calculate the First 1,000-Digit Fibonacci Number

Include `using System.Numerics` and add a reference to `System.Numerics` to the project.

```
using System;
using System.Numerics;

namespace Euler_25
{
    class Program
    {
        static void Main(string[] args)
        {
            BigInteger l1 = 1;
            BigInteger l2 = 1;
            BigInteger current = l1 + l2;
            while (current.ToString().Length < 1000)
            {
                l2 = l1;
                l1 = current;
                current = l1 + l2;
            }
            Console.WriteLine(current);
        }
    }
}
```

This simple algorithm iterates through Fibonacci numbers until it reaches one at least 1000 decimal digits in length, then prints it out. This value is significantly larger than even a `ulong` could hold.

Theoretically, the only limit on the `BigInteger` class is the amount of RAM your application can consume.

Note: `BigInteger` is only available in .NET 4.0 and higher.

Chapter 26: Collection Initializers

Section 26.1: Collection initializers

Initialize a collection type with values:

```
var stringList = new List<string>
{
    "foo",
    "bar",
};
```

Collection initializers are syntactic sugar for `Add()` calls. Above code is equivalent to:

```
var temp = new List<string>();
temp.Add("foo");
temp.Add("bar");
var stringList = temp;
```

Note that the intialization is done atomically using a temporary variable, to avoid race conditions.

For types that offer multiple parameters in their `Add()` method, enclose the comma-separated arguments in curly braces:

```
var numberDictionary = new Dictionary<int, string>
{
    { 1, "One" },
    { 2, "Two" },
};
```

This is equivalent to:

```
var temp = new Dictionary<int, string>();
temp.Add(1, "One");
temp.Add(2, "Two");
var numberDictionarynumberDictionary = temp;
```

Section 26.2: C# 6 Index Initializers

Starting with C# 6, collections with indexers can be initialized by specifying the index to assign in square brackets, followed by an equals sign, followed by the value to assign.

Dictionary Initialization

An example of this syntax using a Dictionary:

```
var dict = new Dictionary<string, int>
{
    ["key1"] = 1,
    ["key2"] = 50
};
```

This is equivalent to:

```
var dict = new Dictionary<string, int>();
```

```
dict["key1"] = 1;
dict["key2"] = 50
```

The collection initializer syntax to do this before C# 6 was:

```
var dict = new Dictionary<string, int>
{
    { "key1", 1 },
    { "key2", 50 }
};
```

Which would correspond to:

```
var dict = new Dictionary<string, int>();
dict.Add("key1", 1);
dict.Add("key2", 50);
```

So there is a significant difference in functionality, as the new syntax uses the *indexer* of the initialized object to assign values instead of using its `Add()` method. This means the new syntax only requires a publicly available indexer, and works for any object that has one.

```
public class IndexableClass
{
    public int this[int index]
    {
        set
        {
            Console.WriteLine("{0} was assigned to index {1}", value, index);
        }
    }
}

var foo = new IndexableClass
{
    [0] = 10,
    [1] = 20
}
```

This would output:

```
10 was assigned to index 0
20 was assigned to index 1
```

Section 26.3: Collection initializers in custom classes

To make a class support collection initializers, it must implement `IEnumerable` interface and have at least one `Add` method. Since C# 6, any collection implementing `IEnumerable` can be extended with custom `Add` methods using extension methods.

```
class Program
{
    static void Main()
    {
        var col = new MyCollection {
            "foo",
            { "bar", 3 },
```

```
            "baz",
            123.45d,
        };
    }
}

class MyCollection : IEnumerable
{
    private IList list = new ArrayList();

    public void Add(string item)
    {
        list.Add(item)
    }

    public void Add(string item, int count)
    {
        for(int i=0;i< count;i++) {
            list.Add(item);
        }
    }

    public IEnumerator GetEnumerator()
    {
        return list.GetEnumerator();
    }
}

static class MyCollectionExtensions
{
    public static void Add(this MyCollection @this, double value) =>
        @this.Add(value.ToString());
}
```

Section 26.4: Using collection initializer inside object initializer

```
public class Tag
{
    public IList<string> Synonyms { get; set; }
}
```

`Synonyms` is a collection-type property. When the `Tag` object is created using object initializer syntax, `Synonyms` can also be initialized with collection initializer syntax:

```
Tag t = new Tag
{
    Synonyms = new List<string> {"c#", "c-sharp"}
};
```

The collection property can be readonly and still support collection initializer syntax. Consider this modified example (`Synonyms` property now has a private setter):

```
public class Tag
{
    public Tag()
    {
        Synonyms = new List<string>();
    }
```

```
    public IList<string> Synonyms { get; private set; }
}
```

A new `Tag` object can be created like this:

```
Tag t = new Tag
{
    Synonyms = {"c#", "c-sharp"}
};
```

This works because collection initializers are just syntatic sugar over calls to `Add()`. There's no new list being created here, the compiler is just generating calls to `Add()` on the exiting object.

Section 26.5: Collection Initializers with Parameter Arrays

You can mix normal parameters and parameter arrays:

```
public class LotteryTicket : IEnumerable{
    public int[] LuckyNumbers;
    public string UserName;

    public void Add(string userName, params int[] luckyNumbers){
        UserName = userName;
        Lottery = luckyNumbers;
    }
}
```

This syntax is now possible:

```
var Tickets = new List<LotteryTicket>{
    {"Mr Cool"  , 35663, 35732, 12312, 75685},
    {"Bruce"    , 26874, 66677, 24546, 36483, 46768, 24632, 24527},
    {"John Cena", 25446, 83356, 65536, 23783, 24567, 89337}
}
```

Chapter 27: An overview of C# collections

Section 27.1: HashSet<T>

This is a collection of unique items, with O(1) lookup.

```
HashSet<int> validStoryPointValues = new HashSet<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(1)
```

By way of comparison, doing a `Contains` on a List yields poorer performance:

```
List<int> validStoryPointValues = new List<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(n)
```

`HashSet.Contains` uses a hash table, so that lookups are extremely fast, regardless of the number of items in the collection.

Section 27.2: Dictionary<TKey, TValue>

Dictionary<TKey, TValue> is a map. For a given key there can be one value in the dictionary.

```
using System.Collections.Generic;

var people = new Dictionary<string, int>
{
    { "John", 30 }, {"Mary", 35}, {"Jack", 40}
};

// Reading data
Console.WriteLine(people["John"]); // 30
Console.WriteLine(people["George"]); // throws KeyNotFoundException

int age;
if (people.TryGetValue("Mary", out age))
{
    Console.WriteLine(age); // 35
}

// Adding and changing data
people["John"] = 40;    // Overwriting values this way is ok
people.Add("John", 40); // Throws ArgumentException since "John" already exists

// Iterating through contents
foreach(KeyValuePair<string, int> person in people)
{
    Console.WriteLine("Name={0}, Age={1}", person.Key, person.Value);
}

foreach(string name in people.Keys)
{
    Console.WriteLine("Name={0}", name);
}

foreach(int age in people.Values)
{
    Console.WriteLine("Age={0}", age);
}
```

Duplicate key when using collection initialization

```
var people = new Dictionary<string, int>
{
    { "John", 30 }, {"Mary", 35}, {"Jack", 40}, {"Jack", 40}
}; // throws ArgumentException since "Jack" already exists
```

Section 27.3: SortedSet<T>

```
// create an empty set
var mySet = new SortedSet<int>();

// add something
// note that we add 2 before we add 1
mySet.Add(2);
mySet.Add(1);

// enumerate through the set
foreach(var item in mySet)
{
    Console.WriteLine(item);
}

// output:
// 1
// 2
```

Section 27.4: T[ ] (Array of T)

```
// create an array with 2 elements
var myArray = new [] { "one", "two" };

// enumerate through the array
foreach(var item in myArray)
{
    Console.WriteLine(item);
}

// output:
// one
// two

// exchange the element on the first position
// note that all collections start with the index 0
myArray[0] = "something else";


// enumerate through the array again
foreach(var item in myArray)
{
    Console.WriteLine(item);
}

// output:
// something else
// two
```

Section 27.5: List<T>

`List<T>` is a list of a given type. Items can be added, inserted, removed and addressed by index.

```
using System.Collections.Generic;

var list = new List<int>() { 1, 2, 3, 4, 5 };
list.Add(6);
Console.WriteLine(list.Count); // 6
list.RemoveAt(3);
Console.WriteLine(list.Count); // 5
Console.WriteLine(list[3]);    // 5
```

`List<T>` can be thought of as an array that you can resize. Enumerating over the collection in order is quick, as is access to individual elements via their index. To access elements based on some aspect of their value, or some other key, a `Dictionary<T>` will provide faster lookup.

Section 27.6: Stack<T>

```
// Initialize a stack object of integers
var stack = new Stack<int>();

// add some data
stack.Push(3);
stack.Push(5);
stack.Push(8);

// elements are stored with "first in, last out" order.
// stack from top to bottom is: 8, 5, 3

// We can use peek to see the top element of the stack.
Console.WriteLine(stack.Peek()); // prints 8

// Pop removes the top element of the stack and returns it.
Console.WriteLine(stack.Pop()); // prints 8
Console.WriteLine(stack.Pop()); // prints 5
Console.WriteLine(stack.Pop()); // prints 3
```

Section 27.7: LinkedList<T>

```
// initialize a LinkedList of integers
LinkedList list = new LinkedList<int>();

// add some numbers to our list.
list.AddLast(3);
list.AddLast(5);
list.AddLast(8);

// the list currently is 3, 5, 8

list.AddFirst(2);
// the list now is 2, 3, 5, 8

list.RemoveFirst();
// the list is now 3, 5, 8

list.RemoveLast();
// the list is now 3, 5
```

Note that `LinkedList<T>` represents the *doubly* linked list. So, it's simply collection of nodes and each node contains an element of type `T`. Each node is linked to the preceding node and the following node.

Section 27.8: Queue

```
// Initalize a new queue of integers
var queue = new Queue<int>();

// Add some data
queue.Enqueue(6);
queue.Enqueue(4);
queue.Enqueue(9);

// Elements in a queue are stored in "first in, first out" order.
// The queue from first to last is: 6, 4, 9

// View the next element in the queue, without removing it.
Console.WriteLine(queue.Peek()); // prints 6

// Removes the first element in the queue, and returns it.
Console.WriteLine(queue.Dequeue()); // prints 6
Console.WriteLine(queue.Dequeue()); // prints 4
Console.WriteLine(queue.Dequeue()); // prints 9
```

> Thread safe heads up! Use ConcurrentQueue in multi-thread environments.

Chapter 28: Looping

Section 28.1: For Loop

A For Loop is great for doing things a certain amount of time. It's like a While Loop but the increment is included with the condition.

A For Loop is set up like this:

```
for (Initialization; Condition; Increment)
{
    // Code
}
```

> Initialization - Makes a new local variable that can only be used in the loop.
> Condition - The loop only runs when the condition is true.
> Increment - How the variable changes every time the loop runs.

An example:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

Output:

> 0
> 1
> 2
> 3
> 4

You can also leave out spaces in the For Loop, but you have to have all semicolons for it to function.

```
int input = Console.ReadLine();

for ( ; input < 10; input + 2)
{
    Console.WriteLine(input);
}
```

Output for 3:

> 3
> 5
> 7
> 9
> 11

Section 28.2: Do - While Loop

It is similar to a `while` loop, except that it tests the condition at the *end* of the loop body. The Do - While loop executes the loop once irrespective of whether the condition is true or not.

```
int[] numbers = new int[] { 6, 7, 8, 10 };

// Sum values from the array until we get a total that's greater than 10,
// or until we run out of values.
int sum = 0;
int i = 0;
do
{
    sum += numbers[i];
    i++;
} while (sum <= 10 && i < numbers.Length);

System.Console.WriteLine(sum); // 13
```

Section 28.3: Foreach Loop

foreach will iterate over any object of a class that implements `IEnumerable` (take note that `IEnumerable<T>` inherits from it). Such objects include some built-in ones, but not limit to: `List<T>`, `T[]` (arrays of any type), `Dictionary<TKey, TSource>`, as well as interfaces like `IQueryable` and `ICollection`, etc.

syntax

```
foreach(ItemType itemVariable in enumerableObject)
    statement;
```

remarks

1. The type `ItemType` does not need to match the precise type of the items, it just needs to be assignable from the type of the items
2. Instead of `ItemType`, alternatively `var` can be used which will infer the items type from the enumerableObject by inspecting the generic argument of the `IEnumerable` implementation
3. The statement can be a block, a single statement or even an empty statement (`;`)
4. If `enumerableObject` is not implementing `IEnumerable`, the code will not compile
5. During each iteration the current item is cast to `ItemType` (even if this is not specified but compiler-inferred via `var`) and if the item cannot be cast an `InvalidCastException` will be thrown.

Consider this example:

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
foreach(var name in list)
{
    Console.WriteLine("Hello " + name);
}
```

is equivalent to:

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
```

```
IEnumerator enumerator;
try
{
    enumerator = list.GetEnumerator();
    while(enumerator.MoveNext())
    {
        string name = (string)enumerator.Current;
        Console.WriteLine("Hello " + name);
    }
}
finally
{
    if (enumerator != null)
        enumerator.Dispose();
}
```

Section 28.4: Looping styles

While

The most trivial loop type. Only drawback is there is no intrinsic clue to know where you are in the loop.

```
/// loop while the condition satisfies
while(condition)
{
    /// do something
}
```

Do

Similar to `while`, but the condition is evaluated at the end of the loop instead of the beginning. This results in executing the loops at least once.

```
do
{
    /// do something
} while(condition) /// loop while the condition satisfies
```

For

Another trivial loop style. While looping an index (`i`) gets increased and you can use it. It is usually used for handling arrays.

```
for ( int i = 0; i < array.Count; i++ )
{
    var currentItem = array[i];
    /// do something with "currentItem"
}
```

Foreach

Modernized way of looping through `IEnumarable` objects. Good thing that you don't have to think about the index of the item or the item count of the list.

```
foreach ( var item in someList )
{
    /// do something with "item"
```

```
}
```

Foreach Method

While the other styles are used for selecting or updating the elements in collections, this style is usually used for *calling a method* straight away for all elements in a collection.

```
list.ForEach(item => item.DoSomething());

// or
list.ForEach(item => DoSomething(item));

// or using a method group
list.ForEach(Console.WriteLine);

// using an array
Array.ForEach(myArray, Console.WriteLine);
```

It is important to note that this method in only available on `List<T>` instances and as a static method on `Array` - it is **not** part of Linq.

Linq Parallel Foreach

Just like Linq Foreach, except this one does the job in a parallel manner. Meaning that all the items in the collection will run the given action at the same time, simultaneously.

```
collection.AsParallel().ForAll(item => item.DoSomething());

/// or
collection.AsParallel().ForAll(item => DoSomething(item));
```

Section 28.5: Nested loops

```
// Print the multiplication table up to 5s
for (int i = 1; i <= 5; i++)
{
    for (int j = 1; j <= 5; j++)
    {
        int product = i * j;
        Console.WriteLine("{0} times {1} is {2}", i, j, product);
    }
}
```

Section 28.6: continue

In addition to `break`, there is also the keyword `continue`. Instead of breaking completely the loop, it will simply skip the current iteration. It could be useful if you don't want some code to be executed if a particular value is set.

Here's a simple example:

```
for (int i = 1; i <= 10; i++)
{
    if (i < 9)
        continue;

    Console.WriteLine(i);
```

```
}
```

Will result in:

```
9
10
```

Note: `Continue` is often most useful in while or do-while loops. For-loops, with well-defined exit conditions, may not benefit as much.

Section 28.7: While loop

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

Output:

> 0
> 1
> 2
> 3
> 4

IEnumerators can be iterated with a while loop:

```
// Call a custom method that takes a count, and returns an IEnumerator for a list
// of strings with the names of theh largest city metro areas.
IEnumerator<string> largestMetroAreas = GetLargestMetroAreas(4);

while (largestMetroAreas.MoveNext())
{
    Console.WriteLine(largestMetroAreas.Current);
}
```

Sample output:

> Tokyo/Yokohama
> New York Metro
> Sao Paulo
> Seoul/Incheon

Section 28.8: break

Sometimes loop condition should be checked in the middle of the loop. The former is arguably more elegant than the latter:

```
for (;;)
{
```

```
    // precondition code that can change the value of should_end_loop expression

    if (should_end_loop)
        break;

    // do something
}
```

Alternative:

```
bool endLoop = false;
for (; !endLoop;)
{
    // precondition code that can set endLoop flag

    if (!endLoop)
    {
        // do something
    }
}
```

Note: In nested loops and/or `switch` must use more than just a simple `break`.

Chapter 29: Iterators

Section 29.1: Creating Iterators Using Yield

Iterators *produce* enumerators. In C#, enumerators are produced by defining methods, properties or indexers that contain `yield` statements.

Most methods will return control to their caller through normal `return` statements, which disposes all state local to that method. In contrast, methods that use `yield` statements allow them to return multiple values to the caller on request while *preserving* local state in-between returning those values. These returned values constitute a sequence. There are two types of `yield` statements used within iterators:

- `yield return`, which returns control to the caller but preserves state. The callee will continue execution from this line when control is passed back to it.
- `yield break`, which functions similarly to a normal `return` statement - this signifies the end of the sequence. Normal `return` statements themselves are illegal within an iterator block.

This example below demonstrates an iterator method that can be used to generate the Fibonacci sequence:

```
IEnumerable<int> Fibonacci(int count)
{
    int prev = 1;
    int curr = 1;

    for (int i = 0; i < count; i++)
    {
        yield return prev;
        int temp = prev + curr;
        prev = curr;
        curr = temp;
    }
}
```

This iterator can then be used to produce an enumerator of the Fibonacci sequence that can be consumed by a calling method. The code below demonstrates how the first ten terms within the Fibonacci sequence can be enumerated:

```
void Main()
{
    foreach (int term in Fibonacci(10))
    {
        Console.WriteLine(term);
    }
}
```

Output

```
1
1
2
3
5
8
13
21
```

```
34
55
```

Section 29.2: Simple Numeric Iterator Example

A common use-case for iterators is to perform some operation over a collection of numbers. The example below demonstrates how each element within an array of numbers can be individually printed out to the console.

This is possible because arrays implement the `IEnumerable` interface, allowing clients to obtain an iterator for the array using the `GetEnumerator()` method. This method returns an *enumerator*, which is a read-only, forward-only cursor over each number in the array.

```
int[] numbers = { 1, 2, 3, 4, 5 };

IEnumerator iterator = numbers.GetEnumerator();

while (iterator.MoveNext())
{
    Console.WriteLine(iterator.Current);
}
```

Output

```
1
2
3
4
5
```

It's also possible to achieve the same results using a `foreach` statement:

```
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

Chapter 30: IEnumerable

`IEnumerable` is the base interface for all non-generic collections like ArrayList that can be enumerated.
`IEnumerator<T>` is the base interface for all generic enumerators like List<>.

`IEnumerable` is an interface which implements the method `GetEnumerator`. The `GetEnumerator` method returns an `IEnumerator` which provides options to iterate through the collection like foreach.

Section 30.1: IEnumerable with custom Enumerator

Implementing the IEnumerable interface allows classes to be enumerated in the same way as BCL collections. This requires extending the Enumerator class which tracks the state of the enumeration.

Other than iterating over a standard collection, examples include:

- Using ranges of numbers based on a function rather than a collection of objects
- Implementing different iteration algorithms over collections, like DFS or BFS on a graph collection

```
public static void Main(string[] args) {

    foreach (var coffee in new CoffeeCollection()) {
        Console.WriteLine(coffee);
    }
}

public class CoffeeCollection : IEnumerable {
    private CoffeeEnumerator enumerator;

    public CoffeeCollection() {
        enumerator = new CoffeeEnumerator();
    }

    public IEnumerator GetEnumerator() {
        return enumerator;
    }

    public class CoffeeEnumerator : IEnumerator {
        string[] beverages = new string[3] { "espresso", "macchiato", "latte" };
        int currentIndex = -1;

        public object Current {
            get {
                return beverages[currentIndex];
            }
        }

        public bool MoveNext() {
            currentIndex++;

            if (currentIndex < beverages.Length) {
                return true;
            }

            return false;
        }

        public void Reset() {
            currentIndex = 0;
        }
```

```
    }
}
```

Section 30.2: IEnumerable<int>

In its most basic form, an object that implements IEnumerable represents a series of objects. The objects in question can be iterated using the c# `foreach` keyword.

In the example below, the object `sequenceOfNumbers` implements IEnumerable. It represents a series of integers. The `foreach` loop iterates through each in turn.

```
int AddNumbers(IEnumerable<int> sequenceOfNumbers) {
    int returnValue = 0;
    foreach(int i in sequenceOfNumbers) {
        returnValue += i;
    }
    return returnValue;
}
```

Chapter 31: Value type vs Reference type

Section 31.1: Passing by reference using ref keyword

From the documentation :

> In C#, arguments can be passed to parameters either by value or by reference. Passing by reference enables function members, methods, properties, indexers, operators, and constructors to change the value of the parameters and have that change persist in the calling environment. To pass a parameter by reference, use the `ref` or `out` keyword.

The difference between `ref` and `out` is that `out` means that the passed parameter has to be assigned before the function ends.in contrast parameters passed with `ref` can be changed or left unchanged.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);

        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);

        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a = 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a = 6;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        a = 7;
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}
```

Output :

```
Inside Main - Before Callee: a = 20
Inside Callee a : 5
Inside Main - After Callee: a = 20
Inside Main - Before CalleeRef: a = 20
Inside CalleeRef a : 6
Inside Main - After CalleeRef: a = 6
Inside Main - Before CalleeOut: a = 6
Inside CalleeOut a : 7
Inside Main - After CalleeOut: a = 7
```

Section 31.2: Changing values elsewhere

```
public static void Main(string[] args)
{
    var studentList = new List<Student>();
    studentList.Add(new Student("Scott", "Nuke"));
    studentList.Add(new Student("Vincent", "King"));
    studentList.Add(new Student("Craig", "Bertt"));

    // make a separate list to print out later
    var printingList = studentList; // this is a new list object, but holding the same student
objects inside it

    // oops, we've noticed typos in the names, so we fix those
    studentList[0].LastName = "Duke";
    studentList[1].LastName = "Kong";
    studentList[2].LastName = "Brett";

    // okay, we now print the list
    PrintPrintingList(printingList);
}

private static void PrintPrintingList(List<Student> students)
{
    foreach (Student student in students)
    {
        Console.WriteLine(string.Format("{0} {1}", student.FirstName, student.LastName));
    }
}
```

You'll notice that even though the printingList list was made before the corrections to student names after the typos, the PrintPrintingList method still prints out the corrected names:

```
Scott Duke
Vincent Kong
Craig Brett
```

This is because both lists hold a list of references to the same students. SO changing the underlying student object propogates to usages by either list.

Here's what the student class would look like.

```
public class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Student(string firstName, string lastName)
    {
```

```
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}
```

Section 31.3: ref vs out parameters

Code

```
class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);
        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a += 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a += 10;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        // can't use a+=15 since for this method 'a' is not intialized only declared in the method
declaration
        a = 25; //has to be initialized
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}
```

Output

```
Inside Main - Before Callee: a = 20
Inside Callee a : 25
Inside Main - After Callee: a = 20

Inside Main - Before CalleeRef: a = 20
Inside CalleeRef a : 30
Inside Main - After CalleeRef: a = 30
```

```
Inside Main - Before CalleeOut: a = 30
Inside CalleeOut a : 25
Inside Main - After CalleeOut: a = 25
```

Section 31.4: Assignment

```
var a = new List<int>();
var b = a;
a.Add(5);
Console.WriteLine(a.Count); // prints 1
Console.WriteLine(b.Count); // prints 1 as well
```

Assigning to a variable of a `List<int>` does not create a copy of the `List<int>`. Instead, it copies the reference to the `List<int>`. We call types that behave this way *reference types*.

Section 31.5: Difference with method parameters ref and out

There are two possible ways to pass a value type by reference: `ref` and `out`. The difference is that by passing it with `ref` the value must be initialized but not when passing it with `out`. Using `out` ensures that the variable has a value after the method call:

```
public void ByRef(ref int value)
{
    Console.WriteLine(nameof(ByRef) + value);
    value += 4;
    Console.WriteLine(nameof(ByRef) + value);
}

public void ByOut(out int value)
{
    value += 4 // CS0269: Use of unassigned out parameter `value'
    Console.WriteLine(nameof(ByOut) + value); // CS0269: Use of unassigned out parameter `value'

    value = 4;
    Console.WriteLine(nameof(ByOut) + value);
}

public void TestOut()
{
    int outValue1;
    ByOut(out outValue1); // prints 4

    int outValue2 = 10;   // does not make any sense for out
    ByOut(out outValue2); // prints 4
}

public void TestRef()
{
    int refValue1;
    ByRef(ref refValue1); // S0165  Use of unassigned local variable 'refValue'

    int refValue2 = 0;
    ByRef(ref refValue2); // prints 0 and 4

    int refValue3 = 10;
    ByRef(ref refValue3); // prints 10 and 14
}
```

The catch is that by using `out` the parameter `must` be initialized before leaving the method, therefore the following

method is possible with `ref` but not with `out`:

```
public void EmtyRef(bool condition, ref int value)
{
    if (condition)
    {
        value += 10;
    }
}

public void EmtyOut(bool condition, out int value)
{
    if (condition)
    {
        value = 10;
    }
} //CS0177: The out parameter 'value' must be assigned before control leaves the current method
```

This is because if `condition` does not hold, `value` goes unassigned.

Section 31.6: Passing by reference

If you want the Value Types vs Reference Types in methods example to work properly, use the ref keyword in your method signature for the parameter you want to pass by reference, as well as when you call the method.

```
public static void Main(string[] args)
{
    ...
    DoubleNumber(ref number); // calling code
    Console.WriteLine(number); // outputs 8
    ...
}
```

```
public void DoubleNumber(ref int number)
{
    number += number;
}
```

Making these changes would make the number update as expected, meaning the console output for number would be 8.

Chapter 32: Built-in Types

Section 32.1: Conversion of boxed value types

Boxed value types can only be unboxed into their original `Type`, even if a conversion of the two `Types` is valid, e.g.:

```
object boxedInt = (int)1; // int boxed in an object

long unboxedInt1 = (long)boxedInt; // invalid cast
```

This can be avoided by first unboxing into the original `Type`, e.g.:

```
long unboxedInt2 = (long)(int)boxedInt; // valid
```

Section 32.2: Comparisons with boxed value types

If value types are assigned to variables of type `object` they are *boxed* - the value is stored in an instance of a `System.Object`. This can lead to unintended consequences when comparing values with `==`, e.g.:

```
object left = (int)1;  // int in an object box
object right = (int)1; // int in an object box

var comparison1 = left == right;      // false
```

This can be avoided by using the overloaded `Equals` method, which will give the expected result.

```
var comparison2 = left.Equals(right); // true
```

Alternatively, the same could be done by unboxing the `left` and `right` variables so that the `int` values are compared:

```
var comparison3 = (int)left == (int)right; // true
```

Section 32.3: Immutable reference type - string

```
// assign string from a string literal
string s = "hello";

// assign string from an array of characters
char[] chars = new char[] { 'h', 'e', 'l', 'l', 'o' };
string s = new string(chars, 0, chars.Length);

// assign string from a char pointer, derived from a string
string s;
unsafe
{
    fixed (char* charPointer = "hello")
    {
        s = new string(charPointer);
    }
}
```

Section 32.4: Value type - char

```
// single character s
char c = 's';

// character s: casted from integer value
char c = (char)115;

// unicode character: single character s
char c = '\u0073';

// unicode character: smiley face
char c = '\u263a';
```

Section 32.5: Value type - short, int, long (signed 16 bit, 32 bit, 64 bit integers)

```
// assigning a signed short to its minimum value
short s = -32768;

// assigning a signed short to its maximum value
short s = 32767;

// assigning a signed int to its minimum value
int i = -2147483648;

// assigning a signed int to its maximum value
int i = 2147483647;

// assigning a signed long to its minimum value (note the long postfix)
long l = -9223372036854775808L;

// assigning a signed long to its maximum value (note the long postfix)
long l = 9223372036854775807L;
```

It is also possible to make these types nullable, meaning that additionally to the usual values, null can be assigned, too. If a variable of a nullable type is not initialized, it will be null instead of 0. Nullable types are marked by adding a question mark (?) after the type.

```
int a; //This is now 0.
int? b; //This is now null.
```

Section 32.6: Value type - ushort, uint, ulong (unsigned 16 bit, 32 bit, 64 bit integers)

```
// assigning an unsigned short to its minimum value
ushort s = 0;

// assigning an unsigned short to its maximum value
ushort s = 65535;

// assigning an unsigned int to its minimum value
uint i = 0;

// assigning an unsigned int to its maximum value
uint i = 4294967295;
```

```
// assigning an unsigned long to its minimum value (note the unsigned long postfix)
ulong l = 0UL;

// assigning an unsigned long to its maximum value (note the unsigned long postfix)
ulong l = 18446744073709551615UL;
```

It is also possible to make these types nullable, meaning that additionally to the usual values, null can be assigned, too. If a variable of a nullable type is not initialized, it will be null instead of 0. Nullable types are marked by adding a question mark (?) after the type.

```
uint a; //This is now 0.
uint? b; //This is now null.
```

Section 32.7: Value type - bool

```
// default value of boolean is false
bool b;
//default value of nullable boolean is null
bool? z;
b = true;
if(b) {
    Console.WriteLine("Boolean has true value");
}
```

The bool keyword is an alias of System.Boolean. It is used to declare variables to store the Boolean values, `true` and `false`.

Chapter 33: Aliases of built-in types

Section 33.1: Built-In Types Table

The following table shows the keywords for built-in C# types, which are aliases of predefined types in the System namespaces.

C# Type	.NET Framework Type
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

The C# type keywords and their aliases are interchangeable. For example, you can declare an integer variable by using either of the following declarations:

```
int number = 123;
System.Int32 number = 123;
```

Chapter 34: Anonymous types

Section 34.1: Anonymous vs dynamic

Anonymous types allow the creation of objects without having to explicitly define their types ahead of time, while maintaining static type checking.

```
var anon = new { Value = 1 };
Console.WriteLine(anon.Id); // compile time error
```

Conversely, `dynamic` has dynamic type checking, opting for runtime errors, instead of compile-time errors.

```
dynamic val = "foo";
Console.WriteLine(val.Id); // compiles, but throws runtime error
```

Section 34.2: Creating an anonymous type

Since anonymous types are not named, variables of those types must be implicitly typed (`var`).

```
var anon = new { Foo = 1, Bar = 2 };
// anon.Foo == 1
// anon.Bar == 2
```

If the member names are not specified, they are set to the name of the property/variable used to initialize the object.

```
int foo = 1;
int bar = 2;
var anon2 = new { foo, bar };
// anon2.foo == 1
// anon2.bar == 2
```

Note that names can only be omitted when the expression in the anonymous type declaration is a simple property access; for method calls or more complex expressions, a property name must be specified.

```
string foo = "some string";
var anon3 = new { foo.Length };
// anon3.Length == 11
var anon4 = new { foo.Length <= 10 ? "short string" : "long string" };
// compiler error - Invalid anonymous type member declarator.
var anon5 = new { Description = foo.Length <= 10 ? "short string" : "long string" };
// OK
```

Section 34.3: Anonymous type equality

Anonymous type equality is given by the `Equals` instance method. Two objects are equal if they have the same type and equal values (through `a.Prop.Equals(b.Prop)`) for every property.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 1, Bar = 2 };
var anon3 = new { Foo = 5, Bar = 10 };
var anon3 = new { Foo = 5, Bar = 10 };
var anon4 = new { Bar = 2, Foo = 1 };
// anon.Equals(anon2) == true
// anon.Equals(anon3) == false
```

```
// anon.Equals(anon4) == false (anon and anon4 have different types, see below)
```

Two anonymous types are considered the same if and only if their properties have the same name and type and appear in the same order.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 7, Bar = 1 };
var anon3 = new { Bar = 1, Foo = 3 };
var anon4 = new { Fa = 1, Bar = 2 };
// anon and anon2 have the same type
// anon and anon3 have diferent types (Bar and Foo appear in different orders)
// anon and anon4 have different types (property names are different)
```

Section 34.4: Generic methods with anonymous types

Generic methods allow the use of anonymous types through type inference.

```
void Log<T>(T obj) {
    // ...
}
Log(new { Value = 10 });
```

This means LINQ expressions can be used with anonymous types:

```
var products = new[] {
    new { Amount = 10, Id = 0 },
    new { Amount = 20, Id = 1 },
    new { Amount = 15, Id = 2 }
};
var idsByAmount = products.OrderBy(x => x.Amount).Select(x => x.Id);
// idsByAmount: 0, 2, 1
```

Section 34.5: Instantiating generic types with anonymous types

Using generic constructors would require the anonymous types to be named, which is not possible. Alternatively, generic methods may be used to allow type inference to occur.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 5, Bar = 10 };
List<T> CreateList<T>(params T[] items) {
    return new List<T>(items);
}

var list1 = CreateList(anon, anon2);
```

In the case of `List<T>`, implicitly typed arrays may be converted to a `List<T>` through the `ToList` LINQ method:

```
var list2 = new[] {anon, anon2}.ToList();
```

Section 34.6: Implicitly typed arrays

Arrays of anonymous types may be created with implicit typing.

```
var arr = new[] {
    new { Id = 0 },
```

```
    new { Id = 1 }
};
```

Chapter 35: Dynamic type

Section 35.1: Creating a dynamic object with properties

```
using System;
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
// 456

Console.WriteLine(info.DoesntExist);
// Throws RuntimeBinderException
```

Section 35.2: Creating a dynamic variable

```
dynamic foo = 123;
Console.WriteLine(foo + 234);
// 357    Console.WriteLine(foo.ToUpper())
// RuntimeBinderException, since int doesn't have a ToUpper method

foo = "123";
Console.WriteLine(foo + 234);
// 123234
Console.WriteLine(foo.ToUpper()):
// NOW A STRING
```

Section 35.3: Returning dynamic

```
using System;

public static void Main()
{
    var value = GetValue();
    Console.WriteLine(value);
    // dynamics are useful!
}

private static dynamic GetValue()
{
    return "dynamics are useful!";
}
```

Section 35.4: Handling Specific Types Unknown at Compile Time

The following output equivalent results:

```
class IfElseExample
{
    public string DebugToString(object a)
    {
        if (a is StringBuilder)
```

```
        {
            return DebugToStringInternal(a as StringBuilder);
        }
        else if (a is List<string>)
        {
            return DebugToStringInternal(a as List<string>);
        }
        else
        {
            return a.ToString();
        }
    }

    private string DebugToStringInternal(object a)
    {
        // Fall Back
        return a.ToString();
    }

    private string DebugToStringInternal(StringBuilder sb)
    {
        return $"StringBuilder - Capacity: {sb.Capacity}, MaxCapacity: {sb.MaxCapacity}, Value:
{sb.ToString()}";
    }

    private string DebugToStringInternal(List<string> list)
    {
        return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "\t" +
string.Join(Environment.NewLine + "\t", list.ToArray())}";
    }
}

class DynamicExample
{
    public string DebugToString(object a)
    {
        return DebugToStringInternal((dynamic)a);
    }

    private string DebugToStringInternal(object a)
    {
        // Fall Back
        return a.ToString();
    }

    private string DebugToStringInternal(StringBuilder sb)
    {
        return $"StringBuilder - Capacity: {sb.Capacity}, MaxCapacity: {sb.MaxCapacity}, Value:
{sb.ToString()}";
    }

    private string DebugToStringInternal(List<string> list)
    {
        return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "\t" +
string.Join(Environment.NewLine + "\t", list.ToArray())}";
    }
}
```

The advantage to the dynamic, is adding a new Type to handle just requires adding an overload of DebugToStringInternal of the new type. Also eliminates the need to manually cast it to the type as well.

Chapter 36: Type Conversion

Section 36.1: Explicit Type Conversion

```
using System;
namespace TypeConversionApplication
{
   class ExplicitConversion
   {
      static void Main(string[] args)
      {
         double d = 5673.74;
         int i;

         // cast double to int.
         i = (int)d;
         Console.WriteLine(i);
         Console.ReadKey();
      }
   }
}
```

Section 36.2: MSDN implicit operator example

```
class Digit
{
    public Digit(double d) { val = d; }
    public double val;

    // User-defined conversion from Digit to double
    public static implicit operator double(Digit d)
    {
        Console.WriteLine("Digit to double implict conversion called");
        return d.val;
    }
    //  User-defined conversion from double to Digit
    public static implicit operator Digit(double d)
    {
        Console.WriteLine("double to Digit implict conversion called");
        return new Digit(d);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Digit dig = new Digit(7);
        //This call invokes the implicit "double" operator
        double num = dig;
        //This call invokes the implicit "Digit" operator
        Digit dig2 = 12;
        Console.WriteLine("num = {0} dig2 = {1}", num, dig2.val);
        Console.ReadLine();
    }
}
```

Output:

> Digit to double implict conversion called
> double to Digit implict conversion called
> num = 7 dig2 = 12

Live Demo on .NET Fiddle

Chapter 37: Casting

Section 37.1: Checking compatibility without casting

If you need to know whether a value's type extends or implements a given type, but you don't want to actually cast it as that type, you can use the `is` operator.

```
if(value is int)
{
   Console.WriteLine(value + "is an int");
}
```

Section 37.2: Cast an object to a base type

Given the following definitions :

```
public interface IMyInterface1
{
    string GetName();
}

public interface IMyInterface2
{
    string GetName();
}

public class MyClass : IMyInterface1, IMyInterface2
{
    string IMyInterface1.GetName()
    {
        return "IMyInterface1";
    }

    string IMyInterface2.GetName()
    {
        return "IMyInterface2";
    }
}
```

Casting an object to a base type example :

```
    MyClass obj = new MyClass();

    IMyInterface1 myClass1 = (IMyInterface1)obj;
    IMyInterface2 myClass2 = (IMyInterface2)obj;

    Console.WriteLine("I am : {0}", myClass1.GetName());
    Console.WriteLine("I am : {0}", myClass2.GetName());

    // Outputs :
    // I am : IMyInterface1
    // I am : IMyInterface2
```

Section 37.3: Conversion Operators

In C#, types can define custom *Conversion Operators*, which allow values to be converted to and from other types using either explicit or implicit casts. For example, consider a class that is meant to represent a JavaScript

expression:

```
public class JsExpression
{
    private readonly string expression;
    public JsExpression(string rawExpression)
    {
        this.expression = rawExpression;
    }
    public override string ToString()
    {
        return this.expression;
    }
    public JsExpression IsEqualTo(JsExpression other)
    {
        return new JsExpression("(" + this + " == " + other + ")");
    }
}
```

If we wanted to create a JsExpression representing a comparison of two JavaScript values, we could do something like this:

```
JsExpression intExpression = new JsExpression("-1");
JsExpression doubleExpression = new JsExpression("-1.0");
Console.WriteLine(intExpression.IsEqualTo(doubleExpression)); // (-1 == -1.0)
```

But we can add some *explicit conversion operators* to `JsExpression`, to allow a simple conversion when using explicit casting.

```
public static explicit operator JsExpression(int value)
{
    return new JsExpression(value.ToString());
}
public static explicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = (JsExpression)(-1);
JsExpression doubleExpression = (JsExpression)(-1.0);
Console.WriteLine(intExpression.IsEqualTo(doubleExpression)); // (-1 == -1.0)
```

Or, we could change these operators to *implicit* to make the syntax much simpler.

```
public static implicit operator JsExpression(int value)
{
    return new JsExpression(value.ToString());
}
public static implicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = -1;
Console.WriteLine(intExpression.IsEqualTo(-1.0)); // (-1 == -1.0)
```

Section 37.4: LINQ Casting operations

Suppose you have types like the following:

```
interface IThing {  }
class Thing : IThing {  }
```

LINQ allows you to create a projection that changes the compile-time generic type of an `IEnumerable<>` via the `Enumerable.Cast<>()` and `Enumerable.OfType<>()` extension methods.

```
IEnumerable<IThing> things = new IThing[] {new Thing()};
IEnumerable<Thing> things2 = things.Cast<Thing>();
IEnumerable<Thing> things3 = things.OfType<Thing>();
```

When `things2` is evaluated, the `Cast<>()` method will try to cast all of the values in `things` into `Things`. If it encounters a value that cannot be cast, an `InvalidCastException` will be thrown.

When `things3` is evaluated, the `OfType<>()` method will do the same, except that if it encounters a value that cannot be cast, it will simply omit that value rather than throw an exception.

Due to the generic type of these methods, they cannot invoke Conversion Operators or perform numeric conversions.

```
double[] doubles = new[]{1,2,3}.Cast<double>().ToArray(); // Throws InvalidCastException
```

You can simply perform a cast inside a `.Select()` as a workaround:

```
double[] doubles = new[]{1,2,3}.Select(i => (double)i).ToArray();
```

Section 37.5: Explicit Casting

If you know that a value is of a specific type, you can explicitly cast it to that type in order to use it in a context where that type is needed.

```
object value = -1;
int number = (int) value;
Console.WriteLine(Math.Abs(number));
```

If we tried passing `value` directly to `Math.Abs()`, we would get a compile-time exception because `Math.Abs()` doesn't have an overload that takes an `object` as a parameter.

If `value` could not be cast to an `int`, then the second line in this example would throw an `InvalidCastException`

Section 37.6: Safe Explicit Casting (`as` operator)

If you aren't sure whether a value is of the type you think it is, you can safely cast it using the `as` operator. If the value is not of that type, the resulting value will be `null`.

```
object value = "-1";
int? number = value as int?;
if(number != null)
{
    Console.WriteLine(Math.Abs(number.Value));
}
```

Note that `null` values have no type, so the `as` keyword will safely yield `null` when casting any `null` value.

Section 37.7: Implicit Casting

A value will automatically be cast to the appropriate type if the compiler knows that it can always be converted to that type.

```
int number = -1;
object value = number;
Console.WriteLine(value);
```

In this example, we didn't need to use the typical explicit casting syntax because the compiler knows all `int`s can be cast to `object`s. In fact, we could avoid creating variables and pass `-1` directly as the argument of `Console.WriteLine()` that expects an `object`.

```
Console.WriteLine(-1);
```

Section 37.8: Explicit Numeric Conversions

Explicit casting operators can be used to perform conversions of numeric types, even though they don't extend or implement one another.

```
double value = -1.1;
int number = (int) value;
```

Note that in cases where the destination type has less precision than the original type, precision will be lost. For example, `-1.1` as a double value in the above example becomes `-1` as an integer value.

Also, numeric conversions rely on compile-time types, so they won't work if the numeric types have been "boxed" into objects.

```
object value = -1.1;
int number = (int) value; // throws InvalidCastException
```

Chapter 38: Nullable types

Section 38.1: Initialising a nullable

For `null` values:

```
Nullable<int> i = null;
```

Or:

```
int? i = null;
```

Or:

```
var i = (int?)null;
```

For non-null values:

```
Nullable<int> i = 0;
```

Or:

```
int? i = 0;
```

Section 38.2: Check if a Nullable has a value

```
int? i = null;

if (i != null)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

Which is the same as:

```
if (i.HasValue)
{
    Console.WriteLine("i is not null");
}
else
{
    Console.WriteLine("i is null");
}
```

Section 38.3: Get the value of a nullable type

Given following nullable `int`

```
int? i = 10;
```

In case default value is needed, you can assign one using null coalescing operator, `GetValueOrDefault` method or check if nullable int `HasValue` before assignment.

```
int j = i ?? 0;
int j = i.GetValueOrDefault(0);
int j = i.HasValue ? i.Value : 0;
```

The following usage is always *unsafe*. If `i` is null at runtime, a `System.InvalidOperationException` will be thrown. At design time, if a value is not set, you'll get a `Use of unassigned local variable 'i'` error.

```
int j = i.Value;
```

Section 38.4: Getting a default value from a nullable

The `.GetValueOrDefault()` method returns a value even if the `.HasValue` property is false (unlike the Value property, which throws an exception).

```
class Program
{
    static void Main()
    {
        int? nullableExample = null;
        int result = nullableExample.GetValueOrDefault();
        Console.WriteLine(result); // will output the default value for int - 0
        int secondResult = nullableExample.GetValueOrDefault(1);
        Console.WriteLine(secondResult) // will output our specified default - 1
        int thirdResult = nullableExample ?? 1;
        Console.WriteLine(secondResult) // same as the GetValueOrDefault but a bit shorter
    }
}
```

Output:

```
0
1
```

Section 38.5: Default value of nullable types is null

```
public class NullableTypesExample
{
    static int? _testValue;

    public static void Main()
    {
        if(_testValue == null)
            Console.WriteLine("null");
        else
            Console.WriteLine(_testValue.ToString());
    }
}
```

Output:

> null

Section 38.6: Effective usage of underlying Nullable<T> argument

Any nullable type is a **generic** type. And any nullable type is a **value** type.

There are some tricks which allow to **effectively use** the result of the Nullable.GetUnderlyingType method when creating code related to reflection/code-generation purposes:

```
public static class TypesHelper {
    public static bool IsNullable(this Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType);
    }
    public static bool IsNullable(this Type type, out Type underlyingType) {
        underlyingType = Nullable.GetUnderlyingType(type);
        return underlyingType != null;
    }
    public static Type GetNullable(Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType) ? type : NullableTypesCache.Get(type);
    }
    public static bool IsExactOrNullable(this Type type, Func<Type, bool> predicate) {
        Type underlyingType;
        if(IsNullable(type, out underlyingType))
            return IsExactOrNullable(underlyingType, predicate);
        return predicate(type);
    }
    public static bool IsExactOrNullable<T>(this Type type)
        where T : struct {
        return IsExactOrNullable(type, t => Equals(t, typeof(T)));
    }
}
```

The usage:

```
Type type = typeof(int).GetNullable();
Console.WriteLine(type.ToString());

if(type.IsNullable())
    Console.WriteLine("Type is nullable.");
Type underlyingType;
if(type.IsNullable(out underlyingType))
    Console.WriteLine("The underlying type is " + underlyingType.Name + ".");
if(type.IsExactOrNullable<int>())
    Console.WriteLine("Type is either exact or nullable Int32.");
if(!type.IsExactOrNullable(t => t.IsEnum))
    Console.WriteLine("Type is neither exact nor nullable enum.");
```

Output:

```
System.Nullable`1[System.Int32]
Type is nullable.
The underlying type is Int32.
Type is either exact or nullable Int32.
Type is neither exact nor nullable enum.
```

PS. The `NullableTypesCache` is defined as follows:

```
static class NullableTypesCache {
    readonly static ConcurrentDictionary<Type, Type> cache = new ConcurrentDictionary<Type,
Type>();
    static NullableTypesCache() {
        cache.TryAdd(typeof(byte), typeof(Nullable<byte>));
        cache.TryAdd(typeof(short), typeof(Nullable<short>));
        cache.TryAdd(typeof(int), typeof(Nullable<int>));
        cache.TryAdd(typeof(long), typeof(Nullable<long>));
        cache.TryAdd(typeof(float), typeof(Nullable<float>));
        cache.TryAdd(typeof(double), typeof(Nullable<double>));
        cache.TryAdd(typeof(decimal), typeof(Nullable<decimal>));
        cache.TryAdd(typeof(sbyte), typeof(Nullable<sbyte>));
        cache.TryAdd(typeof(ushort), typeof(Nullable<ushort>));
        cache.TryAdd(typeof(uint), typeof(Nullable<uint>));
        cache.TryAdd(typeof(ulong), typeof(Nullable<ulong>));
        //...
    }
    readonly static Type NullableBase = typeof(Nullable<>);
    internal static Type Get(Type type) {
        // Try to avoid the expensive MakeGenericType method call
        return cache.GetOrAdd(type, t => NullableBase.MakeGenericType(t));
    }
}
```

Section 38.7: Check if a generic type parameter is a nullable type

```
public bool IsTypeNullable<T>()
{
    return Nullable.GetUnderlyingType( typeof(T) )!=null;
}
```

Chapter 39: Constructors and Finalizers

Constructors are methods in a class that are invoked when an instance of that class is created. Their main responsibility is to leave the new object in a useful and consistent state.

Destructors/Finalizers are methods in a class that are invoked when an instance of that is destroyed. In C# they are rarely explicitely written/used.

Section 39.1: Static constructor

A static constructor is called the first time any member of a type is initialized, a static class member is called or a static method. The static constructor is thread safe. A static constructor is commonly used to:

- Initialize static state, that is state which is shared across different instances of the same class.
- Create a singleton

Example:

```
class Animal
{
    // * A static constructor is executed only once,
    //   when a class is first accessed.
    // * A static constructor cannot have any access modifiers
    // * A static constructor cannot have any parameters
    static Animal()
    {
        Console.WriteLine("Animal initialized");
    }

    // Instance constructor, this is executed every time the class is created
    public Animal()
    {
        Console.WriteLine("Animal created");
    }

    public static void Yawn()
    {
        Console.WriteLine("Yawn!");
    }
}

var turtle = new Animal();
var giraffe = new Animal();
```

Output:

> Animal initialized
> Animal created
> Animal created

View Demo

If the first call is to a static method, the static constructor is invoked without the instance constructor. This is OK, because the static method can't access instance state anyways.

```
Animal.Yawn();
```

This will output:

> Animal initialized
> Yawn!

See also Exceptions in static constructors and Generic Static Constructors .

Singleton example:

```
public class SessionManager
{
    public static SessionManager Instance;

    static SessionManager()
    {
        Instance = new SessionManager();
    }
}
```

Section 39.2: Singleton constructor pattern

```
public class SingletonClass
{
    public static SingletonClass Instance { get; } = new SingletonClass();

    private SingletonClass()
    {
        // Put custom constructor code here
    }
}
```

Because the constructor is private, no new instances of `SingletonClass` can be made by consuming code. The only way to access the single instance of `SingletonClass` is by using the static property `SingletonClass.Instance`.

The `Instance` property is assigned by a static constructor that the C# compiler generates. The .NET runtime guarantees that the static constructor is run at most once and is run before `Instance` is first read. Therefore, all synchronization and initialization concerns are carried out by the runtime.

Note, that if the static constructor fails the `Singleton` class becomes permanently unusable for the life of the AppDomain.

Also, the static constructor is not guaranteed to run at the time of the first access of `Instance`. Rather, it will run *at some point before that*. This makes the time at which initialization happens non-deterministic. In practical cases the JIT often calls the static constructor during *compilation* (not execution) of a method referencing `Instance`. This is a performance optimization.

See the Singleton Implementations page for other ways to implement the singleton pattern.

Section 39.3: Default Constructor

When a type is defined without a constructor:

```
public class Animal
{
}
```

then the compiler generates a default constructor equivalent to the following:

```
public class Animal
{
    public Animal() {}
}
```

The definition of any constructor for the type will suppress the default constructor generation. If the type were defined as follows:

```
public class Animal
{
    public Animal(string name) {}
}
```

then an `Animal` could only be created by calling the declared constructor.

```
// This is valid
var myAnimal = new Animal("Fluffy");
// This fails to compile
var unnamedAnimal = new Animal();
```

For the second example, the compiler will display an error message:

> 'Animal' does not contain a constructor that takes 0 arguments

If you want a class to have both a parameterless constructor and a constructor that takes a parameter, you can do it by explicitly implementing both constructors.

```
public class Animal
{

    public Animal() {} //Equivalent to a default constructor.
    public Animal(string name) {}
}
```

The compiler will not be able to generate a default constructor if the class extends another class which doesn't have a parameterless constructor. For example, if we had a class `Creature`:

```
public class Creature
{
    public Creature(Genus genus) {}
}
```

then `Animal` defined as `class Animal : Creature {}` would not compile.

Section 39.4: Forcing a static constructor to be called

While static constructors are always called before the first usage of a type it's sometimes useful to be able to force them to be called and the `RuntimeHelpers` class provide an helper for it:

```
using System.Runtime.CompilerServices;
// ...
RuntimeHelpers.RunClassConstructor(typeof(Foo).TypeHandle);
```

***Remark*:** All static initialization (fields initializers for example) will run, not only the constructor itself.

***Potential usages*:** Forcing initialization during the splash screen in an UI application or ensuring that a static constructor doesn't fail in an unit test.

Section 39.5: Calling a constructor from another constructor

```
public class Animal
{
    public string Name { get; set; }

    public Animal() : this("Dog")
    {
    }

    public Animal(string name)
    {
        Name = name;
    }
}

var dog = new Animal();      // dog.Name will be set to "Dog" by default.
var cat = new Animal("Cat"); // cat.Name is "Cat", the empty constructor is not called.
```

Section 39.6: Calling the base class constructor

A constructor of a base class is called before a constructor of a derived class is executed. For example, if `Mammal` extends `Animal`, then the code contained in the constructor of `Animal` is called first when creating an instance of a `Mammal`.

If a derived class doesn't explicitly specify which constructor of the base class should be called, the compiler assumes the parameterless constructor.

```
public class Animal
{
    public Animal() { Console.WriteLine("An unknown animal gets born."); }
    public Animal(string name) { Console.WriteLine(name + " gets born"); }
}

public class Mammal : Animal
{
    public Mammal(string name)
    {
        Console.WriteLine(name + " is a mammal.");
    }
}
```

In this case, instantiating a `Mammal` by calling `new Mammal("George the Cat")` will print

> An unknown animal gets born.
> George the Cat is a mammal.

View Demo

Calling a different constructor of the base class is done by placing `: base(args)` between the constructor's signature and its body:

```
public class Mammal : Animal
{
    public Mammal(string name) : base(name)
    {
        Console.WriteLine(name + " is a mammal.");
    }
}
```

Calling `new Mammal("George the Cat")` will now print:

> George the Cat gets born.
> George the Cat is a mammal.

View Demo

Section 39.7: Finalizers on derived classes

When an object graph is finalized, the order is the reverse of the construction. E.g. the super-type is finalized before the base-type as the following code demonstrates:

```
class TheBaseClass
{
    ~TheBaseClass()
    {
        Console.WriteLine("Base class finalized!");
    }
}

class TheDerivedClass : TheBaseClass
{
    ~TheDerivedClass()
    {
        Console.WriteLine("Derived class finalized!");
    }
}

//Don't assign to a variable
//to make the object unreachable
new TheDerivedClass();

//Just to make the example work;
//this is otherwise NOT recommended!
GC.Collect();

//Derived class finalized!
//Base class finalized!
```

Section 39.8: Exceptions in static constructors

If a static constructor throws an exception, it is never retried. The type is unusable for the lifetime of the AppDomain. Any further usages of the type will raise a `TypeInitializationException` wrapped around the original

exception.

```
public class Animal
{
    static Animal()
    {
        Console.WriteLine("Static ctor");
        throw new Exception();
    }

    public static void Yawn() {}
}

try
{
    Animal.Yawn();
}
catch (Exception e)
{
    Console.WriteLine(e.ToString());
}

try
{
    Animal.Yawn();
}
catch (Exception e)
{
    Console.WriteLine(e.ToString());
}
```

This will output:

> Static ctor
>
> System.TypeInitializationException: The type initializer for 'Animal' threw an exception. --->
> System.Exception: Exception of type 'System.Exception' was thrown.

[...]

> System.TypeInitializationException: The type initializer for 'Animal' threw an exception. --->
> System.Exception: Exception of type 'System.Exception' was thrown.

where you can see that the actual constructor is only executed once, and the exception is re-used.

Section 39.9: Constructor and Property Initialization

Shall the property value's assignment be executed *before* or *after* the class' constructor?

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
        if (TestProperty == 1)
```

```
        {
            Console.WriteLine("Shall this be executed?");
        }

        if (TestProperty == 2)
        {
            Console.WriteLine("Or shall this be executed");
        }
    }
}

var testInstance = new TestClass() { TestProperty = 1 };
```

In the example above, shall the `TestProperty` value be `1` in the class' constructor or after the class constructor?

Assigning property values in the instance creation like this:

```
var testInstance = new TestClass() {TestProperty = 1};
```

Will be executed ***after*** the constructor is run. However, initializing the property value in the class' property in C# 6.0 like this:

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
    }
}
```

will be done ***before*** the constructor is run.

Combining the two concepts above in a single example:

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
        if (TestProperty == 1)
        {
            Console.WriteLine("Shall this be executed?");
        }

        if (TestProperty == 2)
        {
            Console.WriteLine("Or shall this be executed");
        }
    }
}

static void Main(string[] args)
{
    var testInstance = new TestClass() { TestProperty = 1 };
    Console.WriteLine(testInstance.TestProperty); //resulting in 1
}
```

Final result:

```
"Or shall this be executed"
"1"
```

Explanation:

The `TestProperty` value will first be assigned as `2`, then the `TestClass` constructor will be run, resulting in printing of

```
"Or shall this be executed"
```

And then the `TestProperty` will be assigned as `1` due to `new TestClass() { TestProperty = 1 }`, making the final value for the `TestProperty` printed by `Console.WriteLine(testInstance.TestProperty)` to be

```
"1"
```

Section 39.10: Generic Static Constructors

If the type on which the static constructor is declared is generic, the static constructor will be called once for each unique combination of generic arguments.

```
class Animal<T>
{
    static Animal()
    {
        Console.WriteLine(typeof(T).FullName);
    }

    public static void Yawn() { }
}

Animal<Object>.Yawn();
Animal<String>.Yawn();
```

This will output:

> System.Object
> System.String

See also How do static constructors for generic types work ?

Section 39.11: Calling virtual methods in constructor

Unlike C++ in C# you can call a virtual method from class constructor (OK, you can also in C++ but behavior at first is surprising). For example:

```
abstract class Base
{
    protected Base()
    {
        _obj = CreateAnother();
    }
```

```
    protected virtual AnotherBase CreateAnother()
    {
        return new AnotherBase();
    }

    private readonly AnotherBase _obj;
}

sealed class Derived : Base
{
    public Derived() { }

    protected override AnotherBase CreateAnother()
    {
        return new AnotherDerived();
    }
}

var test = new Derived();
// test._obj is AnotherDerived
```

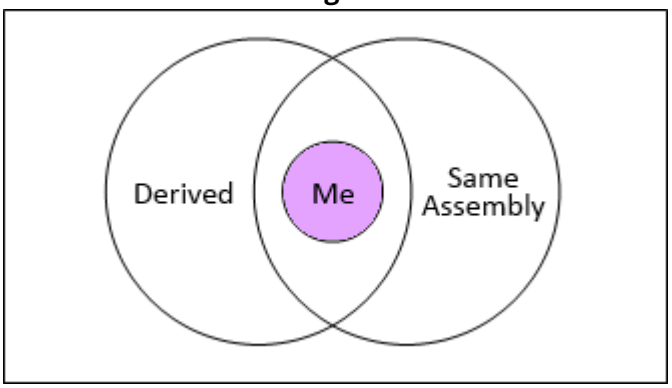
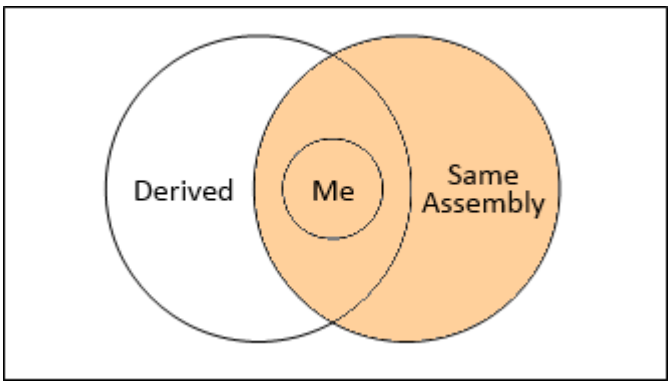
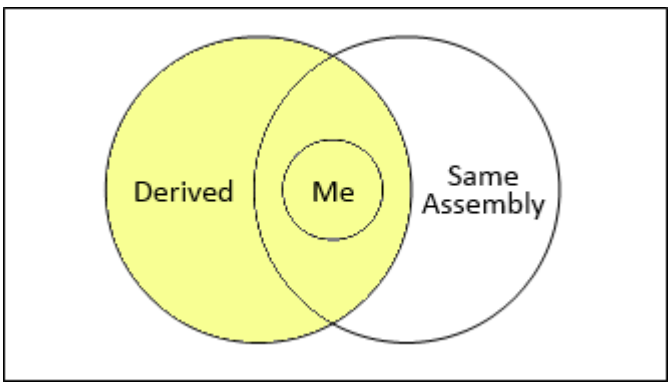
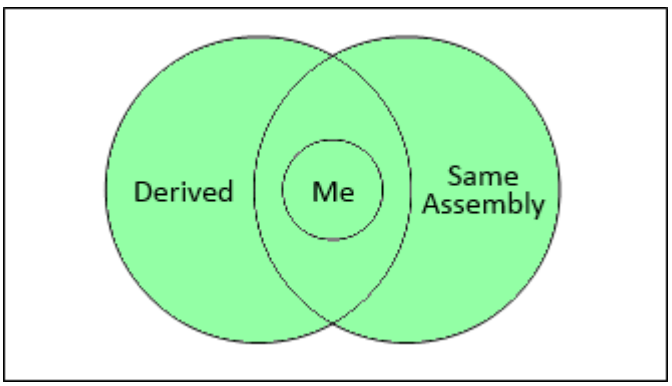
If you come from a C++ background this is surprising, base class constructor already sees derived class virtual method table!

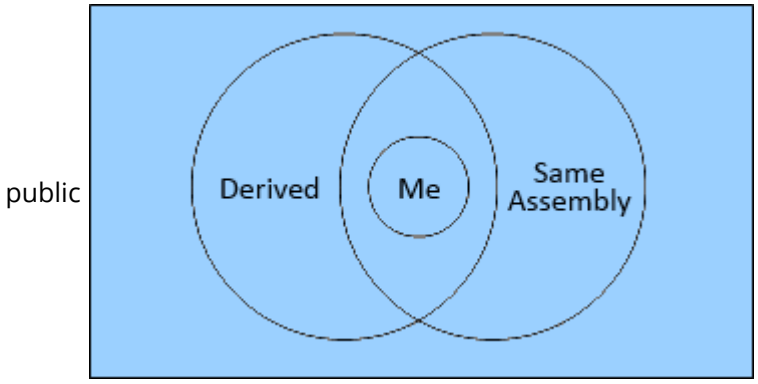
Be careful: derived class may not been fully initialized yet (its constructor will be executed after base class constructor) and this technique is dangerous (there is also a StyleCop warning for this). Usually this is regarded as bad practice.

Chapter 40: Access Modifiers

Section 40.1: Access Modifiers Diagrams

Here are all access modifiers in venn diagrams, from more limiting to more accessible:

Access Modifier	Diagram
private	
internal	
protected	
protected internal	



Below you could find more information.

Section 40.2: public

The `public` keyword makes a class (including nested classes), property, method or field available to every consumer:

```
public class Foo()
{
    public string SomeProperty { get; set; }

    public class Baz
    {
        public int Value { get; set; }
    }
}

public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();
        var someValue = foo.SomeProperty;
        var myNestedInstance = new Foo.Baz();
        var otherValue = myNestedInstance.Value;
    }
}
```

Section 40.3: private

The `private` keyword marks properties, methods, fields and nested classes for use inside the class only:

```
public class Foo()
{
    private string someProperty { get; set; }

    private class Baz
    {
        public string Value { get; set; }
    }

    public void Do()
    {
        var baz = new Baz { Value = 42 };
    }
}
```

```
public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();

        // Compile Error - not accessible due to private modifier
        var someValue = foo.someProperty;
        // Compile Error - not accessible due to private modifier
        var baz = new Foo.Baz();
    }
}
```

Section 40.4: protected internal

The `protected internal` keyword marks field, methods, properties and nested classes for use inside the same assembly or derived classes in another assembly:

> Assembly 1

```
public class Foo
{
    public string MyPublicProperty { get; set; }
    protected internal string MyProtectedInternalProperty  { get; set; }

    protected internal class MyProtectedInternalNestedClass
    {
        private string blah;
        public int N { get; set; }
    }
}

public class Bar
{
    void MyMethod1()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}
```

> Assembly 2

```
public class Baz : Foo
{
    void MyMethod1()
    {
        var myPublicProperty = MyPublicProperty;
        var myProtectedInternalProperty = MyProtectedInternalProperty;
        var thing = new MyProtectedInternalNestedClass();
    }

    void MyMethod2()
```

```
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;

        // Compile Error
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }

}

public class Qux
{
    void MyMethod1()
    {
        Baz baz = new Baz();
        var myPublicProperty = baz.MyPublicProperty;

        // Compile Error
        var myProtectedInternalProperty = baz.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Baz.MyProtectedInternalNestedClass();
    }

    void MyMethod2()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;

        //Compile Error
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}
```

Section 40.5: internal

The internal keyword makes a class (including nested classes), property, method or field available to every consumer in the same assembly:

```
internal class Foo
{
    internal string SomeProperty {get; set;}
}

internal class Bar
{
    var myInstance = new Foo();
    internal string SomeField = foo.SomeProperty;

    internal class Baz
    {
        private string blah;
        public int N { get; set; }
    }
```

```
}
```

This can be broken to allow a testing assembly to access the code via adding code to AssemblyInfo.cs file:

```
using System.Runtime.CompilerServices;

[assembly:InternalsVisibleTo("MyTests")]
```

Section 40.6: protected

The `protected` keyword marks field, methods properties and nested classes for use inside the same class and derived classes only:

```
public class Foo()
{
    protected void SomeFooMethod()
    {
        //do something
    }

    protected class Thing
    {
        private string blah;
        public int N { get; set; }
    }
}

public class Bar() : Foo
{
    private void someBarMethod()
    {
        SomeFooMethod(); // inside derived class
        var thing = new Thing(); // can use nested class
    }
}

public class Baz()
{
    private void someBazMethod()
    {
        var foo = new Foo();
        foo.SomeFooMethod(); //not accessible due to protected modifier
    }
}
```

Chapter 41: Interfaces

Section 41.1: Implementing an interface

An interface is used to enforce the presence of a method in any class that 'implements' it. The interface is defined with the keyword `interface` and a class can 'implement' it by adding `: InterfaceName` after the class name. A class can implement multiple interfaces by separating each interface with a comma.
`: InterfaceName, ISecondInterface`

```
public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : INoiseMaker
{
    public string MakeNoise()
    {
        return "Nyan";
    }
}

public class Dog : INoiseMaker
{
    public string MakeNoise()
    {
        return "Woof";
    }
}
```

Because they implement `INoiseMaker`, both `cat` and `dog` are required to include the `string MakeNoise()` method and will fail to compile without it.

Section 41.2: Explicit interface implementation

Explicit interface implementation is necessary when you implement multiple interfaces who define a common method, but different implementations are required depending on which interface is being used to call the method (note that you don't need explicit implementations if multiple interfaces share the same method and a common implementation is possible).

```
interface IChauffeur
{
    string Drive();
}

interface IGolfPlayer
{
    string Drive();
}

class GolfingChauffeur : IChauffeur, IGolfPlayer
{
    public string Drive()
    {
        return "Vroom!";
    }
```

```
    string IGolfPlayer.Drive()
    {
        return "Took a swing...";
    }
}


GolfingChauffeur obj = new GolfingChauffeur();
IChauffeur chauffeur = obj;
IGolfPlayer golfer = obj;

Console.WriteLine(obj.Drive()); // Vroom!
Console.WriteLine(chauffeur.Drive()); // Vroom!
Console.WriteLine(golfer.Drive()); // Took a swing...
```

The implementation cannot be called from anywhere else except by using the interface:

```
public class Golfer : IGolfPlayer
{
    string IGolfPlayer.Drive()
    {
        return "Swinging hard...";
    }
    public void Swing()
    {
        Drive(); // Compiler error: No such method
    }
}
```

Due to this, it may be advantageous to put complex implementation code of an explicitly implemented interface in a separate, private method.

An explicit interface implementation can of course only be used for methods that actually exist for that interface:

```
public class ProGolfer : IGolfPlayer
{
    string IGolfPlayer.Swear() // Error
    {
        return "The ball is in the pit";
    }
}
```

Similarly, using an explicit interface implementation without declaring that interface on the class causes an error, too.

Hint:

Implementing interfaces explicitly can also be used to avoid dead code. When a method is no longer needed and gets removed from the interface, the compiler will complain about each still existing implementation.

Note:

Programmers expect the contract to be the same regardless of the context of the type and explicit implementation should not expose different behavior when called. So unlike the example above, `IGolfPlayer.Drive` and `Drive` should do the same thing when possible.

Section 41.3: Interface Basics

An Interface's function known as a "contract" of functionality. It means that it declares properties and methods but it doesn't implement them.

So unlike classes Interfaces:

- Can't be instantiated
- Can't have any functionality
- Can only contain methods * *(Properties and Events are methods internally)*
- Inheriting an interface is called "Implementing"
- You can inherit from 1 class, but you can "Implement" multiple Interfaces

```
public interface ICanDoThis{
    void TheThingICanDo();
    int SomeValueProperty { get; set; }
}
```

Things to notice:

- The "I" prefix is a naming convention used for interfaces.
- The function body is replaced with a semicolon ";".
- Properties are also allowed because internally they are also methods

```
public class MyClass : ICanDoThis {
    public void TheThingICanDo(){
        // do the thing
    }

    public int SomeValueProperty { get; set; }
    public int SomeValueNotImplemtingAnything { get; set; }
}
```

.

```
ICanDoThis obj = new MyClass();

// ok
obj.TheThingICanDo();

// ok
obj.SomeValueProperty = 5;

// Error, this member doesn't exist in the interface
obj.SomeValueNotImplemtingAnything = 5;

// in order to access the property in the class you must "down cast" it
((MyClass)obj).SomeValueNotImplemtingAnything = 5; // ok
```

This is especially useful when you're working with UI frameworks such as WinForms or WPF because it's mandatory to inherit from a base class to create user control and you loose the ability to create abstraction over different control types. An example? Coming up:

```
public class MyTextBlock : TextBlock {
    public void SetText(string str){
        this.Text = str;
    }
```

```
}

public class MyButton : Button {
    public void SetText(string str){
        this.Content = str;
    }
}
```

The problem proposed is that both contain some concept of "Text" but the property names differ. And you can't create create a *abstract base class* because they have a mandatory inheritance to 2 different classes. An interface can alleviate that

```
public interface ITextControl{
    void SetText(string str);
}

public class MyTextBlock : TextBlock, ITextControl {
    public void SetText(string str){
        this.Text = str;
    }
}

public class MyButton : Button, ITextControl {
    public void SetText(string str){
        this.Content = str;
    }

    public int Clicks { get; set; }
}
```

Now MyButton and MyTextBlock is interchangeable.

```
var controls = new List<ITextControls>{
    new MyTextBlock(),
    new MyButton()
};

foreach(var ctrl in controls){
    ctrl.SetText("This text will be applied to both controls despite them being different");


    // Compiler Error, no such member in interface
    ctrl.Clicks = 0;

    // Runtime Error because 1 class is in fact not a button which makes this cast invalid
    ((MyButton)ctrl).Clicks = 0;


    /* the solution is to check the type first.
    This is usually considered bad practice since
    it's a symptom of poor abstraction */
    var button = ctrl as MyButton;
    if(button != null)
        button.Clicks = 0; // no errors


}
```

Section 41.4: IComparable<T> as an Example of Implementing an Interface

Interfaces can seem abstract until you seem them in practice. The `IComparable` and `IComparable<T>` are great examples of why interfaces can be helpful to us.

Let's say that in a program for a online store, we have a variety of items you can buy. Each item has a name, an ID number, and a price.

```
public class Item {

    public string name; // though public variables are generally bad practice,
    public int idNumber; // to keep this example simple we will use them instead
    public decimal price; // of a property.

    // body omitted for brevity

}
```

We have our `Item`s stored inside of a `List<Item>`, and in our program somewhere, we want to sort our list by ID number from smallest to largest. Instead of writing our own sorting algorithm, we can instead use the `Sort()` method that `List<T>` already has. However, as our `Item` class is right now, there is no way for the `List<T>` to understand what order to sort the list. Here is where the `IComparable` interface comes in.

To correctly implement the `CompareTo` method, `CompareTo` should return a positive number if the parameter is "less than" the current one, zero if they are equal, and a negative number if the parameter is "greater than".

```
Item apple = new Item();
apple.idNumber = 15;
Item banana = new Item();
banana.idNumber = 4;
Item cow = new Item();
cow.idNumber = 15;
Item diamond = new Item();
diamond.idNumber = 18;

Console.WriteLine(apple.CompareTo(banana)); // 11
Console.WriteLine(apple.CompareTo(cow)); // 0
Console.WriteLine(apple.CompareTo(diamond)); // -3
```

Here's the example `Item`'s implementation of the interface:

```
public class Item : IComparable<Item> {

    private string name;
    private int idNumber;
    private decimal price;

    public int CompareTo(Item otherItem) {

        return (this.idNumber - otherItem.idNumber);

    }

    // rest of code omitted for brevity

}
```

On a surface level, the `CompareTo` method in our item simply returns the difference in their ID numbers, but what does the above do in practice?

Now, when we call `Sort()` on a `List<Item>` object, the `List` will automatically call the `Item`'s `CompareTo` method when it needs to determine what order to put objects in. Furthermore, besides `List<T>`, any other objects that need the ability to compare two objects will work with the `Item` because we have defined the ability for two different `Items` to be compared with one another.

Section 41.5: Implementing multiple interfaces

```
public interface IAnimal
{
    string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : IAnimal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
    }

    public string Name { get; set; }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

Section 41.6: Why we use interfaces

An interface is a definition of a contract between the user of the interface and the class that implement it. One way to think of an interface is as a declaration that an object can perform certain functions.

Let's say that we define an interface `IShape` to represent different type of shapes, we expect a shape to have an area, so we will define a method to force the interface implementations to return their area :

```
public interface IShape
{
    double ComputeArea();
}
```

Let's that we have the following two shapes : a `Rectangle` and a `Circle`

```
public class Rectangle : IShape
{
    private double length;
    private double width;

    public Rectangle(double length, double width)
    {
        this.length = length;
```

```
        this.width = width;
    }

    public double ComputeArea()
    {
        return length * width;
    }
}

public class Circle : IShape
{
    private double radius;

    public Circle(double radius)
    {
        this.radius = radius;
    }

    public double ComputeArea()
    {
        return Math.Pow(radius, 2.0) * Math.PI;
    }
}
```

Each one of them have its own definition of its area, but both of them are shapes. So it's only logical to see them as `IShape` in our program :

```
private static void Main(string[] args)
{
    var shapes = new List<IShape>() { new Rectangle(5, 10), new Circle(5) };
    ComputeArea(shapes);

    Console.ReadKey();
}

private static void ComputeArea(IEnumerable<IShape> shapes)
{
    foreach (shape in shapes)
    {
        Console.WriteLine("Area: {0:N}, shape.ComputeArea());
    }
}

// Output:
// Area : 50.00
// Area : 78.54
```

Section 41.7: "Hiding" members with Explicit Implementation

Don't you hate it when interfaces pollute you class with too many members you don't even care about? Well I got a solution! Explicit Implementations

```
public interface IMessageService {
    void OnMessageRecieve();
    void SendMessage();
    string Result { get; set; }
    int Encoding { get; set; }
    // yadda yadda
}
```

Normally you'd implement the class like this.

```
public class MyObjectWithMessages : IMessageService {
     public void OnMessageRecieve(){

     }

     public void SendMessage(){

     }

     public string Result { get; set; }
     public int Encoding { get; set; }
}
```

Every member is public.

```
var obj = new MyObjectWithMessages();

// why would i want to call this function?
obj.OnMessageRecieve();
```

Answer: I don't. So neither should it be declared public but simply declaring the members as private will make the compiler throw an error

The solution is to use explicit implementation:

```
public class MyObjectWithMessages : IMessageService{
    void IMessageService.OnMessageRecieve() {

    }

    void IMessageService.SendMessage() {

    }

    string IMessageService.Result { get; set; }
    int IMessageService.Encoding { get; set; }
}
```

So now you have implemented the members as required and they won't expose any members in as public.

```
var obj = new MyObjectWithMessages();

/* error member does not exist on type MyObjectWithMessages.
 * We've succesfully made it "private" */
obj.OnMessageRecieve();
```

If you seriously still want to access the member even though is explicitly implement all you have to do is cast the object to the interface and you good to go.

```
((IMessageService)obj).OnMessageRecieve();
```

Chapter 42: Static Classes

Section 42.1: Static Classes

The "static" keyword when referring to a class has three effects:

1. You **cannot** create an instance of a static class (this even removes the default constructor)
2. All properties and methods in the class **must** be static as well.
3. A `static` class is a `sealed` class, meaning it cannot be inherited.

```
public static class Foo
{
    //Notice there is no constructor as this cannot be an instance
    public static int Counter { get; set; }
    public static int GetCount()
    {
        return Counter;
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Foo.Counter++;
        Console.WriteLine(Foo.GetCount()); //this will print 1

        //var foo1 = new Foo();
        //this line would break the code as the Foo class does not have a constructor
    }
}
```

Section 42.2: Static class lifetime

A `static` class is lazily initialized on member access and lives for the duration of the application domain.

```
void Main()
{
    Console.WriteLine("Static classes are lazily initialized");
    Console.WriteLine("The static constructor is only invoked when the class is first accessed");
    Foo.SayHi();

    Console.WriteLine("Reflecting on a type won't trigger its static .ctor");
    var barType = typeof(Bar);

    Console.WriteLine("However, you can manually trigger it with
System.Runtime.CompilerServices.RuntimeHelpers");
    RuntimeHelpers.RunClassConstructor(barType.TypeHandle);
}

// Define other methods and classes here
public static class Foo
{
    static Foo()
    {
        Console.WriteLine("static Foo.ctor");
    }
    public static void SayHi()
```

```
    {
        Console.WriteLine("Foo: Hi");
    }
}
public static class Bar
{
    static Bar()
    {
        Console.WriteLine("static Bar.ctor");
    }
}
```

Section 42.3: Static keyword

The static keyword means 2 things:

1. This value does not change from object to object but rather changes on a class as a whole
2. Static properties and methods don't require an instance.

```
public class Foo
{
    public Foo{
        Counter++;
        NonStaticCounter++;
    }

    public static int Counter { get; set; }
    public int NonStaticCounter { get; set; }
}

public class Program
{
    static void Main(string[] args)
    {
        //Create an instance
        var foo1 = new Foo();
        Console.WriteLine(foo1.NonStaticCounter); //this will print "1"

        //Notice this next call doesn't access the instance but calls by the class name.
        Console.WriteLine(Foo.Counter); //this will also print "1"

        //Create a second instance
        var foo2 = new Foo();

        Console.WriteLine(foo2.NonStaticCounter); //this will print "1"

        Console.WriteLine(Foo.Counter); //this will now print "2"
        //The static property incremented on both instances and can persist for the whole class

    }
}
```

Chapter 43: Singleton Implementation

Section 43.1: Statically Initialized Singleton

```
public class Singleton
{
    private readonly static Singleton instance = new Singleton();
    private Singleton() { }
    public static Singleton Instance => instance;
}
```

This implementation is thread-safe because in this case `instance` object is initialized in the static constructor. The CLR already ensures that all static constructors are executed thread-safe.

Mutating `instance` is not a thread-safe operation, therefore the `readonly` attribute guarantees immutability after initialization.

Section 43.2: Lazy, thread-safe Singleton (using Lazy<T>)

.Net 4.0 type Lazy guarantees thread-safe object initialization, so this type could be used to make Singletons.

```
public class LazySingleton
{
    private static readonly Lazy<LazySingleton> _instance =
        new Lazy<LazySingleton>(() => new LazySingleton());

    public static LazySingleton Instance
    {
        get { return _instance.Value; }
    }

    private LazySingleton() { }
}
```

Using `Lazy<T>` will make sure that the object is only instantiated when it is used somewhere in the calling code.

A simple usage will be like:

```
using System;

public class Program
{
    public static void Main()
    {
        var instance = LazySingleton.Instance;
    }
}
```

Live Demo on .NET Fiddle

Section 43.3: Lazy, thread-safe Singleton (using Double Checked Locking)

This thread-safe version of a singleton was necessary in the early versions of .NET where `static` initialization was not guaranteed to be thread-safe. In more modern versions of the framework a statically initialized singleton is

usually preferred because it is very easy to make implementation mistakes in the following pattern.

```
public sealed class ThreadSafeSingleton
{
   private static volatile ThreadSafeSingleton instance;
   private static object lockObject = new Object();

   private ThreadSafeSingleton()
   {
   }

   public static ThreadSafeSingleton Instance
   {
      get
      {
         if (instance == null)
         {
            lock (lockObject)
            {
               if (instance == null)
               {
                  instance = new ThreadSafeSingleton();
               }
            }
         }

         return instance;
      }
   }
}
```

Notice that the `if (instance == null)` check is done twice: once before the lock is acquired, and once afterwards. This implementation would still be thread-safe even without the first null check. However, that would mean that a lock would be acquired *every time* the instance is requested, and that would cause performance to suffer. The first null check is added so that the lock is not acquired unless it's necessary. The second null check makes sure that only the first thread to acquire the lock then creates the instance. The other threads will find the instance to be populated and skip ahead.

Section 43.4: Lazy, thread safe singleton (for .NET 3.5 or older, alternate implementation)

Because in .NET 3.5 and older you don't have `Lazy<T>` class you use the following pattern:

```
public class Singleton
{
    private Singleton() // prevents public instantiation
    {
    }

    public static Singleton Instance
    {
        get
        {
            return Nested.instance;
        }
    }

    private class Nested
    {
```

```
        // Explicit static constructor to tell C# compiler
        // not to mark type as beforefieldinit
        static Nested()
        {
        }

        internal static readonly Singleton instance = new Singleton();
    }
}
```

This is inspired from Jon Skeet's blog post.

Because the `Nested` class is nested and private the instantiation of the singleton instance will not be triggered by accessing other members of the `Sigleton` class (such as a public readonly property, for example).

Chapter 44: Dependency Injection

Section 44.1: Dependency Injection C# and ASP.NET with Unity

First why we should use depedency injection in our code ? We want to decouple other components from other classes in our program. For example we have class AnimalController which have code like this :

```
public class AnimalController()
{
    private SantaAndHisReindeer _SantaAndHisReindeer = new SantaAndHisReindeer();

    public AnimalController(){
            Console.WriteLine("");
    }
}
```

We look at this code and we think everything is ok but now our AnimalController is reliant on object _SantaAndHisReindeer. Automatically my Controller is bad to testing and reusability of my code will be very hard.

Very good explanation why we should use Depedency Injection and interfaces here.

If we want Unity to handle DI, the road to achieve this is very simple :) With NuGet(package manager) we can easily import unity to our code.

> in Visual Studio Tools -> NuGet Package Manager -> Manage Packages for Solution -> in search input write unity -> choose our project-> click install

Now two files with nice comments will be created.

> in App-Data folder UnityConfig.cs and UnityMvcActivator.cs

> UnityConfig - in RegisterTypes method, we can see type that will be injection in our constructors.

```
namespace Vegan.WebUi.App_Start
{

public class UnityConfig
{
    #region Unity Container
    private static Lazy<IUnityContainer> container = new Lazy<IUnityContainer>(() =>
    {
        var container = new UnityContainer();
        RegisterTypes(container);
        return container;
    });

    /// <summary>
    /// Gets the configured Unity container.
    /// </summary>
    public static IUnityContainer GetConfiguredContainer()
    {
        return container.Value;
```

```
    }
    #endregion

    /// <summary>Registers the type mappings with the Unity container.</summary>
    /// <param name="container">The unity container to configure.</param>
    /// <remarks>There is no need to register concrete types such as controllers or API controllers
(unless you want to
    /// change the defaults), as Unity allows resolving a concrete type even if it was not previously
registered.</remarks>
    public static void RegisterTypes(IUnityContainer container)
    {
        // NOTE: To load from web.config uncomment the line below. Make sure to add a
Microsoft.Practices.Unity.Configuration to the using statements.
        // container.LoadConfiguration();

        // TODO: Register your types here
        // container.RegisterType<IProductRepository, ProductRepository>();

        container.RegisterType<ISanta, SantaAndHisReindeer>();

     }
 }
}
```

> UnityMvcActivator - > also with nice comments which say that this class integrates Unity with ASP.NET MVC

```
using System.Linq;
using System.Web.Mvc;
using Microsoft.Practices.Unity.Mvc;

[assembly:
WebActivatorEx.PreApplicationStartMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator), "Start")]
[assembly:
WebActivatorEx.ApplicationShutdownMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator),
"Shutdown")]

namespace Vegan.WebUi.App_Start
{
/// <summary>Provides the bootstrapping for integrating Unity with ASP.NET MVC.</summary>
public static class UnityWebActivator
{
    /// <summary>Integrates Unity when the application starts.</summary>
    public static void Start()
    {
        var container = UnityConfig.GetConfiguredContainer();


FilterProviders.Providers.Remove(FilterProviders.Providers.OfType<FilterAttributeFilterProvider>().
First());
        FilterProviders.Providers.Add(new UnityFilterAttributeFilterProvider(container));

        DependencyResolver.SetResolver(new UnityDependencyResolver(container));

        // TODO: Uncomment if you want to use PerRequestLifetimeManager
        //
Microsoft.Web.Infrastructure.DynamicModuleHelper.DynamicModuleUtility.RegisterModule(typeof(UnityPerR
equestHttpModule));
    }
```

```
    /// <summary>Disposes the Unity container when the application is shut down.</summary>
    public static void Shutdown()
    {
        var container = UnityConfig.GetConfiguredContainer();
        container.Dispose();
    }
}
}
```

Now we can decouple our Controller from class SantAndHisReindeer :)

```
public class AnimalController()
    {
        private readonly SantaAndHisReindeer _SantaAndHisReindeer;

        public AnimalController(SantaAndHisReindeer SantaAndHisReindeer){

                _SantAndHisReindeer = SantaAndHisReindeer;
        }
    }
```

> There is one final thing we must do before running our application.

In Global.asax.cs we must add new line: UnityWebActivator.Start() which will start, configure Unity and register our types.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using Vegan.WebUi.App_Start;

namespace Vegan.WebUi
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
            UnityWebActivator.Start();
        }
    }
}
```

Section 44.2: Dependency injection using MEF

```
public interface ILogger
{
    void Log(string message);
}

[Export(typeof(ILogger))]
```

```
[ExportMetadata("Name", "Console")]
public class ConsoleLogger:ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}

[Export(typeof(ILogger))]
[ExportMetadata("Name", "File")]
public class FileLogger:ILogger
{
    public void Log(string message)
    {
        //Write the message to file
    }
}

public class User
{
    private readonly ILogger logger;
    public User(ILogger logger)
    {
        this.logger = logger;
    }
    public void LogUser(string message)
    {
        logger.Log(message)  ;
    }
}

public interface ILoggerMetaData
{
    string Name { get; }
}

internal class Program
{
    private CompositionContainer _container;

    [ImportMany]
    private IEnumerable<Lazy<ILogger, ILoggerMetaData>> _loggers;

    private static void Main()
    {
        ComposeLoggers();
        Lazy<ILogger, ILoggerMetaData> loggerNameAndLoggerMapping = _ loggers.First((n) =>
((n.Metadata.Name.ToUpper() =="Console"));
        ILogger logger= loggerNameAndLoggerMapping.Value
        var user = new User(logger);
        user.LogUser("user name");
    }

    private void ComposeLoggers()
    {
        //An aggregate catalog that combines multiple catalogs
        var catalog = new AggregateCatalog();
        string loggersDllDirectory =Path.Combine(Utilities.GetApplicationDirectory(), "Loggers");
        if (!Directory.Exists(loggersDllDirectory ))
        {
            Directory.CreateDirectory(loggersDllDirectory );
```

```
        }
        //Adds all the parts found in the same assembly as the PluginManager class
        catalog.Catalogs.Add(new AssemblyCatalog(typeof(Program).Assembly));
        catalog.Catalogs.Add(new DirectoryCatalog(loggersDllDirectory ));

        //Create the CompositionContainer with the parts in the catalog
        _container = new CompositionContainer(catalog);

        //Fill the imports of this object
        try
        {
            this._container.ComposeParts(this);
        }
        catch (CompositionException compositionException)
        {
            throw new CompositionException(compositionException.Message);
        }
    }
}
```

Chapter 45: Partial class and methods

Partial classes provides us an option to split classes into multiple parts and in multiple source files. All parts are combined into one single class during compile time. All parts should contain the keyword `partial`,should be of the same accessibility. All parts should be present in the same assembly for it to be included during compile time.

Section 45.1: Partial classes

Partial classes provide an ability to split class declaration (usually into separate files). A common problem that can be solved with partial classes is allowing users to modify auto-generated code without fearing that their changes will be overwritten if the code is regenerated. Also multiple developers can work on same class or methods.

```
using System;

namespace PartialClassAndMethods
{
    public partial class PartialClass
    {
        public void ExampleMethod() {
            Console.WriteLine("Method call from the first declaration.");
        }
    }

    public partial class PartialClass
    {
        public void AnotherExampleMethod()
        {
            Console.WriteLine("Method call from the second declaration.");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            PartialClass partial = new PartialClass();
            partial.ExampleMethod(); // outputs "Method call from the first declaration."
            partial.AnotherExampleMethod(); // outputs "Method call from the second declaration."
        }
    }
}
```

Section 45.2: Partial classes inheriting from a base class

When inheriting from any base class, only one partial class needs to have the base class specified.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {}

// PartialClass2.cs
public partial class PartialClass {}
```

You *can* specify the *same* base class in more than one partial class. It will get flagged as redundant by some IDE tools, but it does compile correctly.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {}
```

```
// PartialClass2.cs
public partial class PartialClass : BaseClass {} // base class here is redundant
```

You *cannot* specify *different* base classes in multiple partial classes, it will result in a compiler error.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {} // compiler error

// PartialClass2.cs
public partial class PartialClass : OtherBaseClass {} // compiler error
```

Section 45.3: Partial methods

Partial method consists of the definition in one partial class declaration (as a common scenario - in the auto-generated one) and the implementation in another partial class declaration.

```
using System;

namespace PartialClassAndMethods
{
    public partial class PartialClass // Auto-generated
    {
        partial void PartialMethod();
    }

    public partial class PartialClass // Human-written
    {
        public void PartialMethod()
        {
            Console.WriteLine("Partial method called.");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            PartialClass partial = new PartialClass();
            partial.PartialMethod(); // outputs "Partial method called."
        }
    }
}
```

Chapter 46: Object initializers

Section 46.1: Simple usage

Object initializers are handy when you need to create an object and set a couple of properties right away, but the available constructors are not sufficient. Say you have a class

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }

    // the rest of class definition
}
```

To initialize a new instance of the class with an initializer:

```
Book theBook = new Book { Title = "Don Quixote", Author = "Miguel de Cervantes" };
```

This is equivalent to

```
Book theBook = new Book();
theBook.Title = "Don Quixote";
theBook.Author = "Miguel de Cervantes";
```

Section 46.2: Usage with non-default constructors

You can combine object initializers with constructors to initialize types if necessary. Take for example a class defined as such:

```
public class Book {
    public string Title { get; set; }
    public string Author { get; set; }

    public Book(int id) {
        //do things
    }

    // the rest of class definition
}

var someBook = new Book(16) { Title = "Don Quixote", Author = "Miguel de Cervantes" }
```

This will first instantiate a `Book` with the `Book(int)` constructor, then set each property in the initializer. It is equivalent to:

```
var someBook = new Book(16);
someBook.Title = "Don Quixote";
someBook.Author = "Miguel de Cervantes";
```

Section 46.3: Usage with anonymous types

Object initializers are the only way to initialize anonymous types, which are types generated by the compiler.

```
var album = new { Band = "Beatles", Title = "Abbey Road" };
```

For that reason object initializers are widely used in LINQ select queries, since they provide a convenient way to specify which parts of a queried object you are interested in.

```
var albumTitles = from a in albums
                  select new
                  {
                     Title = a.Title,
                     Artist = a.Band
                  };
```

Chapter 47: Methods

Section 47.1: Calling a Method

Calling a static method:

```
// Single argument
System.Console.WriteLine("Hello World");

// Multiple arguments
string name = "User";
System.Console.WriteLine("Hello, {0}!", name);
```

Calling a static method and storing its return value:

```
string input = System.Console.ReadLine();
```

Calling an instance method:

```
int x = 42;
// The instance method called here is Int32.ToString()
string xAsString = x.ToString();
```

Calling a generic method

```
// Assuming a method 'T[] CreateArray<T>(int size)'
DateTime[] dates = CreateArray<DateTime>(8);
```

Section 47.2: Anonymous method

Anonymous methods provide a technique to pass a code block as a delegate parameter. They are methods with a body, but no name.

```
delegate int IntOp(int lhs, int rhs);
```

```
class Program
{
    static void Main(string[] args)
    {
        // C# 2.0 definition
        IntOp add = delegate(int lhs, int rhs)
        {
            return lhs + rhs;
        };

        // C# 3.0 definition
        IntOp mul = (lhs, rhs) =>
        {
            return lhs * rhs;
        };

        // C# 3.0 definition - shorthand
        IntOp sub = (lhs, rhs) => lhs - rhs;

        // Calling each method
        Console.WriteLine("2 + 3 = " + add(2, 3));
        Console.WriteLine("2 * 3 = " + mul(2, 3));
```

```
        Console.WriteLine("2 - 3 = " + sub(2, 3));
    }
}
```

Section 47.3: Declaring a Method

Every method has a unique signature consisting of a accessor (`public`, `private`, ...) ,optional modifier (`abstract`), a name and if needed method parameters. Note, that the return type is not part of the signature. A method prototype looks like the following:

```
AccessModifier OptionalModifier ReturnType MethodName(InputParameters)
{
    //Method body
}
```

`AccessModifier` can be `public`, `protected`, `pirvate` or by default `internal`.

`OptionalModifier` can be `static` `abstract` `virtual` `override` `new` or `sealed`.

`ReturnType` can be `void` for no return or can be any type from the basic ones, as `int` to complex classes.

a Method may have some or no input parameters. to set parameters for a method, you should declare each one like normal variable declarations (like `int a`), and for more than one parameter you should use comma between them (like `int a, int b`).

Parameters may have default values. for this you should set a value for the parameter (like `int a = 0`). if a parameter has a default value, setting the input value is optional.

The following method example returns the sum of two integers:

```
private int Sum(int a, int b)
{
    return a + b;
}
```

Section 47.4: Parameters and Arguments

A method can declare any number of parameters (in this example, `i`, `s` and `o` are the parameters):

```
static void DoSomething(int i, string s, object o) {
    Console.WriteLine(String.Format("i={0}, s={1}, o={2}", i, s, o));
}
```

Parameters can be used to pass values into a method, so that the method can work with them. This can be every kind of work like printing the values, or making modifications to the object referenced by a parameter, or storing the values.

When you call the method, you need to pass an actual value for every parameter. At this point, the values that you actually pass to the method call are called Arguments:

```
DoSomething(x, "hello", new object());
```

Section 47.5: Return Types

A method can return either nothing (`void`), or a value of a specified type:

```
// If you don't want to return a value, use void as return type.
static void ReturnsNothing() {
    Console.WriteLine("Returns nothing");
}

// If you want to return a value, you need to specify its type.
static string ReturnsHelloWorld() {
    return "Hello World";
}
```

If your method specifies a return value, the method *must* return a value. You do this using the `return` statement. Once a `return` statement has been reached, it returns the specified value and any code after it will not be run anymore (exceptions are `finally` blocks, which will still be executed before the method returns).

If your method returns nothing (`void`), you can still use the `return` statement without a value if you want to return from the method immediately. At the end of such a method, a `return` statement would be unnecessary though.

Examples of valid `return` statements:

```
return;
return 0;
return x * 2;
return Console.ReadLine();
```

Throwing an exception can end method execution without returning a value. Also, there are iterator blocks, where return values are generated by using the yield keyword, but those are special cases that will not be explained at this point.

Section 47.6: Default Parameters

You can use default parameters if you want to provide the option to leave out parameters:

```
static void SaySomething(string what = "ehh") {
    Console.WriteLine(what);
}

static void Main() {
    // prints "hello"
    SaySomething("hello");
    // prints "ehh"
    SaySomething(); // The compiler compiles this as if we had typed SaySomething("ehh")
}
```

When you call such a method and omit a parameter for which a default value is provided, the compiler inserts that default value for you.

Keep in mind that parameters with default values need to be written **after** parameters without default values.

```
static void SaySomething(string say, string what = "ehh") {
        //Correct
        Console.WriteLine(say + what);
    }

static void SaySomethingElse(string what = "ehh", string say) {
        //Incorrect
        Console.WriteLine(say + what);
    }
```

WARNING: Because it works that way, default values can be problematic in some cases. If you change the default value of a method parameter and don't recompile all callers of that method, those callers will still use the default value that was present when they were compiled, possibly causing inconsistencies.

Section 47.7: Method overloading

Definition : When multiple methods with the same name are declared with different parameters, it is referred to as method overloading. Method overloading typically represents functions that are identical in their purpose but are written to accept different data types as their parameters.

Factors affecting

- Number of Arguments
- Type of arguments
- Return Type**

Consider a method named `Area` that will perform calculation functions, which will accepts various arguments and return the result.

Example

```
public string Area(int value1)
{
    return String.Format("Area of Square is {0}", value1 * value1);
}
```

This method will accepts one argument and return a string, if we call the method with an integer(say `5`) the output will be `"Area of Square is 25"`.

```
public  double Area(double value1, double value2)
{
    return value1 * value2;
}
```

Similarly if we pass two double values to this method the output will be the product of the two values and are of type double. This can be used of multiplication as well as finding the Area of rectangles

```
public double Area(double value1)
{
    return 3.14 * Math.Pow(value1,2);
}
```

This can be used specially for finding the area of circle, which will accepts a double value(`radius`) and return another double value as its Area.

Each of these methods can be called normally without conflict - the compiler will examine the parameters of each method call to determine which version of `Area` needs to be used.

```
string squareArea = Area(2);
double rectangleArea = Area(32.0, 17.5);
double circleArea = Area(5.0); // all of these are valid and will compile.
```

**Note that return type *alone* cannot differentiate between two methods. For instance, if we had two definitions for Area that had the same parameters, like so:

```
public string Area(double width, double height) { ... }
public double Area(double width, double height) { ... }
// This will NOT compile.
```

If we need to have our class use the same method names that return different values, we can remove the issues of ambiguity by implementing an interface and explicitly defining its usage.

```
public interface IAreaCalculatorString {

    public string Area(double width, double height);

}

public class AreaCalculator : IAreaCalculatorString {

    public string IAreaCalculatorString.Area(double width, double height) { ... }
    // Note that the method call now explicitly says it will be used when called through
    // the IAreaCalculatorString interface, allowing us to resolve the ambiguity.
    public double Area(double width, double height) { ... }
```

Section 47.8: Access rights

```
// static: is callable on a class even when no instance of the class has been created
public static void MyMethod()

// virtual: can be called or overridden in an inherited class
public virtual  void MyMethod()

// internal: access is limited within the current assembly
internal  void MyMethod()

//private: access is limited only within the same class
private  void MyMethod()

//public: access right from every class / assembly
public void MyMethod()

//protected: access is limited to the containing class or types derived from it
protected void MyMethod()

//protected internal: access is limited to the current assembly or types derived from the containing
class.
protected internal void MyMethod()
```

Chapter 48: Extension Methods

Parameter	Details
this	The first parameter of an extension method should always be preceded by the `this` keyword, followed by the identifier with which to refer to the "current" instance of the object you are extending

Section 48.1: Extension methods - overview

Extension methods were introduced in C# 3.0. Extension methods extend and add behavior to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. *They are especially helpful when you cannot modify the source of a type you are looking to enhance.* Extension methods may be created for system types, types defined by third parties, and types that you have defined yourself. The extension method can be invoked as though it were a member method of the original type. This allows for **Method Chaining** used to implement a **Fluent Interface**.

An extension method is created by adding a **static method** to a **static class** which is distinct from the original type being extended. The static class holding the extension method is often created for the sole purpose of holding extension methods.

Extension methods take a special first parameter that designates the original type being extended. This first parameter is decorated with the keyword `this` (which constitutes a special and distinct use of `this` in C#—it should be understood as different from the use of `this` which allows referring to members of the current object instance).

In the following example, the original type being extended is the class `string`. `String` has been extended by a method `Shorten()`, which provides the additional functionality of shortening. The static class `StringExtensions` has been created to hold the extension method. The extension method `Shorten()` shows that it is an extension of `string` via the specially marked first parameter. To show that the `Shorten()` method extends `string`, the first parameter is marked with `this`. Therefore, the full signature of the first parameter is `this string text`, where `string` is the original type being extended and `text` is the chosen parameter name.

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}

class Program
{
    static void Main()
    {
        // This calls method String.ToUpper()
        var myString = "Hello World!".ToUpper();

        // This calls the extension method StringExtensions.Shorten()
        var newString = myString.Shorten(5);

        // It is worth noting that the above call is purely syntactic sugar
        // and the assignment below is functionally equivalent
        var newString2 = StringExtensions.Shorten(myString, 5);
    }
}
```

Live Demo on .NET Fiddle

The object passed as the *first argument of an extension method* (which is accompanied by the `this` keyword) is the instance the extension method is called upon.

For example, when this code is executed:

```
"some string".Shorten(5);
```

The values of the arguments are as below:

```
text: "some string"
length: 5
```

Note that extension methods are only usable if they are in the same namespace as their definition, if the namespace is imported explicitly by the code using the extension method, or if the extension class is namespace-less. The .NET framework guidelines recommend putting extension classes in their own namespace. However, this may lead to discovery issues.

This results in no conflicts between the extension methods and the libraries being used, unless namespaces which might conflict are explicitly pulled in. For example LINQ Extensions:

```
using System.Linq; // Allows use of extension methods from the System.Linq namespace

class Program
{
    static void Main()
    {
        var ints = new int[] {1, 2, 3, 4};

        // Call Where() extension method from the System.Linq namespace
        var even = ints.Where(x => x % 2 == 0);
    }
}
```

Live Demo on .NET Fiddle

Since C# 6.0, it is also possible to put a `using static` directive to the *class* containing the extension methods. For example, `using static System.Linq.Enumerable;`. This makes extension methods from that particular class available without bringing other types from the same namespace into scope.

When a class method with the same signature is available, the compiler prioritizes it over the extension method call. For example:

```
class Test
{
   public void Hello()
   {
       Console.WriteLine("From Test");
   }
}

static class TestExtensions
{
    public static void Hello(this Test test)
    {
        Console.WriteLine("From extension method");
    }
}
```

```
class Program
{
    static void Main()
    {
        Test t = new Test();
        t.Hello(); // Prints "From Test"
    }
}
```

Live demo on .NET Fiddle

Note that if there are two extension functions with the same signature, and one of them is in the same namespace, then that one will be prioritized. On the other hand, if both of them are accessed by `using`, then a compile time error will ensue with the message:

> **The call is ambiguous between the following methods or properties**

Note that the syntactic convenience of calling an extension method via `originalTypeInstance.ExtensionMethod()` is an optional convenience. The method can also be called in the traditional manner, so that the special first parameter is used as a parameter to the method.

I.e., both of the following work:

```
//Calling as though method belongs to string--it seamlessly extends string
String s = "Hello World";
s.Shorten(5);

//Calling as a traditional static method with two parameters
StringExtensions.Shorten(s, 5);
```

Section 48.2: Null checking

Extension methods are static methods which behave like instance methods. However, unlike what happens when calling an instance method on a `null` reference, when an extension method is called with a `null` reference, it does not throw a `NullReferenceException`. This can be quite useful in some scenarios.

For example, consider the following static class:

```
public static class StringExtensions
{
    public static string EmptyIfNull(this string text)
    {
        return text ?? String.Empty;
    }

    public static string NullIfEmpty(this string text)
    {
        return String.Empty == text ? null : text;
    }
}
```

```
string nullString = null;
string emptyString = nullString.EmptyIfNull();// will return ""
string anotherNullString = emptyString.NullIfEmpty(); // will return null
```

Live Demo on .NET Fiddle

Section 48.3: Explicitly using an extension method

Extension methods can also be used like ordinary static class methods. This way of calling an extension method is more verbose, but is necessary in some cases.

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}
```

Usage:

```
var newString = StringExtensions.Shorten("Hello World", 5);
```

When to call extension methods as static methods

There are still scenarios where you would need to use an extension method as a static method:

- Resolving conflict with a member method. This can happen if a new version of a library introduces a new member method with the same signature. In this case, the member method will be preferred by the compiler.
- Resolving conflicts with another extension method with the same signature. This can happen if two libraries include similar extension methods and namespaces of both classes with extension methods are used in the same file.
- Passing extension method as a method group into delegate parameter.
- Doing your own binding through `Reflection`.
- Using the extension method in the Immediate window in Visual Studio.

Using static

If a `using static` directive is used to bring static members of a static class into global scope, extension methods are skipped. Example:

```
using static OurNamespace.StringExtensions; // refers to class in previous example

// OK: extension method syntax still works.
"Hello World".Shorten(5);
// OK: static method syntax still works.
OurNamespace.StringExtensions.Shorten("Hello World", 5);
// Compile time error: extension methods can't be called as static without specifying class.
Shorten("Hello World", 5);
```

If you remove the `this` modifier from the first argument of the `Shorten` method, the last line will compile.

Section 48.4: Generic Extension Methods

Just like other methods, extension methods can use generics. For example:

```
static class Extensions
{
    public static bool HasMoreThanThreeElements<T>(this IEnumerable<T> enumerable)
    {
        return enumerable.Take(4).Count() > 3;
    }
```

```
}
```

and calling it would be like:

```
IEnumerable<int> numbers = new List<int> {1,2,3,4,5,6};
var hasMoreThanThreeElements = numbers.HasMoreThanThreeElements();
```

View Demo

Likewise for multiple Type Arguments:

```
public static TU GenericExt<T, TU>(this T obj)
{
     TU ret = default(TU);
     // do some stuff with obj
     return ret;
}
```

Calling it would be like:

```
IEnumerable<int> numbers = new List<int> {1,2,3,4,5,6};
var result = numbers.GenericExt<IEnumerable<int>,String>();
```

View Demo

You can also create extension methods for partially bound types in multi generic types:

```
class MyType<T1, T2>
{
}

static class Extensions
{
    public static void Example<T>(this MyType<int, T> test)
    {
    }
}
```

Calling it would be like:

```
MyType<int, string> t = new MyType<int, string>();
t.Example();
```

View Demo

You can also specify type constraints with `where`:

```
public static bool IsDefault<T>(this T obj) where T : struct, IEquatable<T>
{
     return EqualityComparer<T>.Default.Equals(obj, default(T));
}
```

Calling code:

```
int number = 5;
var IsDefault = number.IsDefault();
```

View Demo

Section 48.5: Extension methods can only see public (or internal) members of the extended class

```
public class SomeClass
{
    public void DoStuff()
    {

    }

    protected void DoMagic()
    {

    }
}

public static class SomeClassExtensions
{
    public static void DoStuffWrapper(this SomeClass someInstance)
    {
        someInstance.DoStuff(); // ok
    }

    public static void DoMagicWrapper(this SomeClass someInstance)
    {
        someInstance.DoMagic(); // compilation error
    }
}
```

Extension methods are just a syntactic sugar, and are not actually members of the class they extend. This means that they cannot break encapsulation—they only have access to `public` (or when implemented in the same assembly, `internal`) fields, properties and methods.

Section 48.6: Extension methods for chaining

When an extension method returns a value that has the same type as its `this` argument, it can be used to "chain" one or more method calls with a compatible signature. This can be useful for sealed and/or primitive types, and allows the creation of so-called "fluent" APIs if the method names read like natural human language.

```
void Main()
{
    int result = 5.Increment().Decrement().Increment();
    // result is now 6
}

public static class IntExtensions
{
    public static int Increment(this int number) {
        return ++number;
    }

    public static int Decrement(this int number) {
        return --number;
    }
}
```

Or like this

```
void Main()
{
    int[] ints = new[] { 1, 2, 3, 4, 5, 6};
    int[] a = ints.WhereEven();
    //a is { 2, 4, 6 };
    int[] b = ints.WhereEven().WhereGreaterThan(2);
    //b is { 4, 6 };
}

public static class IntArrayExtensions
{
    public static int[] WhereEven(this int[] array)
    {
        //Enumerable.* extension methods use a fluent approach
        return array.Where(i => (i%2) == 0).ToArray();
    }

    public static int[] WhereGreaterThan(this int[] array, int value)
    {
        return array.Where(i => i > value).ToArray();
    }
}
```

Section 48.7: Extension methods with Enumeration

Extension methods are useful for adding functionality to enumerations.

One common use is to implement a conversion method.

```
public enum YesNo
{
    Yes,
    No,
}

public static class EnumExtentions
{
    public static bool ToBool(this YesNo yn)
    {
        return yn == YesNo.Yes;
    }
    public static YesNo ToYesNo(this bool yn)
    {
        return yn ? YesNo.Yes : YesNo.No;
    }
}
```

Now you can quickly convert your enum value to a different type. In this case a bool.

```
bool yesNoBool = YesNo.Yes.ToBool(); // yesNoBool == true
YesNo yesNoEnum = false.ToYesNo();   // yesNoEnum == YesNo.No
```

Alternatively extension methods can be used to add property like methods.

```
public enum Element
{
    Hydrogen,
```

```
    Helium,
    Lithium,
    Beryllium,
    Boron,
    Carbon,
    Nitrogen,
    Oxygen
    //Etc
}

public static class ElementExtensions
{
    public static double AtomicMass(this Element element)
    {
        switch(element)
        {
            case Element.Hydrogen:  return 1.00794;
            case Element.Helium:    return 4.002602;
            case Element.Lithium:   return 6.941;
            case Element.Beryllium: return 9.012182;
            case Element.Boron:     return 10.811;
            case Element.Carbon:    return 12.0107;
            case Element.Nitrogen:  return 14.0067;
            case Element.Oxygen:    return 15.9994;
            //Etc
        }
        return double.Nan;
    }
}

var massWater = 2*Element.Hydrogen.AtomicMass() + Element.Oxygen.AtomicMass();
```

Section 48.8: Extension methods dispatch based on static type

The static (compile-time) type is used rather than the dynamic (run-time type) to match parameters.

```
public class Base
{
    public virtual string GetName()
    {
        return "Base";
    }
}

public class Derived : Base
{
    public override string GetName()
    {
        return "Derived";
    }
}

public static class Extensions
{
    public static string GetNameByExtension(this Base item)
    {
        return "Base";
    }

    public static string GetNameByExtension(this Derived item)
```

```
    {
        return "Derived";
    }
}

public static class Program
{
    public static void Main()
    {
        Derived derived = new Derived();
        Base @base = derived;

        // Use the instance method "GetName"
        Console.WriteLine(derived.GetName()); // Prints "Derived"
        Console.WriteLine(@base.GetName()); // Prints "Derived"

        // Use the static extension method "GetNameByExtension"
        Console.WriteLine(derived.GetNameByExtension()); // Prints "Derived"
        Console.WriteLine(@base.GetNameByExtension()); // Prints "Base"
    }
}
```

Live Demo on .NET Fiddle

Also the dispatch based on static type does not allow an extension method to be called on a `dynamic` object:

```
public class Person
{
    public string Name { get; set; }
}

public static class ExtenionPerson
{
    public static string GetPersonName(this Person person)
    {
        return person.Name;
    }
}

dynamic person = new Person { Name = "Jon" };
var name = person.GetPersonName(); // RuntimeBinderException is thrown
```

Section 48.9: Extension methods on Interfaces

One useful feature of extension methods is that you can create common methods for an interface. Normally an interface cannot have shared implementations, but with extension methods they can.

```
public interface IVehicle
{
    int MilesDriven { get; set; }
}

public static class Extensions
{
    public static int FeetDriven(this IVehicle vehicle)
    {
        return vehicle.MilesDriven * 5028;
    }
}
```

In this example, the method `FeetDriven` can be used on any `IVehicle`. This logic in this method would apply to all `IVehicles`, so it can be done this way so that there doesn't have to be a `FeetDriven` in the `IVehicle` definition which would be implemented the same way for all children.

Section 48.10: Extension methods in combination with interfaces

It is very convenient to use extension methods with interfaces as implementation can be stored outside of class and all it takes to add some functionality to class is to decorate class with interface.

```
public interface IInterface
{
   string Do()
}

public static class ExtensionMethods{
    public static string DoWith(this IInterface obj){
      //does something with IInterface instance
    }
}

public class Classy : IInterface
{
   // this is a wrapper method; you could also call DoWith() on a Classy instance directly,
   // provided you import the namespace containing the extension method
   public Do(){
       return this.DoWith();
   }
}
```

use like:

```
var classy = new Classy();
classy.Do(); // will call the extension
classy.DoWith(); // Classy implements IInterface so it can also be called this way
```

Section 48.11: Extension methods aren't supported by dynamic code

```
static class Program
{
    static void Main()
    {
        dynamic dynamicObject = new ExpandoObject();

        string awesomeString = "Awesome";

        // Prints True
        Console.WriteLine(awesomeString.IsThisAwesome());

        dynamicObject.StringValue = awesomeString;

        // Prints True
        Console.WriteLine(StringExtensions.IsThisAwesome(dynamicObject.StringValue));

        // No compile time error or warning, but on runtime throws RuntimeBinderException
        Console.WriteLine(dynamicObject.StringValue.IsThisAwesome());
    }
```

```
}

static class StringExtensions
{
    public static bool IsThisAwesome(this string value)
    {
        return value.Equals("Awesome");
    }
}
```

> The reason [calling extension methods from dynamic code] doesn't work is because in regular, non-dynamic code extension methods work by doing a full search of all the classes known to the compiler for a static class that has an extension method that matches. The search goes in order based on the namespace nesting and available `using` directives in each namespace.
>
> That means that in order to get a dynamic extension method invocation resolved correctly, somehow the DLR has to know *at runtime* what all the namespace nestings and `using` directives were *in your source code*. We do not have a mechanism handy for encoding all that information into the call site. We considered inventing such a mechanism, but decided that it was too high cost and produced too much schedule risk to be worth it.

Source

Section 48.12: Extensions and interfaces together enable DRY code and mixin-like functionality

Extension methods enable you to simplify your interface definitions by only including core required functionality in the interface itself and allowing you to define convenience methods and overloads as extension methods. Interfaces with fewer methods are easier to implement in new classes. Keeping overloads as extensions rather than including them in the interface directly saves you from copying boilerplate code into every implementation, helping you keep your code DRY. This in fact is similar to the mixin pattern which C# does not support.

`System.Linq.Enumerable`'s extensions to `IEnumerable<T>` is a great example of this. `IEnumerable<T>` only requires the implementing class to implement two methods: generic and non-generic `GetEnumerator()`. But `System.Linq.Enumerable` provides countless useful utilities as extensions enabling concise and clear consumption of `IEnumerable<T>`.

The following is a very simple interface with convenience overloads provided as extensions.

```
public interface ITimeFormatter
{
   string Format(TimeSpan span);
}

public static class TimeFormatter
{
    // Provide an overload to *all* implementers of ITimeFormatter.
    public static string Format(
        this ITimeFormatter formatter,
        int millisecondsSpan)
        => formatter.Format(TimeSpan.FromMilliseconds(millisecondsSpan));
}

// Implementations only need to provide one method. Very easy to
// write additional implementations.
```

```
public class SecondsTimeFormatter : ITimeFormatter
{
   public string Format(TimeSpan span)
   {
       return $"{(int)span.TotalSeconds}s";
   }
}

class Program
{
    static void Main(string[] args)
    {
        var formatter = new SecondsTimeFormatter();
        // Callers get two method overloads!
        Console.WriteLine($"4500ms is rougly {formatter.Format(4500)}");
        var span = TimeSpan.FromSeconds(5);
        Console.WriteLine($"{span} is formatted as {formatter.Format(span)}");
    }
}
```

Section 48.13: IList<T> Extension Method Example: Comparing 2 Lists

You can use the following extension method for comparing the contents of two IList< T > instances of the same type.

By default the items are compared based on their order within the list and the items themselves, passing false to the `isOrdered` parameter will compare only the items themselves regardless of their order.

For this method to work, the generic type (`T`) must override both `Equals` and `GetHashCode` methods.

Usage:

```
List<string> list1 = new List<string> {"a1", "a2", null, "a3"};
List<string> list2 = new List<string> {"a1", "a2", "a3", null};

list1.Compare(list2);//this gives false
list1.Compare(list2, false);//this gives true. they are equal when the order is disregarded
```

Method:

```
public static bool Compare<T>(this IList<T> list1, IList<T> list2, bool isOrdered = true)
{
    if (list1 == null && list2 == null)
        return true;
    if (list1 == null || list2 == null || list1.Count != list2.Count)
        return false;

    if (isOrdered)
    {
        for (int i = 0; i < list2.Count; i++)
        {
            var l1 = list1[i];
            var l2 = list2[i];
            if (
                 (l1 == null && l2 != null) ||
                 (l1 != null && l2 == null) ||
                 (!l1.Equals(l2)))
            {
```

```
                    return false;
                }
            }
            return true;
        }
        else
        {
            List<T> list2Copy = new List<T>(list2);
            //Can be done with Dictionary without O(n^2)
            for (int i = 0; i < list1.Count; i++)
            {
                if (!list2Copy.Remove(list1[i]))
                    return false;
            }
            return true;
        }
    }
```

Section 48.14: Extension methods as strongly typed wrappers

Extension methods can be used for writing strongly typed wrappers for dictionary-like objects. For example a cache, `HttpContext.Items` at cetera...

```
public static class CacheExtensions
{
    public static void SetUserInfo(this Cache cache, UserInfo data) =>
        cache["UserInfo"] = data;

    public static UserInfo GetUserInfo(this Cache cache) =>
        cache["UserInfo"] as UserInfo;
}
```

This approach removes the need of using string literals as keys all over the codebase as well as the need of casting to the required type during the read operation. Overall it creates a more secure, strongly typed way of interacting with such loosely typed objects as Dictionaries.

Section 48.15: Using Extension methods to create beautiful mapper classes

We can create a better mapper classes with extension methods, Suppose if i have some DTO classes like

```
public class UserDTO
{
        public AddressDTO Address { get; set; }
}

public class AddressDTO
{
        public string Name { get; set; }
}
```

and i need to map to corresponding view model classes

```
public class UserViewModel
{
    public AddressViewModel Address { get; set; }
}
```

```
public class AddressViewModel
{
    public string Name { get; set; }
}
```

then I can create my mapper class like below

```
public static class ViewModelMapper
{
      public static UserViewModel ToViewModel(this UserDTO user)
      {
            return user == null ?
                null :
                new UserViewModel()
                {
                    Address = user.Address.ToViewModel()
                    // Job = user.Job.ToViewModel(),
                    // Contact = user.Contact.ToViewModel() .. and so on
                };
      }

      public static AddressViewModel ToViewModel(this AddressDTO userAddr)
      {
            return userAddr == null ?
                null :
                new AddressViewModel()
                {
                    Name = userAddr.Name
                };
      }
}
```

Then finally i can invoke my mapper like below

```
    UserDTO userDTOObj = new UserDTO() {
            Address = new AddressDTO() {
                Name = "Address of the user"
            }
       };

    UserViewModel user = userDTOObj.ToViewModel(); // My DTO mapped to Viewmodel
```

The beauty here is all the mapping method have a common name (ToViewModel) and we can reuse it several ways

Section 48.16: Using Extension methods to build new collection types (e.g. DictList)

You can create extension methods to improve usability for nested collections like a `Dictionary` with a `List<T>` value.

Consider the following extension methods:

```
public static class DictListExtensions
{
    public static void Add<TKey, TValue, TCollection>(this Dictionary<TKey, TCollection> dict, TKey
key, TValue value)
            where TCollection : ICollection<TValue>, new()
    {
        TCollection list;
```

```
        if (!dict.TryGetValue(key, out list))
        {
            list = new TCollection();
            dict.Add(key, list);
        }

        list.Add(value);
    }

    public static bool Remove<TKey, TValue, TCollection>(this Dictionary<TKey, TCollection> dict,
TKey key, TValue value)
        where TCollection : ICollection<TValue>
    {
        TCollection list;
        if (!dict.TryGetValue(key, out list))
        {
            return false;
        }

        var ret = list.Remove(value);
        if (list.Count == 0)
        {
            dict.Remove(key);
        }
        return ret;
    }
}
```

you can use the extension methods as follows:

```
var dictList = new Dictionary<string, List<int>>();

dictList.Add("example", 5);
dictList.Add("example", 10);
dictList.Add("example", 15);

Console.WriteLine(String.Join(", ", dictList["example"])); // 5, 10, 15

dictList.Remove("example", 5);
dictList.Remove("example", 10);

Console.WriteLine(String.Join(", ", dictList["example"])); // 15

dictList.Remove("example", 15);

Console.WriteLine(dictList.ContainsKey("example")); // False
```

View Demo

Section 48.17: Extension methods for handling special cases

Extension methods can be used to "hide" processing of inelegant business rules that would otherwise require cluttering up a calling function with if/then statements. This is similar to and analogous to handling nulls with extension methods. For example,

```
public static class CakeExtensions
{
    public static Cake EnsureTrueCake(this Cake cake)
    {
        //If the cake is a lie, substitute a cake from grandma, whose cakes aren't as tasty but are
```

```
        known never to be lies. If the cake isn't a lie, don't do anything and return it.
        return CakeVerificationService.IsCakeLie(cake) ? GrandmasKitchen.Get1950sCake() : cake;
    }
}
```

```
Cake myCake = Bakery.GetNextCake().EnsureTrueCake();
myMouth.Eat(myCake);//Eat the cake, confident that it is not a lie.
```

Section 48.18: Using Extension methods with Static methods and Callbacks

Consider using Extension Methods as Functions which wrap other code, here's a great example that uses both a static method and and extension method to wrap the Try Catch construct. Make your code Bullet Proof...

```
using System;
using System.Diagnostics;

namespace Samples
{
    /// <summary>
    /// Wraps a try catch statement as a static helper which uses
    /// Extension methods for the exception
    /// </summary>
    public static class Bullet
    {
        /// <summary>
        /// Wrapper for Try Catch Statement
        /// </summary>
        /// <param name="code">Call back for code</param>
        /// <param name="error">Already handled and logged exception</param>
        public static void Proof(Action code, Action<Exception> error)
        {
            try
            {
                code();
            }
            catch (Exception iox)
            {
                //extension method used here
                iox.Log("BP2200-ERR-Unexpected Error");
                //callback, exception already handled and logged
                error(iox);
            }
        }
        /// <summary>
        /// Example of a logging method helper, this is the extension method
        /// </summary>
        /// <param name="error">The Exception to log</param>
        /// <param name="messageID">A unique error ID header</param>
        public static void Log(this Exception error, string messageID)
        {
            Trace.WriteLine(messageID);
            Trace.WriteLine(error.Message);
            Trace.WriteLine(error.StackTrace);
            Trace.WriteLine("");
        }
    }
    /// <summary>
    /// Shows how to use both the wrapper and extension methods.
    /// </summary>
    public class UseBulletProofing
```

```
    {
        public UseBulletProofing()
        {
            var ok = false;
            var result = DoSomething();
            if (!result.Contains("ERR"))
            {
                ok = true;
                DoSomethingElse();
            }
        }

        /// <summary>
        /// How to use Bullet Proofing in your code.
        /// </summary>
        /// <returns>A string</returns>
        public string DoSomething()
        {
            string result = string.Empty;
            //Note that the Bullet.Proof method forces this construct.
            Bullet.Proof(() =>
            {
                //this is the code callback
                result = "DST5900-INF-No Exceptions in this code";
            }, error =>
            {
                //error is the already logged and handled exception
                //determine the base result
                result = "DTS6200-ERR-An exception happened look at console log";
                if (error.Message.Contains("SomeMarker"))
                {
                    //filter the result for Something within the exception message
                    result = "DST6500-ERR-Some marker was found in the exception";
                }
            });
            return result;
        }

        /// <summary>
        /// Next step in workflow
        /// </summary>
        public void DoSomethingElse()
        {
            //Only called if no exception was thrown before
        }
    }
}
```

Chapter 49: Named Arguments

Section 49.1: Argument order is not necessary

You can place named arguments in any order you want.

Sample Method:

```
public static string Sample(string left, string right)
{
     return string.Join("-",left,right);
}
```

Call Sample:

```
Console.WriteLine (Sample(left:"A",right:"B"));
Console.WriteLine (Sample(right:"A",left:"B"));
```

Results:

```
A-B
B-A
```

Section 49.2: Named arguments and optional parameters

You can combine named arguments with optional parameters.

Let see this method:

```
public sealed class SmsUtil
{
    public static bool SendMessage(string from, string to, string message, int retryCount = 5,
object attachment = null)
    {
         // Some code
    }
}
```

When you want to call this method *without* set `retryCount` argument :

```
var result = SmsUtil.SendMessage(
                        from       : "Cihan",
                        to         : "Yakar",
                        message    : "Hello there!",
                        attachment : new object());
```

Section 49.3: Named Arguments can make your code more clear

Consider this simple class:

```
class SmsUtil
{
    public bool SendMessage(string from, string to, string message, int retryCount, object
attachment)
```

```
    {
        // Some code
    }
}
```

Before C# 3.0 it was:

```
var result = SmsUtil.SendMessage("Mehran", "Maryam", "Hello there!", 12, null);
```

you can make this method call even more clear with **named arguments**:

```
var result = SmsUtil.SendMessage(
    from: "Mehran",
    to:  "Maryam",
    message "Hello there!",
    retryCount: 12,
    attachment: null);
```

Chapter 50: Named and Optional Arguments

Section 50.1: Optional Arguments

Consider preceding is our function definition with optional arguments.

```
private static double FindAreaWithOptional(int length, int width=56)
       {
           try
           {
               return (length * width);
           }
           catch (Exception)
           {
               throw new NotImplementedException();
           }
       }
```

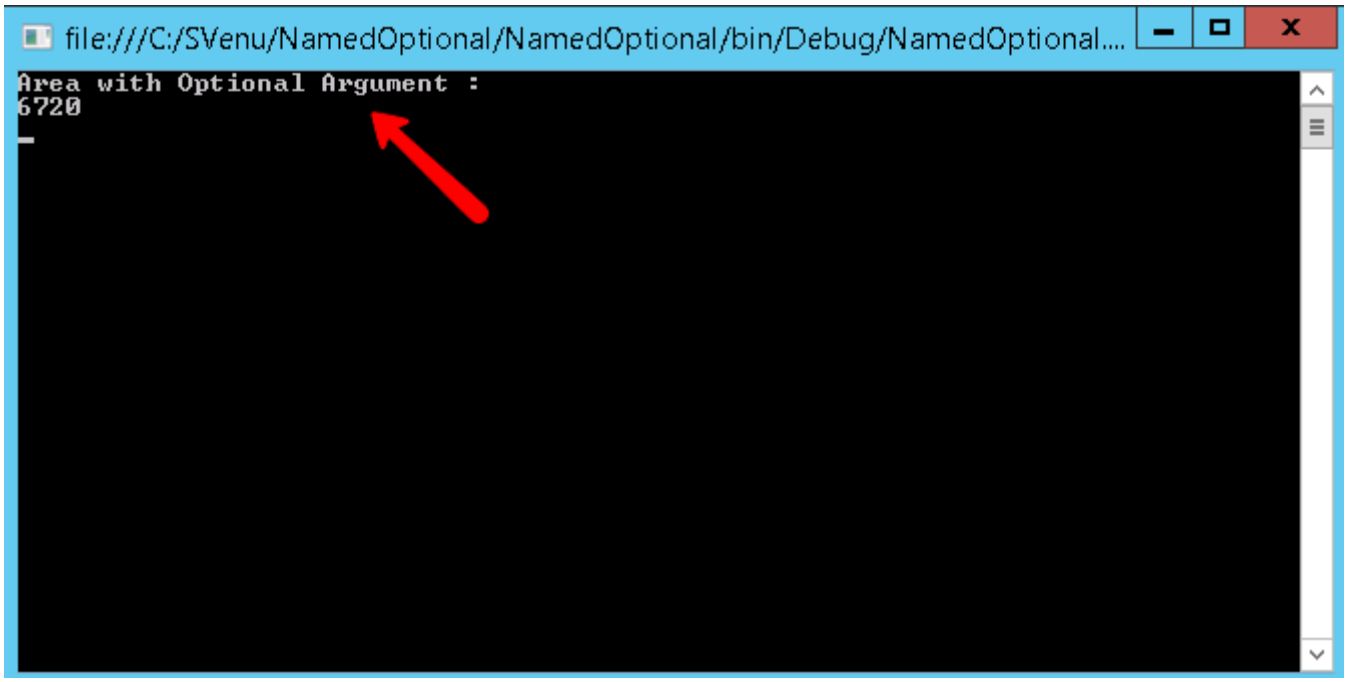
Here we have set the value for width as optional and gave value as 56. If you note, the IntelliSense itself shows you the optional argument as shown in the below image.

```
area=FindAreaWithOptional(
```

double Program.FindAreaWithOptional(**int length**, [int width = 56])

```
Console.WriteLine("Area with Optional Argument : ");
area = FindAreaWithOptional(120);
Console.WriteLine(area);
Console.Read();
```

Note that we did not get any error while compiling and it will give you an output as follows.



Using Optional Attribute.

Another way of implementing the optional argument is by using the `[Optional]` keyword. If you do not pass the

value for the optional argument, the default value of that datatype is assigned to that argument. The `Optional` keyword is present in “Runtime.InteropServices” namespace.

```
using System.Runtime.InteropServices;
private static double FindAreaWithOptional(int length, [Optional]int width)
   {
       try
       {
           return (length * width);
       }
       catch (Exception)
       {
           throw new NotImplementedException();
       }
   }

area = FindAreaWithOptional(120);  //area=0
```

And when we call the function, we get 0 because the second argument is not passed and the default value of int is 0 and so the product is 0.

Section 50.2: Named Arguments

Consider following is our function call.

```
FindArea(120, 56);
```

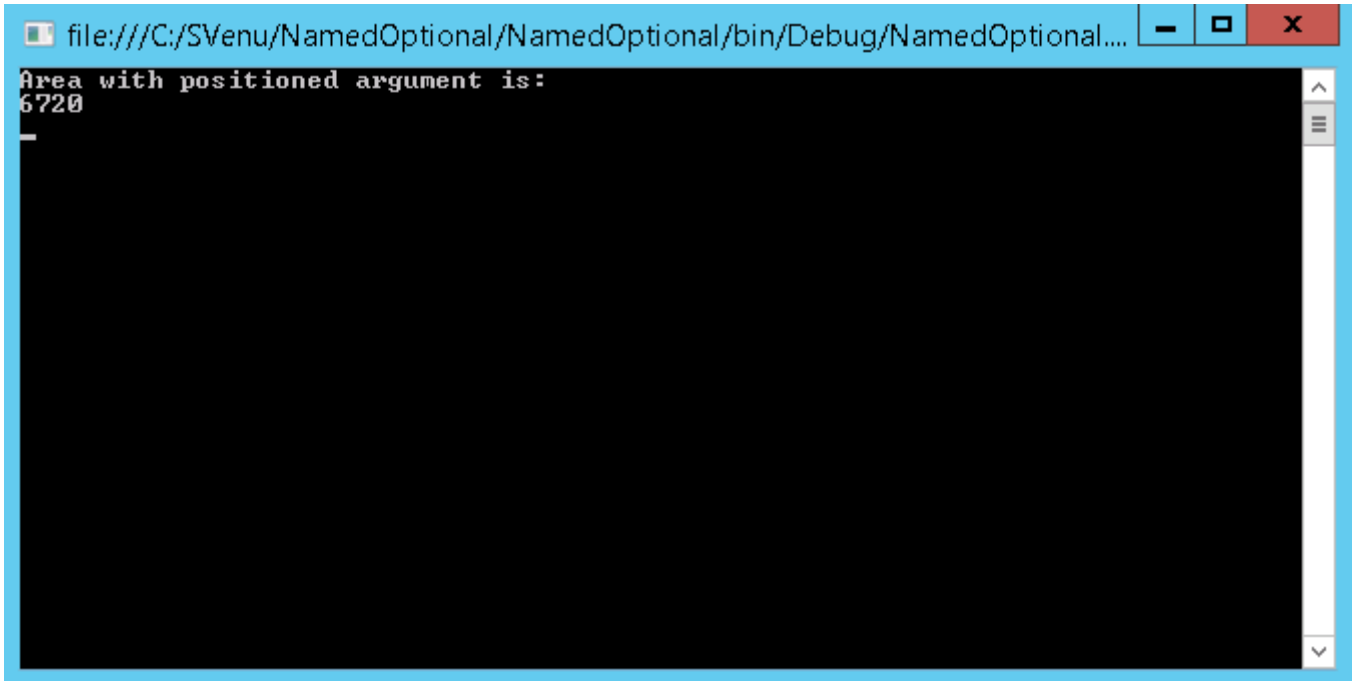
In this our first argument is length (ie 120) and second argument is width (ie 56). And we are calculating the area by that function. And following is the function definition.

```
private static double FindArea(int length, int width)
       {
           try
           {
               return (length* width);
           }
           catch (Exception)
           {
               throw new NotImplementedException();
           }
       }
```

So in the first function call, we just passed the arguments by its position. Right?

```
double area;
Console.WriteLine("Area with positioned argument is: ");
area = FindArea(120, 56);
Console.WriteLine(area);
Console.Read();
```

If you run this, you will get an output as follows.



Now here it comes the features of a named arguments. Please see the preceding function call.

```
Console.WriteLine("Area with Named argument is: ");
area = FindArea(length: 120, width: 56);
Console.WriteLine(area);
Console.Read();
```

Here we are giving the named arguments in the method call.

```
area = FindArea(length: 120, width: 56);
```

Now if you run this program, you will get the same result. We can give the names vice versa in the method call if we are using the named arguments.

```
Console.WriteLine("Area with Named argument vice versa is: ");
area = FindArea(width: 120, length: 56);
Console.WriteLine(area);
Console.Read();
```

One of the important use of a named argument is, when you use this in your program it improves the readability of your code. It simply says what your argument is meant to be, or what it is?.

You can give the positional arguments too. That means, a combination of both positional argument and named argument.

```
Console.WriteLine("Area with Named argument Positional Argument : ");
            area = FindArea(120, width: 56);
            Console.WriteLine(area);
            Console.Read();
```

In the above example we passed 120 as the length and 56 as a named argument for the parameter width.

There are some limitations too. We will discuss the limitation of a named arguments now.

Limitation of using a Named Argument

Named argument specification must appear after all fixed arguments have been specified.

If you use a named argument before a fixed argument you will get a compile time error as follows.

Named argument specification must appear after all fixed arguments have been specified

Chapter 51: Data Annotation

Section 51.1: Data Annotation Basics

Data annotations are a way of adding more contextual information to classes or members of a class. There are three main categories of annotations:

- Validation Attributes: add validation criteria to data
- Display Attributes: specify how the data should be displayed to the user
- Modelling Attributes: add information on usage and relationship with other classes

Usage

Here is an example where two `ValidationAttribute` and one `DisplayAttribute` are used:

```
class Kid
{
    [Range(0, 18)] // The age cannot be over 18 and cannot be negative
    public int Age { get; set; }
    [StringLength(MaximumLength = 50, MinimumLength = 3)] // The name cannot be under 3 chars or
more than 50 chars
    public string Name { get; set; }
    [DataType(DataType.Date)] // The birthday will be displayed as a date only (without the time)
    public DateTime Birthday { get; set; }
}
```

Data annotations are mostly used in frameworks such as ASP.NET. For example, in `ASP.NET MVC`, when a model is received by a controller method, `ModelState.IsValid()` can be used to tell if the received model respects all its `ValidationAttribute`. `DisplayAttribute` is also used in `ASP.NET MVC` to determine how to display values on a web page.

Section 51.2: Creating a custom validation attribute

Custom validation attributes can be created by deriving from the `ValidationAttribute` base class, then overriding `virtual` methods as needed.

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited = false)]
public class NotABananaAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        var inputValue = value as string;
        var isValid = true;

        if (!string.IsNullOrEmpty(inputValue))
        {
            isValid = inputValue.ToUpperInvariant() != "BANANA";
        }

        return isValid;
    }
}
```

This attribute can then be used like this:

```
public class Model
```

```
{
    [NotABanana(ErrorMessage = "Bananas are not allowed.")]
    public string FavoriteFruit { get; set; }
}
```

Section 51.3: Manually Execute Validation Attributes

Most of the times, validation attributes are use inside frameworks (such as ASP.NET). Those frameworks take care of executing the validation attributes. But what if you want to execute validation attributes manually? Just use the `Validator` class (no reflection needed).

Validation Context

Any validation needs a context to give some information about what is being validated. This can include various information such as the object to be validated, some properties, the name to display in the error message, etc.

```
ValidationContext vc = new ValidationContext(objectToValidate); // The simplest form of validation context. It contains only a reference to the object being validated.
```

Once the context is created, there are multiple ways of doing validation.

Validate an Object and All of its Properties

```
ICollection<ValidationResult> results = new List<ValidationResult>(); // Will contain the results of the validation
bool isValid = Validator.TryValidateObject(objectToValidate, vc, results, true); // Validates the object and its properties using the previously created context.
// The variable isValid will be true if everything is valid
// The results variable contains the results of the validation
```

Validate a Property of an Object

```
ICollection<ValidationResult> results = new List<ValidationResult>(); // Will contain the results of the validation
bool isValid = Validator.TryValidatePropery(objectToValidate.PropertyToValidate, vc, results, true); // Validates the property using the previously created context.
// The variable isValid will be true if everything is valid
// The results variable contains the results of the validation
```

And More

To learn more about manual validation see:

- ValidationContext Class Documentation
- Validator Class Documentation

Section 51.4: Validation Attributes

Validation attributes are used to enforce various validation rules in a declarative fashion on classes or class members. All validation attributes derive from the ValidationAttribute base class.

Example: RequiredAttribute

When validated through the `ValidationAttribute.Validate` method, this attribute will return an error if the `Name` property is null or contains only whitespace.

```
public class ContactModel
{
    [Required(ErrorMessage = "Please provide a name.")]
    public string Name { get; set; }
}
```

Example: StringLengthAttribute

The `StringLengthAttribute` validates if a string is less than the maximum length of a string. It can optionally specify a minimum length. Both values are inclusive.

```
public class ContactModel
{
    [StringLength(20, MinimumLength = 5, ErrorMessage = "A name must be between five and twenty
characters.")]
    public string Name { get; set; }
}
```

Example: RangeAttribute

The `RangeAttribute` gives the maximum and minimum value for a numeric field.

```
public class Model
{
    [Range(0.01, 100.00,ErrorMessage = "Price must be between 0.01 and 100.00")]
    public decimal Price { get; set; }
}
```

Example: CustomValidationAttribute

The `CustomValidationAttribute` class allows a custom `static` method to be invoked for validation. The custom method must be `static ValidationResult [MethodName] (object input)`.

```
public class Model
{
    [CustomValidation(typeof(MyCustomValidation), "IsNotAnApple")]
    public string FavoriteFruit { get; set; }
}
```

Method declaration:

```
public static class MyCustomValidation
{
    public static ValidationResult IsNotAnApple(object input)
    {
        var result = ValidationResult.Success;

        if (input?.ToString()?.ToUpperInvariant() == "APPLE")
        {
            result = new ValidationResult("Apples are not allowed.");
        }

        return result;
    }
}
```

Section 51.5: EditableAttribute (data modeling attribute)

`EditableAttribute` sets whether users should be able to change the value of the class property.

```
public class Employee
{
    [Editable(false)]
    public string FirstName { get; set; }
}
```

Simple usage example in XAML application

```
<Window x:Class="WpfApplication.MainWindow"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       xmlns:wpfApplication="clr-namespace:WpfApplication"
       Height="70" Width="360" Title="Display name example">

    <Window.Resources>
        <wpfApplication:EditableConverter x:Key="EditableConverter"/>
    </Window.Resources>

    <StackPanel Margin="5">
        <!-- TextBox Text (FirstName property value) -->
        <!-- TextBox IsEnabled (Editable attribute) -->
        <TextBox Text="{Binding Employee.FirstName, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}"
                 IsEnabled="{Binding Employee, Converter={StaticResource EditableConverter},
ConverterParameter=FirstName}"/>
    </StackPanel>

</Window>
```

```
namespace WpfApplication
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private Employee _employee = new Employee() { FirstName = "This is not editable"};

        public MainWindow()
        {
            InitializeComponent();
            DataContext = this;
        }

        public Employee Employee
        {
            get { return _employee; }
            set { _employee = value; }
        }
    }
}
```

```
namespace WpfApplication
{
    public class EditableConverter : IValueConverter
```

```
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
        {
            // return editable attribute's value for given instance property,
            // defaults to true if not found
            var attribute = value.GetType()
                .GetProperty(parameter.ToString())
                .GetCustomAttributes(false)
                .OfType<EditableAttribute>()
                .FirstOrDefault();

            return attribute != null ? attribute.AllowEdit : true;
        }

        public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            throw new NotImplementedException();
        }
    }
}
```

Chapter 52: Keywords

Keywords are predefined, reserved identifiers with special meaning to the compiler. They cannot be used as identifiers in your program without the `@` prefix. For example `@if` is a legal identifier but not the keyword `if`.

Section 52.1: as

The `as` keyword is an operator similar to a *cast*. If a cast is not possible, using `as` produces `null` rather than resulting in an `InvalidCastException`.

`expression as type` is equivalent to `expression is type ? (type)expression : (type)null` with the caveat that `as` is only valid on reference conversions, nullable conversions, and boxing conversions. User-defined conversions are *not* supported; a regular cast must be used instead.

For the expansion above, the compiler generates code such that `expression` will only be evaluated once and use single dynamic type check (unlike the two in the sample above).

`as` can be useful when expecting an argument to facilitate several types. Specifically it grants the user multiple options - rather than checking every possibility with `is` before casting, or just casting and catching exceptions. It is best practice to use 'as' when casting/checking an object which will cause only one unboxing penalty. Using `is` to check, then casting will cause two unboxing penalties.

If an argument is expected to be an instance of a specific type, a regular cast is preferred as its purpose is more clear to the reader.

Because a call to `as` may produce `null`, always check the result to avoid a `NullReferenceException`.

Example usage

```
object something = "Hello";
Console.WriteLine(something as string);        //Hello
Console.Writeline(something as Nullable<int>); //null
Console.WriteLine(something as int?);          //null

//This does NOT compile:
//destination type must be a reference type (or a nullable value type)
Console.WriteLine(something as int);
```

Live Demo on .NET Fiddle

Equivalent example without using as:

```
Console.WriteLine(something is string ? (string)something : (string)null);
```

This is useful when overriding the `Equals` function in custom classes.

```
class MyCustomClass
{

    public override bool Equals(object obj)
    {
        MyCustomClass customObject = obj as MyCustomClass;

        // if it is null it may be really null
        // or it may be of a different type
```

```
        if (Object.ReferenceEquals(null, customObject))
        {
            // If it is null then it is not equal to this instance.
            return false;
        }

        // Other equality controls specific to class
    }

}
```

Section 52.2: goto

`goto` can be used to jump to a specific line inside the code, specified by a label.

`goto` as a:

Label:

```
void InfiniteHello()
{
    sayHello:
    Console.WriteLine("Hello!");
    goto sayHello;
}
```

Live Demo on .NET Fiddle

Case statement:

```
enum Permissions { Read, Write };

switch (GetRequestedPermission())
{
    case Permissions.Read:
        GrantReadAccess();
        break;

    case Permissions.Write:
        GrantWriteAccess();
        goto case Permissions.Read; //People with write access also get read
}
```

Live Demo on .NET Fiddle

This is particularly useful in executing multiple behaviors in a switch statement, as C# does not support fall-through case blocks.

Exception Retry

```
var exCount = 0;
retry:
try
{
    //Do work
}
catch (IOException)
{
    exCount++;
    if (exCount < 3)
    {
        Thread.Sleep(100);
```

```
            goto retry;
        }
        throw;
    }
```

Live Demo on .NET Fiddle

Similar to many languages, use of goto keyword is discouraged except the cases below.

Valid usages of `goto` which apply to C#:

- Fall-through case in switch statement.
- Multi-level break. LINQ can often be used instead, but it usually has worse performance.
- Resource deallocation when working with unwrapped low-level objects. In C#, low-level objects should usually be wrapped in separate classes.
- Finite state machines, for example, parsers; used internally by compiler generated async/await state machines.

Section 52.3: volatile

Adding the `volatile` keyword to a field indicates to the compiler that the field's value may be changed by multiple separate threads. The primary purpose of the `volatile` keyword is to prevent compiler optimizations that assume only single-threaded access. Using `volatile` ensures that the value of the field is the most recent value that is available, and the value is not subject to the caching that non-volatile values are.

It is good practice to mark *every variable* that may be used by multiple threads as `volatile` to prevent unexpected behavior due to behind-the-scenes optimizations. Consider the following code block:

```
public class Example
{
    public int x;

    public void DoStuff()
    {
        x = 5;

        // the compiler will optimize this to y = 15
        var y = x + 10;

        /* the value of x will always be the current value, but y will always be "15" */
        Debug.WriteLine("x = " + x + ", y = " + y);
    }
}
```

In the above code-block, the compiler reads the statements `x = 5` and `y = x + 10` and determines that the value of `y` will always end up as 15. Thus, it will optimize the last statement as `y = 15`. However, the variable `x` is in fact a `public` field and the value of `x` may be modified at runtime through a different thread acting on this field separately. Now consider this modified code-block. Do note that the field `x` is now declared as `volatile`.

```
public class Example
{
    public volatile int x;

    public void DoStuff()
```

```
    {
        x = 5;

        // the compiler no longer optimizes this statement
        var y = x + 10;

        /* the value of x and y will always be the correct values */
        Debug.WriteLine("x = " + x + ", y = " + y);
    }
}
```

Now, the compiler looks for *read* usages of the field `x` and ensures that the current value of the field is always retrieved. This ensures that even if multiple threads are reading and writing to this field, the current value of `x` is always retrieved.

`volatile` can only be used on fields within `class`es or `struct`s. The following is *not* valid:

public void MyMethod() { volatile int x; }

`volatile` can only be applied to fields of following types:

- reference types or generic type parameters known to be reference types
- primitive types such as `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, and `bool`
- enums types based on `byte`, `sbyte`, `short`, `ushort`, `int` or `uint`
- `IntPtr` and `UIntPtr`

Remarks:

- The `volatile` modifier is usually used for a field that is accessed by multiple threads without using the lock statement to serialize access.
- The `volatile` keyword can be applied to fields of reference types
- The `volatile` keyword will not make operating on 64-bit primitives on a 32-bit platform atomic. Interlocked operations such as `Interlocked.Read` and `Interlocked.Exchange` must still be used for safe multi-threaded access on these platforms.

Section 52.4: checked, unchecked

The `checked` and `unchecked` keywords define how operations handle mathematical overflow. "Overflow" in the context of the `checked` and `unchecked` keywords is when an integer arithmetic operation results in a value which is greater in magnitude than the target data type can represent.

When overflow occurs within a `checked` block (or when the compiler is set to globally use checked arithmetic), an exception is thrown to warn of undesired behavior. Meanwhile, in an `unchecked` block, overflow is silent: no exceptions are thrown, and the value will simply wrap around to the opposite boundary. This can lead to subtle, hard to find bugs.

Since most arithmetic operations are done on values that are not large or small enough to overflow, most of the time, there is no need to explicitly define a block as `checked`. Care needs to be taken when doing arithmetic on unbounded input that may cause overflow, for example when doing arithmetic in recursive functions or while taking user input.

Neither `checked` nor `unchecked` affect floating point arithmetic operations.

When a block or expression is declared as `unchecked`, any arithmetic operations inside it are allowed to overflow without causing an error. An example where this behavior is *desired* would be the calculation of a checksum, where

the value is allowed to "wrap around" during calculation:

```
byte Checksum(byte[] data) {
    byte result = 0;
    for (int i = 0; i < data.Length; i++) {
        result = unchecked(result + data[i]); // unchecked expression
    }
    return result;
}
```

One of the most common uses for `unchecked` is implementing a custom override for `object.GetHashCode()`, a type of checksum. You can see the keyword's use in the answers to this question: What is the best algorithm for an overridden System.Object.GetHashCode?.

When a block or expression is declared as `checked`, any arithmetic operation that causes an overflow results in an `OverflowException` being thrown.

```
int SafeSum(int x, int y) {
    checked { // checked block
        return x + y;
    }
}
```

Both checked and unchecked may be in block and expression form.

Checked and unchecked blocks do not affect called methods, only operators called directly in the current method. For example, `Enum.ToObject()`, `Convert.ToInt32()`, and user-defined operators are not affected by custom checked/unchecked contexts.

***Note**: The default overflow default behavior (checked vs. unchecked) may be changed in the **Project Properties** or through the **/checked[+|-]** command line switch. It is common to default to checked operations for debug builds and unchecked for release builds. The `checked` and `unchecked` keywords would then be used only where a default approach does not apply and you need an explicit behavior to ensure correctness.*

Section 52.5: virtual, override, new

virtual and override

The `virtual` keyword allows a method, property, indexer or event to be overridden by derived classes and present polymorphic behavior. (Members are non-virtual by default in C#)

```
public class BaseClass
{
    public virtual void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}
```

In order to override a member, the `override` keyword is used in the derived classes. (Note the signature of the members must be identical)

```
public class DerivedClass: BaseClass
{
    public override void Foo()
    {
```

```
        Console.WriteLine("Foo from DerivedClass");
    }
}
```

The polymorphic behavior of virtual members means that when invoked, the actual member being executed is determined at runtime instead of at compile time. The overriding member in the most derived class the particular object is an instance of will be the one executed.

In short, object can be declared of type `BaseClass` at compile time but if at runtime it is an instance of `DerivedClass` then the overridden member will be executed:

```
BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from DerivedClass"
```

Overriding a method is optional:

```
public class SecondDerivedClass: DerivedClass {}

var obj1 = new SecondDerivedClass();
obj1.Foo(); //Outputs "Foo from DerivedClass"
```

new

Since only members defined as `virtual` are overridable and polymorphic, a derived class redefining a non virtual member might lead to unexpected results.

```
public class BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}

public class DerivedClass: BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from BaseClass" too!
```

When this happens, the member executed is always determined at compile time based on the type of the object.

- If the object is declared of type `BaseClass` (even if at runtime is of a derived class) then the method of `BaseClass` is executed
- If the object is declared of type `DerivedClass` then the method of `DerivedClass` is executed.

This is usually an accident (When a member is added to the base type after an identical one was added to the

derived type) and a compiler warning **CS0108** is generated in those scenarios.

If it was intentional, then the `new` keyword is used to suppress the compiler warning (And inform other developers of your intentions!). the behavior remains the same, the `new` keyword just suppresses the compiler warning.

```
public class BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}

public class DerivedClass: BaseClass
{
    public new void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from BaseClass" too!
```

The usage of override is *not* optional

Unlike in C++, the usage of the `override` keyword is *not* optional:

```
public class A
{
    public virtual void Foo()
    {
    }
}

public class B : A
{
    public void Foo() // Generates CS0108
    {
    }
}
```

The above example also causes warning **CS0108**, because `B.Foo()` is not automatically overriding `A.Foo()`. Add `override` when the intention is to override the base class and cause polymorphic behavior, add `new` when you want non-polymorphic behavior and resolve the call using the static type. The latter should be used with caution, as it may cause severe confusion.

The following code even results in an error:

```
public class A
{
    public void Foo()
    {
    }
}

public class B : A
```

```
{
    public override void Foo() // Error: Nothing to override
    {
    }
}
```

Derived classes can introduce polymorphism

The following code is perfectly valid (although rare):

```
    public class A
    {
        public void Foo()
        {
            Console.WriteLine("A");
        }
    }

    public class B : A
    {
        public new virtual void Foo()
        {
            Console.WriteLine("B");
        }
    }
```

Now all objects with a static reference of B (and its derivatives) use polymorphism to resolve `Foo()`, while references of A use `A.Foo()`.

```
A a = new A();
a.Foo(); // Prints "A";
a = new B();
a.Foo(); // Prints "A";
B b = new B();
b.Foo(); // Prints "B";
```

Virtual methods cannot be private

The C# compiler is strict in preventing senseless constructs. Methods marked as `virtual` cannot be private. Because a private method cannot be seen from a derived type, it couldn't be overwritten either. This fails to compile:

```
public class A
{
    private virtual void Foo() // Error: virtual methods cannot be private
    {
    }
}
```

Section 52.6: stackalloc

The `stackalloc` keyword creates a region of memory on the stack and returns a pointer to the start of that memory. Stack allocated memory is automatically removed when the scope it was created in is exited.

```
//Allocate 1024 bytes. This returns a pointer to the first byte.
byte* ptr = stackalloc byte[1024];

//Assign some values...
ptr[0] = 109;
ptr[1] = 13;
```

```
ptr[2] = 232;
...
```

Used in an unsafe context.

As with all pointers in C# there is no bounds checking on reads and assignments. Reading beyond the bounds of the allocated memory will have unpredictable results - it may access some arbitrary location within memory or it may cause an access violation exception.

```
//Allocate 1 byte
byte* ptr = stackalloc byte[1];

//Unpredictable results...
ptr[10] = 1;
ptr[-1] = 2;
```

Stack allocated memory is automatically removed when the scope it was created in is exited. This means that you should never return the memory created with stackalloc or store it beyond the lifetime of the scope.

```
unsafe IntPtr Leak() {
    //Allocate some memory on the stack
    var ptr = stackalloc byte[1024];

    //Return a pointer to that memory (this exits the scope of "Leak")
    return new IntPtr(ptr);
}

unsafe void Bad() {
     //ptr is now an invalid pointer, using it in any way will have
     //unpredictable results. This is exactly the same as accessing beyond
     //the bounds of the pointer.
     var ptr = Leak();
}
```

`stackalloc` can only be used when declaring *and* initialising variables. The following is *not* valid:

```
byte* ptr;
...
ptr = stackalloc byte[1024];
```

Remarks:

`stackalloc` should only be used for performance optimizations (either for computation or interop). This is due to the fact that:

- The garbage collector is not required as the memory is allocated on the stack rather than the heap - the memory is released as soon as the variable goes out of scope
- It is faster to allocate memory on the stack rather than the heap
- Increase the chance of cache hits on the CPU due to the locality of data

Section 52.7: break

In a loop (for, foreach, do, while) the `break` statement aborts the execution of the innermost loop and returns to the code after it. Also it can be used with `yield` in which it specifies that an iterator has come to an end.

```
for (var i = 0; i < 10; i++)
{
```

```
    if (i == 5)
    {
        break;
    }
    Console.WriteLine("This will appear only 5 times, as the break will stop the loop.");
}
```

Live Demo on .NET Fiddle

```
foreach (var stuff in stuffCollection)
{
    if (stuff.SomeStringProp == null)
        break;
    // If stuff.SomeStringProp for any "stuff" is null, the loop is aborted.
    Console.WriteLine(stuff.SomeStringProp);
}
```

The break-statement is also used in switch-case constructs to break out of a case or default segment.

```
switch(a)
{
    case 5:
        Console.WriteLine("a was 5!");
        break;

    default:
        Console.WriteLine("a was something else!");
        break;
}
```

In switch statements, the 'break' keyword is required at the end of each case statement. This is contrary to some languages that allow for 'falling through' to the next case statement in the series. Workarounds for this would include 'goto' statements or stacking the 'case' statements sequentially.

Following code will give numbers `0`, `1`, `2`, `...`, `9` and the last line will not be executed. `yield break` signifies the end of the function (not just a loop).

```
public static IEnumerable<int> GetNumbers()
{
    int i = 0;
    while (true) {
        if (i < 10) {
            yield return i++;
        } else {
            yield break;
        }
    }
    Console.WriteLine("This line will not be executed");
}
```

Live Demo on .NET Fiddle

Note that unlike some other languages, there is no way to label a particular break in C#. This means that in the case of nested loops, only the innermost loop will be stopped:

```
foreach (var outerItem in outerList)
{
    foreach (var innerItem in innerList)
```

```
    {
        if (innerItem.ShoudBreakForWhateverReason)
            // This will only break out of the inner loop, the outer will continue:
            break;
    }
}
```

If you want to break out of the *outer* loop here, you can use one of several different strategies, such as:

- A **goto** statement to jump out of the whole looping structure.
- A specific flag variable (`shouldBreak` in the following example) that can be checked at the end of each iteration of the outer loop.
- Refactoring the code to use a `return` statement in the innermost loop body, or avoid the whole nested loop structure altogether.

```
bool shouldBreak = false;
while(comeCondition)
{
    while(otherCondition)
    {
        if (conditionToBreak)
        {
            // Either tranfer control flow to the label below...
            goto endAllLooping;

            // OR use a flag, which can be checked in the outer loop:
            shouldBreak = true;
        }
    }

    if(shouldBreakNow)
    {
        break; // Break out of outer loop if flag was set to true
    }
}

endAllLooping: // label from where control flow will continue
```

Section 52.8: const

`const` is used to represent values that **will never change** throughout the lifetime of the program. Its value is constant from **compile-time**, as opposed to the `readonly` keyword, whose value is constant from run-time.

For example, since the speed of light will never change, we can store it in a constant.

```
const double c = 299792458;  // Speed of light

double CalculateEnergy(double mass)
{
    return mass * c * c;
}
```

This is essentially the same as having `return mass * 299792458 * 299792458`, as the compiler will directly substitute `c` with its constant value.

As a result, `c` cannot be changed once declared. The following will produce a compile-time error:

```
const double c = 299792458;  // Speed of light
```

```
c = 500;  //compile-time error
```

A constant can be prefixed with the same access modifiers as methods:

```
private const double c = 299792458;
public const double c = 299792458;
internal const double c = 299792458;
```

`const` members are `static` by nature. However using `static` explicitly is not permitted.

You can also define method-local constants:

```
double CalculateEnergy(double mass)
{
    const c = 299792458;
    return mass * c * c;
}
```

These can not be prefixed with a `private` or `public` keyword, since they are implicitly local to the method they are defined in.

Not all types can be used in a `const` declaration. The value types that are allowed, are the pre-defined types `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, and all `enum` types. Trying to declare `const` members with other value types (such as `TimeSpan` or `Guid`) will fail at compile-time.

For the special pre-defined reference type `string`, constants can be declared with any value. For all other reference types, constants can be declared but must always have the value `null`.

Because `const` values are known at compile-time, they are allowed as `case` labels in a `switch` statement, as standard arguments for optional parameters, as arguments to attribute specifications, and so on.

If `const` values are used across different assemblies, care must be taken with versioning. For example, if assembly A defines a `public const int MaxRetries = 3;`, and assembly B uses that constant, then if the value of `MaxRetries` is later changed to `5` in assembly A (which is then re-compiled), that change will not be effective in assembly B *unless* assembly B is also re-compiled (with a reference to the new version of A).

For that reason, if a value might change in future revisions of the program, and if the value needs to be publicly visible, do not declare that value `const` unless you know that all dependent assemblies will be re-compiled whenever something is changed. The alternative is using `static readonly` instead of `const`, which is resolved at runtime.

Section 52.9: async, await

The `await` keyword was added as part of C# 5.0 release which is supported from Visual Studio 2012 onwards. It leverages Task Parallel Library (TPL) which made the multi-threading relatively easier. The `async` and `await` keywords are used in pair in the same function as shown below. The `await` keyword is used to pause the current asynchronous method's execution until the awaited asynchronous task is completed and/or its results returned. In order to use the `await` keyword, the method that uses it must be marked with the `async` keyword.

Using `async` with `void` is strongly discouraged. For more info you can look here.

Example:

```
public async Task DoSomethingAsync()
{
```

```
    Console.WriteLine("Starting a useless process...");
    Stopwatch stopwatch = Stopwatch.StartNew();
    int delay = await UselessProcessAsync(1000);
    stopwatch.Stop();
    Console.WriteLine("A useless process took {0} milliseconds to execute.",
stopwatch.ElapsedMilliseconds);
}

public async Task<int> UselessProcessAsync(int x)
{
    await Task.Delay(x);
    return x;
}
```

Output:

> "Starting a useless process..."
>
> **... 1 second delay... **
>
> "A useless process took 1000 milliseconds to execute."

The keyword pairs `async` and `await` can be omitted if a `Task` or `Task<T>` returning method only returns a single asynchronous operation.

Rather than this:

```
public async Task PrintAndDelayAsync(string message, int delay)
{
    Debug.WriteLine(message);
    await Task.Delay(x);
}
```

It is preferred to do this:

```
public Task PrintAndDelayAsync(string message, int delay)
{
    Debug.WriteLine(message);
    return Task.Delay(x);
}
```

`Version = 5.0`

In C# 5.0 `await` cannot be used in `catch` and `finally`.

`Version ≥ 6.0`

With C# 6.0 `await` can be used in `catch` and `finally`.

Section 52.10: for

Syntax: `for (initializer; condition; iterator)`

- The `for` loop is commonly used when the number of iterations is known.
- The statements in the `initializer` section run only once, before you enter the loop.
- The `condition` section contains a boolean expression that's evaluated at the end of every loop iteration to

determine whether the loop should exit or should run again.
- The `iterator` section defines what happens after each iteration of the body of the loop.

This example shows how `for` can be used to iterate over the characters of a string:

```
string str = "Hello";
for (int i = 0; i < str.Length; i++)
{
    Console.WriteLine(str[i]);
}
```

Output:

> H
> e
> l
> l
> o

Live Demo on .NET Fiddle

All of the expressions that define a `for` statement are optional; for example, the following statement is used to create an infinite loop:

```
for( ; ; )
{
    // Your code here
}
```

The `initializer` section can contain multiple variables, so long as they are of the same type. The `condition` section can consist of any expression which can be evaluated to a `bool`. And the `iterator` section can perform multiple actions separated by comma:

```
string hello = "hello";
for (int i = 0, j = 1, k = 9; i < 3 && k > 0; i++, hello += i) {
    Console.WriteLine(hello);
}
```

Output:

> hello
> hello1
> hello12

Live Demo on .NET Fiddle

Section 52.11: abstract

A class marked with the keyword `abstract` cannot be instantiated.

A class *must* be marked as abstract if it contains abstract members or if it inherits abstract members that it doesn't implement. A class *may* be marked as abstract even if no abstract members are involved.

Abstract classes are usually used as base classes when some part of the implementation needs to be specified by another component.

```
abstract class Animal
{
    string Name { get; set; }
    public abstract void MakeSound();
}

public class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Meov meov");
    }
}

public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Bark bark");
    }
}

Animal cat = new Cat();       // Allowed due to Cat deriving from Animal
cat.MakeSound();              // will print out "Meov meov"

Animal dog = new Dog();       // Allowed due to Dog deriving from Animal
dog.MakeSound();              // will print out "Bark bark"

Animal animal = new Animal(); // Not allowed due to being an abstract class
```

A method, property, or event marked with the keyword `abstract` indicates that the implementation for that member is expected to be provided in a subclass. As mentioned above, abstract members can only appear in abstract classes.

```
abstract class Animal
{
   public abstract string Name { get; set; }
}

public class Cat : Animal
{
    public override string Name { get; set; }
}

public class Dog : Animal
{
    public override string Name { get; set; }
}
```

Section 52.12: fixed

The fixed statement fixes memory in one location. Objects in memory are usually moving arround, this makes garbage collection possible. But when we use unsafe pointers to memory addresses, that memory must not be moved.

- We use the fixed statement to ensure that the garbage collector does not relocate the string data.

Fixed Variables

```
var myStr = "Hello world!";

fixed (char* ptr = myStr)
{
    // myStr is now fixed (won't be [re]moved by the Garbage Collector).
    // We can now do something with ptr.
}
```

Used in an unsafe context.

Fixed Array Size

```
unsafe struct Example
{
    public fixed byte SomeField[8];
    public fixed char AnotherField[64];
}
```

`fixed` can only be used on fields in a `struct` (must also be used in an unsafe context).

Section 52.13: default

For classes, interfaces, delegate, array, nullable (such as int?) and pointer types, `default(TheType)` returns `null`:

```
class MyClass {}
Debug.Assert(default(MyClass) == null);
Debug.Assert(default(string) == null);
```

For structs and enums, `default(TheType)` returns the same as `new TheType()`:

```
struct Coordinates
{
    public int X { get; set; }
    public int Y { get; set; }
}

struct MyStruct
{
    public string Name { get; set; }
    public Coordinates Location { get; set; }
    public Coordinates? SecondLocation { get; set; }
    public TimeSpan Duration { get; set; }
}

var defaultStruct = default(MyStruct);
Debug.Assert(defaultStruct.Equals(new MyStruct()));
Debug.Assert(defaultStruct.Location.Equals(new Coordinates()));
Debug.Assert(defaultStruct.Location.X == 0);
Debug.Assert(defaultStruct.Location.Y == 0);
Debug.Assert(defaultStruct.SecondLocation == null);
Debug.Assert(defaultStruct.Name == null);
Debug.Assert(defaultStruct.Duration == TimeSpan.Zero);
```

`default(T)` can be particularly useful when `T` is a generic parameter for which no constraint is present to decide whether `T` is a reference type or a value type, for example:

```
public T GetResourceOrDefault<T>(string resourceName)
{
   if (ResourceExists(resourceName))
   {
      return (T)GetResource(resourceName);
   }
   else
   {
      return default(T);
   }
}
```

Section 52.14: sealed

When applied to a class, the `sealed` modifier prevents other classes from inheriting from it.

```
class A { }
sealed class B : A { }
class C : B { } //error : Cannot derive from the sealed class
```

When applied to a `virtual` method (or virtual property), the `sealed` modifier prevents this method (property) from being *overridden* in derived classes.

```
public class A
{
    public sealed override string ToString() // Virtual method inherited from class Object
    {
        return "Do not override me!";
    }
}

public class B: A
{
    public override string ToString() // Compile time error
    {
        return "An attempt to override";
    }
}
```

Section 52.15: is

Checks if an object is compatible with a given type, i.e. if an object is an instance of the `BaseInterface` type, or a type that derives from `BaseInterface`:

```
interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass);  // True
Console.WriteLine(d is BaseClass);     // True
Console.WriteLine(d is BaseInterface); // True
Console.WriteLine(d is object);        // True
Console.WriteLine(d is string);        // False

var b = new BaseClass();
Console.WriteLine(b is DerivedClass);  // False
Console.WriteLine(b is BaseClass);     // True
```

```
Console.WriteLine(b is BaseInterface); // True
Console.WriteLine(b is object);        // True
Console.WriteLine(b is string);        // False
```

If the intent of the cast is to use the object, it is best practice to use the `as` keyword'

```
interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass);  // True - valid use of 'is'
Console.WriteLine(d is BaseClass);     // True - valid use of 'is'

if(d is BaseClass){
    var castedD = (BaseClass)d;
    castedD.Method(); // valid, but not best practice
}

var asD = d as BaseClass;

if(asD!=null){
    asD.Method(); //preferred method since you incur only one unboxing penalty
}
```

But, from C# 7 `pattern matching` feature extends the is operator to check for a type and declare a new variable at the same time. Same code part with C# 7 :

`Version ≥ 7.0`

```
if(d is BaseClass asD ){
    asD.Method();
}
```

Section 52.16: this

The `this` keyword refers to the current instance of class(object). That way two variables with the same name, one at the class-level (a field) and one being a parameter (or local variable) of a method, can be distinguished.

```
public MyClass {
    int a;

    void set_a(int a)
    {
        //this.a refers to the variable defined outside of the method,
        //while a refers to the passed parameter.
        this.a = a;
    }
}
```

Other usages of the keyword are chaining non-static constructor overloads:

```
public MyClass(int arg) : this(arg, null)
{
}
```

and writing indexers:

```
public string this[int idx1, string idx2]
```

```
{
    get { /* ... */ }
    set { /* ... */ }
}
```

and declaring extension methods:

```
public static int Count<TItem>(this IEnumerable<TItem> source)
{
    // ...
}
```

If there is no conflict with a local variable or parameter, it is a matter of style whether to use `this` or not, so `this.MemberOfType` and `MemberOfType` would be equivalent in that case. Also see `base` keyword.

Note that if an extension method is to be called on the current instance, `this` is required. For example if your are inside a non-static method of a class which implements `IEnumerable<>` and you want to call the extension `Count` from before, you must use:

```
this.Count()  // works like StaticClassForExtensionMethod.Count(this)
```

and `this` cannot be omitted there.

Section 52.17: readonly

The `readonly` keyword is a field modifier. When a field declaration includes a `readonly` modifier, assignments to that field can only occur as part of the declaration or in a constructor in the same class.

The `readonly` keyword is different from the `const` keyword. A `const` field can only be initialized at the declaration of the field. A `readonly` field can be initialized either at the declaration or in a constructor. Therefore, `readonly` fields can have different values depending on the constructor used.

The `readonly` keyword is often used when injecting dependencies.

```
class Person
{
    readonly string _name;
    readonly string _surname = "Surname";

    Person(string name)
    {
        _name = name;
    }
    void ChangeName()
    {
        _name = "another name"; // Compile error
        _surname = "another surname"; // Compile error
    }
}
```

> Note: Declaring a field *readonly* does not imply *immutability*. If the field is a *reference type* then the **content** of the object can be changed. *Readonly* is typically used to prevent having the object being **overwritten** and assigned only during **instantiation** of that object.

> Note: Inside the constructor a readonly field can be reassigned

```
public class Car
{
    public double Speed {get; set;}
}

//In code

private readonly Car car = new Car();

private void SomeMethod()
{
    car.Speed = 100;
}
```

Section 52.18: typeof

Returns the `Type` of an object, without the need to instantiate it.

```
Type type = typeof(string);
Console.WriteLine(type.FullName); //System.String
Console.WriteLine("Hello".GetType() == type); //True
Console.WriteLine("Hello".GetType() == typeof(string)); //True
```

Section 52.19: foreach

`foreach` is used to iterate over the elements of an array or the items within a collection which implements `IEnumerable`†.

```
var lines = new string[] {
    "Hello world!",
    "How are you doing today?",
    "Goodbye"
};

foreach (string line in lines)
{
    Console.WriteLine(line);
}
```

This will output

> "Hello world!"
> "How are you doing today?"
> "Goodbye"

Live Demo on .NET Fiddle

You can exit the `foreach` loop at any point by using the break keyword or move on to the next iteration using the continue keyword.

```
var numbers = new int[] {1, 2, 3, 4, 5, 6};

foreach (var number in numbers)
```

```
{
    // Skip if 2
    if (number == 2)
        continue;

    // Stop iteration if 5
    if (number == 5)
        break;

    Console.Write(number + ", ");
}

// Prints: 1, 3, 4,
```

Live Demo on .NET Fiddle

Notice that the order of iteration is guaranteed *only* for certain collections such as arrays and `List`, but **not** guaranteed for many other collections.

† While `IEnumerable` is typically used to indicate enumerable collections, `foreach` only requires that the collection expose publicly the `object GetEnumerator()` method, which should return an object that exposes the `bool MoveNext()` method and the `object Current { get; }` property.

Section 52.20: dynamic

The `dynamic` keyword is used with dynamically typed objects. Objects declared as `dynamic` forego compile-time static checks, and are instead evaluated at runtime.

```
using System;
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
// 456

Console.WriteLine(info.DoesntExist);
// Throws RuntimeBinderException
```

The following example uses `dynamic` with Newtonsoft's library Json.NET, in order to easily read data from a deserialized JSON file.

```
try
{
    string json = @"{ x : 10, y : ""ho""}";
    dynamic deserializedJson = JsonConvert.DeserializeObject(json);
    int x = deserializedJson.x;
    string y = deserializedJson.y;
    // int z = deserializedJson.z; // throws RuntimeBinderException
}
catch (RuntimeBinderException e)
{
    // This exception is thrown when a property
    // that wasn't assigned to a dynamic variable is used
}
```

There are some limitations associated with the dynamic keyword. One of them is the use of extension methods. The following example adds an extension method for string: `SayHello`.

```
static class StringExtensions
{
    public static string SayHello(this string s) => $"Hello {s}!";
}
```

The first approach will be to call it as usual (as for a string):

```
var person = "Person";
Console.WriteLine(person.SayHello());

dynamic manager = "Manager";
Console.WriteLine(manager.SayHello()); // RuntimeBinderException
```

No compilation error, but at runtime you get a `RuntimeBinderException`. The workaround for this will be to call the extension method via the static class:

```
var helloManager = StringExtensions.SayHello(manager);
Console.WriteLine(helloManager);
```

Section 52.21: try, catch, finally, throw

`try`, `catch`, `finally`, and `throw` allow you to handle exceptions in your code.

```
var processor = new InputProcessor();

// The code within the try block will be executed. If an exception occurs during execution of
// this code, execution will pass to the catch block corresponding to the exception type.
try
{
    processor.Process(input);
}
// If a FormatException is thrown during the try block, then this catch block
// will be executed.
catch (FormatException ex)
{
    // Throw is a keyword that will manually throw an exception, triggering any catch block that is
    // waiting for that exception type.
    throw new InvalidOperationException("Invalid input", ex);
}
// catch can be used to catch all or any specific exceptions. This catch block,
// with no type specified, catches any exception that hasn't already been caught
// in a prior catch block.
catch
{
    LogUnexpectedException();
    throw; // Re-throws the original exception.
}
// The finally block is executed after all try-catch blocks have been; either after the try has
// succeeded in running all commands or after all exceptions have been caught.
finally
{
    processor.Dispose();
}
```

Note: The `return` keyword can be used in `try` block, and the `finally` block will still be executed (just before

returning). For example:

```
try
{
    connection.Open();
    return connection.Get(query);
}
finally
{
    connection.Close();
}
```

The statement `connection.Close()` will execute before the result of `connection.Get(query)` is returned.

Section 52.22: void

The reserved word `"void"` is an alias of `System.Void` type, and has two uses:

1. Declare a method that doesn't have a return value:

```
public void DoSomething()
{
    // Do some work, don't return any value to the caller.
}
```

A method with a return type of void can still have the `return` keyword in its body. This is useful when you want to exit the method's execution and return the flow to the caller:

```
public void DoSomething()
{
    // Do some work...

    if (condition)
        return;

    // Do some more work if the condition evaluated to false.
}
```

2. Declare a pointer to an unknown type in an unsafe context.

In an unsafe context, a type may be a pointer type, a value type, or a reference type. A pointer type declaration is usually `type* identifier`, where the type is a known type - i.e `int* myInt`, but can also be `void* identifier`, where the type is unknown.

Note that declaring a void pointer type is discouraged by Microsoft.

Section 52.23: namespace

The `namespace` keyword is an organization construct that helps us understand how a codebase is arranged. Namespaces in C# are virtual spaces rather than being in a physical folder.

```
namespace StackOverflow
{
    namespace Documentation
    {
        namespace CSharp.Keywords
        {
```

```
            public class Program
            {
                public static void Main()
                {
                    Console.WriteLine(typeof(Program).Namespace);
                    //StackOverflow.Documentation.CSharp.Keywords
                }
            }
        }
    }
}
```

Namespaces in C# can also be written in chained syntax. The following is equivalent to above:

```
namespace StackOverflow.Documentation.CSharp.Keywords
{
    public class Program
    {
        public static void Main()
        {
            Console.WriteLine(typeof(Program).Namespace);
            //StackOverflow.Documentation.CSharp.Keywords
        }
    }
}
```

Section 52.24: ref, out

The `ref` and `out` keywords cause an argument to be passed by reference, not by value. For value types, this means that the value of the variable can be changed by the callee.

```
int x = 5;
ChangeX(ref x);
// The value of x could be different now
```

For reference types, the instance in the variable can not only be modified (as is the case without `ref`), but it can also be replaced altogether:

```
Address a = new Address();
ChangeFieldInAddress(a);
// a will be the same instance as before, even if it is modified
CreateANewInstance(ref a);
// a could be an entirely new instance now
```

The main difference between the `out` and `ref` keyword is that `ref` requires the variable to be initialized by the caller, while `out` passes that responsibility to the callee.

To use an `out` parameter, both the method definition and the calling method must explicitly use the `out` keyword.

```
int number = 1;
Console.WriteLine("Before AddByRef: " + number); // number = 1
AddOneByRef(ref number);
Console.WriteLine("After AddByRef: " + number);  // number = 2
SetByOut(out number);
Console.WriteLine("After SetByOut: " + number);  // number = 34

void AddOneByRef(ref int value)
{
```

```
    value++;
}

void SetByOut(out int value)
{
    value = 34;
}
```

Live Demo on .NET Fiddle

The following does *not* compile, because `out` parameters must have a value assigned before the method returns (it would compile using `ref` instead):

```
void PrintByOut(out int value)
{
    Console.WriteLine("Hello!");
}
```

using out keyword as Generic Modifier

`out` keyword can also be used in generic type parameters when defining generic interfaces and delegates. In this case, the `out` keyword specifies that the type parameter is covariant.

> Covariance enables you to use a more derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement variant interfaces and implicit conversion of delegate types. Covariance and contravariance are supported for reference types, but they are not supported for value types. - MSDN

```
//if we have an interface like this
interface ICovariant<out R> { }

//and two variables like
ICovariant<Object> iobj = new Sample<Object>();
ICovariant<String> istr = new Sample<String>();

// then the following statement is valid
// without the out keyword this would have thrown error
iobj = istr; // implicit conversion occurs here
```

Section 52.25: base

The **`base`** keyword is used to access members from a base class. It is commonly used to call base implementations of virtual methods, or to specify which base constructor should be called.

Choosing a constructor

```
public class Child : SomeBaseClass {
    public Child() : base("some string for the base class")
    {
    }
}

public class SomeBaseClass {
    public SomeBaseClass()
    {
        // new Child() will not call this constructor, as it does not have a parameter
```

```
    }
    public SomeBaseClass(string message)
    {
        // new Child() will use this base constructor because of the specified parameter in Child's
constructor
        Console.WriteLine(message);
    }
}
```

Calling base implementation of virtual method

```
public override void SomeVirtualMethod() {
    // Do something, then call base implementation
    base.SomeVirtualMethod();
}
```

It is possible to use the base keyword to call a base implementation from any method. This ties the method call directly to the base implementation, which means that even if new child classes override a virtual method, the base implementation will still be called so this needs to be used with caution.

```
public class Parent
{
    public virtual int VirtualMethod()
    {
        return 1;
    }
}

public class Child : Parent
{
    public override int VirtualMethod() {
        return 11;
    }

    public int NormalMethod()
    {
        return base.VirtualMethod();
    }

    public void CallMethods()
    {
        Assert.AreEqual(11, VirtualMethod());

        Assert.AreEqual(1, NormalMethod());
        Assert.AreEqual(1, base.VirtualMethod());
    }
}

public class GrandChild : Child
{
    public override int VirtualMethod()
    {
        return 21;
    }

    public void CallAgain()
    {
        Assert.AreEqual(21, VirtualMethod());
        Assert.AreEqual(11, base.VirtualMethod());
```

```
        // Notice that the call to NormalMethod below still returns the value
        // from the extreme base class even though the method has been overridden
        // in the child class.
        Assert.AreEqual(1, NormalMethod());
    }
}
```

Section 52.26: float, double, decimal

float

`float` is an alias to the .NET datatype `System.Single`. It allows IEEE 754 single-precision floating point numbers to be stored. This data type is present in `mscorlib.dll` which is implicitly referenced by every C# project when you create them.

Approximate range: -3.4 × 1038 to 3.4 × 1038

Decimal precision: 6-9 significant digits

Notation:

```
float f = 0.1259;
var f1 = 0.7895f; // f is literal suffix to represent float values
```

> It should be noted that the `float` type often results in significant rounding errors. In applications where precision is important, other data types should be considered.

double

`double` is an alias to the .NET datatype `System.Double`. It represents a double-precision 64-bit floating-point number. This datatype is present in `mscorlib.dll` which is implicitly referenced in any C# project.

Range: ±5.0 × 10−324 to ±1.7 × 10308

Decimal precision: 15-16 significant digits

Notation:

```
double distance = 200.34; // a double value
double salary = 245; // an integer implicitly type-casted to double value
var marks = 123.764D; // D is literal suffix to represent double values
```

decimal

`decimal` is an alias to the .NET datatype `System.Decimal`. It represents a keyword indicates a 128-bit data type. Compared to floating-point types, the decimal type has more precision and a smaller range, which makes it appropriate for financial and monetary calculations. This datatype is present in `mscorlib.dll` which is implicitly referenced in any C# project.

Range: -7.9 × 1028 to 7.9 × 1028

Decimal precision: 28-29 significant digits

Notation:

```
decimal payable = 152.25m; // a decimal value
var marks = 754.24m; // m is literal suffix to represent decimal values
```

Section 52.27: operator

Most of the built-in operators (including conversion operators) can be overloaded by using the `operator` keyword along with the `public` and `static` modifiers.

The operators comes in three forms: unary operators, binary operators and conversion operators.

Unary and binary operators requires at least one parameter of same type as the containing type, and some requires a complementary matching operator.

Conversion operators must convert to or from the enclosing type.

```
public struct Vector32
{

    public Vector32(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; }
    public int Y { get; }

    public static bool operator ==(Vector32 left, Vector32 right)
        => left.X == right.X && left.Y == right.Y;

    public static bool operator !=(Vector32 left, Vector32 right)
        => !(left == right);

    public static Vector32 operator +(Vector32 left, Vector32 right)
        => new Vector32(left.X + right.X, left.Y + right.Y);

    public static Vector32 operator +(Vector32 left, int right)
        => new Vector32(left.X + right, left.Y + right);

    public static Vector32 operator +(int left, Vector32 right)
        => right + left;

    public static Vector32 operator -(Vector32 left, Vector32 right)
        => new Vector32(left.X - right.X, left.Y - right.Y);

    public static Vector32 operator -(Vector32 left, int right)
        => new Vector32(left.X - right, left.Y - right);

    public static Vector32 operator -(int left, Vector32 right)
        => right - left;

    public static implicit operator Vector64(Vector32 vector)
        => new Vector64(vector.X, vector.Y);

    public override string ToString() => $"{{{X}, {Y}}}";

}
```

```
public struct Vector64
{

    public Vector64(long x, long y)
    {
        X = x;
        Y = y;
    }

    public long X { get; }
    public long Y { get; }

    public override string ToString() => $"{{{X}, {Y}}}";

}
```

Example

```
var vector1 = new Vector32(15, 39);
var vector2 = new Vector32(87, 64);

Console.WriteLine(vector1 == vector2); // false
Console.WriteLine(vector1 != vector2); // true
Console.WriteLine(vector1 + vector2);  // {102, 103}
Console.WriteLine(vector1 - vector2);  // {-72, -25}
```

Section 52.28: char

A char is single letter stored inside a variable. It is built-in value type which takes two bytes of memory space. It represents `System.Char` data type found in `mscorlib.dll` which is implicitly referenced by every C# project when you create them.

There are multiple ways to do this.

1. `char c = 'c';`
2. `char c = '\u0063'; //Unicode`
3. `char c = '\x0063'; //Hex`
4. `char c = (char)99;//Integral`

A char can be implicitly converted to `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal` and it will return the integer value of that char.

```
ushort u = c;
```

returns 99 etc.

However, there are no implicit conversions from other types to char. Instead you must cast them.

```
ushort u = 99;
 char c = (char)u;
```

Section 52.29: params

`params` allows a method parameter to receive a variable number of arguments, i.e. zero, one or multiple arguments are allowed for that parameter.

```
static int AddAll(params int[] numbers)
{
    int total = 0;
    foreach (int number in numbers)
    {
        total += number;
    }

    return total;
}
```

This method can now be called with a typical list of `int` arguments, or an array of ints.

```
AddAll(5, 10, 15, 20);                // 50
AddAll(new int[] { 5, 10, 15, 20 });  // 50
```

`params` must appear at most once and if used, it must be **last** in the argument list, even if the succeeding type is different to that of the array.

Be careful when overloading functions when using the `params` keyword. C# prefers matching more specific overloads before resorting to trying to use overloads with `params`. For example if you have two methods:

```
static double Add(params double[] numbers)
{
    Console.WriteLine("Add with array of doubles");
    double total = 0.0;
    foreach (double number in numbers)
    {
        total += number;
    }

    return total;
}

static int Add(int a, int b)
{
    Console.WriteLine("Add with 2 ints");
    return a + b;
}
```

Then the specific 2 argument overload will take precedence before trying the `params` overload.

```
Add(2, 3);      //prints "Add with 2 ints"
Add(2, 3.0);    //prints "Add with array of doubles" (doubles are not ints)
Add(2, 3, 4);   //prints "Add with array of doubles" (no 3 argument overload)
```

Section 52.30: while

The `while` operator iterates over a block of code until the conditional query equals false or the code is interrupted with a `goto`, `return`, `break` or `throw` statement.

Syntax for `while` keyword:

> while(*condition*) { *code block;* }

Example:

```
int i = 0;
while (i++ < 5)
{
    Console.WriteLine("While is on loop number {0}.", i);
}
```

Output:

> "While is on loop number 1."
> "While is on loop number 2."
> "While is on loop number 3."
> "While is on loop number 4."
> "While is on loop number 5."

Live Demo on .NET Fiddle

A while loop is **Entry Controlled**, as the condition is checked **before** the execution of the enclosed code block. This means that the while loop wouldn't execute its statements if the condition is false.

```
bool a = false;

while (a == true)
{
    Console.WriteLine("This will never be printed.");
}
```

Giving a `while` condition without provisioning it to become false at some point will result in an infinite or endless loop. As far as possible, this should be avoided, however, there may be some exceptional circumstances when you need this.

You can create such a loop as follows:

```
while (true)
{
//...
}
```

Note that the C# compiler will transform loops such as

```
while (true)
{
// ...
}
```

or

```
for(;;)
{
// ...
}
```

into

```
{
:label
```

```
// ...
goto label;
}
```

Note that a while loop may have any condition, no matter how complex, as long as it evaluates to (or returns) a boolean value (bool). It may also contain a function that returns a boolean value (as such a function evaluates to the same type as an expression such as `a==x'). For example,

```
while (AgriculturalService.MoreCornToPick(myFarm.GetAddress()))
{
    myFarm.PickCorn();
}
```

Section 52.31: null

A variable of a reference type can hold either a valid reference to an instance or a null reference. The null reference is the default value of reference type variables, as well as nullable value types.

`null` is the keyword that represents a null reference.

As an expression, it can be used to assign the null reference to variables of the aforementioned types:

```
object a = null;
string b = null;
int? c = null;
List<int> d  = null;
```

Non-nullable value types cannot be assigned a null reference. All the following assignments are invalid:

```
int a = null;
float b = null;
decimal c = null;
```

The null reference should *not* be confused with valid instances of various types such as:

- an empty list (`new List<int>()`)
- an empty string (`""`)
- the number zero (`0`, `0f`, `0m`)
- the null character (`'\0'`)

Sometimes, it is meaningful to check if something is either null or an empty/default object. The System.String.IsNullOrEmpty(String) method may be used to check this, or you may implement your own equivalent method.

```
private void GreetUser(string userName)
{
    if (String.IsNullOrEmpty(userName))
    {
        //The method that called us either sent in an empty string, or they sent us a null reference.
Either way, we need to report the problem.
        throw new InvalidOperationException("userName may not be null or empty.");
    }
    else
    {
        //userName is acceptable.
        Console.WriteLine("Hello, " + userName + "!");
    }
```

```
}
```

Section 52.32: continue

Immediately pass control to the next iteration of the enclosing loop construct (for, foreach, do, while):

```
for (var i = 0; i < 10; i++)
{
    if (i < 5)
    {
        continue;
    }
    Console.WriteLine(i);
}
```

Output:

```
5
6
7
8
9
```

Live Demo on .NET Fiddle

```
var stuff = new [] {"a", "b", null, "c", "d"};

foreach (var s in stuff)
{
    if (s == null)
    {
        continue;
    }
    Console.WriteLine(s);
}
```

Output:

```
a
b
c
d
```

Live Demo on .NET Fiddle

Section 52.33: string

`string` is an alias to the .NET datatype `System.String`, which allows text (sequences of characters) to be stored.

Notation:

```
string a = "Hello";
var b = "world";
```

```
var f = new string(new []{ 'h', 'i', '!' }); // hi!
```

Each character in the string is encoded in UTF-16, which means that each character will require a minimum 2 bytes of storage space.

Section 52.34: return

> MSDN: The return statement terminates execution of the method in which it appears and returns control to the calling method. It can also return an optional value. If the method is a void type, the return statement can be omitted.

```
public int Sum(int valueA, int valueB)
{
    return valueA + valueB;
}


public void Terminate(bool terminateEarly)
{
    if (terminateEarly) return; // method returns to caller if true was passed in
    else Console.WriteLine("Not early"); // prints only if terminateEarly was false
}
```

Section 52.35: unsafe

The `unsafe` keyword can be used in type or method declarations or to declare an inline block.

The purpose of this keyword is to enable the use of the *unsafe subset* of C# for the block in question. The unsafe subset includes features like pointers, stack allocation, C-like arrays, and so on.

Unsafe code is not verifiable and that's why its usage is discouraged. Compilation of unsafe code requires passing a switch to the C# compiler. Additionally, the CLR requires that the running assembly has full trust.

Despite these limitations, unsafe code has valid usages in making some operations more performant (e.g. array indexing) or easier (e.g. interop with some unmanaged libraries).

As a very simple example

```
// compile with /unsafe
class UnsafeTest
{
   unsafe static void SquarePtrParam(int* p)
   {
      *p *= *p; // the '*' dereferences the pointer.
      //Since we passed in "the address of i", this becomes "i *= i"
   }

   unsafe static void Main()
   {
      int i = 5;
      // Unsafe method: uses address-of operator (&):
      SquarePtrParam(&i); // "&i" means "the address of i". The behavior is similar to "ref i"
      Console.WriteLine(i); // Output: 25
   }
}
```

While working with pointers, we can change the values of memory locations directly, rather than having to address them by name. Note that this often requires the use of the fixed keyword to prevent possible memory corruption as the garbage collector moves things around (otherwise, you may get error CS0212). Since a variable that has been "fixed" cannot be written to, we also often have to have a second pointer that starts out pointing to the same location as the first.

```
void Main()
{
    int[] intArray = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    UnsafeSquareArray(intArray);
    foreach(int i in intArray)
        Console.WriteLine(i);
}

unsafe static void UnsafeSquareArray(int[] pArr)
{
    int len = pArr.Length;

    //in C or C++, we could say
    // int* a = &(pArr[0])
    // however, C# requires you to "fix" the variable first
    fixed(int* fixedPointer = &(pArr[0]))
    {
        //Declare a new int pointer because "fixedPointer" cannot be written to.
        // "p" points to the same address space, but we can modify it
        int* p = fixedPointer;

        for (int i = 0; i < len; i++)
        {
            *p *= *p; //square the value, just like we did in SquarePtrParam, above
            p++;      //move the pointer to the next memory space.
                      // NOTE that the pointer will move 4 bytes since "p" is an
                      // int pointer and an int takes 4 bytes

            //the above 2 lines could be written as one, like this:
            // "*p *= *p++;"
        }
    }
}
```

Output:

```
1
4
9
16
25
36
49
64
81
100
```

`unsafe` also allows the use of stackalloc which will allocate memory on the stack like _alloca in the C run-time library. We can modify the above example to use `stackalloc` as follows:

```
unsafe void Main()
{
```

```
    const int len=10;
    int* seedArray = stackalloc int[len];

    //We can no longer use the initializer "{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}" as before.
    // We have at least 2 options to populate the array. The end result of either
    // option will be the same (doing both will also be the same here).

    //FIRST OPTION:
    int* p = seedArray; // we don't want to lose where the array starts, so we
                        // create a shadow copy of the pointer
    for(int i=1; i<=len; i++)
        *p++ = i;
    //end of first option

    //SECOND OPTION:
    for(int i=0; i<len; i++)
        seedArray[i] = i+1;
    //end of second option

    UnsafeSquareArray(seedArray, len);
    for(int i=0; i< len; i++)
        Console.WriteLine(seedArray[i]);
}

//Now that we are dealing directly in pointers, we don't need to mess around with
// "fixed", which dramatically simplifies the code
unsafe static void UnsafeSquareArray(int* p, int len)
{
    for (int i = 0; i < len; i++)
        *p *= *p++;
}
```

(Output is the same as above)

Section 52.36: switch

The `switch` statement is a control statement that selects a switch section to execute from a list of candidates. A switch statement includes one or more switch sections. Each switch section contains one or more `case` labels followed by one or more statements. If no case label contains a matching value, control is transferred to the `default` section, if there is one. Case fall-through is not supported in C#, strictly speaking. However, if 1 or more `case` labels are empty, execution will follow the code of the next `case` block which contains code. This allows grouping of multiple `case` labels with the same implementation. In the following example, if `month` equals 12, the code in `case 2` will be executed since the `case` labels 12 1 and 2 are grouped. If a `case` block is not empty, a `break` must be present before the next `case` label, otherwise the compiler will flag an error.

```
int month = DateTime.Now.Month; // this is expected to be 1-12 for Jan-Dec

switch (month)
{
    case 12:
    case 1:
    case 2:
        Console.WriteLine("Winter");
        break;
    case 3:
    case 4:
    case 5:
        Console.WriteLine("Spring");
        break;
```

```
    case 6:
    case 7:
    case 8:
        Console.WriteLine("Summer");
        break;
    case 9:
    case 10:
    case 11:
        Console.WriteLine("Autumn");
        break;
    default:
        Console.WriteLine("Incorrect month index");
        break;
}
```

A `case` can only be labeled by a value known at *compile time* (e.g. `1`, `"str"`, `Enum.A`), so a `variable` isn't a valid `case` label, but a `const` or an `Enum` value is (as well as any literal value).

Section 52.37: var

An implicitly-typed local variable that is strongly typed just as if the user had declared the type. Unlike other variable declarations, the compiler determines the type of variable that this represents based on the value that is assigned to it.

```
var i = 10; // implicitly typed, the compiler must determine what type of variable this is
int i = 10; // explicitly typed, the type of variable is explicitly stated to the compiler

// Note that these both represent the same type of variable (int) with the same value (10).
```

Unlike other types of variables, variable definitions with this keyword need to be initialized when declared. This is due to the **var** keyword representing an implicitly-typed variable.

```
var i;
i = 10;

// This code will not run as it is not initialized upon declaration.
```

The **var** keyword can also be used to create new datatypes on the fly. These new datatypes are known as *anonymous types*. They are quite useful, as they allow a user to define a set of properties without having to explicitly declare any kind of object type first.

Plain anonymous type

```
var a = new { number = 1, text = "hi" };
```

LINQ query that returns an anonymous type

```
public class Dog
{
    public string Name { get; set; }
    public int Age { get; set; }
}

public class DogWithBreed
{
    public Dog Dog { get; set; }
    public string BreedName  { get; set; }
```

```
}

public void GetDogsWithBreedNames()
{
    var db = new DogDataContext(ConnectString);
    var result = from d in db.Dogs
             join b in db.Breeds on d.BreedId equals b.BreedId
             select new
                    {
                        DogName = d.Name,
                        BreedName = b.BreedName
                    };

    DoStuff(result);
}
```

You can use var keyword in foreach statement

```
public bool hasItemInList(List<String> list, string stringToSearch)
{
    foreach(var item in list)
    {
        if( ( (string)item ).equals(stringToSearch) )
            return true;
    }

    return false;
}
```

Section 52.38: when

The `when` is a keyword added in **C# 6**, and it is used for exception filtering.

Before the introduction of the `when` keyword, you could have had one catch clause for each type of exception; with the addition of the keyword, a more fine-grained control is now possible.

A `when` expression is attached to a `catch` branch, and only if the `when` condition is `true`, the `catch` clause will be executed. It is possible to have several `catch` clauses with the same exception class types, and different `when` conditions.

```
private void CatchException(Action action)
{
    try
    {
        action.Invoke();
    }

    // exception filter
    catch (Exception ex) when (ex.Message.Contains("when"))
    {
        Console.WriteLine("Caught an exception with when");
    }

    catch (Exception ex)
    {
        Console.WriteLine("Caught an exception without when");
    }
}
```

```
private void Method1() { throw new Exception("message for exception with when"); }
private void Method2() { throw new Exception("message for general exception"); }


CatchException(Method1);
CatchException(Method2);
```

Section 52.39: lock

`lock` provides thread-safety for a block of code, so that it can be accessed by only one thread within the same process. Example:

```
private static object _lockObj = new object();
static void Main(string[] args)
{
    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());

    Console.ReadKey();
}

private static void TaskWork()
{
    lock(_lockObj)
    {
        Console.WriteLine("Entered");

        Task.Delay(3000);
        Console.WriteLine("Done Delaying");

        // Access shared resources safely

        Console.WriteLine("Leaving");
    }
}

Output:

Entered
Done Delaying
Leaving
Entered
Done Delaying
Leaving
Entered
Done Delaying
Leaving
```

Use cases:

Whenever you have a block of code that might produce side-effects if executed by multiple threads at the same time. The lock keyword along with a **shared synchronization object** (`_objLock` in the example) can be used to prevent that.

Note that `_objLock` can't be `null` and multiple threads executing the code must use the same object instance (either by making it a `static` field, or by using the same class instance for both threads)

From the compiler side, the lock keyword is a syntactic sugar that is replaced by `Monitor.Enter(_lockObj);` and

`Monitor.Exit(_lockObj);`. So if you replace the lock by surrounding the block of code with these two methods, you would get the same results. You can see actual code in Syntactic sugar in C# - lock example

Section 52.40: uint

An **unsigned integer**, or **uint**, is a numeric datatype that only can hold positive integers. Like it's name suggests, it represents an unsigned 32-bit integer. The **uint** keyword itself is an alias for the Common Type System type `System.UInt32`. This datatype is present in `mscorlib.dll`, which is implicitly referenced by every C# project when you create them. It occupies four bytes of memory space.

Unsigned integers can hold any value from 0 to 4,294,967,295.

Examples on how and now not to declare unsigned integers

```
uint i = 425697; // Valid expression, explicitly stated to compiler
var i1 = 789247U; // Valid expression, suffix allows compiler to determine datatype
uint x = 3.0; // Error, there is no implicit conversion
```

Please note: According to Microsoft, it is recommended to use the **int** datatype wherever possible as the **uint** datatype is not CLS-compliant.

Section 52.41: if, if...else, if... else if

The `if` statement is used to control the flow of the program. An `if` statement identifies which statement to run based on the value of a `Boolean` expression.

For a single statement, the `braces{}` are optional but recommended.

```
int a = 4;
if(a % 2 == 0)
{
     Console.WriteLine("a contains an even number");
}
// output: "a contains an even number"
```

The `if` can also have an `else` clause, that will be executed in case the condition evaluates to false:

```
int a = 5;
if(a % 2 == 0)
{
     Console.WriteLine("a contains an even number");
}
else
{
     Console.WriteLine("a contains an odd number");
}
// output: "a contains an odd number"
```

The `if...else if` construct lets you specify multiple conditions:

```
int a = 9;
if(a % 2 == 0)
{
     Console.WriteLine("a contains an even number");
}
else if(a % 3 == 0)
```

```
{
     Console.WriteLine("a contains an odd number that is a multiple of 3");
}
else
{
     Console.WriteLine("a contains an odd number");
}
// output: "a contains an odd number that is a multiple of 3"
```

***Important to note* that if a condition is met in the above example , the control skips other tests and jumps to the end of that particular if else construct.So, the *order* of tests is important if you are using if .. else if construct**

C# Boolean expressions use short-circuit evaluation. This is important in cases where evaluating conditions may have side effects:

```
if (someBooleanMethodWithSideEffects() && someOtherBooleanMethodWithSideEffects()) {
  //...
}
```

There's no guarantee that `someOtherBooleanMethodWithSideEffects` will actually run.

It's also important in cases where earlier conditions ensure that it's "safe" to evaluate later ones. For example:

```
if (someCollection != null && someCollection.Count > 0) {
   // ..
}
```

The order is very important in this case because, if we reverse the order:

```
if (someCollection.Count > 0 && someCollection != null) {
```

it will throw a `NullReferenceException` if `someCollection` is `null`.

Section 52.42: static

The `static` modifier is used to declare a static member, which does not need to be instantiated in order to be accessed, but instead is accessed simply through its name, i.e. `DateTime.Now`.

`static` can be used with classes, fields, methods, properties, operators, events, and constructors.

While an instance of a class contains a separate copy of all instance fields of the class, there is only one copy of each static field.

```
class A
{
    static public int count = 0;

    public A()
    {
        count++;
    }
}

class Program
{
    static void Main(string[] args)
```

```
    {
        A a = new A();
        A b = new A();
        A c = new A();

        Console.WriteLine(A.count); // 3
    }
}
```

`count` equals to the total number of instances of `A` class.

The static modifier can also be used to declare a static constructor for a class, to initialize static data or run code that only needs to be called once. Static constructors are called before the class is referenced for the first time.

```
class A
{
    static public DateTime InitializationTime;

    // Static constructor
    static A()
    {
        InitializationTime = DateTime.Now;
        // Guaranteed to only run once
        Console.WriteLine(InitializationTime.ToString());
    }
}
```

A `static class` is marked with the `static` keyword, and can be used as a beneficial container for a set of methods that work on parameters, but don't necessarily require being tied to an instance. Because of the `static` nature of the class, it cannot be instantiated, but it can contain a `static constructor`. Some features of a `static class` include:

- Can't be inherited
- Can't inherit from anything other than `Object`
- Can contain a static constructor but not an instance constructor
- Can only contain static members
- Is sealed

The compiler is also friendly and will let the developer know if any instance members exist within the class. An example would be a static class that converts between US and Canadian metrics:

```
static class ConversionHelper {
    private static double oneGallonPerLitreRate = 0.264172;

    public static double litreToGallonConversion(int litres) {
        return litres * oneGallonPerLitreRate;
    }
}
```

When classes are declared static:

```
public static class Functions
{
  public static int Double(int value)
  {
    return value + value;
  }
```

```
}
```

all function, properties or members within the class also need to be declared static. No instance of the class can be created. In essence a static class allows you to create bundles of functions that are grouped together logically.

Since C#6 `static` can also be used alongside `using` to import static members and methods. They can be used then without class name.

Old way, without `using static`:

```
using System;

public class ConsoleApplication
{
    public static void Main()
    {
         Console.WriteLine("Hello World!"); //Writeline is method belonging to static class Console
    }

}
```

Example with `using static`

```
using static System.Console;

public class ConsoleApplication
{
    public static void Main()
    {
         WriteLine("Hello World!"); //Writeline is method belonging to static class Console
    }

}
```

Drawbacks

While static classes can be incredibly useful, they do come with their own caveats:

- Once the static class has been called, the class is loaded into memory and cannot be run through the garbage collector until the AppDomain housing the static class is unloaded.
- A static class cannot implement an interface.

Section 52.43: internal

The `internal` keyword is an access modifier for types and type members. Internal types or members are **accessible only within files in the same assembly**

usage:

```
public class BaseClass
{
    // Only accessible within the same assembly
    internal static int x = 0;
}
```

The difference between different access modifiers is clarified here

Access modifiers

public

> The type or member can be accessed by any other code in the same assembly or another assembly that references it.

private

> The type or member can only be accessed by code in the same class or struct.

protected

> The type or member can only be accessed by code in the same class or struct, or in a derived class.

internal

> The type or member can be accessed by any code in the same assembly, but not from another assembly.

protected internal

> The type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.

When **no access modifier** is set, a default access modifier is used. So there is always some form of access modifier even if it's not set.

Section 52.44: using

There are two types of `using` keyword usage, `using statement` and `using directive`:

1. **using statement**:

   The `using` keyword ensures that objects that implement the `IDisposable` interface are properly disposed after usage. There is a separate topic for the using statement

2. **using directive**

   The `using` directive has three usages, see the msdn page for the using directive. There is a separate topic for the using directive.

Section 52.45: where

`where` can serve two purposes in C#: type constraining in a generic argument, and filtering LINQ queries.

In a generic class, let's consider

```
public class Cup<T>
{
    // ...
}
```

T is called a type parameter. The class definition can impose constraints on the actual types that can be supplied for T.

The following kinds of constraints can be applied:

- value type
- reference type
- default constructor
- inheritance and implementation

value type

In this case only `struct`s (this includes 'primitive' data types such as `int`, `boolean` etc) can be supplied

```
public class Cup<T> where T : struct
{
    // ...
}
```

reference type

In this case only class types can be supplied

```
public class Cup<T> where T : class
{
    // ...
}
```

hybrid value/reference type

Occasionally it is desired to restrict type arguments to those available in a database, and these will usually map to value types and strings. As all type restrictions must be met, it is not possible to specify `where T : struct or string` (this is not valid syntax). A workaround is to restrict type arguments to `IConvertible` which has built in types of "... Boolean, SByte, Byte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, Decimal, DateTime, Char, and String." It is possible other objects will implement IConvertible, though this is rare in practice.

```
public class Cup<T> where T : IConvertible
{
    // ...
}
```

default constructor

Only types that contain a default constructor will be allowed. This includes value types and classes that contain a default (parameterless) constructor

```
public class Cup<T> where T : new
{
    // ...
```

```
}
```

inheritance and implementation

Only types that inherit from a certain base class or implement a certain interface can be supplied.

```
public class Cup<T> where T : Beverage
{
    // ...
}


public class Cup<T> where T : IBeer
{
    // ...
}
```

The constraint can even reference another type parameter:

```
public class Cup<T, U> where U : T
{
    // ...
}
```

Multiple constraints can be specified for a type argument:

```
public class Cup<T> where T : class, new()
{
    // ...
}
```

The previous examples show generic constraints on a class definition, but constraints can be used anywhere a type argument is supplied: classes, structs, interfaces, methods, etc.

`where` can also be a LINQ clause. In this case it is analogous to `WHERE` in SQL:

```
int[] nums = { 5, 2, 1, 3, 9, 8, 6, 7, 2, 0 };

var query =
    from num in nums
    where num < 5
    select num;

    foreach (var n in query)
    {
        Console.Write(n + " ");
    }
    // prints 2 1 3 2 0
```

Section 52.46: int

`int` is an alias for `System.Int32`, which is a data type for signed 32-bit integers. This data type can be found in `mscorlib.dll` which is implicitly referenced by every C# project when you create them.

Range: -2,147,483,648 to 2,147,483,647

```
int int1 = -10007;
```

```
var int2 = 2132012521;
```

Section 52.47: ulong

Keyword used for unsigned 64-bit integers. It represents `System.UInt64` data type found in `mscorlib.dll` which is implicitly referenced by every C# project when you create them.

Range: 0 to 18,446,744,073,709,551,615

```
ulong veryLargeInt = 18446744073609451315;
var anotherVeryLargeInt = 1544674406360945131 5UL;
```

Section 52.48: true, false

The `true` and `false` keywords have two uses:

1. As literal Boolean values

```
var myTrueBool = true;
var myFalseBool = false;
```

2. As operators that can be overloaded

```
public static bool operator true(MyClass x)
{
    return x.value >= 0;
}

public static bool operator false(MyClass x)
{
    return x.value < 0;
}
```

Overloading the false operator was useful prior to C# 2.0, before the introduction of `Nullable` types.
A type that overloads the `true` operator, must also overload the `false` operator.

Section 52.49: struct

A `struct` type is a value type that is typically used to encapsulate small groups of related variables, such as the coordinates of a rectangle or the characteristics of an item in an inventory.

> Classes are reference types, structs are value types.

```
using static System.Console;

namespace ConsoleApplication1
{
    struct Point
    {
        public int X;
        public int Y;

        public override string ToString()
        {
            return $"X = {X}, Y = {Y}";
```

```
        }

        public void Display(string name)
        {
            WriteLine(name + ": " + ToString());
        }
    }

    class Program
    {
        static void Main()
        {
            var point1 = new Point {X = 10, Y = 20};
            // it's not a reference but value type
            var point2 = point1;
            point2.X = 777;
            point2.Y = 888;
            point1.Display(nameof(point1)); // point1: X = 10, Y = 20
            point2.Display(nameof(point2)); // point2: X = 777, Y = 888

            ReadKey();
        }
    }
}
```

Structs can also contain constructors, constants, fields, methods, properties, indexers, operators, events, and nested types, although if several such members are required, you should consider making your type a class instead.

Some suggestions from MS on when to use struct and when to use class:

CONSIDER

defining a struct instead of a class if instances of the type are small and commonly short-lived or are commonly embedded in other objects.

AVOID

defining a struct unless the type has all of the following characteristics:

- It logically represents a single value, similar to primitive types (int, double, etc.)
- It has an instance size under 16 bytes.
- It is immutable.
- It will not have to be boxed frequently.

Section 52.50: extern

The `extern` keyword is used to declare methods that are implemented externally. This can be used in conjunction with the DllImport attribute to call into unmanaged code using Interop services. which in this case it will come with `static` modifier

For Example:

```
using System.Runtime.InteropServices;
public class MyClass
{
    [DllImport("User32.dll")]
    private static extern int SetForegroundWindow(IntPtr point);
```

```
    public void ActivateProcessWindow(Process p)
    {
        SetForegroundWindow(p.MainWindowHandle);
    }
}
```

This uses the SetForegroundWindow method imported from the User32.dll library

This can also be used to define an external assembly alias. which let us to reference different versions of same components from single assembly.

To reference two assemblies with the same fully-qualified type names, an alias must be specified at a command prompt, as follows:

```
/r:GridV1=grid.dll
/r:GridV2=grid20.dll
```

This creates the external aliases GridV1 and GridV2. To use these aliases from within a program, reference them by using the extern keyword. For example:

```
extern alias GridV1;
extern alias GridV2;
```

Section 52.51: bool

Keyword for storing the Boolean values `true` and `false`. bool is an alias of System.Boolean.

The default value of a bool is false.

```
bool b; // default value is false
b = true; // true
b = ((5 + 2) == 6); // false
```

For a bool to allow null values it must be initialized as a bool?.

The default value of a bool? is null.

```
bool? a // default value is null
```

Section 52.52: interface

An `interface` contains the signatures of methods, properties and events. The derived classes defines the members as the interface contains only the declaration of the members.

An interface is declared using the `interface` keyword.

```
interface IProduct
{
    decimal Price { get; }
}

class Product : IProduct
{
    const decimal vat = 0.2M;

    public Product(decimal price)
```

```
    {
        _price = price;
    }

    private decimal _price;
    public decimal Price { get { return _price * (1 + vat); } }
}
```

Section 52.53: delegate

Delegates are types that represent a reference to a method. They are used for passing methods as arguments to other methods.

Delegates can hold static methods, instance methods, anonymous methods, or lambda expressions.

```
class DelegateExample
{
    public void Run()
    {
        //using class method
        InvokeDelegate( WriteToConsole );

        //using anonymous method
        DelegateInvoker di = delegate ( string input )
        {
            Console.WriteLine( string.Format( "di: {0} ", input ) );
            return true;
        };
        InvokeDelegate( di );

        //using lambda expression
        InvokeDelegate( input => false );
    }

    public delegate bool DelegateInvoker( string input );

    public void InvokeDelegate(DelegateInvoker func)
    {
        var ret = func( "hello world" );
        Console.WriteLine( string.Format( " > delegate returned {0}", ret ) );
    }

    public bool WriteToConsole( string input )
    {
        Console.WriteLine( string.Format( "WriteToConsole: '{0}'", input ) );
        return true;
    }
}
```

When assigning a method to a delegate it is important to note that the method must have the same return type as well as parameters. This differs from 'normal' method overloading, where only the parameters define the signature of the method.

Events are built on top of delegates.

Section 52.54: unchecked

The `unchecked` keyword prevents the compiler from checking for overflows/underflows.

For example:

```
const int ConstantMax = int.MaxValue;
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

Without the `unchecked` keyword, neither of the two addition operations will compile.

When is this useful?

This is useful as it may help speed up calculations that definitely will not overflow since checking for overflow takes time, or when an overflow/underflow is desired behavior (for instance, when generating a hash code).

Section 52.55: ushort

A numeric type used to store 16-bit positive integers. `ushort` is an alias for `System.UInt16`, and takes up 2 bytes of memory.

Valid range is `0` to `65535`.

```
ushort a = 50; // 50
ushort b = 65536; // Error, cannot be converted
ushort c = unchecked((ushort)65536); // Overflows (wraps around to 0)
```

Section 52.56: sizeof

Used to obtain the size in bytes for an unmanaged type

```
int byteSize = sizeof(byte) // 1
int sbyteSize = sizeof(sbyte) // 1
int shortSize = sizeof(short) // 2
int ushortSize = sizeof(ushort) // 2
int intSize = sizeof(int) // 4
int uintSize = sizeof(uint) // 4
int longSize = sizeof(long) // 8
int ulongSize = sizeof(ulong) // 8
int charSize = sizeof(char) // 2(Unicode)
int floatSize = sizeof(float) // 4
int doubleSize = sizeof(double) // 8
int decimalSize = sizeof(decimal) // 16
int boolSize = sizeof(bool) // 1
```

Section 52.57: in

The `in` keyword has three uses:

a) As part of the syntax in a `foreach` statement or as part of the syntax in a LINQ query

```
foreach (var member in sequence)
{
    // ...
}
```

b) In the context of generic interfaces and generic delegate types signifies *contravariance* for the type parameter in question:

```
public interface IComparer<in T>
{
    // ...
}
```

c) In the context of LINQ query refers to the collection that is being queried

```
var query = from x in source select new { x.Name, x.ID, };
```

Section 52.58: implicit

The `implicit` keyword is used to overload a conversion operator. For example, you may declare a `Fraction` class that should automatically be converted to a `double` when needed, and that can be automatically converted from `int`:

```
class Fraction(int numerator, int denominator)
{
    public int Numerator { get; } = numerator;
    public int Denominator { get; } = denominator;
    // ...
    public static implicit operator double(Fraction f)
    {
        return f.Numerator / (double) f.Denominator;
    }
    public static implicit operator Fraction(int i)
    {
        return new Fraction(i, 1);
    }
}
```

Section 52.59: do

The do operator iterates over a block of code until a conditional query equals false. The do-while loop can also be interrupted by a `goto`, `return`, `break` or `throw` statement.

The syntax for the `do` keyword is:

> do { *code block;* } while(*condition*);

Example:

```
int i = 0;

do
{
    Console.WriteLine("Do is on loop number {0}.", i);
} while (i++ < 5);
```

Output:

> "Do is on loop number 1."

> "Do is on loop number 2."
> "Do is on loop number 3."
> "Do is on loop number 4."
> "Do is on loop number 5."

Unlike the `while` loop, the do-while loop is **Exit Controlled**. This means that the do-while loop would execute its statements at least once, even if the condition fails the first time.

```
bool a = false;

do
{
    Console.WriteLine("This will be printed once, even if a is false.");
} while (a == true);
```

Section 52.60: long

The **long** keyword is used to represent signed 64-bit integers. It is an alias for the `System.Int64` datatype present in `mscorlib.dll`, which is implicitly referenced by every C# project when you create them.

*Any **long** variable can be declared both explicitly and implicitly:*

```
long long1 = 9223372036854775806;  // explicit declaration, long keyword used
var long2 = -9223372036854775806L; // implicit declaration, 'L' suffix used
```

A **long** variable can hold any value from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, and can be useful in situations which a variable must hold a value that exceeds the bounds of what other variables (such as the **int** variable) can hold.

Section 52.61: enum

The `enum` keyword tells the compiler that this class inherits from the abstract class `Enum`, without the programmer having to explicitly inherit it. `Enum` is a descendant of `ValueType`, which is intended for use with distinct set of named constants.

```
public enum DaysOfWeek
{
    Monday,
    Tuesday,
}
```

You can optionally specify a specific value for each one (or some of them):

```
public enum NotableYear
{
   EndOfWwI = 1918;
   EnfOfWwII = 1945,
}
```

In this example I omitted a value for 0, this is usually a bad practice. An `enum` will always have a default value produced by explicit conversion `(YourEnumType) 0`, where `YourEnumType` is your declared `enume` type. Without a value of 0 defined, an `enum` will not have a defined value at initiation.

The default underlying type of `enum` is `int`, you can change the underlying type to any integral type including `byte`,

`sbyte`, `short`, `ushort`, `int`, `uint`, `long` and `ulong`. Below is an enum with underlying type `byte`:

```
enum Days : byte
{
    Sunday = 0,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};
```

Also note that you can convert to/from underlying type simply with a cast:

```
int value = (int)NotableYear.EndOfWwI;
```

For these reasons you'd better always check if an `enum` is valid when you're exposing library functions:

```
void PrintNotes(NotableYear year)
{
    if (!Enum.IsDefined(typeof(NotableYear), year))
        throw InvalidEnumArgumentException("year", (int)year, typeof(NotableYear));

    // ...
}
```

Section 52.62: partial

The keyword `partial` can be used during type definition of class, struct, or interface to allow the type definition to be split into several files. This is useful to incorporate new features in auto generated code.

File1.cs

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
    }
}
```

File2.cs

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
    }
}
```

Note: A class can be split into any number of files. However, all declaration must be under same namespace and the same assembly.

Methods can also be declared partial using the `partial` keyword. In this case one file will contain only the method definition and another file will contain the implementation.

> A partial method has its signature defined in one part of a partial type, and its implementation defined in another part of the type. Partial methods enable class designers to provide method hooks, similar to event handlers, that developers may decide to implement or not. If the developer does not supply an implementation, the compiler removes the signature at compile time. The following conditions apply to partial methods:
>
> - Signatures in both parts of the partial type must match.
> - The method must return void.
> - No access modifiers are allowed. Partial methods are implicitly private.
>
> -- MSDN

File1.cs

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
        public partial Method1(string str);
    }
}
```

File2.cs

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
        public partial Method1(string str)
        {
            Console.WriteLine(str);
        }
    }
}
```

Note: The type containing the partial method must also be declared partial.

Section 52.63: event

An `event` allows the developer to implement a notification pattern.

Simple example

```
public class Server
{
    // defines the event
    public event EventHandler DataChangeEvent;

    void RaiseEvent()
    {
        var ev = DataChangeEvent;
        if(ev != null)
        {
            ev(this, EventArgs.Empty);
        }
```

```
    }
}

public class Client
{
    public void Client(Server server)
    {
        // client subscribes to the server's DataChangeEvent
        server.DataChangeEvent += server_DataChanged;
    }

    private void server_DataChanged(object sender, EventArgs args)
    {
        // notified when the server raises the DataChangeEvent
    }
}
```

MSDN reference

Section 52.64: sbyte

A numeric type used to store 8-bit *signed* integers. `sbyte` is an alias for `System.SByte`, and takes up 1 byte of memory. For the unsigned equivalent, use `byte`.

Valid range is `-127` to `127` (the leftover is used to store the sign).

```
sbyte a = 127; // 127
sbyte b = -127; // -127
sbyte c = 200; // Error, cannot be converted
sbyte d = unchecked((sbyte)129); // -127 (overflows)
```

Chapter 53: Object Oriented Programming In C#

This topic try to tell us how we can write programs based on OOP approach.But we don't try to teach Object Oriented Programming paradigm. We'll be covering following topics:
Classes,Properties,Inheritance,Polymorphism,Interfaces and so on.

Section 53.1: Classes:

Skeleton of declaring class is:

<>:Required

[]:Optional

```
[private/public/protected/internal] class <Desired Class Name> [:[Inherited class][,][[Interface
Name 1],[Interface Name 2],...]
{
    //Your code
}
```

Don't worry if you can't understand whole syntax,We'll be get familiar with all part of that.for first example consider following class:

```
class MyClass
{
    int i = 100;
    public void getMyValue()
    {
        Console.WriteLine(this.i);//Will print number 100 in output
    }
}
```

in this class we create variable `i` with `int` type and with default private Access Modifiers and `getMyValue()` method with public access modifiers.

Chapter 54: Recursion

Section 54.1: Recursion in plain English

Recursion can be defined as:

> A method that calls itself until a specific condition is met.

An excellent and simple example of recursion is a method that will get the factorial of a given number:

```
public int Factorial(int number)
{
    return number == 0 ? 1 : n * Factorial(number - 1);
}
```

In this method, we can see that the method will take an argument, `number`.

Step by step:

Given the example, executing `Factorial(4)`

1. Is `number (4) == 1`?
2. No? return `4 * Factorial(number-1)` (3)
3. Because the method is called once again, it now repeats the first step using `Factorial(3)` as the new argument.
4. This continues until `Factorial(1)` is executed and `number (1) == 1` returns 1.
5. Overall, the calculation "builds up" `4 * 3 * 2 * 1` and finally returns 24.

The key to understanding recursion is that the method calls a *new instance* of itself. After returning, the execution of the calling instance continues.

Section 54.2: Fibonacci Sequence

You can calculate a number in the Fibonacci sequence using recursion.

Following the math theory of F(n) = F(n-2) + F(n-1), for any i > 0,

```
// Returns the i'th Fibonacci number
public int fib(int i) {
    if(i <= 2) {
        // Base case of the recursive function.
        // i is either 1 or 2, whose associated Fibonacci sequence numbers are 1 and 1.
        return 1;
    }
    // Recursive case. Return the sum of the two previous Fibonacci numbers.
    // This works because the definition of the Fibonacci sequence specifies
    // that the sum of two adjacent elements equals the next element.
    return  fib(i - 2) + fib(i - 1);

}

fib(10); // Returns 55
```

Section 54.3: PowerOf calculation

Calculating the power of a given number can be done recursively as well. Given a base number `n` and exponent `e`, we need to make sure to split the problem in chunks by decreasing the exponent `e`.

Theoretical Example:

- $2^2 = 2x2$
- $2^3 = 2x2x2$ or, $2^3 = 2^2 \times 2$
  In there lies the secret of our recursive algorithm (see the code below). This is about taking the problem and separating it into smaller and simpler to solve chunks.
- **Notes**
  - when the base number is 0, we have to be aware to return 0 as $0^3 = 0 \times 0 \times 0$
  - when the exponent is 0, we have to be aware to always return 1, as this is a mathematical rule.

Code Example:

```
public int CalcPowerOf(int b, int e) {
    if (b == 0) { return 0; } // when base is 0, it doesn't matter, it will always return 0
    if (e == 0) { return 1; } // math rule, exponent 0 always returns 1
    return b * CalcPowerOf(b, e - 1); // actual recursive logic, where we split the problem, aka: 2³
= 2 * 2² etc..
}
```

Tests in xUnit to verify the logic:
Although this is not necessary, it's always good to write tests to verify your logic. I include those here written in the xUnit framework.

```
    [Theory]
    [MemberData(nameof(PowerOfTestData))]
    public void PowerOfTest(int @base, int exponent, int expected) {
        Assert.Equal(expected, CalcPowerOf(@base, exponent));
    }

    public static IEnumerable<object[]> PowerOfTestData() {
        yield return new object[] { 0, 0, 0 };
        yield return new object[] { 0, 1, 0 };
        yield return new object[] { 2, 0, 1 };
        yield return new object[] { 2, 1, 2 };
        yield return new object[] { 2, 2, 4 };
        yield return new object[] { 5, 2, 25 };
        yield return new object[] { 5, 3, 125 };
        yield return new object[] { 5, 4, 625 };
}
```

Section 54.4: Recursively describe an object structure

Recursion is when a method calls itself. Preferably it will do so until a specific condition is met and then it will exit the method normally, returning to the point from which the method was called. If not, a stack overflow exception might occur due to too many recursive calls.

```
/// <summary>
/// Create an object structure the code can recursively describe
/// </summary>
public class Root
{
```

```
    public string Name { get; set; }
    public ChildOne Child { get; set; }
}
public class ChildOne
{
    public string ChildOneName { get; set; }
    public ChildTwo Child { get; set; }
}
public class ChildTwo
{
    public string ChildTwoName { get; set; }
}
/// <summary>
/// The console application with the recursive function DescribeTypeOfObject
/// </summary>
public class Program
{
    static void Main(string[] args)
    {
        // point A, we call the function with type 'Root'
        DescribeTypeOfObject(typeof(Root));
        Console.WriteLine("Press a key to exit");
        Console.ReadKey();
    }

    static void DescribeTypeOfObject(Type type)
    {
        // get all properties of this type
        Console.WriteLine($"Describing type {type.Name}");
        PropertyInfo[] propertyInfos = type.GetProperties();
        foreach (PropertyInfo pi in propertyInfos)
        {
            Console.WriteLine($"Has property {pi.Name} of type {pi.PropertyType.Name}");
            // is a custom class type? describe it too
            if (pi.PropertyType.IsClass && !pi.PropertyType.FullName.StartsWith("System."))
            {
                // point B, we call the function type this property
                DescribeTypeOfObject(pi.PropertyType);
            }
        }
        // done with all properties
        // we return to the point where we were called
        // point A for the first call
        // point B for all properties of type custom class
    }
}
```

Section 54.5: Using Recursion to Get Directory Tree

One of the uses of recursion is to navigate through a hierarchical data structure, like a file system directory tree, without knowing how many levels the tree has or the number of objects on each level. In this example, you will see how to use recursion on a directory tree to find all sub-directories of a specified directory and print the whole tree to the console.

```
internal class Program
{
    internal const int RootLevel = 0;
    internal const char Tab = '\t';

    internal static void Main()
```

```
    {
        Console.WriteLine("Enter the path of the root directory:");
        var rootDirectorypath = Console.ReadLine();

        Console.WriteLine(
            $"Getting directory tree of '{rootDirectorypath}'");

        PrintDirectoryTree(rootDirectorypath);
        Console.WriteLine("Press 'Enter' to quit...");
        Console.ReadLine();
    }

    internal static void PrintDirectoryTree(string rootDirectoryPath)
    {
        try
        {
            if (!Directory.Exists(rootDirectoryPath))
            {
                throw new DirectoryNotFoundException(
                    $"Directory '{rootDirectoryPath}' not found.");
            }

            var rootDirectory = new DirectoryInfo(rootDirectoryPath);
            PrintDirectoryTree(rootDirectory, RootLevel);
        }
        catch (DirectoryNotFoundException e)
        {
            Console.WriteLine(e.Message);
        }
    }

    private static void PrintDirectoryTree(
        DirectoryInfo directory, int currentLevel)
    {
        var indentation = string.Empty;
        for (var i = RootLevel; i < currentLevel; i++)
        {
            indentation += Tab;
        }

        Console.WriteLine($"{indentation}-{directory.Name}");
        var nextLevel = currentLevel + 1;
        try
        {
            foreach (var subDirectory in directory.GetDirectories())
            {
                PrintDirectoryTree(subDirectory, nextLevel);
            }
        }
        catch (UnauthorizedAccessException e)
        {
            Console.WriteLine($"{indentation}-{e.Message}");
        }
    }
}
```

This code is somewhat more complicated than the bare minimum to complete this task, as it includes exception checking to handle any issues with getting the directories. Below you will find a break-down of the code into smaller segments with explanations of each.

`Main`:

The main method takes an input from a user as a string, which is to be used as the path to the root directory. It then calls the `PrintDirectoryTree` method with this string as the parameter.

`PrintDirectoryTree(string)`:

This is the first of two methods that handle the actual directory tree printing. This method takes a string representing the path to the root directory as a parameter. It checks if the path is an actual directory, and if not, throws a `DirectoryNotFoundException` which is then handled in the catch block. If the path is a real directory, a `DirectoryInfo` object `rootDirectory` is created from the path, and the second `PrintDirectoryTree` method is called with the `rootDirectory` object and `RootLevel`, which is an integer constant with a value of zero.

`PrintDirectoryTree(DirectoryInfo, int)`:

This second method handles the brunt of the work. It takes a `DirectoryInfo` and an integer as parameters. The `DirectoryInfo` is the current directory, and the integer is the depth of the directory relative to the root. For ease of reading, the output is indented for each level deep the current directory is, so that the output looks like this:

```
-Root
    -Child 1
    -Child 2
        -Grandchild 2.1
    -Child 3
```

Once the current directory is printed, its sub directories are retrieved, and this method is then called on each of them with a depth level value of one more than the current. That part is the recursion: the method calling itself. The program will run in this manner until it has visited every directory in the tree. When it reached a directory with no sub directories, the method will return automatically.

This method also catches an `UnauthorizedAccessException`, which is thrown if any of the sub directories of the current directory are protected by the system. The error message is printed at the current indentation level for consistency.

The method below provides a more basic approach to this problem:

```
internal static void PrintDirectoryTree(string directoryName)
{
    try
    {
        if (!Directory.Exists(directoryName)) return;
        Console.WriteLine(directoryName);
        foreach (var d in Directory.GetDirectories(directoryName))
        {
            PrintDirectoryTree(d);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

This does not include the specific error checking or output formatting of the first approach, but it effectively does the same thing. Since it only uses strings as opposed to `DirectoryInfo`, it cannot provide access to other directory properties like permissions.

Section 54.6: Factorial calculation

The factorial of a number (denoted with !, as for instance 9!) is the multiplication of that number with the factorial of one lower. So, for instance, 9! = 9 x 8! = 9 x 8 x 7! = 9 x 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1.

So in code that becomes, using recursion:

```
long Factorial(long x)
{
    if (x < 1)
    {
        throw new OutOfRangeException("Factorial can only be used with positive numbers.");
    }

    if (x == 1)
    {
        return 1;
    } else {
        return x * Factorial(x - 1);
    }
}
```

Chapter 55: Naming Conventions

This topic outlines some basic naming conventions used when writing in the C# language. Like all conventions, they are not enforced by the compiler, but will ensure readability between developers.

For comprehensive .NET framework design guidelines, see docs.microsoft.com/dotnet/standard/design-guidelines.

Section 55.1: Capitalization conventions

The following terms describe different ways to case identifiers.

Pascal Casing

The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters. For example: `BackColor`

Camel Casing

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example: `backColor`

Uppercase

All letters in the identifier are capitalized. For example: `IO`

Rules

When an identifier consists of multiple words, do not use separators, such as underscores ("_") or hyphens ("-"), between words. Instead, use casing to indicate the beginning of each word.

The following table summarizes the capitalization rules for identifiers and provides examples for the different types of identifiers:

Identifier	Case	Example
Local variable	Camel	carName
Class	Pascal	AppDomain
Enumeration type	Pascal	ErrorLevel
Enumeration values	Pascal	FatalError
Event	Pascal	ValueChanged
Exception class	Pascal	WebException
Read-only static field	Pascal	RedValue
Interface	Pascal	IDisposable
Method	Pascal	ToString
Namespace	Pascal	System.Drawing
Parameter	Camel	typeName
Property	Pascal	BackColor

More information can be found on MSDN.

Section 55.2: Enums

Use a singular name for most Enums

```
public enum Volume
{
   Low,
   Medium,
   High
}
```

Use a plural name for Enum types that are bit fields

```
[Flags]
public enum MyColors
{
    Yellow = 1,
    Green = 2,
    Red = 4,
    Blue = 8
}
```

Note: Always add the `FlagsAttribute` to a bit field Enum type.

Do not add 'enum' as a suffix

```
public enum VolumeEnum // Incorrect
```

Do not use the enum name in each entry

```
public enum Color
{
    ColorBlue, // Remove Color, unnecessary
    ColorGreen,
}
```

Section 55.3: Interfaces

Interfaces should be named with nouns or noun phrases, or adjectives that describe behaviour. For example `IComponent` uses a descriptive noun, `ICustomAttributeProvider` uses a noun phrase and `IPersistable` uses an adjective.

Interface names should be prefixed with the letter `I`, to indicate that the type is an interface, and Pascal case should be used.

Below are correctly named interfaces:

```
public interface IServiceProvider
public interface IFormatable
```

Section 55.4: Exceptions

Add 'exception' as a suffix

Custom exception names should be suffixed with "-Exception".

Below are correctly named exceptions:

```
public class MyCustomException : Exception
public class FooException : Exception
```

Section 55.5: Private fields

There are two common conventions for private fields: `camelCase` and `_camelCaseWithLeadingUnderscore`.

Camel case

```
public class Rational
{
    private readonly int numerator;
    private readonly int denominator;

    public Rational(int numerator, int denominator)
    {
        // "this" keyword is required to refer to the class-scope field
        this.numerator = numerator;
        this.denominator = denominator;
    }
}
```

Camel case with underscore

```
public class Rational
{
    private readonly int _numerator;
    private readonly int _denominator;

    public Rational(int numerator, int denominator)
    {
        // Names are unique, so "this" keyword is not required
        _numerator = numerator;
        _denominator = denominator;
    }
}
```

Section 55.6: Namespaces

The general format for namespaces is:

```
<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>].
```

Examples include:

```
Fabrikam.Math
Litware.Security
```

Prefixing namespace names with a company name prevents namespaces from different companies from having the same name.

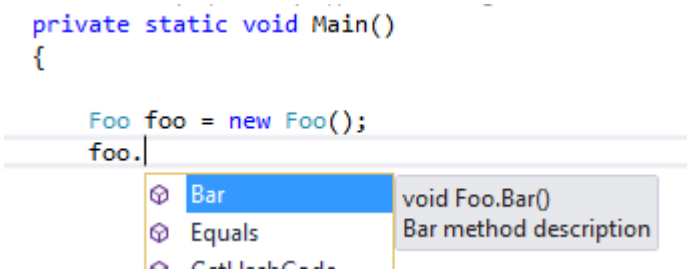
Chapter 56: XML Documentation Comments

Section 56.1: Simple method annotation

Documentation comments are placed directly above the method or class they describe. They begin with three forward slashes `///`, and allow meta information to be stored via XML.

```
/// <summary>
/// Bar method description
/// </summary>
public void Bar()
{

}
```

Information inside the tags can be used by Visual Studio and other tools to provide services such as IntelliSense:

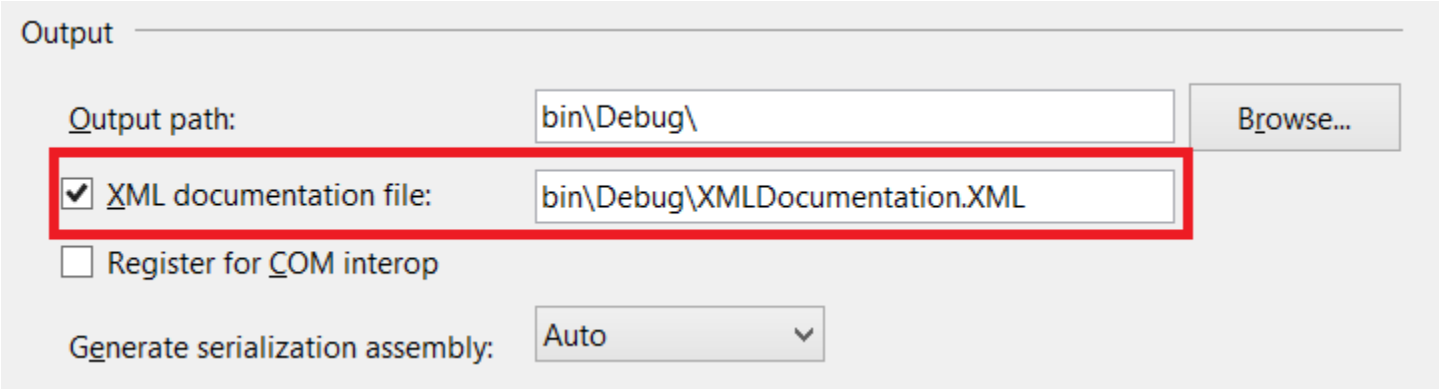


See also Microsoft's list of common documentation tags.

Section 56.2: Generating XML from documentation comments

To generate an XML documentation file from documentation comments in the code, use the `/doc` option with the `csc.exe` C# compiler.

In Visual Studio 2013/2015, In **Project** -> **Properties** -> **Build** -> **Output**, check the `XML documentation file` checkbox:



When you build the project, an XML file will be produced by the compiler with a name corresponding to the project name (e.g. `XMLDocumentation.dll` -> `XMLDocumentation.xml`).

When you use the assembly in another project, make sure that the XML file is in the same directory as the DLL being referenced.

This example:

```
/// <summary>
/// Data class description
/// </summary>
public class DataClass
{
    /// <summary>
    /// Name property description
    /// </summary>
    public string Name { get; set; }
}


/// <summary>
/// Foo function
/// </summary>
public class Foo
{
    /// <summary>
    /// This method returning some data
    /// </summary>
    /// <param name="id">Id parameter</param>
    /// <param name="time">Time parameter</param>
    /// <returns>Data will be returned</returns>
    public DataClass GetData(int id, DateTime time)
    {
        return new DataClass();
    }
}
```

Produces this xml on build:

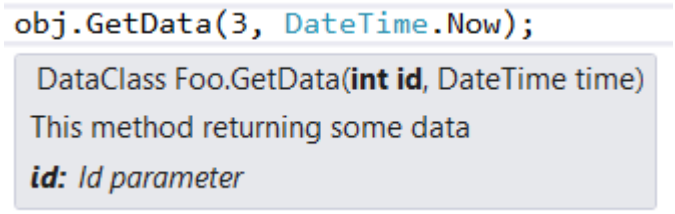
```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>XMLDocumentation</name>
    </assembly>
    <members>
        <member name="T:XMLDocumentation.DataClass">
            <summary>
            Data class description
            </summary>
        </member>
        <member name="P:XMLDocumentation.DataClass.Name">
            <summary>
            Name property description
            </summary>
        </member>
        <member name="T:XMLDocumentation.Foo">
            <summary>
            Foo function
            </summary>
        </member>
        <member name="M:XMLDocumentation.Foo.GetData(System.Int32,System.DateTime)">
            <summary>
            This method returning some data
            </summary>
            <param name="id">Id parameter</param>
            <param name="time">Time parameter</param>
            <returns>Data will be returned</returns>
```

```
        </member>
    </members>
</doc>
```

Section 56.3: Method documentation comment with param and returns elements

```
/// <summary>
/// Returns the data for the specified ID and timestamp.
/// </summary>
/// <param name="id">The ID for which to get data. </param>
/// <param name="time">The DateTime for which to get data. </param>
/// <returns>A DataClass instance with the result. </returns>
public DataClass GetData(int id, DateTime time)
{
   // ...
}
```

IntelliSense shows you the description for each parameter:



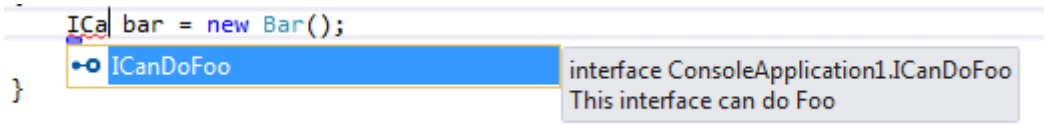
Tip: If Intellisense doesn't display in Visual Studio, delete the first bracket or comma and then type it again.

Section 56.4: Interface and class documentation comments

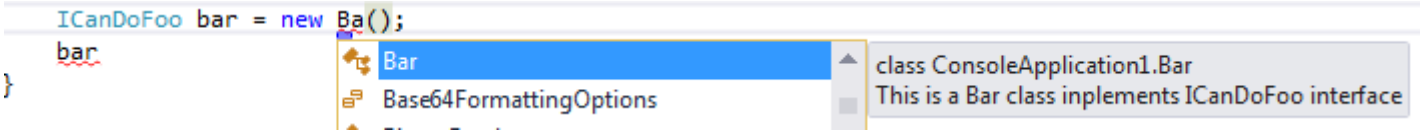
```
/// <summary>
/// This interface can do Foo
/// </summary>
public interface ICanDoFoo
{
    // ...
}

/// <summary>
/// This Bar class implements ICanDoFoo interface
/// </summary>
public class Bar : ICanDoFoo
{
    // ...
}
```

Result

Interface summary



Class summary



Section 56.5: Referencing another class in documentation

The **<see>** tag can be used to link to another class. It contains the `cref` member which should contain the name of the class that is to be referenced. Visual Studio will provide Intellsense when writing this tag and such references will be processed when renaming the referenced class, too.

```
/// <summary>
/// You might also want to check out <see cref="SomeOtherClass"/>.
/// </summary>
public class SomeClass
{
}
```

In Visual Studio Intellisense popups such references will also be displayed colored in the text.

To reference a generic class, use something similar to the following:

```
/// <summary>
/// An enhanced version of <see cref="List{T}"/>.
/// </summary>
public class SomeGenericClass<T>
{
}
```

Chapter 57: Comments and regions

Section 57.1: Comments

Using comments in your projects is a handy way of leaving explanations of your design choices, and should aim to make your (or someone else's) life easier when maintaining or adding to the code.

There are a two ways of adding a comment to your code.

Single line comments

Any text placed after `//` will be treated as a comment.

```
public class Program
{
    // This is the entry point of my program.
    public static void Main()
    {
        // Prints a message to the console. - This is a comment!
        System.Console.WriteLine("Hello, World!");

        // System.Console.WriteLine("Hello, World again!"); // You can even comment out code.
        System.Console.ReadLine();
    }
}
```

Multi line or delimited comments

Any text between `/*` and `*/` will be treated as a comment.

```
public class Program
{
    public static void Main()
    {
        /*
            This is a multi line comment
            it will be ignored by the compiler.
        */
        System.Console.WriteLine("Hello, World!");

        // It's also possible to make an inline comment with /* */
        // although it's rarely used in practice
        System.Console.WriteLine(/* Inline comment */ "Hello, World!");

        System.Console.ReadLine();
    }
}
```

Section 57.2: Regions

A region is a collapsible block of code, that can help with the readability and organisation of your code.

NOTE: StyleCop's rule SA1124 DoNotUseRegions discourages use of regions. They are usually a sign of badly organized code, as C# includes partial classes and other features which make regions obsolete.

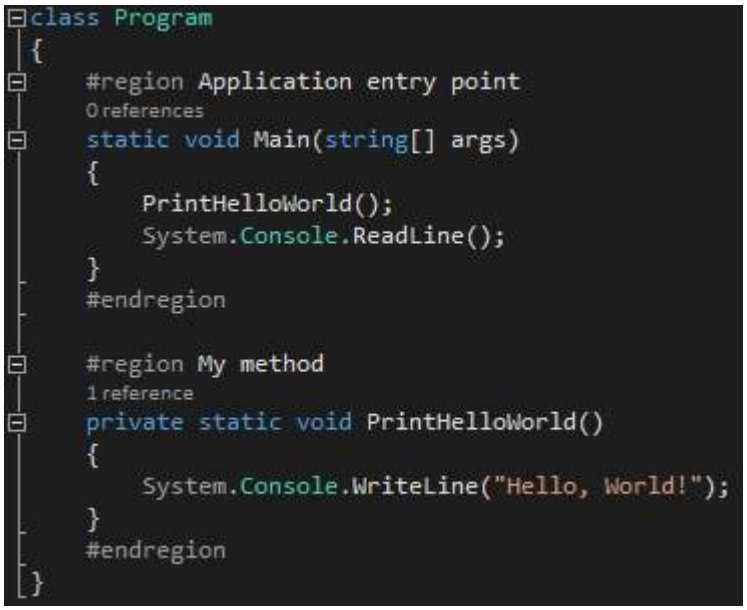
You can use regions in the following way:

```
class Program
```

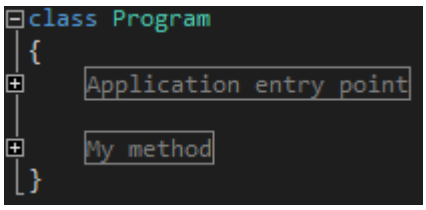
```
{
    #region Application entry point
    static void Main(string[] args)
    {
        PrintHelloWorld();
        System.Console.ReadLine();
    }
    #endregion

    #region My method
    private static void PrintHelloWorld()
    {
        System.Console.WriteLine("Hello, World!");
    }
    #endregion
}
```

When the above code is view in an IDE, you will be able to collapse and expand the code using the + and - symbols.

Expanded



Collapsed



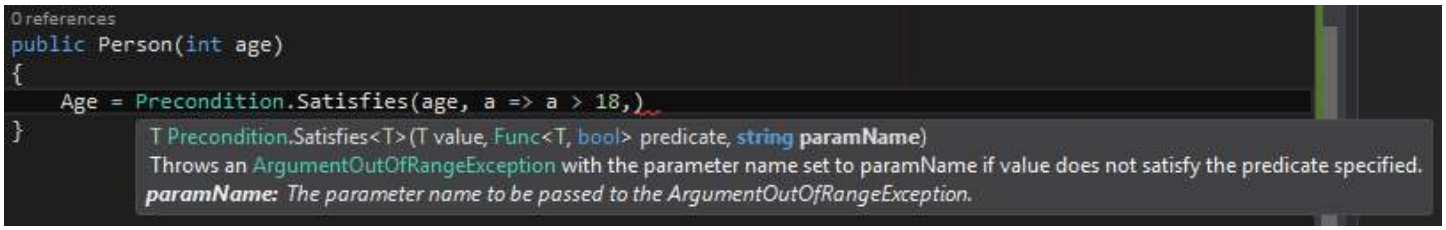
Section 57.3: Documentation comments

XML documentation comments can be used to provide API documentation that can be easily processed by tools:

```
/// <summary>
/// A helper class for validating method arguments.
/// </summary>
public static class Precondition
{
    /// <summary>
    ///     Throws an <see cref="ArgumentOutOfRangeException"/> with the parameter
```

```
    ///     name set to <c>paramName</c> if <c>value</c> does not satisfy the
    ///     <c>predicate</c> specified.
    /// </summary>
    /// <typeparam name="T">
    ///     The type of the argument checked
    /// </typeparam>
    /// <param name="value">
    ///     The argument to be checked
    /// </param>
    /// <param name="predicate">
    ///     The predicate the value is required to satisfy
    /// </param>
    /// <param name="paramName">
    ///     The parameter name to be passed to the
    ///     <see cref="ArgumentOutOfRangeException"/>.
    /// </param>
    /// <returns>The value specified</returns>
    public static T Satisfies<T>(T value, Func<T, bool> predicate, string paramName)
    {
        if (!predicate(value))
            throw new ArgumentOutOfRangeException(paramName);

        return value;
    }
}
```

Documentation is instantly picked up by IntelliSense:



Chapter 58: Inheritance

Section 58.1: Inheritance. Constructors' calls sequence

Consider we have a class `Animal` which has a child class `Dog`

```
class Animal
{
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}

class Dog : Animal
{
    public Dog()
    {
        Console.WriteLine("In Dog's constructor");
    }
}
```

By default every class implicitly inherits the `Object` class.

This is same as the above code.

```
class Animal : Object
{
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}
```

When creating an instance of `Dog` class, the **base classes's default constructor (without parameters) will be called if there is no explicit call to another constructor in the parent class**. In our case, first will be called `Object's` constructor, then `Animal's` and at the end `Dog's` constructor.

```
public class Program
{
    public static void Main()
    {
        Dog dog = new Dog();
    }
}
```

Output will be

> In Animal's constructor
> In Dog's constructor

View Demo

Call parent's constructor explicitly.

In the above examples, our `Dog` class constructor calls the **default** constructor of the `Animal` class. If you want, you can specify which constructor should be called: it is possible to call any constructor which is defined in the parent class.

Consider we have these two classes.

```
class Animal
{
    protected string name;

    public Animal()
    {
        Console.WriteLine("Animal's default constructor");
    }

    public Animal(string name)
    {
        this.name = name;
        Console.WriteLine("Animal's constructor with 1 parameter");
        Console.WriteLine(this.name);
    }
}

class Dog : Animal
{
    public Dog() : base()
    {
        Console.WriteLine("Dog's default constructor");
    }

    public Dog(string name) : base(name)
    {
        Console.WriteLine("Dog's constructor with 1 parameter");
        Console.WriteLine(this.name);
    }
}
```

What is going here?

We have 2 constructors in each class.

What does `base` mean?

`base` is a reference to the parent class. In our case, when we create an instance of `Dog` class like this

```
Dog dog = new Dog();
```

The runtime first calls the `Dog()`, which is the parameterless constructor. But its body doesn't work immediately. After the parentheses of the constructor we have a such call: `base()`, which means that when we call the default `Dog` constructor, it will in turn call the parent's **default** constructor. After the parent's constructor runs, it will return and then, finally, run the `Dog()` constructor body.

So output will be like this:

> Animal's default constructor
> Dog's default constructor

View Demo

Now what if we call the `Dog's` constructor with a parameter?

```
Dog dog = new Dog("Rex");
```

You know that members in the parent class which are not private are inherited by the child class, meaning that `Dog` will also have the `name` field.
In this case we passed an argument to our constructor. It in his turn passes the argument to the parent class' **constructor with a parameter**, which initializes the `name` field.

Output will be

```
Animal's constructor with 1 parameter
Rex
Dog's constructor with 1 parameter
Rex
```

Summary:

Every object creation starts from the base class. In the inheritance, the classes which are in the hierarchy are chained. As all classes derive from `Object`, the first constructor to be called when any object is created is the `Object` class constructor; Then the next constructor in the chain is called and only after all of them are called the object is created

base keyword

1. The base keyword is used to access members of the base class from within a derived class:
2. Call a method on the base class that has been overridden by another method. Specify which base-class constructor should be called when creating instances of the derived class.

Section 58.2: Inheriting from a base class

To avoid duplicating code, define common methods and attributes in a general class as a base:

```
public class Animal
{
    public string Name { get; set; }
    // Methods and attributes common to all animals
    public void Eat(Object dinner)
    {
        // ...
    }
    public void Stare()
    {
        // ...
    }
    public void Roll()
    {
        // ...
    }
}
```

Now that you have a class that represents `Animal` in general, you can define a class that describes the peculiarities of specific animals:

```
public class Cat : Animal
{
    public Cat()
    {
        Name = "Cat";
    }
    // Methods for scratching furniture and ignoring owner
    public void Scratch(Object furniture)
    {
        // ...
    }
}
```

The Cat class gets access to not only the methods described in its definition explicitly, but also all the methods defined in the general `Animal` base class. Any Animal (whether or not it was a Cat) could Eat, Stare, or Roll. An Animal would not be able to Scratch, however, unless it was also a Cat. You could then define other classes describing other animals. (Such as Gopher with a method for destroying flower gardens and Sloth with no extra methods at all.)

Section 58.3: Inheriting from a class and implementing an interface

```
public class Animal
{
    public string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

//Note that in C#, the base class name must come before the interface names
public class Cat : Animal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
    }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

Section 58.4: Inheriting from a class and implementing multiple interfaces

```
public class LivingBeing
{
    string Name { get; set; }
}

public interface IAnimal
{
    bool HasHair { get; set; }
}
```

```
public interface INoiseMaker
{
    string MakeNoise();
}

//Note that in C#, the base class name must come before the interface names
public class Cat : LivingBeing, IAnimal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
        HasHair = true;
    }

    public bool HasHair { get; set; }

    public string Name { get; set; }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

Section 58.5: Constructors In A Subclass

When you make a subclass of a base class, you can construct the base class by using `: base` after the subclass constructor's parameters.

```
class Instrument
{
    string type;
    bool clean;

    public Instrument (string type, bool clean)
    {
        this.type = type;
        this.clean = clean;
    }
}

class Trumpet : Instrument
{
    bool oiled;

    public Trumpet(string type, bool clean, bool oiled) : base(type, clean)
    {
        this.oiled = oiled;
    }
}
```

Section 58.6: Inheritance Anti-patterns

Improper Inheritance

Lets say there are 2 classes class `Foo` and `Bar`. `Foo` has two features `Do1` and `Do2`. `Bar` needs to use `Do1` from `Foo`, but it doesn't need `Do2` or needs feature that is equivalent to `Do2` but does something completely different.

Bad way: make `Do2()` on `Foo` virtual then override it in `Bar` or just `throw Exception` in `Bar` for `Do2()`

```
public class Bar : Foo
{
    public override void Do2()
    {
        //Does something completely different that you would expect Foo to do
        //or simply throws new Exception
    }
}
```

Good way

Take out `Do1()` from `Foo` and put it into new class `Baz` then inherit both `Foo` and `Bar` from `Baz` and implement `Do2()` separately

```
public class Baz
{
    public void Do1()
    {
        // magic
    }
}

public class Foo : Baz
{
    public void Do2()
    {
        // foo way
    }
}

public class Bar : Baz
{
    public void Do2()
    {
        // bar way or not have Do2 at all
    }
}
```

Now why first example is bad and second is good: When developer nr2 has to do a change in `Foo`, chances are he will break implementation of `Bar` because `Bar` is now inseparable from `Foo`. When doing it by latter example `Foo` and `Bar` commonalty has been moved to `Baz` and they do not affect each other (like the shouldn't).

Section 58.7: Extending an abstract base class

Unlike interfaces, which can be described as contracts for implementation, abstract classes act as contracts for extension.

An abstract class cannot be instantiated, it must be extended and the resulting class (or derived class) can then be instantiated.

Abstract classes are used to provide generic implementations

```
public abstract class Car
{
    public void HonkHorn() {
        // Implementation of horn being honked
    }
}
```

```
public class Mustang : Car
{
    // Simply by extending the abstract class Car, the Mustang can HonkHorn()
    // If Car were an interface, the HonkHorn method would need to be included
    // in every class that implemented it.
}
```

The above example shows how any class extending Car will automatically receive the HonkHorn method with the implementation. This means that any developer creating a new Car will not need to worry about how it will honk it's horn.

Section 58.8: Testing and navigating inheritance

```
interface BaseInterface {}
class BaseClass : BaseInterface {}

interface DerivedInterface {}
class DerivedClass : BaseClass, DerivedInterface {}

var baseInterfaceType = typeof(BaseInterface);
var derivedInterfaceType = typeof(DerivedInterface);
var baseType = typeof(BaseClass);
var derivedType = typeof(DerivedClass);

var baseInstance = new BaseClass();
var derivedInstance = new DerivedClass();

Console.WriteLine(derivedInstance is DerivedClass); //True
Console.WriteLine(derivedInstance is DerivedInterface); //True
Console.WriteLine(derivedInstance is BaseClass); //True
Console.WriteLine(derivedInstance is BaseInterface); //True
Console.WriteLine(derivedInstance is object); //True

Console.WriteLine(derivedType.BaseType.Name);  //BaseClass
Console.WriteLine(baseType.BaseType.Name);  //Object
Console.WriteLine(typeof(object).BaseType);  //null

Console.WriteLine(baseType.IsInstanceOfType(derivedInstance));  //True
Console.WriteLine(derivedType.IsInstanceOfType(baseInstance));  //False

Console.WriteLine(
    string.Join(",",
    derivedType.GetInterfaces().Select(t => t.Name).ToArray()));
//BaseInterface,DerivedInterface

Console.WriteLine(baseInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(baseType)); //False
```

Section 58.9: Inheriting methods

There are several ways methods can be inherited

```
public abstract class Car
{
    public void HonkHorn() {
        // Implementation of horn being honked
    }
```

```
    // virtual methods CAN be overridden in derived classes
    public virtual void ChangeGear() {
        // Implementation of gears being changed
    }

    // abstract methods MUST be overridden in derived classes
    public abstract void Accelerate();
}

public class Mustang : Car
{
    // Before any code is added to the Mustang class, it already contains
    // implementations of HonkHorn and ChangeGear.

    // In order to compile, it must be given an implementation of Accelerate,
    // this is done using the override keyword
    public override void Accelerate() {
        // Implementation of Mustang accelerating
    }

    // If the Mustang changes gears differently to the implementation in Car
    // this can be overridden using the same override keyword as above
    public override void ChangeGear() {
        // Implementation of Mustang changing gears
    }
}
```

Section 58.10: Base class with recursive type specification

One time definition of a generic base class with recursive type specifier. Each node has one parent and multiple children.

```
/// <summary>
/// Generic base class for a tree structure
/// </summary>
/// <typeparam name="T">The node type of the tree</typeparam>
public abstract class Tree<T> where T : Tree<T>
{
    /// <summary>
    /// Constructor sets the parent node and adds this node to the parent's child nodes
    /// </summary>
    /// <param name="parent">The parent node or null if a root</param>
    protected Tree(T parent)
    {
        this.Parent=parent;
        this.Children=new List<T>();
        if(parent!=null)
        {
            parent.Children.Add(this as T);
        }
    }
    public T Parent { get; private set; }
    public List<T> Children { get; private set; }
    public bool IsRoot { get { return Parent==null; } }
    public bool IsLeaf { get { return Children.Count==0; } }
    /// <summary>
    /// Returns the number of hops to the root object
    /// </summary>
    public int Level { get { return IsRoot ? 0 : Parent.Level+1; } }
}
```

The above can be re-used every time a tree hierarchy of objects needs to be defined. The node object in the tree has to inherit from the base class with

```
public class MyNode : Tree<MyNode>
{
    // stuff
}
```

each node class knows where it is in the hierarchy, what the parent object is as well as what the children objects are. Several built in types use a tree structure, like `Control` or `XmlElement` and the above `Tree<T>` can be used as a base class of *any* type in your code.

For example, to create a hierarchy of parts where the total weight is calculated from the weight of *all* the children, do the following:

```
public class Part : Tree<Part>
{
    public static readonly Part Empty = new Part(null) { Weight=0 };
    public Part(Part parent) : base(parent) { }
    public Part Add(float weight)
    {
        return new Part(this) { Weight=weight };
    }
    public float Weight { get; set; }

    public float TotalWeight { get { return Weight+Children.Sum((part) => part.TotalWeight); } }
}
```

to be used as

```
// [Q:2.5] -- [P:4.2] -- [R:0.4]
//    \
//      - [Z:0.8]
var Q = Part.Empty.Add(2.5f);
var P = Q.Add(4.2f);
var R = P.Add(0.4f);
var Z = Q.Add(0.9f);

// 2.5+(4.2+0.4)+0.9 = 8.0
float weight = Q.TotalWeight;
```

Another example would in the definition of relative coordinate frames. In this case the true position of the coordinate frame depends on the positions of *all* the parent coordinate frames.

```
public class RelativeCoordinate : Tree<RelativeCoordinate>
{
    public static readonly RelativeCoordinate Start = new RelativeCoordinate(null, PointF.Empty) {
};
    public RelativeCoordinate(RelativeCoordinate parent, PointF local_position)
        : base(parent)
    {
        this.LocalPosition=local_position;
    }
    public PointF LocalPosition { get; set; }
    public PointF GlobalPosition
    {
        get
        {
            if(IsRoot) return LocalPosition;
```

```
            var parent_pos = Parent.GlobalPosition;
            return new PointF(parent_pos.X+LocalPosition.X, parent_pos.Y+LocalPosition.Y);
        }
    }
    public float TotalDistance
    {
        get
        {
            float dist =
(float)Math.Sqrt(LocalPosition.X*LocalPosition.X+LocalPosition.Y*LocalPosition.Y);
            return IsRoot ? dist : Parent.TotalDistance+dist;
        }
    }
    public RelativeCoordinate Add(PointF local_position)
    {
        return new RelativeCoordinate(this, local_position);
    }
    public RelativeCoordinate Add(float x, float y)
    {
        return Add(new PointF(x, y));
    }
}
```

to be used as

```
// Define the following coordinate system hierarchy
//
// o--> [A1] --+--> [B1] -----> [C1]
//             |
//             +--> [B2] --+--> [C2]
//                         |
//                         +--> [C3]

var A1 = RelativeCoordinate.Start;
var B1 = A1.Add(100, 20);
var B2 = A1.Add(160, 10);

var C1 = B1.Add(120, -40);
var C2 = B2.Add(80, -20);
var C3 = B2.Add(60, -30);

double dist1 = C1.TotalDistance;
```

Chapter 59: Generics

Parameter(s)	Description
T, V	Type placeholders for generic declarations

Section 59.1: Implicit type inference (methods)

When passing formal arguments to a generic method, relevant generic type arguments can usually be inferred implicitly. If all generic type can be inferred, then specifying them in the syntax is optional.

Consider the following generic method. It has one formal parameter and one generic type parameter. There is a very obvious relationship between them -- the type passed as an argument to the generic type parameter must be the same as the compile-time type of the argument passed to the formal parameter.

```
void M<T>(T obj)
{
}
```

These two calls are equivalent:

```
M<object>(new object());
M(new object());
```

These two calls are also equivalent:

```
M<string>("");
M("");
```

And so are these three calls:

```
M<object>("");
M((object) "");
M("" as object);
```

Notice that if at least one type argument cannot be inferred, then all of them have to be specified.

Consider the following generic method. The first generic type argument is the same as the type of the formal argument. But there is no such relationship for the second generic type argument. Therefore, the compiler has no way of inferring the second generic type argument in any call to this method.

```
void X<T1, T2>(T1 obj)
{
}
```

This doesn't work anymore:

```
X("");
```

This doesn't work either, because the compiler isn't sure if we are specifying the first or the second generic parameter (both would be valid as `object`):

```
X<object>("");
```

We are required to type out both of them, like this:

```
X<string, object>("");
```

Section 59.2: Type inference (classes)

Developers can be caught out by the fact that type inference *doesn't work* for constructors:

```
class Tuple<T1,T2>
{
   public Tuple(T1 value1, T2 value2)
   {
   }
}

var x = new Tuple(2, "two");              // This WON'T work...
var y = new Tuple<int, string>(2, "two"); // even though the explicit form will.
```

The first way of creating instance without explicitly specifying type parameters will cause compile time error which would say:

> Using the generic type 'Tuple<T1, T2>' requires 2 type arguments

A common workaround is to add a helper method in a static class:

```
static class Tuple
{
    public static Tuple<T1, T2> Create<T1, T2>(T1 value1, T2 value2)
    {
         return new Tuple<T1, T2>(value1, value2);
    }
}

var x = Tuple.Create(2, "two");  // This WILL work...
```

Section 59.3: Using generic method with an interface as a constraint type

This is an example of how to use the generic type TFood inside Eat method on the class Animal

```
public interface IFood
{
    void EatenBy(Animal animal);
}

public class Grass: IFood
{
    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Grass was eaten by: {0}", animal.Name);
    }
}

public class Animal
{
    public string Name { get; set; }
```

```
    public void Eat<TFood>(TFood food)
        where TFood : IFood
    {
        food.EatenBy(this);
    }
}

public class Carnivore : Animal
{
    public Carnivore()
    {
        Name = "Carnivore";
    }
}

public class Herbivore : Animal, IFood
{
    public Herbivore()
    {
        Name = "Herbivore";
    }

    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Herbivore was eaten by: {0}", animal.Name);
    }
}
```

You can call the Eat method like this:

```
var grass = new Grass();
var sheep = new Herbivore();
var lion = new Carnivore();

sheep.Eat(grass);
//Output: Grass was eaten by: Herbivore

lion.Eat(sheep);
//Output: Herbivore was eaten by: Carnivore
```

In this case if you try to call:

```
sheep.Eat(lion);
```

It won't be possible because the object lion does not implement the interface IFood. Attempting to make the above call will generate a compiler error: "The type 'Carnivore' cannot be used as type parameter 'TFood' in the generic type or method 'Animal.Eat(TFood)'. There is no implicit reference conversion from 'Carnivore' to 'IFood'."

Section 59.4: Type constraints (new-keyword)

By using the `new()` constraint, it is possible to enforce type parameters to define an empty (default) constructor.

```
class Foo
{
    public Foo () { }
}

class Bar
{
```

```
    public Bar (string s) { ... }
}

class Factory<T>
    where T : new()
{
    public T Create()
    {
        return new T();
    }
}

Foo f = new Factory<Foo>().Create(); // Valid.
Bar b = new Factory<Bar>().Create(); // Invalid, Bar does not define a default/empty constructor.
```

The second call to to `Create()` will give compile time error with following message:

> 'Bar' must be a non-abstract type with a public parameterless constructor in order to use it as parameter 'T' in the generic type or method 'Factory'

There is no constraint for a constructor with parameters, only parameterless constructors are supported.

Section 59.5: Type constraints (classes and interfaces)

Type constraints are able to force a type parameter to implement a certain interface or class.

```
interface IType;
interface IAnotherType;

// T must be a subtype of IType
interface IGeneric<T>
    where T : IType
{
}

// T must be a subtype of IType
class Generic<T>
    where T : IType
{
}

class NonGeneric
{
    // T must be a subtype of IType
    public void DoSomething<T>(T arg)
        where T : IType
    {
    }
}

// Valid definitions and expressions:
class Type : IType { }
class Sub : IGeneric<Type> { }
class Sub : Generic<Type> { }
new NonGeneric().DoSomething(new Type());

// Invalid definitions and expressions:
class AnotherType : IAnotherType { }
```

```
class Sub : IGeneric<AnotherType> { }
class Sub : Generic<AnotherType> { }
new NonGeneric().DoSomething(new AnotherType());
```

Syntax for multiple constraints:

```
class Generic<T, T1>
    where T : IType
    where T1 : Base, new()
{
}
```

Type constraints works in the same way as inheritance, in that it is possible to specify multiple interfaces as constraints on the generic type, but only one class:

```
class A { /* ... */ }
class B { /* ... */ }

interface I1 { }
interface I2 { }

class Generic<T>
    where T : A, I1, I2
{
}

class Generic2<T>
    where T : A, B //Compilation error
{
}
```

Another rule is that the class must be added as the first constraint and then the interfaces:

```
class Generic<T>
    where T : A, I1
{
}

class Generic2<T>
    where T : I1, A //Compilation error
{
}
```

All declared constraints must be satisfied simultaneously for a particular generic instantiation to work. There is no way to specify two or more alternative sets of constraints.

Section 59.6: Checking equality of generic values

If logic of generic class or method requires checking equality of values having generic type, use `EqualityComparer<TType>.Default` property:

```
public void Foo<TBar>(TBar arg1, TBar arg2)
{
    var comparer = EqualityComparer<TBar>.Default;
    if (comparer.Equals(arg1,arg2)
    {
        ...
    }
```

```
}
```

This approach is better than simply calling `Object.Equals()` method, because default comparer implementation checks, whether `TBar` type implements `IEquatale<TBar>` interface and if yes, calls `IEquatable<TBar>.Equals(TBar other)` method. This allows to avoid boxing/unboxing of value types.

Section 59.7: Reflecting on type parameters

The `typeof` operator works on type parameters.

```
class NameGetter<T>
{
    public string GetTypeName()
    {
        return typeof(T).Name;
    }
}
```

Section 59.8: Covariance

When is an `IEnumerable<T>` a subtype of a different `IEnumerable<T1>`? When `T` is a subtype of `T1`. `IEnumerable` is *covariant* in its `T` parameter, which means that `IEnumerable`'s subtype relationship goes in *the same direction* as `T`'s.

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IEnumerable<Dog> dogs = Enumerable.Empty<Dog>();
IEnumerable<Animal> animals = dogs;  // IEnumerable<Dog> is a subtype of IEnumerable<Animal>
// dogs = animals;  // Compilation error - IEnumerable<Animal> is not a subtype of IEnumerable<Dog>
```

An instance of a covariant generic type with a given type parameter is implicitly convertible to the same generic type with a less derived type parameter.

This relationship holds because `IEnumerable` *produces* `T`s but doesn't consume them. An object that produces `Dog`s can be used as if it produces `Animal`s.

Covariant type parameters are declared using the `out` keyword, because the parameter must be used only as an *output*.

```
interface IEnumerable<out T> { /* ... */ }
```

A type parameter declared as covariant may not appear as an input.

```
interface Bad<out T>
{
    void SetT(T t);  // type error
}
```

Here's a complete example:

```
using NUnit.Framework;

namespace ToyStore
{
   enum Taste { Bitter, Sweet };
```

```
   interface IWidget
   {
      int Weight { get; }
   }

   interface IFactory<out TWidget>
       where TWidget : IWidget
   {
      TWidget Create();
   }

   class Toy : IWidget
   {
      public int Weight { get; set; }
      public Taste Taste { get; set; }
   }

   class ToyFactory : IFactory<Toy>
   {
      public const int StandardWeight = 100;
      public const Taste StandardTaste = Taste.Sweet;

      public Toy Create() { return new Toy { Weight = StandardWeight, Taste = StandardTaste }; }
   }

   [TestFixture]
   public class GivenAToyFactory
   {
      [Test]
      public static void WhenUsingToyFactoryToMakeWidgets()
      {
         var toyFactory = new ToyFactory();

         //// Without out keyword, note the verbose explicit cast:
         // IFactory<IWidget> rustBeltFactory = (IFactory<IWidget>)toyFactory;

         // covariance: concrete being assigned to abstract (shiny and new)
         IFactory<IWidget> widgetFactory = toyFactory;
         IWidget anotherToy = widgetFactory.Create();
         Assert.That(anotherToy.Weight, Is.EqualTo(ToyFactory.StandardWeight)); // abstract
contract
         Assert.That(((Toy)anotherToy).Taste, Is.EqualTo(ToyFactory.StandardTaste)); // concrete
contract
      }
   }
}
```

Section 59.9: Contravariance

When is an `IComparer<T>` a subtype of a different `IComparer<T1>`? When `T1` is a subtype of `T`. `IComparer` is *contravariant* in its `T` parameter, which means that `IComparer`'s subtype relationship goes in the *opposite direction* as `T`'s.

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IComparer<Animal> animalComparer = /* ... */;
IComparer<Dog> dogComparer = animalComparer;  // IComparer<Animal> is a subtype of IComparer<Dog>
// animalComparer = dogComparer;  // Compilation error - IComparer<Dog> is not a subtype of
IComparer<Animal>
```

An instance of a contravariant generic type with a given type parameter is implicitly convertible to the same generic type with a more derived type parameter.

This relationship holds because `IComparer` *consumes* `T`s but doesn't produce them. An object which can compare any two `Animal`s can be used to compare two `Dog`s.

Contravariant type parameters are declared using the `in` keyword, because the parameter must be used only as an *input*.

```
interface IComparer<in T> { /* ... */ }
```

A type parameter declared as contravariant may not appear as an output.

```
interface Bad<in T>
{
    T GetT();  // type error
}
```

Section 59.10: Invariance

`IList<T>` is never a subtype of a different `IList<T1>`. `IList` is *invariant* in its type parameter.

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IList<Dog> dogs = new List<Dog>();
IList<Animal> animals = dogs;  // type error
```

There is no subtype relationship for lists because you can put values into a list *and* take values out of a list.

If `IList` was covariant, you'd be able to add items of the *wrong subtype* to a given list.

```
IList<Animal> animals = new List<Dog>();  // supposing this were allowed...
animals.Add(new Giraffe());  // ... then this would also be allowed, which is bad!
```

If `IList` was contravariant, you'd be able to extract values of the wrong subtype from a given list.

```
IList<Dog> dogs = new List<Animal> { new Dog(), new Giraffe() };  // if this were allowed...
Dog dog = dogs[1];  // ... then this would be allowed, which is bad!
```

Invariant type parameters are declared by omitting both the `in` and `out` keywords.

```
interface IList<T> { /* ... */ }
```

Section 59.11: Variant interfaces

Interfaces may have variant type parameters.

```
interface IEnumerable<out T>
{
    // ...
}
interface IComparer<in T>
{
    // ...
```

```
}
```

but classes and structures may not

```
class BadClass<in T1, out T2>  // not allowed
{
}

struct BadStruct<in T1, out T2>  // not allowed
{
}
```

nor do generic method declarations

```
class MyClass
{
    public T Bad<out T, in T1>(T1 t1)  // not allowed
    {
        // ...
    }
}
```

The example below shows multiple variance declarations on the same interface

```
interface IFoo<in T1, out T2, T3>
//  T1 : Contravariant type
//  T2 : Covariant type
//  T3 : Invariant type
{
    // ...
}

IFoo<Animal, Dog, int> foo1 = /* ... */;
IFoo<Dog, Animal, int> foo2 = foo1;
// IFoo<Animal, Dog, int> is a subtype of IFoo<Dog, Animal, int>
```

Section 59.12: Variant delegates

Delegates may have variant type parameters.

```
delegate void Action<in T>(T t);    // T is an input
delegate T Func<out T>();           // T is an output
delegate T2 Func<in T1, out T2>();  // T1 is an input, T2 is an output
```

This follows from the Liskov Substitution Principle, which states (among other things) that a method D can be considered more derived than a method B if:

- D has an equal or more derived return type than B
- D has equal or more general corresponding parameter types than B

Therefore the following assignments are all type safe:

```
Func<object, string> original = SomeMethod;
Func<object, object> d1 = original;
Func<string, string> d2 = original;
Func<string, object> d3 = original;
```

Section 59.13: Variant types as parameters and return values

If a covariant type appears as an output, the containing type is covariant. Producing a producer of `T`s is like producing `T`s.

```
interface IReturnCovariant<out T>
{
    IEnumerable<T> GetTs();
}
```

If a contravariant type appears as an output, the containing type is contravariant. Producing a consumer of `T`s is like consuming `T`s.

```
interface IReturnContravariant<in T>
{
    IComparer<T> GetTComparer();
}
```

If a covariant type appears as an input, the containing type is contravariant. Consuming a producer of `T`s is like consuming `T`s.

```
interface IAcceptCovariant<in T>
{
    void ProcessTs(IEnumerable<T> ts);
}
```

If a contravariant type appears as an input, the containing type is covariant. Consuming a consumer of `T`s is like producing `T`s.

```
interface IAcceptContravariant<out T>
{
    void CompareTs(IComparer<T> tComparer);
}
```

Section 59.14: Type Parameters (Interfaces)

Declaration:

```
interface IMyGenericInterface<T1, T2, T3, ...> { ... }
```

Usage (in inheritance):

```
class ClassA<T1, T2, T3> : IMyGenericInterface<T1, T2, T3> { ... }

class ClassB<T1, T2> : IMyGenericInterface<T1, T2, int> { ... }

class ClassC<T1> : IMyGenericInterface<T1, char, int> { ... }

class ClassD : IMyGenericInterface<bool, char, int> { ... }
```

Usage (as the type of a parameter):

```
void SomeMethod(IMyGenericInterface<int, char, bool> arg) { ... }
```

Section 59.15: Type constraints (class and struct)

It is possible to specify whether or not the type argument should be a reference type or a value type by using the respective constraints `class` or `struct`. If these constraints are used, they *must* be defined *before all* other constraints (for example a parent type or `new()`) can be listed.

```
// TRef must be a reference type, the use of Int32, Single, etc. is invalid.
// Interfaces are valid, as they are reference types
class AcceptsRefType<TRef>
    where TRef : class
{
    // TStruct must be a value type.
    public void AcceptStruct<TStruct>()
        where TStruct : struct
    {
    }

    // If multiple constraints are used along with class/struct
    // then the class or struct constraint MUST be specified first
    public void Foo<TComparableClass>()
        where TComparableClass : class, IComparable
    {
    }
}
```

Section 59.16: Explicit type parameters

There are different cases where you must Explicitly specify the type parameters for a generic method. In both of the below cases, the compiler is not able to infer all of the type parameters from the specified method parameters.

One case is when there are no parameters:

```
public void SomeMethod<T, V>()
{
   // No code for simplicity
}

SomeMethod(); // doesn't compile
SomeMethod<int, bool>(); // compiles
```

Second case is when one (or more) of the type parameters is not part of the method parameters:

```
public K SomeMethod<K, V>(V input)
{
    return default(K);
}

int num1 = SomeMethod(3); // doesn't compile
int num2 = SomeMethod<int>("3"); // doesn't compile
int num3 = SomeMethod<int, string>("3"); // compiles.
```

Section 59.17: Type Parameters (Classes)

Declaration:

```
class MyGenericClass<T1, T2, T3, ...>
{
```

```
    // Do something with the type parameters.
}
```

Initialisation:

```
var x = new MyGenericClass<int, char, bool>();
```

Usage (as the type of a parameter):

```
void AnotherMethod(MyGenericClass<float, byte, char> arg) { ... }
```

Section 59.18: Type Parameters (Methods)

Declaration:

```
void MyGenericMethod<T1, T2, T3>(T1 a, T2 b, T3 c)
{
    // Do something with the type parameters.
}
```

Invocation:

There is no need to supply type arguements to a genric method, because the compiler can implicitly infer the type.

```
int x =10;
int y =20;
string z = "test";
MyGenericMethod(x,y,z);
```

However, if there is an ambiguity, generic methods need to be called with type arguemnts as

```
MyGenericMethod<int, int, string>(x,y,z);
```

Section 59.19: Generic type casting

```
    /// <summary>
    /// Converts a data type to another data type.
    /// </summary>
    public static class Cast
    {
        /// <summary>
        /// Converts input to Type of default value or given as typeparam T
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it could be
any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="input">Input that need to be converted to specified type</param>
        /// <param name="defaultValue">defaultValue will be returned in case of value is null or any
exception occures</param>
        /// <returns>Input is converted in Type of default value or given as typeparam T and
returned</returns>
        public static T To<T>(object input, T defaultValue)
        {
            var result = defaultValue;
            try
            {
                if (input == null || input == DBNull.Value) return result;
                if (typeof (T).IsEnum)
```

```
                {
                    result = (T) Enum.ToObject(typeof (T), To(input,
Convert.ToInt32(defaultValue)));
                }
                else
                {
                    result = (T) Convert.ChangeType(input, typeof (T));
                }
            }
            catch (Exception ex)
            {
                Tracer.Current.LogException(ex);
            }

            return result;
        }

        /// <summary>
        /// Converts input to Type of typeparam T
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it could be
any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="input">Input that need to be converted to specified type</param>
        /// <returns>Input is converted in Type of default value or given as typeparam T and
returned</returns>
        public static T To<T>(object input)
        {
            return To(input, default(T));
        }



    }
```

Usages:

```
std.Name = Cast.To<string>(drConnection["Name"]);
std.Age = Cast.To<int>(drConnection["Age"]);
std.IsPassed = Cast.To<bool>(drConnection["IsPassed"]);


// Casting type using default value
//Following both ways are correct
// Way 1 (In following style input is converted into type of default value)
std.Name = Cast.To(drConnection["Name"], "");
std.Marks = Cast.To(drConnection["Marks"], 0);
// Way 2
std.Name = Cast.To<string>(drConnection["Name"], "");
std.Marks = Cast.To<int>(drConnection["Marks"], 0);
```

Section 59.20: Configuration reader with generic type casting

```
    /// <summary>
    /// Read configuration values from app.config and convert to specified types
    /// </summary>
    public static class ConfigurationReader
    {
        /// <summary>
        /// Get value from AppSettings by key, convert to Type of default value or typeparam T and
return
```

```
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it could be
any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="strKey">key to find value from AppSettings</param>
        /// <param name="defaultValue">defaultValue will be returned in case of value is null or any
exception occures</param>
        /// <returns>AppSettings value against key is returned in Type of default value or given as
typeparam T</returns>
        public static T GetConfigKeyValue<T>(string strKey, T defaultValue)
        {
            var result = defaultValue;
            try
            {
                if (ConfigurationManager.AppSettings[strKey] != null)
                    result = (T)Convert.ChangeType(ConfigurationManager.AppSettings[strKey],
typeof(T));
            }
            catch (Exception ex)
            {
                Tracer.Current.LogException(ex);
            }

            return result;
        }
        /// <summary>
        /// Get value from AppSettings by key, convert to Type of default value or typeparam T and
return
        /// </summary>
        /// <typeparam name="T">typeparam is the type in which value will be returned, it could be
any type eg. int, string, bool, decimal etc.</typeparam>
        /// <param name="strKey">key to find value from AppSettings</param>
        /// <returns>AppSettings value against key is returned in Type given as typeparam T</returns>
        public static T GetConfigKeyValue<T>(string strKey)
        {
            return GetConfigKeyValue(strKey, default(T));
        }

    }
```

Usages:

```
var timeOut = ConfigurationReader.GetConfigKeyValue("RequestTimeout", 2000);
var url = ConfigurationReader.GetConfigKeyValue("URL", "www.someurl.com");
var enabled = ConfigurationReader.GetConfigKeyValue("IsEnabled", false);
```

Chapter 60: Using Statement

Provides a convenient syntax that ensures the correct use of IDisposable objects.

Section 60.1: Using Statement Basics

`using` is syntactic sugar that allows you to guarantee that a resource is cleaned up without needing an explicit `try-finally` block. This means your code will be much cleaner, and you won't leak non-managed resources.

Standard `Dispose` cleanup pattern, for objects that implement the `IDisposable` interface (which the `FileStream`'s base class `Stream` does in .NET):

```
int Foo()
{
    var fileName = "file.txt";

    {
        FileStream disposable = null;

        try
        {
            disposable = File.Open(fileName, FileMode.Open);

            return disposable.ReadByte();
        }
        finally
        {
            // finally blocks are always run
            if (disposable != null) disposable.Dispose();
        }
    }
}
```

`using` simplifies your syntax by hiding the explicit `try-finally`:

```
int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        return disposable.ReadByte();
    }
    // disposable.Dispose is called even if we return earlier
}
```

Just like `finally` blocks always execute regardless of errors or returns, `using` always calls `Dispose()`, even in the event of an error:

```
int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        throw new InvalidOperationException();
    }
    // disposable.Dispose is called even if we throw an exception earlier
```

```
}
```

Note: Since `Dispose` is guaranteed to be called irrespective of the code flow, it's a good idea to make sure that `Dispose` never throws an exception when you implement `IDisposable`. Otherwise an actual exception would get overridden by the new exception resulting in a debugging nightmare.

Returning from using block

```
using ( var disposable = new DisposableItem() )
{
    return disposable.SomeProperty;
}
```

Because of the semantics of `try..finally` to which the `using` block translates, the `return` statement works as expected - the return value is evaluated before `finally` block is executed and the value disposed. The order of evaluation is as follows:

1. Evaluate the `try` body
2. Evaluate and cache the returned value
3. Execute finally block
4. Return the cached return value

However, you may not return the variable `disposable` itself, as it would contain invalid, disposed reference - see related example.

Section 60.2: Gotcha: returning the resource which you are disposing

The following is a bad idea because it would dispose the `db` variable before returning it.

```
public IDBContext GetDBContext()
{
    using (var db = new DBContext())
    {
        return db;
    }
}
```

This can also create more subtle mistakes:

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DBContext())
    {
        return db.Persons.Where(p => p.Age == age);
    }
}
```

This looks ok, but the catch is that the LINQ expression evaluation is lazy, and will possibly only be executed later when the underlying `DBContext` has already been disposed.

So in short the expression isn't evaluated before leaving the `using`. One possible solution to this problem, which still makes use of `using`, is to cause the expression to evaluate immediately by calling a method that will enumerate the result. For example `ToList()`, `ToArray()`, etc. If you are using the newest version of Entity Framework you could use the `async` counterparts like `ToListAsync()` or `ToArrayAsync()`.

Below you find the example in action:

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DBContext())
    {
        return db.Persons.Where(p => p.Age == age).ToList();
    }
}
```

It is important to note, though, that by calling `ToList()` or `ToArray()`, the expression will be eagerly evaluated, meaning that all the persons with the specified age will be loaded to memory even if you do not iterate on them.

Section 60.3: Multiple using statements with one block

It is possible to use multiple nested `using` statements without added multiple levels of nested braces. For example:

```
using (var input = File.OpenRead("input.txt"))
{
    using (var output = File.OpenWrite("output.txt"))
    {
        input.CopyTo(output);
    } // output is disposed here
} // input is disposed here
```

An alternative is to write:

```
using (var input = File.OpenRead("input.txt"))
using (var output = File.OpenWrite("output.txt"))
{
    input.CopyTo(output);
} // output and then input are disposed here
```

Which is exactly equivalent to the first example.

Note: Nested `using` statements might trigger Microsoft Code Analysis rule CS2002 (see this answer for clarification) and generate a warning. As explained in the linked answer, it is generally safe to nest `using` statements.

When the types within the `using` statement are of the same type you can comma-delimit them and specify the type only once (though this is uncommon):

```
using (FileStream file = File.Open("MyFile.txt"), file2 = File.Open("MyFile2.txt"))
{
}
```

This can also be used when the types have a shared hierarchy:

```
using (Stream file = File.Open("MyFile.txt"), data = new MemoryStream())
{
}
```

The `var` keyword *cannot* be used in the above example. A compilation error would occur. Even the comma separated declaration won't work when the declared variables have types from different hierarchies.

Section 60.4: Gotcha: Exception in Dispose method masking other errors in Using blocks

Consider the following block of code.

```
try
{
    using (var disposable = new MyDisposable())
    {
        throw new Exception("Couldn't perform operation.");
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

class MyDisposable : IDisposable
{
    public void Dispose()
    {
        throw new Exception("Couldn't dispose successfully.");
    }
}
```

You may expect to see "Couldn't perform operation" printed to the Console but you would actually see "Couldn't dispose successfully." as the Dispose method is still called even after the first exception is thrown.

It is worth being aware of this subtlety as it may be masking the real error that prevented the object from being disposed and make it harder to debug.

Section 60.5: Using statements are null-safe

You don't have to check the `IDisposable` object for `null`. `using` will not throw an exception and `Dispose()` will not be called:

```
DisposableObject TryOpenFile()
{
    return null;
}

// disposable is null here, but this does not throw an exception
using (var disposable = TryOpenFile())
{
    // this will throw a NullReferenceException because disposable is null
    disposable.DoSomething();

    if(disposable != null)
    {
        // here we are safe because disposable has been checked for null
        disposable.DoSomething();
    }
}
```

Section 60.6: Using Dispose Syntax to define custom scope

For some use cases, you can use the `using` syntax to help define a custom scope. For example, you can define the following class to execute code in a specific culture.

```
public class CultureContext : IDisposable
{
    private readonly CultureInfo originalCulture;

    public CultureContext(string culture)
    {
        originalCulture = CultureInfo.CurrentCulture;
        Thread.CurrentThread.CurrentCulture = new CultureInfo(culture);
    }

    public void Dispose()
    {
        Thread.CurrentThread.CurrentCulture = originalCulture;
    }
}
```

You can then use use this class to define blocks of code that execute in a specific culture.

```
Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");

using (new CultureContext("nl-NL"))
{
    // Code in this block uses the "nl-NL" culture
    Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 25-12-2016 00:00:00
}

using (new CultureContext("es-ES"))
{
    // Code in this block uses the "es-ES" culture
    Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 25/12/2016 0:00:00
}

// Reverted back to the original culture
Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 12/25/2016 12:00:00 AM
```

Note: as we don't use the `CultureContext` instance we create, we don't assign a variable for it.

This technique is used by the `BeginForm` helper in ASP.NET MVC.

Section 60.7: Using Statements and Database Connections

The `using` keyword ensures that the resource defined within the statement only exists within the scope of the statement itself. Any resources defined within the statement must implement the `IDisposable` interface.

These are incredibly important when dealing with any connections that implement the `IDisposable` interface as it can ensure the connections are not only properly closed but that their resources are freed after the `using` statement is out of scope.

Common `IDisposable` Data Classes

Many of the following are data-related classes that implement the `IDisposable` interface and are perfect candidates for a `using` statement :

- `SqlConnection`,`SqlCommand`,`SqlDataReader`, etc.
- `OleDbConnection`,`OleDbCommand`,`OleDbDataReader`, etc.
- `MySqlConnection`, `MySqlCommand`, `MySqlDbDataReader`, etc.
- `DbContext`

All of these are commonly used to access data through C# and will be commonly encountered throughout building data-centric applications. Many other classes that are not mentioned that implement the same `FooConnection`,`FooCommand`,`FooDataReader` classes can be expected to behave the same way.

Common Access Pattern for ADO.NET Connections

A common pattern that can be used when accessing your data through an ADO.NET connection might look as follows :

```
// This scopes the connection (your specific class may vary)
using(var connection = new SqlConnection("{your-connection-string}")
{
    // Build your query
    var query = "SELECT * FROM YourTable WHERE Property = @property");
    // Scope your command to execute
    using(var command = new SqlCommand(query, connection))
    {
         // Open your connection
         connection.Open();

         // Add your parameters here if necessary

         // Execute your query as a reader (again scoped with a using statement)
         using(var reader = command.ExecuteReader())
         {
               // Iterate through your results here
         }
    }
}
```

Or if you were just performing a simple update and didn't require a reader, the same basic concept would apply :

```
using(var connection = new SqlConnection("{your-connection-string}"))
{
     var query = "UPDATE YourTable SET Property = Value WHERE Foo = @foo";
     using(var command = new SqlCommand(query,connection))
     {
          connection.Open();

          // Add parameters here

          // Perform your update
          command.ExecuteNonQuery();
     }
}
```

Using Statements with DataContexts

Many ORMs such as Entity Framework expose abstraction classes that are used to interact with underlying databases in the form of classes like `DbContext`. These contexts generally implement the `IDisposable` interface as well and should take advantage of this through `using` statements when possible :

```
using(var context = new YourDbContext())
{
      // Access your context and perform your query
      var data = context.Widgets.ToList();
}
```

Section 60.8: Executing code in constraint context

If you have code (a *routine*) you want to execute under a specific (constraint) context, you can use dependency injection.

The following example shows the constraint of executing under an open SSL connection. This first part would be in your library or framework, which you won't expose to the client code.

```
public static class SSLContext
{
    // define the delegate to inject
    public delegate void TunnelRoutine(BinaryReader sslReader, BinaryWriter sslWriter);

    // this allows the routine to be executed under SSL
    public static void ClientTunnel(TcpClient tcpClient, TunnelRoutine routine)
    {
        using (SslStream sslStream = new SslStream(tcpClient.GetStream(), true, _validate))
        {
            sslStream.AuthenticateAsClient(HOSTNAME, null, SslProtocols.Tls, false);

            if (!sslStream.IsAuthenticated)
            {
                throw new SecurityException("SSL tunnel not authenticated");
            }

            if (!sslStream.IsEncrypted)
            {
                throw new SecurityException("SSL tunnel not encrypted");
            }

            using (BinaryReader sslReader = new BinaryReader(sslStream))
            using (BinaryWriter sslWriter = new BinaryWriter(sslStream))
            {
                routine(sslReader, sslWriter);
            }
        }
    }
}
```

Now the client code which wants to do something under SSL but does not want to handle all the SSL details. You can now do whatever you want inside the SSL tunnel, for example exchange a symmetric key:

```
public void ExchangeSymmetricKey(BinaryReader sslReader, BinaryWriter sslWriter)
{
    byte[] bytes = new byte[8];
    (new RNGCryptoServiceProvider()).GetNonZeroBytes(bytes);
    sslWriter.Write(BitConverter.ToUInt64(bytes, 0));
}
```

You execute this routine as follows:

```
SSLContext.ClientTunnel(tcpClient, this.ExchangeSymmetricKey);
```

To do this, you need the `using()` clause because it is the only way (apart from a `try..finally` block) you can guarantee the client code (`ExchangeSymmetricKey`) never exits without properly disposing of the disposable resources. Without `using()` clause, you would never know if a routine could break the context's constraint to dispose of those resources.

Chapter 61: Using Directive

Section 61.1: Associate an Alias to Resolve Conflicts

If you are using multiple namespaces that may have same-name classes(such as `System.Random` and `UnityEngine.Random`), you can use an alias to specify that `Random` comes from one or the other without having to use the entire namespace in the call.

For instance:

```
using UnityEngine;
using System;

Random rnd = new Random();
```

This will cause the compiler to be unsure which `Random` to evaluate the new variable as. Instead, you can do:

```
using UnityEngine;
using System;
using Random = System.Random;

Random rnd = new Random();
```

This doesn't preclude you from calling the other by it's fully qualified namespace, like this:

```
using UnityEngine;
using System;
using Random = System.Random;

Random rnd = new Random();
int unityRandom = UnityEngine.Random.Range(0,100);
```

`rnd` will be a `System.Random` variable and `unityRandom` will be a `UnityEngine.Random` variable.

Section 61.2: Using alias directives

You can use `using` in order to set an alias for a namespace or type. More detail can be found in here.

Syntax:

```
using <identifier> = <namespace-or-type-name>;
```

Example:

```
using NewType = Dictionary<string, Dictionary<string,int>>;
NewType multiDictionary = new NewType();
//Use instances as you are using the original one
multiDictionary.Add("test", new Dictionary<string,int>());
```

Section 61.3: Access Static Members of a Class

`Version ≥ 6.0`

Allows you to import a specific type and use the type's static members without qualifying them with the type name. This shows an example using static methods:

```
using static System.Console;

// ...

string GetName()
{
    WriteLine("Enter your name.");
    return ReadLine();
}
```

And this shows an example using static properties and methods:

```
using static System.Math;

namespace Geometry
{
    public class Circle
    {
        public double Radius { get; set; };

        public double Area => PI * Pow(Radius, 2);
    }
}
```

Section 61.4: Basic Usage

```
using System;
using BasicStuff = System;
using Sayer = System.Console;
using static System.Console;  //From C# 6

class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Ignoring usings and specifying full type name");
        Console.WriteLine("Thanks to the 'using System' directive");
        BasicStuff.Console.WriteLine("Namespace aliasing");
        Sayer.WriteLine("Type aliasing");
        WriteLine("Thanks to the 'using static' directive (from C# 6)");
    }
}
```

Section 61.5: Reference a Namespace

```
using System.Text;
//allows you to access classes within this namespace such as StringBuilder
//without prefixing them with the namespace.  i.e:

//...
var sb = new StringBuilder();
//instead of
var sb = new System.Text.StringBuilder();
```

Section 61.6: Associate an Alias with a Namespace

```
using st = System.Text;
//allows you to access classes within this namespace such as StringBuilder
```

```
//prefixing them with only the defined alias and not the full namespace.  i.e:

//...
var sb = new st.StringBuilder();
//instead of
var sb = new System.Text.StringBuilder();
```

Chapter 62: IDisposable interface

Section 62.1: In a class that contains only managed resources

Managed resources are resources that the runtime's garbage collector is aware and under control of. There are many classes available in the BCL, for example, such as a `SqlConnection` that is a wrapper class for an unmanaged resource. These classes already implement the `IDisposable` interface -- it's up to your code to clean them up when you are done.

It's not necessary to implement a finalizer if your class only contains managed resources.

```
public class ObjectWithManagedResourcesOnly : IDisposable
{
    private SqlConnection sqlConnection = new SqlConnection();

    public void Dispose()
    {
        sqlConnection.Dispose();
    }
}
```

Section 62.2: In a class with managed and unmanaged resources

It's important to let finalization ignore managed resources. The finalizer runs on another thread -- it's possible that the managed objects don't exist anymore by the time the finalizer runs. Implementing a protected `Dispose(bool)` method is a common practice to ensure managed resources do not have their `Dispose` method called from a finalizer.

```
public class ManagedAndUnmanagedObject : IDisposable
{
    private SqlConnection sqlConnection = new SqlConnection();
    private UnmanagedHandle unmanagedHandle = Win32.SomeUnmanagedResource();
    private bool disposed;

    public void Dispose()
    {
        Dispose(true); // client called dispose
        GC.SuppressFinalize(this); // tell the GC to not execute the Finalizer
    }

    protected virtual void Dispose(bool disposeManaged)
    {
        if (!disposed)
        {
            if (disposeManaged)
            {
                if (sqlConnection != null)
                {
                    sqlConnection.Dispose();
                }
            }

            unmanagedHandle.Release();

            disposed = true;
        }
```

```
    }

    ~ManagedAndUnmanagedObject()
    {
        Dispose(false);
    }
}
```

Section 62.3: IDisposable, Dispose

.NET Framework defines a interface for types requiring a tear-down method:

```
public interface IDisposable
{
  void Dispose();
}
```

`Dispose()` is primarily used for cleaning up resources, like unmanaged references. However, it can also be useful to force the disposing of other resources even though they are managed. Instead of waiting for the GC to eventually also clean up your database connection, you can make sure it's done in your own `Dispose()` implementation.

```
public void Dispose()
{
   if (null != this.CurrentDatabaseConnection)
   {
       this.CurrentDatabaseConnection.Dispose();
       this.CurrentDatabaseConnection = null;
   }
}
```

When you need to directly access unmanaged resources such as unmanaged pointers or win32 resources, create a class inheriting from `SafeHandle` and use that class's conventions/tools to do so.

Section 62.4: using keyword

When an object implements the `IDisposable` interface, it can be created within the `using` syntax:

```
using (var foo = new Foo())
{
    // do foo stuff
} // when it reaches here foo.Dispose() will get called

public class Foo : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("dispose called");
    }
}
```

View demo

`using` is syntatic sugar for a `try/finally` block; the above usage would roughly translate into:

```
{
    var foo = new Foo();
    try
```

```
    {
        // do foo stuff
    }
    finally
    {
        if (foo != null)
            ((IDisposable)foo).Dispose();
    }
}
```

Section 62.5: In an inherited class with managed resources

It's fairly common that you may create a class that implements `IDisposable`, and then derive classes that also contain managed resources. It is recommendeded to mark the `Dispose` method with the `virtual` keyword so that clients have the ability to cleanup any resources they may own.

```
public class Parent : IDisposable
{
    private ManagedResource parentManagedResource = new ManagedResource();

    public virtual void Dispose()
    {
        if (parentManagedResource != null)
        {
            parentManagedResource.Dispose();
        }
    }
}

public class Child : Parent
{
    private ManagedResource childManagedResource = new ManagedResource();

    public override void Dispose()
    {
        if (childManagedResource != null)
        {
            childManagedResource.Dispose();
        }
        //clean up the parent's resources
        base.Dispose();
    }
}
```

Chapter 63: Reflection

Reflection is a C# language mechanism for accessing dynamic object properties on runtime. Typically, reflection is used to fetch the information about dynamic object type and object attribute values. In REST application, for example, reflection could be used to iterate through serialized response object.

Remark: According to MS guidelines performance critical code should avoid reflection. See https://msdn.microsoft.com/en-us/library/ff647790.aspx

Section 63.1: Get the members of a type

```
using System;
using System.Reflection;
using System.Linq;

public class Program
{
  public static void Main()
  {
    var members = typeof(object)
                  .GetMembers(BindingFlags.Public |
                              BindingFlags.Static |
                              BindingFlags.Instance);

    foreach (var member in members)
    {
      bool inherited = member.DeclaringType.Equals( typeof(object).Name );
      Console.WriteLine($"{member.Name} is a {member.MemberType}, " +
                        $"it has {(inherited ? "":"not")} been inherited.");
    }
  }
}
```

Output (*see note about output order further down*):

GetType is a Method, it has not been inherited. GetHashCode is a Method, it has not been inherited. ToString is a Method, it has not been inherited. Equals is a Method, it has not been inherited. Equals is a Method, it has not been inherited. ReferenceEquals is a Method, it has not been inherited. .ctor is a Constructor, it has not been inherited.

We can also use the `GetMembers()` without passing any `BindingFlags`. This will return *all* public members of that specific type.

One thing to note that `GetMembers` does not return the members in any particular order, so never rely on the order that `GetMembers` returns you.

View Demo

Section 63.2: Get a method and invoke it

Get Instance method and invoke it

```
using System;

public class Program
{
    public static void Main()
```

```
    {
        var theString = "hello";
        var method = theString
                     .GetType()
                     .GetMethod("Substring",
                                new[] {typeof(int), typeof(int)}); //The types of the method
arguments
         var result = method.Invoke(theString, new object[] {0, 4});
         Console.WriteLine(result);
    }
}
```

Output:

> hell

View Demo

Get Static method and invoke it

On the other hand, if the method is static, you do not need an instance to call it.

```
var method = typeof(Math).GetMethod("Exp");
var result = method.Invoke(null, new object[] {2});//Pass null as the first argument (no need for an
instance)
Console.WriteLine(result); //You'll get e^2
```

Output:

> 7.38905609893065

View Demo

Section 63.3: Creating an instance of a Type

The simplest way is to use the `Activator` class.

However, even though `Activator` performance have been improved since .NET 3.5, using `Activator.CreateInstance()` is bad option sometimes, due to (relatively) low performance: Test 1, Test 2, Test 3...

With `Activator` class

```
Type type = typeof(BigInteger);
object result = Activator.CreateInstance(type); //Requires parameterless constructor.
Console.WriteLine(result); //Output: 0
result = Activator.CreateInstance(type, 123); //Requires a constructor which can receive an 'int'
compatible argument.
Console.WriteLine(result); //Output: 123
```

You can pass an object array to `Activator.CreateInstance` if you have more than one parameter.

```
// With a constructor such as MyClass(int, int, string)
Activator.CreateInstance(typeof(MyClass), new object[] { 1, 2, "Hello World" });
```

```
Type type = typeof(someObject);
var instance = Activator.CreateInstance(type);
```

For a generic type

The `MakeGenericType` method turns an open generic type (like `List<>`) into a concrete type (like `List<string>`) by applying type arguments to it.

```
// generic List with no parameters
Type openType = typeof(List<>);

// To create a List<string>
Type[] tArgs = { typeof(string) };
Type target = openType.MakeGenericType(tArgs);

// Create an instance - Activator.CreateInstance will call the default constructor.
// This is equivalent to calling new List<string>().
List<string> result = (List<string>)Activator.CreateInstance(target);
```

The `List<>` syntax is not permitted outside of a `typeof` expression.

Without `Activator` class

Using `new` keyword (will do for parameterless constructors)

```
T GetInstance<T>() where T : new()
{
    T instance = new T();
    return instance;
}
```

Using Invoke method

```
// Get the instance of the desired constructor (here it takes a string as a parameter).
ConstructorInfo c = typeof(T).GetConstructor(new[] { typeof(string) });
// Don't forget to check if such constructor exists
if (c == null)
    throw new InvalidOperationException(string.Format("A constructor for type '{0}' was not
found.", typeof(T)));
T instance = (T)c.Invoke(new object[] { "test" });
```

Using Expression trees

Expression trees represent code in a tree-like data structure, where each node is an expression. As MSDN explains:

> Expression is a sequence of one or more operands and zero or more operators that can be evaluated to a single value, object, method, or namespace. Expressions can consist of a literal value, a method invocation, an operator and its operands, or a simple name. Simple names can be the name of a variable, type member, method parameter, namespace or type.

```
public class GenericFactory<TKey, TType>
    {
       private readonly Dictionary<TKey, Func<object[], TType>> _registeredTypes; // dictionary,
that holds constructor functions.
       private object _locker = new object(); // object for locking dictionary, to guarantee thread
```

```
safety

        public GenericFactory()
        {
            _registeredTypes = new Dictionary<TKey, Func<object[], TType>>();
        }

        /// <summary>
        /// Find and register suitable constructor for type
        /// </summary>
        /// <typeparam name="TType"></typeparam>
        /// <param name="key">Key for this constructor</param>
        /// <param name="parameters">Parameters</param>
        public void Register(TKey key, params Type[] parameters)
        {
            ConstructorInfo ci = typeof(TType).GetConstructor(BindingFlags.Public |
BindingFlags.Instance, null, CallingConventions.HasThis, parameters, new ParameterModifier[] { });
// Get the instance of ctor.
            if (ci == null)
                throw new InvalidOperationException(string.Format("Constructor for type '{0}' was
not found.", typeof(TType)));

            Func<object[], TType> ctor;

            lock (_locker)
            {
                if (!_registeredTypes.TryGetValue(key, out ctor)) // check if such ctor already
been registered
                {
                    var pExp = Expression.Parameter(typeof(object[]), "arguments"); // create
parameter Expression
                    var ctorParams = ci.GetParameters(); // get parameter info from constructor

                    var argExpressions = new Expression[ctorParams.Length]; // array that will
contains parameter expessions
                    for (var i = 0; i < parameters.Length; i++)
                    {

                        var indexedAcccess = Expression.ArrayIndex(pExp, Expression.Constant(i));

                        if (!parameters[i].IsClass && !parameters[i].IsInterface) // check if
parameter is a value type
                        {
                            var localVariable = Expression.Variable(parameters[i],
"localVariable"); // if so - we should create local variable that will store paraameter value

                            var block = Expression.Block(new[] { localVariable },
                                    Expression.IfThenElse(Expression.Equal(indexedAcccess,
Expression.Constant(null)),
                                        Expression.Assign(localVariable,
Expression.Default(parameters[i])),
                                        Expression.Assign(localVariable,
Expression.Convert(indexedAcccess, parameters[i]))
                                    ),
                                    localVariable
                                );

                            argExpressions[i] = block;

                        }
                        else
                            argExpressions[i] = Expression.Convert(indexedAcccess, parameters[i]);
```

```
                    }
                    var newExpr = Expression.New(ci, argExpressions); // create expression that
represents call to specified ctor with the specified arguments.

                    _registeredTypes.Add(key, Expression.Lambda(newExpr, new[] { pExp }).Compile()
as Func<object[], TType>); // compile expression to create delegate, and add function to dictionary
                }
            }
        }

        /// <summary>
        /// Returns instance of registered type by key.
        /// </summary>
        /// <typeparam name="TType"></typeparam>
        /// <param name="key"></param>
        /// <param name="args"></param>
        /// <returns></returns>
        public TType Create(TKey key, params object[] args)
        {
            Func<object[], TType> foo;
            if (_registeredTypes.TryGetValue(key, out foo))
            {
                return (TType)foo(args);
            }

            throw new ArgumentException("No type registered for this key.");
        }
    }
```

Could be used like this:

```
public class TestClass
{
        public TestClass(string parameter)
        {
            Console.Write(parameter);
        }
}


public void TestMethod()
{
       var factory = new GenericFactory<string, TestClass>();
       factory.Register("key", typeof(string));
       TestClass newInstance = factory.Create("key", "testParameter");
}
```

Using FormatterServices.GetUninitializedObject

```
T instance = (T)FormatterServices.GetUninitializedObject(typeof(T));
```

In case of using `FormatterServices.GetUninitializedObject` constructors and field initializers will not be called. It is meant to be used in serializers and remoting engines

Section 63.4: Get a Strongly-Typed Delegate to a Method or Property via Reflection

When performance is a concern, invoking a method via reflection (i.e. via the `MethodInfo.Invoke` method) is not ideal. However, it is relatively straightforward to obtain a more performant strongly-typed delegate using the

`Delegate.CreateDelegate` function. The performance penalty for using reflection is incurred only during the delegate-creation process. Once the delegate is created, there is little-to-no performance penalty for invoking it:

```
// Get a MethodInfo for the Math.Max(int, int) method...
var maxMethod = typeof(Math).GetMethod("Max", new Type[] { typeof(int), typeof(int) });
// Now get a strongly-typed delegate for Math.Max(int, int)...
var stronglyTypedDelegate = (Func<int, int, int>)Delegate.CreateDelegate(typeof(Func<int, int,
int>), null, maxMethod);
// Invoke the Math.Max(int, int) method using the strongly-typed delegate...
Console.WriteLine("Max of 3 and 5 is: {0}", stronglyTypedDelegate(3, 5));
```

This technique can be extended to properties as well. If we have a class named `MyClass` with an `int` property named `MyIntProperty`, the code to get a strongly-typed getter would be (the following example assumes 'target' is a valid instance of `MyClass`):

```
// Get a MethodInfo for the MyClass.MyIntProperty getter...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theGetter = theProperty.GetGetMethod();
// Now get a strongly-typed delegate for MyIntProperty that can be executed against any MyClass
instance...
var stronglyTypedGetter = (Func<MyClass, int>)Delegate.CreateDelegate(typeof(Func<MyClass, int>),
theGetter);
// Invoke the MyIntProperty getter against MyClass instance 'target'...
Console.WriteLine("target.MyIntProperty is: {0}", stronglyTypedGetter(target));
```

...and the same can be done for the setter:

```
// Get a MethodInfo for the MyClass.MyIntProperty setter...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theSetter = theProperty.GetSetMethod();
// Now get a strongly-typed delegate for MyIntProperty that can be executed against any MyClass
instance...
var stronglyTypedSetter = (Action<MyClass, int>)Delegate.CreateDelegate(typeof(Action<MyClass,
int>), theSetter);
// Set MyIntProperty to 5...
stronglyTypedSetter(target, 5);
```

Section 63.5: Get a generic method and invoke it

Let's say you have class with generic methods. And you need to call its functions with reflection.

```
public class Sample
{
    public void GenericMethod<T>()
    {
        // ...
    }

    public static void StaticMethod<T>()
    {
        //...
    }
}
```

Let's say we want to call the GenericMethod with type string.

```
Sample sample = new Sample();//or you can get an instance via reflection
```

```
MethodInfo method = typeof(Sample).GetMethod("GenericMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(sample, null);//Since there are no arguments, we are passing null
```

For the static method you do not need an instance. Therefore the first argument will also be null.

```
MethodInfo method = typeof(Sample).GetMethod("StaticMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(null, null);
```

Section 63.6: Get a System.Type

For an instance of a type:

```
var theString = "hello";
var theType = theString.GetType();
```

From the type itself:

```
var theType = typeof(string);
```

Section 63.7: Getting and setting properties

Basic usage:

```
PropertyInfo prop = myInstance.GetType().GetProperty("myProperty");
// get the value myInstance.myProperty
object value = prop.GetValue(myInstance);

int newValue = 1;
// set the value myInstance.myProperty to newValue
prop.setValue(myInstance, newValue);
```

Setting read-only automatically-implemented properties can be done through it's backing field (in .NET Framework name of backing field is "k__BackingField"):

```
// get backing field info
FieldInfo fieldInfo = myInstance.GetType()
    .GetField("<myProperty>k__BackingField", BindingFlags.Instance | BindingFlags.NonPublic);

int newValue = 1;
// set the value of myInstance.myProperty backing field to newValue
fieldInfo.SetValue(myInstance, newValue);
```

Section 63.8: Create an instance of a Generic Type and invoke it's method

```
var baseType = typeof(List<>);
var genericType = baseType.MakeGenericType(typeof(String));
var instance = Activator.CreateInstance(genericType);
var method = genericType.GetMethod("GetHashCode");
var result = method.Invoke(instance, new object[] { });
```

Section 63.9: Custom Attributes

Find properties with a custom attribute - `MyAttribute`

```
var props = t.GetProperties(BindingFlags.NonPublic | BindingFlags.Public |
            BindingFlags.Instance).Where(
            prop => Attribute.IsDefined(prop, typeof(MyAttribute)));
```

Find all custom attributes on a given property

```
var attributes = typeof(t).GetProperty("Name").GetCustomAttributes(false);
```

Enumerate all classes with custom attribute - `MyAttribute`

```
static IEnumerable<Type> GetTypesWithAttribute(Assembly assembly) {
    foreach(Type type in assembly.GetTypes()) {
        if (type.GetCustomAttributes(typeof(MyAttribute), true).Length > 0) {
            yield return type;
        }
    }
}
```

Read value of a custom attribute at runtime

```
public static class AttributeExtensions
{

        /// <summary>
        /// Returns the value of a member attribute for any member in a class.
        ///     (a member is a Field, Property, Method, etc...)
        /// <remarks>
        /// If there is more than one member of the same name in the class, it will return the first
one (this applies to overloaded methods)
        /// </remarks>
        /// <example>
        /// Read System.ComponentModel Description Attribute from method 'MyMethodName' in class
'MyClass':
        ///     var Attribute = typeof(MyClass).GetAttribute("MyMethodName", (DescriptionAttribute d)
=> d.Description);
        /// </example>
        /// <param name="type">The class that contains the member as a type</param>
        /// <param name="MemberName">Name of the member in the class</param>
        /// <param name="valueSelector">Attribute type and property to get (will return first
instance if there are multiple attributes of the same type)</param>
        /// <param name="inherit">true to search this member's inheritance chain to find the
attributes; otherwise, false. This parameter is ignored for properties and events</param>
        /// </summary>
        public static TValue GetAttribute<TAttribute, TValue>(this Type type, string MemberName,
Func<TAttribute, TValue> valueSelector, bool inherit = false) where TAttribute : Attribute
        {
            var att =
type.GetMember(MemberName).FirstOrDefault().GetCustomAttributes(typeof(TAttribute),
inherit).FirstOrDefault() as TAttribute;
            if (att != null)
            {
                return valueSelector(att);
            }
            return default(TValue);
        }
```

```
    }
```

Usage

```
//Read System.ComponentModel Description Attribute from method 'MyMethodName' in class 'MyClass'
var Attribute = typeof(MyClass).GetAttribute("MyMethodName", (DescriptionAttribute d) =>
d.Description);
```

Section 63.10: Instantiating classes that implement an interface (e.g. plugin activation)

If you want your application to support a plug-in system, for example to load plug-ins from assemblies located in `plugins` folder:

```
interface IPlugin
{
    string PluginDescription { get; }
    void DoWork();
}
```

This class would be located in a separate dll

```
class HelloPlugin : IPlugin
{
    public string PluginDescription => "A plugin that says Hello";
    public void DoWork()
    {
        Console.WriteLine("Hello");
    }
}
```

Your application's plugin loader would find the dll files, get all types in those assemblies that implement `IPlugin`, and create instances of those.

```
    public IEnumerable<IPlugin> InstantiatePlugins(string directory)
    {
        var pluginAssemblyNames = Directory.GetFiles(directory, "*.addin.dll").Select(name => new
FileInfo(name).FullName).ToArray();
        //load the assemblies into the current AppDomain, so we can instantiate the types later
        foreach (var fileName in pluginAssemblyNames)
            AppDomain.CurrentDomain.Load(File.ReadAllBytes(fileName));
        var assemblies = pluginAssemblyNames.Select(System.Reflection.Assembly.LoadFile);
        var typesInAssembly = assemblies.SelectMany(asm => asm.GetTypes());
        var pluginTypes = typesInAssembly.Where(type => typeof (IPlugin).IsAssignableFrom(type));
        return pluginTypes.Select(Activator.CreateInstance).Cast<IPlugin>();
    }
```

Section 63.11: Get a Type by name with namespace

To do this you need a reference to the assembly which contains the type. If you have another type available which you know is in the same assembly as the one you want you can do this:

```
typeof(KnownType).Assembly.GetType(typeName);
```

- where `typeName` is the name of the type you are looking for (including the namespace) , and `KnownType` is the type you know is in the same assembly.

Less efficient but more general is as follows:

```
Type t = null;
foreach (Assembly ass in AppDomain.CurrentDomain.GetAssemblies())
{
    if (ass.FullName.StartsWith("System."))
        continue;
    t = ass.GetType(typeName);
    if (t != null)
        break;
}
```

Notice the check to exclude scanning System namespace assemblies to speed up the search. If your type may actually be a CLR type, you will have to delete these two lines.

If you happen to have the fully assembly-qualified type name including the assembly you can simply get it with

```
Type.GetType(fullyQualifiedName);
```

Section 63.12: Determining generic arguments of instances of generic types

If you have an instance of a generic type but for some reason don't know the specific type, you might want to determine the generic arguments that were used to create this instance.

Let's say someone created an instance of `List<T>` like that and passes it to a method:

```
var myList = new List<int>();
ShowGenericArguments(myList);
```

where `ShowGenericArguments` has this signature:

```
public void ShowGenericArguments(object o)
```

so at compile time you don't have any idea what generic arguments have been used to create `o`. Reflection provides a lot of methods to inspect generic types. At first, we can determine if the type of `o` is a generic type at all:

```
public void ShowGenericArguments(object o)
{
    if (o == null) return;

    Type t = o.GetType();
    if (!t.IsGenericType) return;
    ...
```

`Type.IsGenericType` returns `true` if the type is a generic type and `false` if not.

But this is not all we want to know. `List<>` itself is a generic type, too. But we only want to examine instances of specific *constructed generic* types. A constructed generic type is for example a `List<int>` that has a specific type *argument* for all its generic *parameters*.

The `Type` class provides two more properties, `IsConstructedGenericType` and `IsGenericTypeDefinition`, to distinguish these constructed generic types from generic type definitions:

```
typeof(List<>).IsGenericType // true
typeof(List<>).IsGenericTypeDefinition // true
```

```
typeof(List<>).IsConstructedGenericType// false

typeof(List<int>).IsGenericType // true
typeof(List<int>).IsGenericTypeDefinition // false
typeof(List<int>).IsConstructedGenericType// true
```

To enumerate the generic arguments of an instance, we can use the `GetGenericArguments()` method that returns an `Type` array containing the generic type arguments:

```
public void ShowGenericArguments(object o)
{
    if (o == null) return;
    Type t = o.GetType();
    if (!t.IsConstructedGenericType) return;

    foreach(Type genericTypeArgument in t.GetGenericArguments())
        Console.WriteLine(genericTypeArgument.Name);
}
```

So the call from above (`ShowGenericArguments(myList)`) results in this output:

```
Int32
```

Section 63.13: Looping through all the properties of a class

```
Type type = obj.GetType();
//To restrict return properties. If all properties are required don't provide flag.
BindingFlags flags = BindingFlags.Public | BindingFlags.Instance;
PropertyInfo[] properties = type.GetProperties(flags);

foreach (PropertyInfo property in properties)
{
    Console.WriteLine("Name: " + property.Name + ", Value: " + property.GetValue(obj, null));
}
```

Chapter 64: IQueryable interface

Section 64.1: Translating a LINQ query to a SQL query

The `IQueryable` and `IQueryable<T>` interfaces allows developers to translate a LINQ query (a 'language-integrated' query) to a specific datasource, for example a relational database. Take this LINQ query written in C#:

```
var query = from book in books
            where book.Author == "Stephen King"
            select book;
```

If the variable `books` is of a type that implements `IQueryable<Book>` then the query above gets passed to the provider (set on the `IQueryable.Provider` property) in the form of an expression tree, a data structure that reflects the structure of the code.

The provider can inspect the expression tree at runtime to determine:

- that there is a predicate for the `Author` property of the `Book` class;
- that the comparison method used is 'equals' (`==`);
- that the value it should equal is `"Stephen King"`.

With this information the provider can translate the C# query to a SQL query at runtime and pass that query to a relational database to fetch only those books that match the predicate:

```
SELECT *
FROM Books
WHERE Author = 'Stephen King'
```

The provider gets called when the `query` variable is iterated over (`IQueryable` implements `IEnumerable`).

(The provider used in this example would require some extra metadata to know which table to query and to know how to match properties of the C# class to columns of the table, but such metadata is outside of the scope of the `IQueryable` interface.)

Chapter 65: Linq to Objects

LINQ to Objects refers to the use of LINQ queries with any IEnumerable collection.

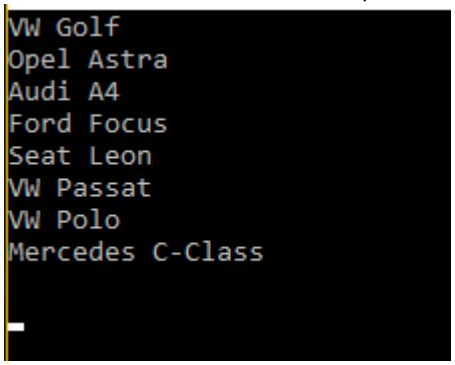
Section 65.1: Using LINQ to Objects in C#

A simple SELECT query in Linq

```
static void Main(string[] args)
{
    string[] cars = { "VW Golf",
                      "Opel Astra",
                      "Audi A4",
                      "Ford Focus",
                      "Seat Leon",
                      "VW Passat",
                      "VW Polo",
                      "Mercedes C-Class" };

    var list = from car in cars
               select car;

    StringBuilder sb = new StringBuilder();

    foreach (string entry in list)
    {
        sb.Append(entry + "\n");
    }

    Console.WriteLine(sb.ToString());
    Console.ReadLine();
}
```

In the example above, an array of strings (cars) is used as a collection of objects to be queried using LINQ. In a LINQ query, the from clause comes first in order to introduce the data source (cars) and the range variable (car). When the query is executed, the range variable will serve as a reference to each successive element in cars. Because the compiler can infer the type of car, you do not have to specify it explicitly

When the above code is compiled and executed, it produces the following result:



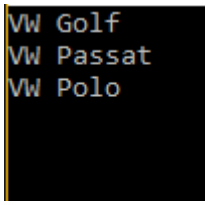
SELECT with a WHERE Clause

```
var list = from car in cars
           where car.Contains("VW")
           select car;
```

The WHERE clause is used to query the string array (cars) to find and return a subset of array which satisfies the

WHERE clause.

When the above code is compiled and executed, it produces the following result:



Generating an Ordered List

```
var list = from car in cars
           orderby car ascending
           select car;
```

Sometimes it is useful to sort the returned data. The orderby clause will cause the elements to be sorted according to the default comparer for the type being sorted.

When the above code is compiled and executed, it produces the following result:

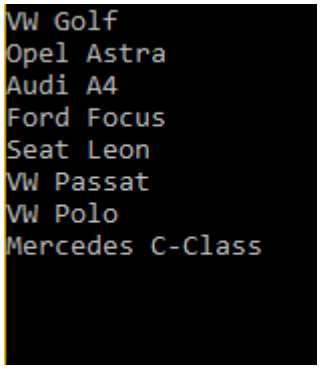
Working with a custom type

In this example, a typed list is created, populated, and then queried

```
public class Car
{
    public String Name { get; private set; }
    public int UnitsSold { get; private set; }

    public Car(string name, int unitsSold)
    {
        Name = name;
        UnitsSold = unitsSold;
    }
}

class Program
{
    static void Main(string[] args)
    {

        var car1 = new Car("VW Golf", 270952);
        var car2 = new Car("Opel Astra", 56079);
        var car3 = new Car("Audi A4", 52493);
        var car4 = new Car("Ford Focus", 51677);
        var car5 = new Car("Seat Leon", 42125);
```

```
        var car6 = new Car("VW Passat", 97586);
        var car7 = new Car("VW Polo", 69867);
        var car8 = new Car("Mercedes C-Class", 67549);

        var cars = new List<Car> {
            car1, car2, car3, car4, car5, car6, car7, car8 };
        var list = from car in cars
                   select car.Name;

        foreach (var entry in list)
        {
            Console.WriteLine(entry);
        }
        Console.ReadLine();
    }
}
```

When the above code is compiled and executed, it produces the following result:

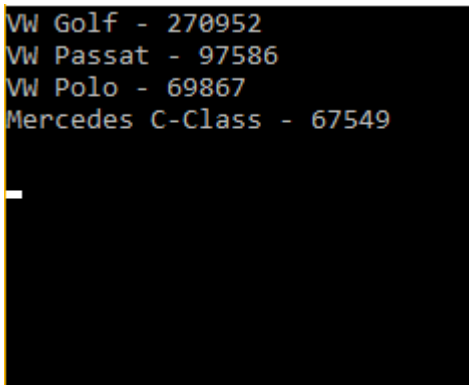


Until now the examples don't seem amazing as one can just iterate through the array to do basically the same. However, with the few examples below you can see how to create more complex queries with LINQ to Objects and achieve more with a lot less of code.

In the example below we can select cars that have been sold over 60000 units and sort them over the number of units sold:

```
var list = from car in cars
           where car.UnitsSold > 60000
           orderby car.UnitsSold descending
           select car;

StringBuilder sb = new StringBuilder();

foreach (var entry in list)
{
    sb.AppendLine($"{entry.Name} - {entry.UnitsSold}");
}
Console.WriteLine(sb.ToString());
```

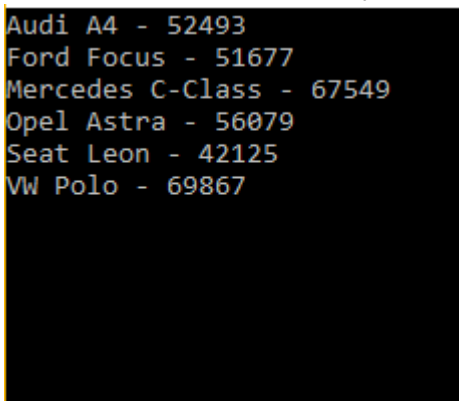
When the above code is compiled and executed, it produces the following result:



In the example below we can select cars that have sold an odd number of units and order them alphabetically over its name:

```
var list = from car in cars
           where car.UnitsSold % 2 != 0
           orderby car.Name ascending
           select car;
```

When the above code is compiled and executed, it produces the following result:



Chapter 66: LINQ Queries

LINQ is an acronym which stands for **L**anguage **IN**tegrated **Q**uery. It is a concept which integrates a query language by offering a consistent model for working with data across various kinds of data sources and formats; you use the same basic coding patterns to query and transform data in XML documents, SQL databases, ADO.NET Datasets, .NET collections, and any other format for which a LINQ provider is available.

Section 66.1: Chaining methods

Many LINQ functions both operate on an `IEnumerable<TSource>` and also return an `IEnumerable<TResult>`. The type parameters `TSource` and `TResult` may or may not refer to the same type, depending on the method in question and any functions passed to it.

A few examples of this are

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector
)

public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate
)

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
)
```

While some method chaining may require an entire set to be worked prior to moving on, LINQ takes advantage of deferred execution by using yield return **MSDN** which creates an Enumerable and an Enumerator behind the scenes. The process of chaining in LINQ is essentially building an enumerable (iterator) for the original set -- which is deferred -- until materialized by enumerating the enumerable.

This allows these functions to be fluently chained **wiki**, where one function can act directly on the result of another. This style of code can be used to perform many sequence based operations in a single statement.

For example, it's possible to combine **`SELECT`**, `Where` and `OrderBy` to transform, filter and sort a sequence in a single statement.

```
var someNumbers = { 4, 3, 2, 1 };

var processed = someNumbers
        .Select(n => n * 2)   // Multiply each number by 2
        .Where(n => n != 6)   // Keep all the results, except for 6
        .OrderBy(n => n);     // Sort in ascending order
```

Output:

```
2
4
8
```

Live Demo on .NET Fiddle

Any functions that both extend and return the generic `IEnumerable<T>` type can be used as chained clauses in a single statement. This style of fluent programming is powerful, and should be considered when creating your own extension methods.

Section 66.2: First, FirstOrDefault, Last, LastOrDefault, Single, and SingleOrDefault

All six methods return a single value of the sequence type, and can be called with or without a predicate.

Depending on the number of elements that match the `predicate` or, if no `predicate` is supplied, the number of elements in the source sequence, they behave as follows:

First()

- Returns the first element of a sequence, or the first element matching the provided `predicate`.
- If the sequence contains no elements, an `InvalidOperationException` is thrown with the message: "Sequence contains no elements".
- If the sequence contains no elements matching the provided `predicate`, an `InvalidOperationException` is thrown with the message "Sequence contains no matching element".

Example

```
// Returns "a":
new[] { "a" }.First();

// Returns "a":
new[] { "a", "b" }.First();

// Returns "b":
new[] { "a", "b" }.First(x => x.Equals("b"));

// Returns "ba":
new[] { "ba", "be" }.First(x => x.Contains("b"));

// Throws InvalidOperationException:
new[] { "ca", "ce" }.First(x => x.Contains("b"));

// Throws InvalidOperationException:
new string[0].First();
```

Live Demo on .NET Fiddle

FirstOrDefault()

- Returns the first element of a sequence, or the first element matching the provided `predicate`.
- If the sequence contains no elements, or no elements matching the provided `predicate`, returns the default value of the sequence type using `default(T)`.

Example

```
// Returns "a":
new[] { "a" }.FirstOrDefault();

// Returns "a":
new[] { "a", "b" }.FirstOrDefault();
```

```
// Returns "b":
new[] { "a", "b" }.FirstOrDefault(x => x.Equals("b"));

// Returns "ba":
new[] { "ba", "be" }.FirstOrDefault(x => x.Contains("b"));

// Returns null:
new[] { "ca", "ce" }.FirstOrDefault(x => x.Contains("b"));

// Returns null:
new string[0].FirstOrDefault();
```

Live Demo on .NET Fiddle

Last()

- Returns the last element of a sequence, or the last element matching the provided `predicate`.
- If the sequence contains no elements, an `InvalidOperationException` is thrown with the message "Sequence contains no elements."
- If the sequence contains no elements matching the provided `predicate`, an `InvalidOperationException` is thrown with the message "Sequence contains no matching element".

Example

```
// Returns "a":
new[] { "a" }.Last();

// Returns "b":
new[] { "a", "b" }.Last();

// Returns "a":
new[] { "a", "b" }.Last(x => x.Equals("a"));

// Returns "be":
new[] { "ba", "be" }.Last(x => x.Contains("b"));

// Throws InvalidOperationException:
new[] { "ca", "ce" }.Last(x => x.Contains("b"));

// Throws InvalidOperationException:
new string[0].Last();
```

LastOrDefault()

- Returns the last element of a sequence, or the last element matching the provided `predicate`.
- If the sequence contains no elements, or no elements matching the provided `predicate`, returns the default value of the sequence type using `default(T)`.

Example

```
// Returns "a":
new[] { "a" }.LastOrDefault();

// Returns "b":
new[] { "a", "b" }.LastOrDefault();

// Returns "a":
new[] { "a", "b" }.LastOrDefault(x => x.Equals("a"));

 // Returns "be":
```

```
new[] { "ba", "be" }.LastOrDefault(x => x.Contains("b"));

// Returns null:
new[] { "ca", "ce" }.LastOrDefault(x => x.Contains("b"));

// Returns null:
new string[0].LastOrDefault();
```

Single()

- If the sequence contains exactly one element, or exactly one element matching the provided `predicate`, that element is returned.
- If the sequence contains no elements, or no elements matching the provided `predicate`, an `InvalidOperationException` is thrown with the message "Sequence contains no elements".
- If the sequence contains more than one element, or more than one element matching the provided `predicate`, an `InvalidOperationException` is thrown with the message "Sequence contains more than one element".
- **Note:** in order to evaluate whether the sequence contains exactly one element, at most two elements has to be enumerated.

Example

```
// Returns "a":
new[] { "a" }.Single();

// Throws InvalidOperationException because sequence contains more than one element:
new[] { "a", "b" }.Single();

// Returns "b":
new[] { "a", "b" }.Single(x => x.Equals("b"));

// Throws InvalidOperationException:
new[] { "a", "b" }.Single(x => x.Equals("c"));

// Throws InvalidOperationException:
new string[0].Single();

// Throws InvalidOperationException because sequence contains more than one element:
new[] { "a", "a" }.Single();
```

SingleOrDefault()

- If the sequence contains exactly one element, or exactly one element matching the provided `predicate`, that element is returned.
- If the sequence contains no elements, or no elements matching the provided `predicate`, `default(T)` is returned.
- If the sequence contains more than one element, or more than one element matching the provided `predicate`, an `InvalidOperationException` is thrown with the message "Sequence contains more than one element".
- If the sequence contains no elements matching the provided `predicate`, returns the default value of the sequence type using `default(T)`.
- **Note:** in order to evaluate whether the sequence contains exactly one element, at most two elements has to be enumerated.

Example

```
// Returns "a":
new[] { "a" }.SingleOrDefault();
```

```
// returns "a"
new[] { "a", "b" }.SingleOrDefault(x => x == "a");

// Returns null:
new[] { "a", "b" }.SingleOrDefault(x => x == "c");

// Throws InvalidOperationException:
new[] { "a", "a" }.SingleOrDefault(x => x == "a");

// Throws InvalidOperationException:
new[] { "a", "b" }.SingleOrDefault();

// Returns null:
new string[0].SingleOrDefault();
```

Recommendations

- Although you can use `FirstOrDefault`, `LastOrDefault` or `SingleOrDefault` to check whether a sequence contains any items, `Any` or `Count` are more reliable. This is because a return value of `default(T)` from one of these three methods doesn't prove that the sequence is empty, as the value of the first / last / single element of the sequence could equally be `default(T)`
- Decide on which methods fits your code's purpose the most. For instance, use `Single` only if you must ensure that there is a single item in the collection matching your predicate — otherwise use `First`; as `Single` throw an exception if the sequence has more than one matching element. This of course applies to the "*OrDefault"-counterparts as well.
- Regarding efficiency: Although it's often appropriate to ensure that there is only one item (`Single`) or, either only one or zero (`SingleOrDefault`) items, returned by a query, both of these methods require more, and often the entirety, of the collection to be examined to ensure there in no second match to the query. This is unlike the behavior of, for example, the `First` method, which can be satisfied after finding the first match.

Section 66.3: Except

The Except method returns the set of items which are contained in the first collection but are not contained in the second. The default `IEqualityComparer` is used to compare the items within the two sets. There is an overload which accepts an `IEqualityComparer` as an argument.

Example:

```
int[] first = { 1, 2, 3, 4 };
int[] second = { 0, 2, 3, 5 };

IEnumerable<int> inFirstButNotInSecond = first.Except(second);
// inFirstButNotInSecond = { 1, 4 }
```

Output:

> 1
> 4

Live Demo on .NET Fiddle

In this case `.Except(second)` excludes elements contained in the array `second`, namely 2 and 3 (0 and 5 are not contained in the `first` array and are skipped).

Note that `Except` implies `Distinct` (i.e., it removes repeated elements). For example:

```
int[] third = { 1, 1, 1, 2, 3, 4 };

IEnumerable<int> inThirdButNotInSecond = third.Except(second);
// inThirdButNotInSecond = { 1, 4 }
```

Output:

> 1
> 4

Live Demo on .NET Fiddle

In this case, the elements 1 and 4 are returned only once.

Implementing `IEquatable` or providing the function an `IEqualityComparer` will allow using a different method to compare the elements. Note that the `GetHashCode` method should also be overridden so that it will return an identical hash code for `object` that are identical according to the `IEquatable` implementation.

Example With IEquatable:

```
class Holiday : IEquatable<Holiday>
{
    public string Name { get; set; }

    public bool Equals(Holiday other)
    {
        return Name == other.Name;
    }

    // GetHashCode must return true whenever Equals returns true.
    public override int GetHashCode()
    {
        //Get hash code for the Name field if it is not null.
        return Name?.GetHashCode() ?? 0;
    }
}

public class Program
{
    public static void Main()
    {
        List<Holiday> holidayDifference = new List<Holiday>();

        List<Holiday> remoteHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Hanukkah" },
            new Holiday { Name = "Ramadan" }
        };

        List<Holiday> localHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Ramadan" }
        };
```

```
        holidayDifference = remoteHolidays
            .Except(localHolidays)
            .ToList();

        holidayDifference.ForEach(x => Console.WriteLine(x.Name));
    }
}
```

Output:

> Hanukkah

Live Demo on .NET Fiddle

Section 66.4: SelectMany

The SelectMany linq method 'flattens' an `IEnumerable<IEnumerable<T>>` into an `IEnumerable<T>`. All of the T elements within the `IEnumerable` instances contained in the source `IEnumerable` will be combined into a single `IEnumerable`.

```
var words = new [] { "a,b,c", "d,e", "f" };
var splitAndCombine = words.SelectMany(x => x.Split(','));
// returns { "a", "b", "c", "d", "e", "f" }
```

If you use a selector function which turns input elements into sequences, the result will be the elements of those sequences returned one by one.

Note that, unlike **`SELECT()`**, the number of elements in the output doesn't need to be the same as were in the input.

More real-world example

```
class School
{
    public Student[] Students { get; set; }
}

class Student
{
    public string Name { get; set; }
}

var schools = new [] {
    new School(){ Students = new [] { new Student { Name="Bob"}, new Student { Name="Jack"} }},
    new School(){ Students = new [] { new Student { Name="Jim"}, new Student { Name="John"} }}
};

var allStudents = schools.SelectMany(s=> s.Students);

foreach(var student in allStudents)
{
    Console.WriteLine(student.Name);
}
```

Output:

> Bob

> Jack
> Jim
> John

Live Demo on .NET Fiddle

Section 66.5: Any

`Any` is used to check if **any** element of a collection matches a condition or not.
see also: .All, Any and FirstOrDefault: best practice

1. Empty parameter

Any: Returns `true` if the collection has any elements and `false` if the collection is empty:

```
var numbers = new List<int>();
bool result = numbers.Any(); // false

var numbers = new List<int>(){ 1, 2, 3, 4, 5};
bool result = numbers.Any(); //true
```

2. Lambda expression as parameter

Any: Returns `true` if the collection has one or more elements that meet the condition in the lambda expression:

```
var arrayOfStrings = new string[] { "a", "b", "c" };
arrayOfStrings.Any(item => item == "a");    // true
arrayOfStrings.Any(item => item == "d");    // false
```

3. Empty collection

Any: Returns `false` if the collection is empty and a lambda expression is supplied:

```
var numbers = new List<int>();
bool result = numbers.Any(i => i >= 0); // false
```

Note: `Any` will stop iteration of the collection as soon as it finds an element matching the condition. This means that the collection will not necessarily be fully enumerated; it will only be enumerated far enough to find the first item matching the condition.

Live Demo on .NET Fiddle

Section 66.6: JOINS

Joins are used to combine different lists or tables holding data via a common key.

Like in SQL, the following kinds of Joins are supported in LINQ:
Inner, Left, Right, Cross and **Full Outer** Joins.

The following two lists are used in the examples below:

```
var first = new List<string>(){ "a","b","c"}; // Left data
var second = new List<string>(){ "a", "c", "d"}; // Right data
```

(Inner) Join

```
var result = from f in first
             join s in second on f equals s
             select new { f, s };

var result = first.Join(second,
                        f => f,
                        s => s,
                        (f, s) => new { f, s });

// Result: {"a","a"}
//         {"c","c"}
```

Left outer join

```
var leftOuterJoin = from f in first
                    join s in second on f equals s into temp
                    from t in temp.DefaultIfEmpty()
                    select new { First = f, Second = t};

// Or can also do:
var leftOuterJoin = from f in first
                    from s in second.Where(x => x == f).DefaultIfEmpty()
                    select new { First = f, Second = s};

// Result: {"a","a"}
//         {"b", null}
//         {"c","c"}


// Left outer join method syntax
var leftOuterJoinFluentSyntax = first.GroupJoin(second,
                                      f => f,
                                      s => s,
                                      (f, s) => new { First = f, Second = s })
                                    .SelectMany(temp => temp.Second.DefaultIfEmpty(),
                                      (f, s) => new { First = f.First, Second = s });
```

Right Outer Join

```
var rightOuterJoin = from s in second
                     join f in first on s equals f into temp
                     from t in temp.DefaultIfEmpty()
                     select new {First=t,Second=s};

// Result: {"a","a"}
//         {"c","c"}
//         {null,"d"}
```

Cross Join

```
var CrossJoin = from f in first
                from s in second
                select new { f, s };

// Result: {"a","a"}
//         {"a","c"}
//         {"a","d"}
//         {"b","a"}
//         {"b","c"}
//         {"b","d"}
```

```
//         {"c","a"}
//         {"c","c"}
//         {"c","d"}
```

Full Outer Join

```
var fullOuterjoin = leftOuterJoin.Union(rightOuterJoin);

// Result: {"a","a"}
//         {"b", null}
//         {"c","c"}
//         {null,"d"}
```

Practical example

The examples above have a simple data structure so you can focus on understanding the different LINQ joins technically, but in the real world you would have tables with columns you need to join.

In the following example, there is just one class `Region` used, in reality you would join two or more different tables which hold the same key (in this example `first` and `second` are joined via the common key `ID`).

Example: Consider the following data structure:

```
public class Region
{
    public Int32 ID;
    public string RegionDescription;

    public Region(Int32 pRegionID, string pRegionDescription=null)
    {
        ID = pRegionID; RegionDescription = pRegionDescription;
    }
}
```

Now prepare the data (i.e. populate with data):

```
// Left data
var first = new List<Region>()
                { new Region(1), new Region(3), new Region(4) };
// Right data
var second = new List<Region>()
                {
                   new Region(1, "Eastern"),  new Region(2, "Western"),
                   new Region(3, "Northern"), new Region(4, "Southern")
                };
```

You can see that in this example `first` doesn't contain any region descriptions so you want to join them from `second`. Then the inner join would look like:

```
// do the inner join
var result = from f in first
             join s in second on f.ID equals s.ID
             select new { f.ID, s.RegionDescription };


 // Result: {1,"Eastern"}
 //         {3, Northern}
 //         {4,"Southern"}
```

This result has created anonymous objects on the fly, which is fine, but we have already created a proper class - so we can specify it: Instead of `SELECT NEW { f.ID, s.RegionDescription };` we can say `SELECT NEW Region(f.ID, s.RegionDescription);`, which will return the same data but will create objects of type `Region` - that will maintain compatibility with the other objects.

Live demo on .NET fiddle

Section 66.7: Skip and Take

The Skip method returns a collection excluding a number of items from the beginning of the source collection. The number of items excluded is the number given as an argument. If there are less items in the collection than specified in the argument then an empty collection is returned.

The Take method returns a collection containing a number of elements from the beginning of the source collection. The number of items included is the number given as an argument. If there are less items in the collection than specified in the argument then the collection returned will contain the same elements as the source collection.

```
var values = new [] { 5, 4, 3, 2, 1 };

var skipTwo        = values.Skip(2);         // { 3, 2, 1 }
var takeThree      = values.Take(3);         // { 5, 4, 3 }
var skipOneTakeTwo = values.Skip(1).Take(2); // { 4, 3 }
var takeZero       = values.Take(0);         // An IEnumerable<int> with 0 items
```

Live Demo on .NET Fiddle

Skip and Take are commonly used together to paginate results, for instance:

```
IEnumerable<T> GetPage<T>(IEnumerable<T> collection, int pageNumber, int resultsPerPage) {
    int startIndex = (pageNumber - 1) * resultsPerPage;
    return collection.Skip(startIndex).Take(resultsPerPage);
}
```

> **Warning:** LINQ to Entities only supports Skip on ordered queries. If you try to use Skip without ordering you will get a **NotSupportedException** with the message "The method 'Skip' is only supported for sorted input in LINQ to Entities. The method 'OrderBy' must be called before the method 'Skip'."

Section 66.8: Defining a variable inside a Linq query (let keyword)

In order to define a variable inside a linq expression, you can use the **let** keyword. This is usually done in order to store the results of intermediate sub-queries, for example:

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

var aboveAverages = from number in numbers
                    let average = numbers.Average()
                    let nSquared = Math.Pow(number,2)
                    where nSquared > average
                    select number;

Console.WriteLine("The average of the numbers is {0}.", numbers.Average());

foreach (int n in aboveAverages)
```

```
{
  Console.WriteLine("Query result includes number {0} with square of {1}.", n, Math.Pow(n,2));
}
```

Output:

> The average of the numbers is 4.5.
> Query result includes number 3 with square of 9.
> Query result includes number 4 with square of 16.
> Query result includes number 5 with square of 25.
> Query result includes number 6 with square of 36.
> Query result includes number 7 with square of 49.
> Query result includes number 8 with square of 64.
> Query result includes number 9 with square of 81.

View Demo

Section 66.9: Zip

The `Zip` extension method acts upon two collections. It pairs each element in the two series together based on position. With a `Func` instance, we use `Zip` to handle elements from the two C# collections in pairs. If the series differ in size, the extra elements of the larger series will be ignored.

To take an example from the book "C# in a Nutshell",

```
int[] numbers = { 3, 5, 7 };
string[] words = { "three", "five", "seven", "ignored" };
IEnumerable<string> zip = numbers.Zip(words, (n, w) => n + "=" + w);
```

Output:

> 3=three
> 5=five
> 7=seven

View Demo

Section 66.10: Range and Repeat

The `Range` and `Repeat` static methods on `Enumerable` can be used to generate simple sequences.

Range

`Enumerable.Range()` generates a sequence of integers given a starting value and a count.

```
// Generate a collection containing the numbers 1-100 ([1, 2, 3, ..., 98, 99, 100])
var range = Enumerable.Range(1,100);
```

Live Demo on .NET Fiddle

Repeat

`Enumerable.Repeat()` generates a sequence of repeating elements given an element and the number of repetitions required.

```
// Generate a collection containing "a", three times (["a","a","a"])
var repeatedValues = Enumerable.Repeat("a", 3);
```

Live Demo on .NET Fiddle

Section 66.11: Basics

LINQ is largely beneficial for querying collections (or arrays).

For example, given the following sample data:

```
var classroom = new Classroom
{
    new Student { Name = "Alice", Grade = 97, HasSnack = true  },
    new Student { Name = "Bob",   Grade = 82, HasSnack = false },
    new Student { Name = "Jimmy", Grade = 71, HasSnack = true  },
    new Student { Name = "Greg",  Grade = 90, HasSnack = false },
    new Student { Name = "Joe",   Grade = 59, HasSnack = false }
}
```

We can "query" on this data using LINQ syntax. For example, to retrieve all students who have a snack today:

```
var studentsWithSnacks = from s in classroom.Students
                         where s.HasSnack
                         select s;
```

Or, to retrieve students with a grade of 90 or above, and only return their names, not the full `Student` object:

```
var topStudentNames = from s in classroom.Students
                      where s.Grade >= 90
                      select s.Name;
```

The LINQ feature is comprised of two syntaxes that perform the same functions, have nearly identical performance, but are written very differently. The syntax in the example above is called **query syntax**. The following example, however, illustrates **method syntax**. The same data will be returned as in the example above, but the way the query is written is different.

```
var topStudentNames = classroom.Students
                              .Where(s => s.Grade >= 90)
                              .Select(s => s.Name);
```

Section 66.12: All

`All` is used to check, if all elements of a collection match a condition or not.
see also: .Any

1. Empty parameter

All: is not allowed to be used with empty parameter.

2. Lambda expression as parameter

All: Returns `true` if all elements of collection satisfies the lambda expression and `false` otherwise:

```
var numbers = new List<int>(){ 1, 2, 3, 4, 5};
bool result = numbers.All(i => i < 10); // true
bool result = numbers.All(i => i >= 3); // false
```

3. Empty collection

All: Returns `true` if the collection is empty and a lambda expression is supplied:

```
var numbers = new List<int>();
bool result = numbers.All(i => i >= 0); // true
```

Note: `All` will stop iteration of the collection as soon as it finds an element **not** matching the condition. This means that the collection will not necessarily be fully enumerated; it will only be enumerated far enough to find the first item **not matching** the condition.

Section 66.13: Aggregate

`Aggregate` Applies an accumulator function over a sequence.

```
int[] intList = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = intList.Aggregate((prevSum, current) => prevSum + current);
// sum = 55
```

- At the first step `prevSum = 1`
- At the second `prevSum = prevSum(at the first step) + 2`
- At the i-th step `prevSum = prevSum(at the (i-1) step) + i-th element of the array`

```
string[] stringList = { "Hello", "World", "!" };
string joinedString = stringList.Aggregate((prev, current) => prev + " " + current);
// joinedString = "Hello World !"
```

A second overload of `Aggregate` also receives an `seed` parameter which is the initial accumulator value. This can be used to calculate multiple conditions on a collection without iterating it more than once.

```
List<int> items = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

For the collection of `items` we want to calculate

1. The total `.Count`
2. The amount of even numbers
3. Collect each forth item

Using `Aggregate` it can be done like this:

```
var result = items.Aggregate(new { Total = 0, Even = 0, FourthItems = new List<int>() },
                (accumelative,item) =>
                new {
                    Total = accumelative.Total + 1,
                    Even = accumelative.Even + (item % 2 == 0 ? 1 : 0),
                    FourthItems = (accumelative.Total + 1)%4 == 0 ?
                        new List<int>(accumelative.FourthItems) { item } :
                        accumelative.FourthItems
                });
// Result:
```

```
// Total = 12
// Even = 6
// FourthItems = [4, 8, 12]
```

Note that using an anonymous type as the seed one has to instantiate a new object each item because the properties are read only. Using a custom class one can simply assign the information and no `new` is needed (only when giving the initial `seed` parameter

Section 66.14: Distinct

Returns unique values from an `IEnumerable`. Uniqueness is determined using the default equality comparer.

```
int[] array = { 1, 2, 3, 4, 2, 5, 3, 1, 2 };

var distinct = array.Distinct();
// distinct = { 1, 2, 3, 4, 5 }
```

To compare a custom data type, we need to implement the `IEquatable<T>` interface and provide `GetHashCode` and `Equals` methods for the type. Or the equality comparer may be overridden:

```
class SSNEqualityComparer : IEqualityComparer<Person> {
    public bool Equals(Person a, Person b) => return a.SSN == b.SSN;
    public int GetHashCode(Person p) => p.SSN;
}

List<Person> people;

distinct = people.Distinct(SSNEqualityComparer);
```

Section 66.15: SelectMany: Flattening a sequence of sequences

```
var sequenceOfSequences = new [] { new [] { 1, 2, 3 }, new [] { 4, 5 }, new [] { 6 } };
var sequence = sequenceOfSequences.SelectMany(x => x);
// returns { 1, 2, 3, 4, 5, 6 }
```

Use `SelectMany()` if you have, or you are creating a sequence of sequences, but you want the result as one long sequence.

In LINQ Query Syntax:

```
var sequence = from subSequence in sequenceOfSequences
               from item in subSequence
               select item;
```

If you have a collection of collections and would like to be able to work on data from parent and child collection at the same time, it is also possible with `SelectMany`.

Let's define simple classes

```
public class BlogPost
{
    public int Id { get; set; }
    public string Content { get; set; }
    public List<Comment> Comments { get; set; }
}
```

```
public class Comment
{
    public int Id { get; set; }
    public string Content { get; set; }
}
```

Let's assume we have following collection.

```
List<BlogPost> posts = new List<BlogPost>()
{
    new BlogPost()
    {
        Id = 1,
        Comments = new List<Comment>()
        {
            new Comment()
            {
                Id = 1,
                Content = "It's really great!",
            },
            new Comment()
            {
                Id = 2,
                Content = "Cool post!"
            }
        }
    },
    new BlogPost()
    {
        Id = 2,
        Comments = new List<Comment>()
        {
            new Comment()
            {
                Id = 3,
                Content = "I don't think you're right",
            },
            new Comment()
            {
                Id = 4,
                Content = "This post is a complete nonsense"
            }
        }
    }
};
```

Now we want to select comments `Content` along with `Id` of `BlogPost` associated with this comment. In order to do so, we can use appropriate `SelectMany` overload.

```
var commentsWithIds = posts.SelectMany(p => p.Comments, (post, comment) => new { PostId = post.Id,
CommentContent = comment.Content });
```

Our `commentsWithIds` looks like this

```
{
    PostId = 1,
    CommentContent = "It's really great!"
},
{
    PostId = 1,
```

```
    CommentContent = "Cool post!"
},
{
    PostId = 2,
    CommentContent = "I don't think you're right"
},
{
    PostId = 2,
    CommentContent = "This post is a complete nonsense"
}
```

Section 66.16: GroupBy

GroupBy is an easy way to sort a `IEnumerable<T>` collection of items into distinct groups.

Simple Example

In this first example, we end up with two groups, odd and even items.

```
List<int> iList = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var grouped = iList.GroupBy(x => x % 2 == 0);

//Groups iList into odd [13579] and even[2468] items

foreach(var group in grouped)
{
    foreach (int item in group)
    {
        Console.Write(item); // 135792468  (first odd then even)
    }
}
```

More Complex Example

Let's take grouping a list of people by age as an example. First, we'll create a Person object which has two properties, Name and Age.

```
public class Person
{
    public int Age {get; set;}
    public string Name {get; set;}
}
```

Then we create our sample list of people with various names and ages.

```
List<Person> people = new List<Person>();
people.Add(new Person{Age = 20, Name = "Mouse"});
people.Add(new Person{Age = 30, Name = "Neo"});
people.Add(new Person{Age = 40, Name = "Morpheus"});
people.Add(new Person{Age = 30, Name = "Trinity"});
people.Add(new Person{Age = 40, Name = "Dozer"});
people.Add(new Person{Age = 40, Name = "Smith"});
```

Then we create a LINQ query to group our list of people by age.

```
var query = people.GroupBy(x => x.Age);
```

Doing so, we can see the Age for each group, and have a list of each person in the group.

```
foreach(var result in query)
{
    Console.WriteLine(result.Key);

    foreach(var person in result)
        Console.WriteLine(person.Name);
}
```

This results in the following output:

```
20
Mouse
30
Neo
Trinity
40
Morpheus
Dozer
Smith
```

You can play with the live demo on .NET Fiddle

Section 66.17: Query collection by type / cast elements to type

```
interface IFoo { }
class Foo : IFoo { }
class Bar : IFoo { }
```

```
var item0 = new Foo();
var item1 = new Foo();
var item2 = new Bar();
var item3 = new Bar();
var collection = new IFoo[] { item0, item1, item2, item3 };
```

Using `OfType`

```
var foos = collection.OfType<Foo>(); // result: IEnumerable<Foo> with item0 and item1
var bars = collection.OfType<Bar>(); // result: IEnumerable<Bar> item item2 and item3
var foosAndBars = collection.OfType<IFoo>(); // result: IEnumerable<IFoo> with all four items
```

Using `Where`

```
var foos = collection.Where(item => item is Foo); // result: IEnumerable<IFoo> with item0 and item1
var bars = collection.Where(item => item is Bar); // result: IEnumerable<IFoo> with item2 and item3
```

Using `Cast`

```
var bars = collection.Cast<Bar>();               // throws InvalidCastException on the 1st item
var foos = collection.Cast<Foo>();               // throws InvalidCastException on the 3rd item
var foosAndBars = collection.Cast<IFoo>();       // OK
```

Section 66.18: Enumerating the Enumerable

The IEnumerable<T> interface is the base interface for all generic enumerators and is a quintessential part of understanding LINQ. At its core, it represents the sequence.

This underlying interface is inherited by all of the generic collections, such as Collection<T>, Array, List<T>, Dictionary<TKey, TValue> Class, and HashSet<T>.

In addition to representing the sequence, any class that inherits from IEnumerable<T> must provide an IEnumerator<T>. The enumerator exposes the iterator for the enumerable, and these two interconnected interfaces and ideas are the source of the saying "enumerate the enumerable".

"Enumerating the enumerable" is an important phrase. The enumerable is simply a structure for how to iterate, it does not hold any materialized objects. For example, when sorting, an enumerable may hold the criteria of the field to sort, but using `.OrderBy()` in itself will return an IEnumerable<T> which only knows *how* to sort. Using a call which will materialize the objects, as in iterate the set, is known as enumerating (for example `.ToList()`). The enumeration process will use the the enumerable definition of *how* in order to move through the series and return the relevant objects (in order, filtered, projected, etc.).

Only once the enumerable has been enumerated does it cause the materialization of the objects, which is when metrics like time complexity (how long it should take related to series size) and spacial complexity (how much space it should use related to series size) can be measured.

Creating your own class that inherits from IEnumerable<T> can be a little complicated depending on the underlying series that needs to be enumerable. In general it is best to use one of the existing generic collections. That said, it is also possible to inherit from the IEnumerable<T> interface without having a defined array as the underlying structure.

For example, using the Fibonacci series as the underlying sequence. Note that the call to `Where` simply builds an `IEnumerable`, and it is not until a call to enumerate that enumerable is made that any of the values are materialized.

```
void Main()
{
    Fibonacci Fibo = new Fibonacci();
    IEnumerable<long> quadrillionplus = Fibo.Where(i => i > 1000000000000);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(quadrillionplus.Take(2).Sum());
    Console.WriteLine(quadrillionplus.Skip(2).First());

    IEnumerable<long> fibMod612 = Fibo.OrderBy(i => i % 612);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(fibMod612.First());//smallest divisible by 612
}

public class Fibonacci : IEnumerable<long>
{
    private int max = 90;

    //Enumerator called typically from foreach
    public IEnumerator GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("Enumerating the Enumerable");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
```

```
        }
    }

    //Enumerable called typically from linq
    IEnumerator<long> IEnumerable<long>.GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("Enumerating the Enumerable");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
        }
    }
}
```

Output

```
Enumerable built
Enumerating the Enumerable
4052739537881
Enumerating the Enumerable
4052739537881
Enumerable built
Enumerating the Enumerable
14930352
```

The strength in the second set (the fibMod612) is that even though we made the call to order our entire set of Fibonacci numbers, since only one value was taken using `.First()` the time complexity was O(n) as only 1 value needed to be compared during the ordering algorithm's execution. This is because our enumerator only asked for 1 value, and so the entire enumerable did not have to be materialized. Had we used `.Take(5)` instead of `.First()` the enumerator would have asked for 5 values, and at most 5 values would need to be materialized. Compared to needing to order an entire set *and then* take the first 5 values, the principle of saves a lot of execution time and space.

Section 66.19: Using Range with various Linq methods

You can use the Enumerable class alongside Linq queries to convert for loops into Linq one liners.

Select Example

Opposed to doing this:

```
var asciiCharacters = new List<char>();
for (var x = 0; x < 256; x++)
{
    asciiCharacters.Add((char)x);
}
```

You can do this:

```
var asciiCharacters = Enumerable.Range(0, 256).Select(a => (char) a);
```

Where Example

In this example, 100 numbers will be generated and even ones will be extracted

```
var evenNumbers = Enumerable.Range(1, 100).Where(a => a % 2 == 0);
```

Section 66.20: Where

Returns a subset of items which the specified predicate is true for them.

```
List<string> trees = new List<string>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

Method syntax

```
// Select all trees with name of length 3
var shortTrees = trees.Where(tree => tree.Length == 3); // Oak, Elm
```

Query syntax

```
var shortTrees = from tree in trees
                 where tree.Length == 3
                 select tree; // Oak, Elm
```

Section 66.21: Using SelectMany instead of nested loops

Given 2 lists

```
var list1 = new List<string> { "a", "b", "c" };
var list2 = new List<string> { "1", "2", "3", "4" };
```

if you want to output all permutations you could use nested loops like

```
var result = new List<string>();
foreach (var s1 in list1)
    foreach (var s2 in list2)
        result.Add($"{s1}{s2}");
```

Using SelectMany you can do the same operation as

```
var result = list1.SelectMany(x => list2.Select(y => $"{x}{y}", x, y)).ToList();
```

Section 66.22: Contains

MSDN:

> Determines whether a sequence contains a specified element by using a specified `IEqualityComparer<T>`

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
var result1 = numbers.Contains(4); // true
var result2 = numbers.Contains(8); // false

List<int> secondNumberCollection = new List<int> { 4, 5, 6, 7 };
// Note that can use the Intersect method in this case
var result3 = secondNumberCollection.Where(item => numbers.Contains(item)); // will be true only
for 4,5
```

Using a user defined object:

```
public class Person
{
   public string Name { get; set; }
}

List<Person> objects = new List<Person>
{
    new Person { Name = "Nikki"},
    new Person { Name = "Gilad"},
    new Person { Name = "Phil"},
    new Person { Name = "John"}
};

//Using the Person's Equals method - override Equals() and GetHashCode() - otherwise it
//will compare by reference and result will be false
var result4 = objects.Contains(new Person { Name = "Phil" }); // true
```

Using the `Enumerable.Contains(value, comparer)` overload:

```
public class Compare : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name;
    }
    public int GetHashCode(Person codeh)
    {
        return codeh.Name.GetHashCode();
    }
}

var result5 = objects.Contains(new Person { Name = "Phil" }, new Compare()); // true
```

A smart usage of `Contains` would be to replace multiple `if` clauses to a `Contains` call.

So instead of doing this:

```
if(status == 1 || status == 3 || status == 4)
{
    //Do some business operation
}
else
{
    //Do something else
}
```

Do this:

```
if(new int[] {1, 3, 4 }.Contains(status)
{
    //Do some business operaion
}
else
{
    //Do something else
}
```

Section 66.23: Joining multiple sequences

Consider entities `Customer`, `Purchase` and `PurchaseItem` as follows:

```
public class Customer
{
   public string Id { get; set } // A unique Id that identifies customer
   public string Name  {get; set; }
}

public class Purchase
{
   public string Id { get; set }
   public string CustomerId {get; set; }
   public string Description { get; set; }
}

public class PurchaseItem
{
   public string Id { get; set }
   public string PurchaseId {get; set; }
   public string Detail { get; set; }
}
```

Consider following sample data for above entities:

```
var customers = new List<Customer>()
 {
    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer1"
    },

    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer2"
    }
 };

 var purchases = new List<Purchase>()
 {
     new Purchase() {
         Id = Guid.NewGuid().ToString(),
         CustomerId = customers[0].Id,
         Description = "Customer1-Purchase1"
     },

     new Purchase() {
         Id = Guid.NewGuid().ToString(),
         CustomerId = customers[0].Id,
         Description = "Customer1-Purchase2"
     },

     new Purchase() {
         Id = Guid.NewGuid().ToString(),
         CustomerId = customers[1].Id,
         Description = "Customer2-Purchase1"
     },

     new Purchase() {
         Id = Guid.NewGuid().ToString(),
```

```
        CustomerId = customers[1].Id,
        Description = "Customer2-Purchase2"
    }
 };

var purchaseItems = new List<PurchaseItem>()
{
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[0].Id,
        Detail = "Purchase1-PurchaseItem1"
    },

    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem1"
    },

    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem2"
    },

    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[3].Id,
        Detail = "Purchase3-PurchaseItem1"
    }
};
```

Now, consider below linq query:

```
var result = from c in customers
             join p in purchases on c.Id equals p.CustomerId           // first join
             join pi in purchaseItems on p.Id equals pi.PurchaseId     // second join
             select new
             {
                c.Name, p.Description, pi.Detail
             };
```

To output the result of above query:

```
foreach(var resultItem in result)
{
    Console.WriteLine($"{resultItem.Name}, {resultItem.Description}, {resultItem.Detail}");
}
```

The output of the query would be:

> Customer1, Customer1-Purchase1, Purchase1-PurchaseItem1
>
> Customer1, Customer1-Purchase2, Purchase2-PurchaseItem1
>
> Customer1, Customer1-Purchase2, Purchase2-PurchaseItem2
>
> Customer2, Customer2-Purchase2, Purchase3-PurchaseItem1

Live Demo on .NET Fiddle

Section 66.24: Joining on multiple keys

```
PropertyInfo[] stringProps = typeof (string).GetProperties();//string properties
PropertyInfo[] builderProps = typeof(StringBuilder).GetProperties();//stringbuilder properties

  var query =
      from s in stringProps
      join b in builderProps
          on new { s.Name, s.PropertyType } equals new { b.Name, b.PropertyType }
      select new
      {
          s.Name,
          s.PropertyType,
          StringToken = s.MetadataToken,
          StringBuilderToken = b.MetadataToken
      };
```

Note that anonymous types in above `join` must contain same properties since objects are considered equal only if all their properties are equal. Otherwise query won't compile.

Section 66.25: ToLookup

> ToLookup returns a data structure that allows indexing. It is an extension method. It produces an ILookup instance that can be indexed or enumerated using a foreach-loop. The entries are combined into groupings at each key. - dotnetperls

```
string[] array = { "one", "two", "three" };
//create lookup using string length as key
var lookup = array.ToLookup(item => item.Length);

//join the values whose lengths are 3
Console.WriteLine(string.Join(",",lookup[3]));
//output: one,two
```

Another Example:

```
int[] array = { 1,2,3,4,5,6,7,8 };
//generate lookup for odd even numbers (keys will be 0 and 1)
var lookup = array.ToLookup(item => item % 2);

//print even numbers after joining
Console.WriteLine(string.Join(",",lookup[0]));
//output: 2,4,6,8

//print odd numbers after joining
Console.WriteLine(string.Join(",",lookup[1]));
//output: 1,3,5,7
```

Section 66.26: SkipWhile

`SkipWhile()` is used to exclude elements until first non-match (this might be counter intuitive to most)

```
int[] list = { 42, 42, 6, 6, 6, 42 };
```

```
var result = list.SkipWhile(i => i == 42);
// Result: 6, 6, 6, 42
```

Section 66.27: Query Ordering - OrderBy() ThenBy() OrderByDescending() ThenByDescending()

```
string[] names= { "mark", "steve", "adam" };
```

Ascending:

Query Syntax

```
var sortedNames =
    from name in names
    orderby name
    select name;
```

Method Syntax

```
var sortedNames = names.OrderBy(name => name);
```

sortedNames contains the names in following order: "adam","mark","steve"

Descending:

Query Syntax

```
var sortedNames =
    from name in names
    orderby name descending
    select name;
```

Method Syntax

```
var sortedNames = names.OrderByDescending(name => name);
```

sortedNames contains the names in following order: "steve","mark","adam"

Order by several fields

```
Person[] people =
{
    new Person { FirstName = "Steve", LastName = "Collins", Age = 30},
    new Person { FirstName = "Phil" , LastName = "Collins", Age = 28},
    new Person { FirstName = "Adam" , LastName = "Ackerman", Age = 29},
    new Person { FirstName = "Adam" , LastName = "Ackerman", Age = 15}
};
```

Query Syntax

```
var sortedPeople = from person in people
                   orderby person.LastName, person.FirstName, person.Age descending
                   select person;
```

Method Syntax

```
sortedPeople = people.OrderBy(person => person.LastName)
                     .ThenBy(person => person.FirstName)
                     .ThenByDescending(person => person.Age);
```

Result

```
1. Adam Ackerman 29
2. Adam Ackerman 15
3. Phil Collins  28
4. Steve Collins 30
```

Section 66.28: Sum

The `Enumerable.Sum` extension method calculates the sum of numeric values.

In case the collection's elements are themselves numbers, you can calculate the sum directly.

```
int[] numbers = new int[] { 1, 4, 6 };
Console.WriteLine( numbers.Sum() ); //outputs 11
```

In case the type of the elements is a complex type, you can use a lambda expression to specify the value that should be calculated:

```
var totalMonthlySalary = employees.Sum( employee => employee.MonthlySalary );
```

Sum extension method can calculate with the following types:

- Int32
- Int64
- Single
- Double
- Decimal

In case your collection contains nullable types, you can use the null-coalescing operator to set a default value for null elements:

```
int?[] numbers = new int?[] { 1, null, 6 };
Console.WriteLine( numbers.Sum( number => number ?? 0 ) ); //outputs 7
```

Section 66.29: GroupBy one or multiple fields

Lets assume we have some Film model:

```
public class Film {
    public string Title { get; set; }
    public string Category { get; set; }
    public int Year { get; set; }
}
```

Group by Category property:

```
foreach (var grp in films.GroupBy(f => f.Category)) {
    var groupCategory = grp.Key;
    var numberOfFilmsInCategory = grp.Count();
```

```
}
```

Group by Category and Year:

```
foreach (var grp in films.GroupBy(f => new { Category = f.Category, Year = f.Year })) {
    var groupCategory = grp.Key.Category;
    var groupYear = grp.Key.Year;
    var numberOfFilmsInCategory = grp.Count();
}
```

Section 66.30: OrderBy

> Orders a collection by a specified value.

When the value is an **integer**, **double** or **float** it starts with the *minimum value*, which means that you get first the negative values, than zero and afterwords the positive values (see Example 1).

When you order by a **char** the method compares the *ascii values* of the chars to sort the collection (see Example 2).

When you sort **strings** the OrderBy method compares them by taking a look at their CultureInfo but normaly starting with the *first letter* in the alphabet (a,b,c...).

This kind of order is called ascending, if you want it the other way round you need descending (see OrderByDescending).

Example 1:

```
int[] numbers = {2, 1, 0, -1, -2};
IEnumerable<int> ascending = numbers.OrderBy(x => x);
// returns {-2, -1, 0, 1, 2}
```

Example 2:

```
 char[] letters = {' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z', 'Z'};
 IEnumerable<char> ascending = letters.OrderBy(x => x);
 // returns { ' ', '!', '+', '1', '9', '?', 'A', 'B', 'Y', 'Z', '[', 'a', 'b', 'y', 'z', '{' }
```

Example:

```
class Person
{
   public string Name { get; set; }
   public int Age { get; set; }
}

var people = new[]
{
    new Person {Name = "Alice", Age = 25},
    new Person {Name = "Bob", Age = 21},
    new Person {Name = "Carol", Age = 43}
};
var youngestPerson = people.OrderBy(x => x.Age).First();
var name = youngestPerson.Name; // Bob
```

Section 66.31: Any and First(OrDefault) - best practice

I won't explain what `Any` and `FirstOrDefault` does because there are already two good example about them. See Any and First, FirstOrDefault, Last, LastOrDefault, Single, and SingleOrDefault for more information.

A pattern I often see in code which **should be avoided** is

```
if (myEnumerable.Any(t=>t.Foo == "Bob"))
{
    var myFoo = myEnumerable.First(t=>t.Foo == "Bob");
    //Do stuff
}
```

It could be written more efficiently like this

```
var myFoo = myEnumerable.FirstOrDefault(t=>t.Foo == "Bob");
if (myFoo != null)
{
    //Do stuff
}
```

By using the second example, the collection is searched only once and give the same result as the first one. The same idea can be applied to `Single`.

Section 66.32: GroupBy Sum and Count

Let's take a sample class:

```
public class Transaction
{
    public string Category { get; set; }
    public DateTime Date { get; set; }
    public decimal Amount { get; set; }
}
```

Now, let us consider a list of transactions:

```
var transactions = new List<Transaction>
{
   new Transaction { Category = "Saving Account", Amount = 56, Date = DateTime.Today.AddDays(1) },
   new Transaction { Category = "Saving Account", Amount = 10, Date = DateTime.Today.AddDays(-10)
},
   new Transaction { Category = "Credit Card", Amount = 15, Date = DateTime.Today.AddDays(1) },
   new Transaction { Category = "Credit Card", Amount = 56, Date = DateTime.Today },
   new Transaction { Category = "Current Account", Amount = 100, Date = DateTime.Today.AddDays(5)
},
};
```

If you want to calculate category wise sum of amount and count, you can use GroupBy as follows:

```
var summaryApproach1 = transactions.GroupBy(t => t.Category)
                           .Select(t => new
                           {
                               Category = t.Key,
                               Count = t.Count(),
                               Amount = t.Sum(ta => ta.Amount),
                           }).ToList();
```

```
Console.WriteLine("-- Summary: Approach 1 --");
summaryApproach1.ForEach(
            row => Console.WriteLine($"Category: {row.Category}, Amount: {row.Amount}, Count:
{row.Count}"));
```

Alternatively, you can do this in one step:

```
var summaryApproach2 = transactions.GroupBy(t => t.Category, (key, t) =>
{
        var transactionArray = t as Transaction[] ?? t.ToArray();
        return new
        {
            Category = key,
            Count = transactionArray.Length,
            Amount = transactionArray.Sum(ta => ta.Amount),
        };
}).ToList();

Console.WriteLine("-- Summary: Approach 2 --");
summaryApproach2.ForEach(
row => Console.WriteLine($"Category: {row.Category}, Amount: {row.Amount}, Count: {row.Count}"));
```

Output for both the above queries would be same:

> Category: Saving Account, Amount: 66, Count: 2
>
> Category: Credit Card, Amount: 71, Count: 2
>
> Category: Current Account, Amount: 100, Count: 1

Live Demo in .NET Fiddle

Section 66.33: SequenceEqual

`SequenceEqual` is used to compare two `IEnumerable<T>` sequences with each other.

```
int[] a = new int[] {1, 2, 3};
int[] b = new int[] {1, 2, 3};
int[] c = new int[] {1, 3, 2};

bool returnsTrue = a.SequenceEqual(b);
bool returnsFalse = a.SequenceEqual(c);
```

Section 66.34: ElementAt and ElementAtOrDefault

`ElementAt` will return the item at index `n`. If `n` is not within the range of the enumerable, throws an `ArgumentOutOfRangeException`.

```
int[] numbers  = { 1, 2, 3, 4, 5 };
numbers.ElementAt(2);  // 3
numbers.ElementAt(10); // throws ArgumentOutOfRangeException
```

`ElementAtOrDefault` will return the item at index `n`. If `n` is not within the range of the enumerable, returns a `default(T)`.

```
int[] numbers  = { 1, 2, 3, 4, 5 };
numbers.ElementAtOrDefault(2);  // 3
numbers.ElementAtOrDefault(10); // 0 = default(int)
```

Both `ElementAt` and `ElementAtOrDefault` are optimized for when the source is an `IList<T>` and normal indexing will be used in those cases.

Note that for `ElementAt`, if the provided index is greater than the size of the `IList<T>`, the list should (but is technically not guaranteed to) throw an `ArgumentOutOfRangeException`.

Section 66.35: DefaultIfEmpty

DefaultIfEmpty is used to return a Default Element if the Sequence contains no elements. This Element can be the Default of the Type or a user defined instance of that Type. Example:

```
var chars = new List<string>() { "a", "b", "c", "d" };

chars.DefaultIfEmpty("N/A").FirstOrDefault(); // returns "a";

chars.Where(str => str.Length > 1)
     .DefaultIfEmpty("N/A").FirstOrDefault(); // return "N/A"

chars.Where(str => str.Length > 1)
        .DefaultIfEmpty().First(); // returns null;
```

Usage in Left Joins:

With `DefaultIfEmpty` the traditional Linq Join can return a default object if no match was found. Thus acting as a SQL's Left Join. Example:

```
var leftSequence = new List<int>() { 99, 100, 5, 20, 102, 105 };
var rightSequence = new List<char>() { 'a', 'b', 'c', 'i', 'd' };

var numbersAsChars = from l in leftSequence
                     join r in rightSequence
                     on l equals (int)r into leftJoin
                     from result in leftJoin.DefaultIfEmpty('?')
                     select new
                     {
                         Number = l,
                         Character = result
                     };

foreach(var item in numbersAsChars)
{
    Console.WriteLine("Num = {0} ** Char = {1}", item.Number, item.Character);
}

output:

Num = 99        Char = c
Num = 100       Char = d
Num = 5         Char = ?
Num = 20        Char = ?
Num = 102       Char = ?
Num = 105       Char = i
```

In the case where a `DefaultIfEmpty` is used (without specifying a default value) and that will result will no matching

items on the right sequence one must make sure that the object is not `null` before accessing its properties. Otherwise it will result in a `NullReferenceException`. Example:

```
var leftSequence = new List<int> { 1, 2, 5 };
var rightSequence = new List<dynamic>()
    {
        new { Value = 1 },
        new { Value = 2 },
        new { Value = 3 },
        new { Value = 4 },
    };

var numbersAsChars = (from l in leftSequence
                      join r in rightSequence
                      on l equals r.Value into leftJoin
                      from result in leftJoin.DefaultIfEmpty()
                      select new
                      {
                          Left = l,
                          // 5 will not have a matching object in the right so result
                          // will be equal to null.
                          // To avoid an error use:
                          //    -  C# 6.0 or above - ?.
                          //    -  Under           - result == null ? 0 : result.Value
                          Right = result?.Value
                      }).ToList();
```

Section 66.36: ToDictionary

The `ToDictionary()` LINQ method can be used to generate a `Dictionary<TKey, TElement>` collection based on a given `IEnumerable<T>` source.

```
IEnumerable<User> users = GetUsers();
Dictionary<int, User> usersById = users.ToDictionary(x => x.Id);
```

In this example, the single argument passed to `ToDictionary` is of type `Func<TSource, TKey>`, which returns the key for each element.

This is a concise way to perform the following operation:

```
Dictionary<int, User> usersById = new Dictionary<int User>();
foreach (User u in users)
{
  usersById.Add(u.Id, u);
}
```

You can also pass a second parameter to the `ToDictionary` method, which is of type `Func<TSource, TElement>` and returns the `Value` to be added for each entry.

```
IEnumerable<User> users = GetUsers();
Dictionary<int, string> userNamesById = users.ToDictionary(x => x.Id, x => x.Name);
```

It is also possible to specify the `IComparer` that is used to compare key values. This can be useful when the key is a string and you want it to match case-insensitive.

```
IEnumerable<User> users = GetUsers();
Dictionary<string, User> usersByCaseInsenstiveName = users.ToDictionary(x => x.Name,
```

```
StringComparer.InvariantCultureIgnoreCase);

var user1 = usersByCaseInsenstiveName["john"];
var user2 = usersByCaseInsenstiveName["JOHN"];
user1 == user2; // Returns true
```

Note: the `ToDictionary` method requires all keys to be unique, there must be no duplicate keys. If there are, then an exception is thrown: `ArgumentException: An item with the same key has already been added.` If you have a scenario where you know that you will have multiple elements with the same key, then you are better off using `ToLookup` instead.

Section 66.37: Concat

Merges two collections (without removing duplicates)

```
List<int> foo = new List<int> { 1, 2, 3 };
List<int> bar = new List<int> { 3, 4, 5 };

// Through Enumerable static class
var result = Enumerable.Concat(foo, bar).ToList(); // 1,2,3,3,4,5

// Through extension method
var result = foo.Concat(bar).ToList(); // 1,2,3,3,4,5
```

Section 66.38: Build your own Linq operators for IEnumerable<T>

One of the great things about Linq is that it is so easy to extend. You just need to create an extension method whose argument is `IEnumerable<T>`.

```
public namespace MyNamespace
{
    public static class LinqExtensions
    {
        public static IEnumerable<List<T>> Batch<T>(this IEnumerable<T> source, int batchSize)
        {
            var batch = new List<T>();
            foreach (T item in source)
            {
                batch.Add(item);
                if (batch.Count == batchSize)
                {
                    yield return batch;
                    batch = new List<T>();
                }
            }
            if (batch.Count > 0)
                yield return batch;
        }
    }
}
```

This example splits the items in an `IEnumerable<T>` into lists of a fixed size, the last list containing the remainder of the items. Notice how the object to which the extension method is applied is passed in (argument `source`) as the initial argument using the `this` keyword. Then the `yield` keyword is used to output the next item in the output `IEnumerable<T>` before continuing with execution from that point (see yield keyword).

This example would be used in your code like this:

```
//using MyNamespace;
var items = new List<int> { 2, 3, 4, 5, 6 };
foreach (List<int> sublist in items.Batch(3))
{
    // do something
}
```

On the first loop, sublist would be `{2, 3, 4}` and on the second `{5, 6}`.

Custom LinQ methods can be combined with standard LinQ methods too. e.g.:

```
//using MyNamespace;
var result = Enumerable.Range(0, 13)          // generate a list
                       .Where(x => x%2 == 0) // filter the list or do something other
                       .Batch(3)              // call our extension method
                       .ToList()              // call other standard methods
```

This query will return even numbers grouped in batches with a size of 3: `{0, 2, 4}, {6, 8, 10}, {12}`

Remember you need a `using MyNamespace;` line in order to be able to access the extension method.

Section 66.39: Select - Transforming elements

Select allows you to apply a transformation to every element in any data structure implementing IEnumerable.

Getting the first character of each string in the following list:

```
List<String> trees = new List<String>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

Using regular (lambda) syntax

```
//The below select stament transforms each element in tree into its first character.
IEnumerable<String> initials = trees.Select(tree => tree.Substring(0, 1));
foreach (String initial in initials) {
    System.Console.WriteLine(initial);
}
```

Output:

> O
> B
> B
> E
> H
> M

Live Demo on .NET Fiddle

Using LINQ Query Syntax

```
initials = from tree in trees
           select tree.Substring(0, 1);
```

Section 66.40: OrderByDescending

> Orders a collection by a specified value.

When the value is an **integer**, **double** or **float** it starts with the *maximal value*, which means that you get first the positive values, than zero and afterwords the negative values (see Example 1).

When you order by a **char** the method compares the *ascii values* of the chars to sort the collection (see Example 2).

When you sort **strings** the OrderBy method compares them by taking a look at their CultureInfo but normaly starting with the *last letter* in the alphabet (z,y,x,...).

This kind of order is called descending, if you want it the other way round you need ascending (see OrderBy).

Example 1:

```
int[] numbers = {-2, -1, 0, 1, 2};
IEnumerable<int> descending = numbers.OrderByDescending(x => x);
// returns {2, 1, 0, -1, -2}
```

Example 2:

```
char[] letters = {' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z', 'Z'};
IEnumerable<char> descending = letters.OrderByDescending(x => x);
// returns { '{', 'z', 'y', 'b', 'a', '[', 'Z', 'Y', 'B', 'A', '?', '9', '1', '+', '!', ' ' }
```

Example 3:

```
class Person
{
   public  string Name { get; set; }
   public  int Age { get; set; }
}

var people = new[]
{
    new Person {Name = "Alice", Age = 25},
    new Person {Name = "Bob", Age = 21},
    new Person {Name = "Carol", Age = 43}
};
var oldestPerson = people.OrderByDescending(x => x.Age).First();
var name = oldestPerson.Name; // Carol
```

Section 66.41: Union

Merges two collections to create a distinct collection using the default equality comparer

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 2, 3, 4, 5 };

var allElement = numbers1.Union(numbers2);   // AllElement now contains 1,2,3,4,5
```

Live Demo on .NET Fiddle

Section 66.42: GroupJoin with outer range variable

```
Customer[] customers = Customers.ToArray();
Purchase[] purchases = Purchases.ToArray();

var groupJoinQuery =
    from c in customers
    join p in purchases on c.ID equals p.CustomerID
    into custPurchases
    select new
    {
        CustName = c.Name,
        custPurchases
    };
```

Section 66.43: Linq Quantifiers

Quantifier operations return a Boolean value if some or all of the elements in a sequence satisfy a condition. In this article, we will see some common LINQ to Objects scenarios where we can use these operators. There are 3 Quantifiers operations that can be used in LINQ:

`All` – used to determine whether all the elements in a sequence satisfy a condition. Eg:

```
int[] array = { 10, 20, 30 };

// Are all elements >= 10? YES
array.All(element => element >= 10);

// Are all elements >= 20? NO
array.All(element => element >= 20);

// Are all elements < 40? YES
array.All(element => element < 40);
```

`Any` - used to determine whether any elements in a sequence satisfy a condition. Eg:

```
int[] query=new int[] { 2, 3, 4 }
query.Any (n => n == 3);
```

`Contains` - used to determine whether a sequence contains a specified element. Eg:

```
//for int array
int[] query =new int[] { 1,2,3 };
query.Contains(1);

//for string array
string[] query={"Tom","grey"};
query.Contains("Tom");

//for a string
var stringValue="hello";
stringValue.Contains("h");
```

Section 66.44: TakeWhile

`TakeWhile` returns elements from a sequence as long as the condition is true

```
int[] list = { 1, 10, 40, 50, 44, 70, 4 };
var result = list.TakeWhile(item => item < 50).ToList();
// result = { 1, 10, 40 }
```

Section 66.45: Reverse

- Inverts the order of the elements in a sequence.
- If there is no items throws a `ArgumentNullException: source is null.`

Example:

```
// Create an array.
int[] array = { 1, 2, 3, 4 };                          //Output:
// Call reverse extension method on the array.        //4
var reverse = array.Reverse();                         //3
// Write contents of array to screen.                 //2
foreach (int value in reverse)                         //1
    Console.WriteLine(value);
```

Live code example

Remeber that `Reverse()` may work diffrent depending on the chain order of your LINQ statements.

```
        //Create List of chars
        List<int> integerlist = new List<int>() { 1, 2, 3, 4, 5, 6 };

        //Reversing the list then taking the two first elements
        IEnumerable<int> reverseFirst = integerlist.Reverse<int>().Take(2);

        //Taking 2 elements and then reversing only thos two
        IEnumerable<int> reverseLast = integerlist.Take(2).Reverse();

        //reverseFirst output: 6, 5
        //reverseLast output:  2, 1
```

Live code example

Reverse() works by buffering everything then walk through it backwards, whitch is not very efficient, but neither is OrderBy from that perspective.

In LINQ-to-Objects, there are buffering operations (Reverse, OrderBy, GroupBy, etc) and non-buffering operations (Where, Take, Skip, etc).

Example: Non-buffering Reverse extention

```
public static IEnumerable<T> Reverse<T>(this IList<T> list) {
    for (int i = list.Count - 1; i >= 0; i--)
        yield return list[i];
}
```

Live code example

This method can encounter problems if u mutate the list while iterating.

Section 66.46: Count and LongCount

`Count` returns the number of elements in an `IEnumerable<T>`. `Count` also exposes an optional predicate parameter that allows you to filter the elements you want to count.

```
int[] array = { 1, 2, 3, 4, 2, 5, 3, 1, 2 };

int n = array.Count(); // returns the number of elements in the array
int x = array.Count(i => i > 2); // returns the number of elements in the array greater than 2
```

`LongCount` works the same way as `Count` but has a return type of `long` and is used for counting `IEnumerable<T>` sequences that are longer than `int.MaxValue`

```
int[] array = GetLargeArray();

long n = array.LongCount(); // returns the number of elements in the array
long x = array.LongCount(i => i > 100); // returns the number of elements in the array greater than
100
```

Section 66.47: Incrementally building a query

Because LINQ uses **deferred execution**, we can have a query object that doesn't actually contain the values, but will return the values when evaluated. We can thus dynamically build the query based on our control flow, and evaluate it once we are finished:

```
IEnumerable<VehicleModel> BuildQuery(int vehicleType, SearchModel search, int start = 1, int count
= -1) {
    IEnumerable<VehicleModel> query = _entities.Vehicles
        .Where(x => x.Active && x.Type == vehicleType)
        .Select(x => new VehicleModel {
            Id = v.Id,
            Year = v.Year,
            Class = v.Class,
            Make = v.Make,
            Model = v.Model,
            Cylinders = v.Cylinders ?? 0
        });
```

We can conditionally apply filters:

```
    if (!search.Years.Contains("all", StringComparer.OrdinalIgnoreCase))
        query = query.Where(v => search.Years.Contains(v.Year));

    if (!search.Makes.Contains("all", StringComparer.OrdinalIgnoreCase)) {
        query = query.Where(v => search.Makes.Contains(v.Make));
    }

    if (!search.Models.Contains("all", StringComparer.OrdinalIgnoreCase)) {
        query = query.Where(v => search.Models.Contains(v.Model));
    }

    if (!search.Cylinders.Equals("all", StringComparer.OrdinalIgnoreCase)) {
        decimal minCylinders = 0;
        decimal maxCylinders = 0;
        switch (search.Cylinders) {
            case "2-4":
                maxCylinders = 4;
                break;
```

```
            case "5-6":
                minCylinders = 5;
                maxCylinders = 6;
                break;
            case "8":
                minCylinders = 8;
                maxCylinders = 8;
                break;
            case "10+":
                minCylinders = 10;
                break;
        }
        if (minCylinders > 0) {
            query = query.Where(v => v.Cylinders >= minCylinders);
        }
        if (maxCylinders > 0) {
            query = query.Where(v => v.Cylinders <= maxCylinders);
        }
    }
```

We can add a sort order to the query based on a condition:

```
    switch (search.SortingColumn.ToLower()) {
        case "make_model":
            query = query.OrderBy(v => v.Make).ThenBy(v => v.Model);
            break;
        case "year":
            query = query.OrderBy(v => v.Year);
            break;
        case "engine_size":
            query = query.OrderBy(v => v.EngineSize).ThenBy(v => v.Cylinders);
            break;
        default:
            query = query.OrderBy(v => v.Year); //The default sorting.
    }
```

Our query can be defined to start from a given point:

```
    query = query.Skip(start - 1);
```

and defined to return a specific number of records:

```
    if (count > -1) {
        query = query.Take(count);
    }
    return query;
}
```

Once we have the query object, we can evaluate the results with a `foreach` loop, or one of the LINQ methods that returns a set of values, such as `ToList` or `ToArray`:

```
SearchModel sm;

// populate the search model here
// ...

List<VehicleModel> list = BuildQuery(5, sm).ToList();
```

Section 66.48: Select with Func<TSource, int, TResult> selector - Use to get ranking of elements

On of the overloads of the **SELECT** extension methods also passes the `index` of the current item in the collection being **SELECT**ed. These are a few uses of it.

Get the "row number" of the items

```
var rowNumbers = collection.OrderBy(item => item.Property1)
                           .ThenBy(item => item.Property2)
                           .ThenByDescending(item => item.Property3)
                           .Select((item, index) => new { Item = item, RowNumber = index })
                           .ToList();
```

Get the rank of an item *within* its group

```
var rankInGroup = collection.GroupBy(item => item.Property1)
                            .OrderBy(group => group.Key)
                            .SelectMany(group => group.OrderBy(item => item.Property2)
                                                      .ThenByDescending(item => item.Property3)
                                                      .Select((item, index) => new
                                                      {
                                                          Item = item,
                                                          RankInGroup = index
                                                      })).ToList();
```

Get the ranking of groups (also known in Oracle as dense_rank)

```
var rankOfBelongingGroup = collection.GroupBy(item => item.Property1)
                            .OrderBy(group => group.Key)
                            .Select((group, index) => new
                            {
                                Items = group,
                                Rank = index
                            })
                            .SelectMany(v => v.Items, (s, i) => new
                            {
                                Item = i,
                                DenseRank = s.Rank
                            }).ToList();
```

For testing this you can use:

```
public class SomeObject
{
    public int Property1 { get; set; }
    public int Property2 { get; set; }
    public int Property3 { get; set; }

    public override string ToString()
    {
        return string.Join(", ", Property1, Property2, Property3);
    }
}
```

And data:

```
List<SomeObject> collection = new List<SomeObject>
```

```
{
    new SomeObject { Property1 = 1, Property2 = 1, Property3 = 1},
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 1},
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 2},
    new SomeObject { Property1 = 2, Property2 = 1, Property3 = 1},
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1},
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1},
    new SomeObject { Property1 = 2, Property2 = 3, Property3 = 1}
};
```

Chapter 67: LINQ to XML

Section 67.1: Read XML using LINQ to XML

```
<?xml version="1.0" encoding="utf-8" ?>
<Employees>
 <Employee>
  <EmpId>1</EmpId>
  <Name>Sam</Name>
  <Sex>Male</Sex>
  <Phone Type="Home">423-555-0124</Phone>
  <Phone Type="Work">424-555-0545</Phone>
  <Address>
   <Street>7A Cox Street</Street>
   <City>Acampo</City>
   <State>CA</State>
   <Zip>95220</Zip>
   <Country>USA</Country>
  </Address>
</Employee>
<Employee>
 <EmpId>2</EmpId>
 <Name>Lucy</Name>
 <Sex>Female</Sex>
 <Phone Type="Home">143-555-0763</Phone>
 <Phone Type="Work">434-555-0567</Phone>
 <Address>
  <Street>Jess Bay</Street>
  <City>Alta</City>
  <State>CA</State>
  <Zip>95701</Zip>
  <Country>USA</Country>
  </Address>
 </Employee>
</Employees>
```

To read that XML file using LINQ

```
XDocument xdocument = XDocument.Load("Employees.xml");
IEnumerable<XElement> employees = xdocument.Root.Elements();
foreach (var employee in employees)
{
    Console.WriteLine(employee);
}
```

To access single element

```
XElement xelement = XElement.Load("Employees.xml");
IEnumerable<XElement> employees = xelement.Root.Elements();
Console.WriteLine("List of all Employee Names :");
foreach (var employee in employees)
{
    Console.WriteLine(employee.Element("Name").Value);
}
```

To access multiple elements

```
XElement xelement = XElement.Load("Employees.xml");
```

```
IEnumerable<XElement> employees = xelement.Root.Elements();
Console.WriteLine("List of all Employee Names along with their ID:");
foreach (var employee in employees)
{
    Console.WriteLine("{0} has Employee ID {1}",
    employee.Element("Name").Value,
    employee.Element("EmpId").Value);
}
```

To access all Elements having a specific attribute

```
XElement xelement = XElement.Load("Employees.xml");
var name = from nm in xelement.Root.Elements("Employee")
       where (string)nm.Element("Sex") == "Female"
       select nm;
Console.WriteLine("Details of Female Employees:");
foreach (XElement xEle in name)
Console.WriteLine(xEle);
```

To access specific element having a specific attribute

```
XElement xelement = XElement.Load("..\\..\\Employees.xml");
var homePhone = from phoneno in xelement.Root.Elements("Employee")
            where (string)phoneno.Element("Phone").Attribute("Type") == "Home"
            select phoneno;
Console.WriteLine("List HomePhone Nos.");
foreach (XElement xEle in homePhone)
{
    Console.WriteLine(xEle.Element("Phone").Value);
}
```

Chapter 68: Parallel LINQ (PLINQ)

Section 68.1: Simple example

This example shows how PLINQ can be used to calculate the even numbers between 1 and 10,000 using multiple threads. Note that the resulting list will won't be ordered!

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .Where(x => x % 2 == 0)
                          .ToList();

// evenNumbers = { 4, 26, 28, 30, ... }
// Order will vary with different runs
```

Section 68.2: WithDegreeOfParallelism

The degree of parallelism is the maximum number of concurrently executing tasks that will be used to process the query.

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .WithDegreeOfParallelism(4)
                          .Where(x => x % 2 == 0);
```

Section 68.3: AsOrdered

This example shows how PLINQ can be used to calculate the even numbers between 1 and 10,000 using multiple threads. Order will be maintained in the resulting list, however keep in mind that `AsOrdered` may hurt performance for a large numbers of elements, so un-ordered processing is preferred when possible.

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
                          .AsOrdered()
                          .Where(x => x % 2 == 0)
                          .ToList();

// evenNumbers = { 2, 4, 6, 8, ..., 10000 }
```

Section 68.4: AsUnordered

Ordered sequences may hurt performance when dealing with a large number of elements. To mitigate this, it's possible to call `AsUnordered` when the sequence order is no longer necessary.

```
var sequence = Enumerable.Range(1, 10000).Select(x => -1 * x); // -1, -2, ...
var evenNumbers = sequence.AsParallel()
                          .OrderBy(x => x)
                          .Take(5000)
                          .AsUnordered()
                          .Where(x => x % 2 == 0) // This line won't be affected by ordering
                          .ToList();
```

Chapter 69: XmlDocument and the System.Xml namespace

Section 69.1: XmlDocument vs XDocument (Example and comparison)

> There are several ways interact with an Xml file.
>
> 1. Xml Document
> 2. XDocument
> 3. XmlReader/XmlWriter
>
> Before LINQ to XML we were used XMLDocument for manipulations in XML like adding attributes, elements and so on. Now LINQ to XML uses XDocument for the same kind of thing. Syntaxes are much easier than XMLDocument and it requires a minimal amount of code.
>
> Also XDocument is mutch faster as XmlDocument. XmlDoucument is an old and dirty solution for query an XML document.
>
> **I am going to show some examples of XmlDocument class and XDocument class** class:

> **Load XML file**

```
string filename = @"C:\temp\test.xml";
```

XmlDocument

```
XmlDocument _doc = new XmlDocument();
_doc.Load(filename);
```

XDocument

```
XDocument _doc = XDocument.Load(fileName);
```

> **Create XmlDocument**

XmlDocument

```
XmlDocument doc = new XmlDocument();
XmlElement root = doc.CreateElement("root");
root.SetAttribute("name", "value");
XmlElement child = doc.CreateElement("child");
child.InnerText = "text node";
root.AppendChild(child);
doc.AppendChild(root);
```

XDocument

```
XDocument doc = new XDocument(
```

```
    new XElement("Root", new XAttribute("name", "value"),
    new XElement("Child", "text node"))
);

/*result*/
<root name="value">
    <child>"TextNode"</child>
</root>
```

Change InnerText of node in XML

XmlDocument

```
XmlNode node = _doc.SelectSingleNode("xmlRootNode");
node.InnerText = value;
```

XDocument

```
 XElement rootNote = _doc.XPathSelectElement("xmlRootNode");
rootNode.Value = "New Value";
```

Save File after edit

Make sure to safe the xml after any change.

```
// Safe XmlDocument and XDocument
_doc.Save(filename);
```

Retrieve Values from XML

XmlDocument

```
 XmlNode node = _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
string text = node.InnerText;
```

XDocument

```
 XElement node = _doc.XPathSelectElement("xmlRootNode/levelOneChildNode");
 string text = node.Value;
```

Retrieve value from all from all child elements where attribute = something.

XmlDocument

```
List<string> valueList = new List<string>();
    foreach (XmlNode n in nodelist)
    {
        if(n.Attributes["type"].InnerText == "City")
        {
            valueList.Add(n.Attributes["type"].InnerText);
```

```
        }
    }
```

XDocument

```
var accounts = _doc.XPathSelectElements("/data/summary/account").Where(c =>
c.Attribute("type").Value == "setting").Select(c => c.Value);
```

> **Append a node**

XmlDocument

```
XmlNode nodeToAppend = doc.CreateElement("SecondLevelNode");
nodeToAppend.InnerText = "This title is created by code";

/* Append node to parent */
XmlNode firstNode= _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
firstNode.AppendChild(nodeToAppend);

/*After a change make sure to safe the document*/
_doc.Save(fileName);
```

XDocument

```
_doc.XPathSelectElement("ServerManagerSettings/TcpSocket").Add(new XElement("SecondLevelNode"));

 /*After a change make sure to safe the document*/
_doc.Save(fileName);
```

Section 69.2: Reading from XML document

An example XML file

```
    <Sample>
    <Account>
        <One number="12"/>
        <Two number="14"/>
    </Account>
    <Account>
        <One number="14"/>
        <Two number="16"/>
    </Account>
    </Sample>
```

Reading from this XML file:

```
    using System.Xml;
    using System.Collections.Generic;

    public static void Main(string fullpath)
    {
        var xmldoc = new XmlDocument();
        xmldoc.Load(fullpath);

        var oneValues = new List<string>();
```

```
        // Getting all XML nodes with the tag name
        var accountNodes = xmldoc.GetElementsByTagName("Account");
        for (var i = 0; i < accountNodes.Count; i++)
        {
            // Use Xpath to find a node
            var account = accountNodes[i].SelectSingleNode("./One");
            if (account != null && account.Attributes != null)
            {
                // Read node attribute
                oneValues.Add(account.Attributes["number"].Value);
            }
        }
}
```

Section 69.3: Basic XML document interaction

```
public static void Main()
{
    var xml  = new XmlDocument();
    var root = xml.CreateElement("element");
        // Creates an attribute, so the element will now be "<element attribute='value' />"
        root.SetAttribute("attribute", "value");

    // All XML documents must have one, and only one, root element
    xml.AppendChild(root);

    // Adding data to an XML document
    foreach (var dayOfWeek in Enum.GetNames((typeof(DayOfWeek))))
    {
        var day = xml.CreateElement("dayOfWeek");
            day.SetAttribute("name", dayOfWeek);

        // Don't forget to add the new value to the current document!
        root.AppendChild(day);
    }

    // Looking for data using XPath; BEWARE, this is case-sensitive
    var monday = xml.SelectSingleNode("//dayOfWeek[@name='Monday']");
    if (monday != null)
    {
        // Once you got a reference to a particular node, you can delete it
        // by navigating through its parent node and asking for removal
        monday.ParentNode.RemoveChild(monday);
    }

    // Displays the XML document in the screen; optionally can be saved to a file
    xml.Save(Console.Out);
}
```

Chapter 70: XDocument and the System.Xml.Linq namespace

Section 70.1: Generate an XML document

The goal is to generate the following XML document:

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit ID="F0001">
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit ID="F0002">
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Code:

```
XNamespace xns = "http://www.fruitauthority.fake";
XDeclaration xDeclaration = new XDeclaration("1.0", "utf-8", "yes");
XDocument xDoc = new XDocument(xDeclaration);
XElement xRoot = new XElement(xns + "FruitBasket");
xDoc.Add(xRoot);

XElement xelFruit1 = new XElement(xns + "Fruit");
XAttribute idAttribute1 = new XAttribute("ID", "F0001");
xelFruit1.Add(idAttribute1);
XElement xelFruitName1 = new XElement(xns + "FruitName", "Banana");
XElement xelFruitColor1 = new XElement(xns + "FruitColor", "Yellow");
xelFruit1.Add(xelFruitName1);
xelFruit1.Add(xelFruitColor1);
xRoot.Add(xelFruit1);

XElement xelFruit2 = new XElement(xns + "Fruit");
XAttribute idAttribute2 = new XAttribute("ID", "F0002");
xelFruit2.Add(idAttribute2);
XElement xelFruitName2 = new XElement(xns + "FruitName", "Apple");
XElement xelFruitColor2 = new XElement(xns + "FruitColor", "Red");
xelFruit2.Add(xelFruitName2);
xelFruit2.Add(xelFruitColor2);
xRoot.Add(xelFruit2);
```

Section 70.2: Generate an XML document using fluent syntax

Goal:

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
```

```
</FruitBasket>
```

Code:

```
XNamespace xns = "http://www.fruitauthority.fake";
XDocument xDoc =
    new XDocument(new XDeclaration("1.0", "utf-8", "yes"),
        new XElement(xns + "FruitBasket",
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "Banana"),
                new XElement(xns + "FruitColor", "Yellow")),
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "Apple"),
                new XElement(xns + "FruitColor", "Red"))
                ));
```

Section 70.3: Modify XML File

To modify an XML file with `XDocument`, you load the file into a variable of type `XDocument`, modify it in memory, then save it, overwriting the original file. A common mistake is to modify the XML in memory and expect the file on disk to change.

Given an XML file:

```
<?xml version="1.0" encoding="utf-8"?>
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

You want to modify the Banana's color to brown:

1. We need to know the path to the file on disk.
2. One overload of `XDocument.Load` receives a URI (file path).
3. Since the xml file uses a namespace, we must query with the namespace AND element name.
4. A Linq query utilizing C# 6 syntax to accommodate for the possibility of null values. Every `.` used in this query has the potential to return a null set if the condition finds no elements. Before C# 6 you would do this in multiple steps, checking for null along the way. The result is the **`<Fruit>`** element that contains the Banana. Actually an `IEnumerable<XElement>`, which is why the next step uses `FirstOfDefault()`.
5. Now we extract the FruitColor element out of the Fruit element we just found. Here we assume there is just one, or we only care about the first one.
6. If it is not null, we set the FruitColor to "Brown".
7. Finally, we save the `XDocument`, overwriting the original file on disk.

```
// 1.
string xmlFilePath = "c:\\users\\public\\fruit.xml";

// 2.
XDocument xdoc = XDocument.Load(xmlFilePath);

// 3.
```

```
XNamespace ns = "http://www.fruitauthority.fake";

//4.
var elBanana = xdoc.Descendants()?.
    Elements(ns + "FruitName")?.
    Where(x => x.Value == "Banana")?.
    Ancestors(ns + "Fruit");

// 5.
var elColor = elBanana.Elements(ns + "FruitColor").FirstOrDefault();

// 6.
if (elColor != null)
{
    elColor.Value = "Brown";
}

// 7.
xdoc.Save(xmlFilePath);
```

The file now looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Brown</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Chapter 71: C# 7.0 Features

C# 7.0 is the seventh version of C#. This version contains some new features: language support for Tuples, local functions, `out var` declarations, digit separators, binary literals, pattern matching, throw expressions, `ref return` and `ref local` and extended expression bodied members list.

Official reference: What's new in C# 7

Section 71.1: Language support for Tuples

Basics

A **tuple** is an ordered, finite list of elements. Tuples are commonly used in programming as a means to work with one single entity collectively instead of individually working with each of the tuple's elements, and to represent individual rows (ie. "records") in a relational database.

In C# 7.0, methods can have multiple return values. Behind the scenes, the compiler will use the new ValueTuple struct.

```
public (int sum, int count) GetTallies()
{
    return (1, 2);
}
```

Side note: for this to work in Visual Studio 2017, you need to get the `System.ValueTuple` package.

If a tuple-returning method result is assigned to a single variable you can access the members by their defined names on the method signature:

```
var result = GetTallies();
// > result.sum
// 1
// > result.count
// 2
```

Tuple Deconstruction

Tuple deconstruction separates a tuple into its parts.

For example, invoking `GetTallies` and assigning the return value to two separate variables deconstructs the tuple into those two variables:

```
(int tallyOne, int tallyTwo) = GetTallies();
```

`var` also works:

```
(var s, var c) = GetTallies();
```

You can also use shorter syntax, with `var` outside of `()`:

```
var (s, c) = GetTallies();
```

You can also deconstruct into existing variables:

```
int s, c;
```

```
(s, c) = GetTallies();
```

Swapping is now much simpler (no temp variable needed):

```
(b, a) = (a, b);
```

Interestingly, any object can be deconstructed by defining a `Deconstruct` method in the class:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public void Deconstruct(out string firstName, out string lastName)
    {
        firstName = FirstName;
        lastName = LastName;
    }
}

var person = new Person { FirstName = "John", LastName = "Smith" };
var (localFirstName, localLastName) = person;
```

In this case, the `(localFirstName, localLastName) = person` syntax is invoking `Deconstruct` on the `person`.

Deconstruction can even be defined in an extension method. This is equivalent to the above:

```
public static class PersonExtensions
{
    public static void Deconstruct(this Person person, out string firstName, out string lastName)
    {
        firstName = person.FirstName;
        lastName = person.LastName;
    }
}

var (localFirstName, localLastName) = person;
```

An alternative approach for the `Person` class is to define the `Name` itself as a `Tuple`. Consider the following:

```
class Person
{
    public (string First, string Last) Name { get; }

    public Person((string FirstName, string LastName) name)
    {
        Name = name;
    }
}
```

Then you can instantiate a person like so (where we can take a tuple as an argument):

```
var person = new Person(("Jane", "Smith"));

var firstName = person.Name.First; // "Jane"
var lastName = person.Name.Last;   // "Smith"
```

Tuple Initialization

You can also arbitrarily create tuples in code:

```
var name = ("John", "Smith");
Console.WriteLine(name.Item1);
// Outputs John

Console.WriteLine(name.Item2);
// Outputs Smith
```

When creating a tuple, you can assign ad-hoc item names to the members of the tuple:

```
var name = (first: "John", middle: "Q", last: "Smith");
Console.WriteLine(name.first);
// Outputs John
```

Type inference

Multiple tuples defined with the same signature (matching types and count) will be inferred as matching types. For example:

```
public (int sum, double average) Measure(List<int> items)
{
    var stats = (sum: 0, average: 0d);
    stats.sum = items.Sum();
    stats.average = items.Average();
    return stats;
}
```

`stats` can be returned since the declaration of the `stats` variable and the method's return signature are a match.

Reflection and Tuple Field Names

Member names do not exist at runtime. Reflection will consider tuples with the same number and types of members the same even if member names do not match. Converting a tuple to an `object` and then to a tuple with the same member types, but different names, will not cause an exception either.

While the ValueTuple class itself does not preserve information for member names the information is available through reflection in a TupleElementNamesAttribute. This attribute is not applied to the tuple itself but to method parameters, return values, properties and fields. This allows tuple item names to be preserved across assemblies i.e. if a method returns (string name, int count) the names name and count will be available to callers of the method in another assembly because the return value will be marked with TupleElementNameAttribute containing the values "name" and "count".

Use with generics and `async`

The new tuple features (using the underlying `ValueTuple` type) fully support generics and can be used as generic type parameter. That makes it possible to use them with the `async`/`await` pattern:

```
public async Task<(string value, int count)> GetValueAsync()
{
    string fooBar = await _stackoverflow.GetStringAsync();
    int num = await _stackoverflow.GetIntAsync();

    return (fooBar, num);
}
```

Use with collections

It may become beneficial to have a collection of tuples in (as an example) a scenario where you're attempting to find a matching tuple based on conditions to avoid code branching.

Example:

```
private readonly List<Tuple<string, string, string>> labels = new List<Tuple<string, string,
string>>()
{
    new Tuple<string, string, string>("test1", "test2", "Value"),
    new Tuple<string, string, string>("test1", "test1", "Value2"),
    new Tuple<string, string, string>("test2", "test2", "Value3"),
};

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.Item1 == firstElement && w.Item2 == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.Item3;
}
```

With the new tuples can become:

```
private readonly List<(string firstThingy, string secondThingyLabel, string foundValue)> labels =
new List<(string firstThingy, string secondThingyLabel, string foundValue)>()
{
    ("test1", "test2", "Value"),
    ("test1", "test1", "Value2"),
    ("test2", "test2", "Value3"),
}

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.firstThingy == firstElement && w.secondThingyLabel == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.foundValue;
}
```

Though the naming on the example tuple above is pretty generic, the idea of relevant labels allows for a deeper understanding of what is being attempted in the code over referencing "item1", "item2", and "item3".

Differences between ValueTuple and Tuple

The primary reason for introduction of `ValueTuple` is performance.

Type name	`ValueTuple`	`Tuple`
Class or structure	`struct`	`class`
Mutability (changing values after creation)	mutable	immutable
Naming members and other language support	yes	no (TBD)

References

- Original Tuples language feature proposal on GitHub
- A runnable VS 15 solution for C# 7.0 features
- NuGet Tuple Package

Section 71.2: Local functions

Local functions are defined within a method and aren't available outside of it. They have access to all local variables and support iterators, `async`/`await` and lambda syntax. This way, repetitions specific to a function can be functionalized without crowding the class. As a side effect, this improves intellisense suggestion performance.

Example

```
double GetCylinderVolume(double radius, double height)
{
    return getVolume();

    double getVolume()
    {
        // You can declare inner-local functions in a local function
        double GetCircleArea(double r) => Math.PI * r * r;

        // ALL parents' variables are accessible even though parent doesn't have any input.
        return GetCircleArea(radius) * height;
    }
}
```

Local functions considerably simplify code for LINQ operators, where you usually have to separate argument checks from actual logic to make argument checks instant, not delayed until after iteration started.

Example

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    if (source == null) throw new ArgumentNullException(nameof(source));
    if (predicate == null) throw new ArgumentNullException(nameof(predicate));

    return iterator();

    IEnumerable<TSource> iterator()
    {
        foreach (TSource element in source)
            if (predicate(element))
                yield return element;
    }
}
```

Local functions also support the `async` and `await` keywords.

Example

```
async Task WriteEmailsAsync()
{
    var emailRegex = new Regex(@"(?i)[a-z0-9_.+-]+@[a-z0-9-]+\.[a-z0-9-.]+");
    IEnumerable<string> emails1 = await getEmailsFromFileAsync("input1.txt");
    IEnumerable<string> emails2 = await getEmailsFromFileAsync("input2.txt");
    await writeLinesToFileAsync(emails1.Concat(emails2), "output.txt");
```

```
    async Task<IEnumerable<string>> getEmailsFromFileAsync(string fileName)
    {
        string text;

        using (StreamReader reader = File.OpenText(fileName))
        {
            text = await reader.ReadToEndAsync();
        }

        return from Match emailMatch in emailRegex.Matches(text) select emailMatch.Value;
    }

    async Task writeLinesToFileAsync(IEnumerable<string> lines, string fileName)
    {
        using (StreamWriter writer = File.CreateText(fileName))
        {
            foreach (string line in lines)
            {
                await writer.WriteLineAsync(line);
            }
        }
    }
}
```

One important thing that you may have noticed is that local functions can be defined under the `return` statement, they do **not** need to be defined above it. Additionally, local functions typically follow the "lowerCamelCase" naming convention as to more easily differentiate themselves from class scope functions.

Section 71.3: out var declaration

A common pattern in C# is using `bool TryParse(object input, out object value)` to safely parse objects.

The `out var` declaration is a simple feature to improve readability. It allows a variable to be declared at the same time that is it passed as an out parameter.

A variable declared this way is scoped to the remainder of the body at the point in which it is declared.

Example

Using `TryParse` prior to C# 7.0, you must declare a variable to receive the value before calling the function:

`Version < 7.0`

```
int value;
if (int.TryParse(input, out value))
{
    Foo(value); // ok
}
else
{
    Foo(value); // value is zero
}

Foo(value); // ok
```

In C# 7.0, you can inline the declaration of the variable passed to the `out` parameter, eliminating the need for a separate variable declaration:

`Version ≥ 7.0`

```
if (int.TryParse(input, out var value))
```

```
{
    Foo(value); // ok
}
else
{
    Foo(value); // value is zero
}

Foo(value); // still ok, the value in scope within the remainder of the body
```

If some of the parameters that a function returns in `out` is not needed you can use the *discard* operator `_`.

```
p.GetCoordinates(out var x, out _); // I only care about x
```

An `out var` declaration can be used with any existing function which already has `out` parameters. The function declaration syntax remains the same, and no additional requirements are needed to make the function compatible with an `out var` declaration. This feature is simply syntactic sugar.

Another feature of `out var` declaration is that it can be used with anonymous types.

Version ≥ 7.0

```
var a = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var groupedByMod2 = a.Select(x => new
                                  {
                                      Source = x,
                                      Mod2 = x % 2
                                  })
                     .GroupBy(x => x.Mod2)
                     .ToDictionary(g => g.Key, g => g.ToArray());
if (groupedByMod2.TryGetValue(1, out var oddElements))
{
    Console.WriteLine(oddElements.Length);
}
```

In this code we create a `Dictionary` with `int` key and array of anonymous type value. In the previous version of C# it was impossible to use `TryGetValue` method here since it required you to declare the `out` variable (which is of anonymous type!). However, with `out var` we do not need to explicitly specify the type of the `out` variable.

Limitations

Note that out var declarations are of limited use in LINQ queries as expressions are interpreted as expression lambda bodies, so the scope of the introduced variables is limited to these lambdas. For example, the following code will not work:

```
var nums =
    from item in seq
    let success = int.TryParse(item, out var tmp)
    select success ? tmp : 0; // Error: The name 'tmp' does not exist in the current context
```

References

- Original out var declaration proposal on GitHub

Section 71.4: Pattern Matching

Pattern matching extensions for C# enable many of the benefits of pattern matching from functional languages, but in a way that smoothly integrates with the feel of the underlying language

`switch` expression

Pattern matching extends the `switch` statement to switch on types:

```
class Geometry {}

class Triangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
    public int Base { get; set; }
}

class Rectangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
}

class Square : Geometry
{
    public int Width { get; set; }
}

public static void PatternMatching()
{
    Geometry g = new Square { Width = 5 };

    switch (g)
    {
        case Triangle t:
            Console.WriteLine($"{t.Width} {t.Height} {t.Base}");
            break;
        case Rectangle sq when sq.Width == sq.Height:
            Console.WriteLine($"Square rectangle: {sq.Width} {sq.Height}");
            break;
        case Rectangle r:
            Console.WriteLine($"{r.Width} {r.Height}");
            break;
        case Square s:
            Console.WriteLine($"{s.Width}");
            break;
        default:
            Console.WriteLine("<other>");
            break;
    }
}
```

`is` expression

Pattern matching extends the `is` operator to check for a type and declare a new variable at the same time.

Example

Version < 7.0

```
string s = o as string;
if(s != null)
{
    // do something with s
}
```

can be rewritten as:

Version ≥ 7.0

```
if(o is string s)
{
    //Do something with s
};
```

Also note that the scope of the pattern variable `s` is extended to outside the `if` block reaching the end of the enclosing scope, example:

```
if(someCondition)
{
   if(o is string s)
   {
      //Do something with s
   }
   else
   {
     // s is unassigned here, but accessible
   }

   // s is unassigned here, but accessible
}
// s is not accessible here
```

Section 71.5: Digit separators

The underscore `_` may be used as a digit separator. Being able to group digits in large numeric literals has a significant impact on readability.

The underscore may occur anywhere in a numeric literal except as noted below. Different groupings may make sense in different scenarios or with different numeric bases.

Any sequence of digits may be separated by one or more underscores. The `_` is allowed in decimals as well as exponents. The separators have no semantic impact - they are simply ignored.

```
int bin = 0b1001_1010_0001_0100;
int hex = 0x1b_a0_44_fe;
int dec = 33_554_432;
int weird = 1_2__3___4____5_____6______7_______8________9;
double real = 1_000.111_1e-1_000;
```

Where the `_` digit separator may not be used:

- at the beginning of the value (`_121`)
- at the end of the value (`121_` or `121.05_`)
- next to the decimal (`10_.0`)
- next to the exponent character (`1.1e_1`)
- next to the type specifier (`10_f`)
- immediately following the `0x` or `0b` in binary and hexadecimal literals (might be changed to allow e.g. `0b_1001_1000`)

Section 71.6: Binary literals

The **0b** prefix can be used to represent Binary literals.

Binary literals allow constructing numbers from zeroes and ones, which makes seeing which bits are set in the binary representation of a number much easier. This can be useful for working with binary flags.

The following are equivalent ways of specifying an `int` with value `34` (=25 + 21):

```
// Using a binary literal:
//   bits: 76543210
int a1 = 0b00100010;          // binary: explicitly specify bits

// Existing methods:
int a2 = 0x22;                // hexadecimal: every digit corresponds to 4 bits
int a3 = 34;                  // decimal: hard to visualise which bits are set
int a4 = (1 << 5) | (1 << 1); // bitwise arithmetic: combining non-zero bits
```

Flags enumerations

Before, specifying flag values for an `enum` could only be done using one of the three methods in this example:

```
[Flags]
public enum DaysOfWeek
{
    // Previously available methods:
    //          decimal        hex       bit shifting
    Monday    =  1,    //    = 0x01    = 1 << 0
    Tuesday   =  2,    //    = 0x02    = 1 << 1
    Wednesday =  4,    //    = 0x04    = 1 << 2
    Thursday  =  8,    //    = 0x08    = 1 << 3
    Friday    = 16,    //    = 0x10    = 1 << 4
    Saturday  = 32,    //    = 0x20    = 1 << 5
    Sunday    = 64,    //    = 0x40    = 1 << 6

    Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,
    Weekends = Saturday | Sunday
}
```

With binary literals it is more obvious which bits are set, and using them does not require understanding hexadecimal numbers and bitwise arithmetic:

```
[Flags]
public enum DaysOfWeek
{
    Monday    = 0b00000001,
    Tuesday   = 0b00000010,
    Wednesday = 0b00000100,
    Thursday  = 0b00001000,
    Friday    = 0b00010000,
    Saturday  = 0b00100000,
    Sunday    = 0b01000000,

    Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,
    Weekends = Saturday | Sunday
}
```

Section 71.7: throw expressions

C# 7.0 allows throwing as an expression in certain places:

```
class Person
{
```

```
    public string Name { get; }

    public Person(string name) => Name = name ?? throw new ArgumentNullException(nameof(name));

    public string GetFirstName()
    {
        var parts = Name.Split(' ');
        return (parts.Length > 0) ? parts[0] : throw new InvalidOperationException("No name!");
    }

    public string GetLastName() => throw new NotImplementedException();
}
```

Prior to C# 7.0, if you wanted to throw an exception from an expression body you would have to:

```
var spoons = "dinner,desert,soup".Split(',');

var spoonsArray = spoons.Length > 0 ? spoons : null;

if (spoonsArray == null)
{
    throw new Exception("There are no spoons");
}
```

Or

```
var spoonsArray = spoons.Length > 0
    ? spoons
    : new Func<string[]>(() =>
      {
          throw new Exception("There are no spoons");
      })();
```

In C# 7.0 the above is now simplified to:

```
var spoonsArray = spoons.Length > 0 ? spoons : throw new Exception("There are no spoons");
```

Section 71.8: Extended expression bodied members list

C# 7.0 adds accessors, constructors and finalizers to the list of things that can have expression bodies:

```
class Person
{
    private static ConcurrentDictionary<int, string> names = new ConcurrentDictionary<int,
string>();

    private int id = GetId();

    public Person(string name) => names.TryAdd(id, name); // constructors

    ~Person() => names.TryRemove(id, out _);              // finalizers

    public string Name
    {
        get => names[id];                                 // getters
        set => names[id] = value;                         // setters
    }
}
```

Also see the out var declaration section for the discard operator.

Section 71.9: ref return and ref local

Ref returns and ref locals are useful for manipulating and returning references to blocks of memory instead of copying memory without resorting to unsafe pointers.

Ref Return

```
public static ref TValue Choose<TValue>(
    Func<bool> condition, ref TValue left, ref TValue right)
{
    return condition() ? ref left : ref right;
}
```

With this you can pass two values by reference with one of them being returned based on some condition:

```
Matrix3D left = ..., right = ...;
Choose(chooser, ref left, ref right).M20 = 1.0;
```

Ref Local

```
public static ref int Max(ref int first, ref int second, ref int third)
{
    ref int max = first > second ? ref first : ref second;
    return max > third ? ref max : ref third;
}
...
int a = 1, b = 2, c = 3;
Max(ref a, ref b, ref c) = 4;
Debug.Assert(a == 1); // true
Debug.Assert(b == 2); // true
Debug.Assert(c == 4); // true
```

Unsafe Ref Operations

In `System.Runtime.CompilerServices.Unsafe` a set of unsafe operations have been defined that allow you to manipulate `ref` values as if they were pointers, basically.

For example, reinterpreting a memory address (`ref`) as a different type:

```
byte[] b = new byte[4] { 0x42, 0x42, 0x42, 0x42 };

ref int r = ref Unsafe.As<byte, int>(ref b[0]);
Assert.Equal(0x42424242, r);

0x0EF00EF0;
Assert.Equal(0xFE, b[0] | b[1] | b[2] | b[3]);
```

Beware of endianness when doing this, though, e.g. check `BitConverter.IsLittleEndian` if needed and handle accordingly.

Or iterate over an array in an unsafe manner:

```
int[] a = new int[] { 0x123, 0x234, 0x345, 0x456 };

ref int r1 = ref Unsafe.Add(ref a[0], 1);
Assert.Equal(0x234, r1);

ref int r2 = ref Unsafe.Add(ref r1, 2);
```

```
Assert.Equal(0x456, r2);

ref int r3 = ref Unsafe.Add(ref r2, -3);
Assert.Equal(0x123, r3);
```

Or the similar `Subtract`:

```
string[] a = new string[] { "abc", "def", "ghi", "jkl" };

ref string r1 = ref Unsafe.Subtract(ref a[0], -2);
Assert.Equal("ghi", r1);

ref string r2 = ref Unsafe.Subtract(ref r1, -1);
Assert.Equal("jkl", r2);

ref string r3 = ref Unsafe.Subtract(ref r2, 3);
Assert.Equal("abc", r3);
```

Additionally, one can check if two `ref` values are the same i.e. same address:

```
long[] a = new long[2];

Assert.True(Unsafe.AreSame(ref a[0], ref a[0]));
Assert.False(Unsafe.AreSame(ref a[0], ref a[1]));
```

Links

Roslyn Github Issue

System.Runtime.CompilerServices.Unsafe on github

Section 71.10: ValueTask<T>

`Task<T>` is a **class** and causes the unnecessary overhead of its allocation when the result is immediately available.

`ValueTask<T>` is a **structure** and has been introduced to prevent the allocation of a `Task` object in case the result of the **async** operation is already available at the time of awaiting.

So `ValueTask<T>` provides two benefits:

1. Performance increase

Here's a `Task<T>` example:

- Requires heap allocation
- Takes 120ns with JIT

```
async Task<int> TestTask(int d)
{
    await Task.Delay(d);
    return 10;
}
```

Here's the analog `ValueTask<T>` example:

- No heap allocation if the result is known synchronously (which it is not in this case because of the `Task.Delay`, but often is in many real-world `async`/`await` scenarios)
- Takes 65ns with JIT

```
async ValueTask<int> TestValueTask(int d)
{
    await Task.Delay(d);
    return 10;
}
```

2. Increased implementation flexibility

Implementations of an async interface wishing to be synchronous would otherwise be forced to use either `Task.Run` or `Task.FromResult` (resulting in the performance penalty discussed above). Thus there's some pressure against synchronous implementations.

But with `ValueTask<T>`, implementations are more free to choose between being synchronous or asynchronous without impacting callers.

For example, here's an interface with an asynchronous method:

```
interface IFoo<T>
{
    ValueTask<T> BarAsync();
}
```

...and here's how that method might be called:

```
IFoo<T> thing = getThing();
var x = await thing.BarAsync();
```

With `ValueTask`, the above code will work with **either synchronous or asynchronous implementations**:

Synchronous implementation:

```
class SynchronousFoo<T> : IFoo<T>
{
    public ValueTask<T> BarAsync()
    {
        var value = default(T);
        return new ValueTask<T>(value);
    }
}
```

Asynchronous implementation

```
class AsynchronousFoo<T> : IFoo<T>
{
    public async ValueTask<T> BarAsync()
    {
        var value = default(T);
        await Task.Delay(1);
        return value;
    }
}
```

Notes

Although `ValueTask` struct was being planned to be added to C# 7.0, it has been kept as another library for the time being. ValueTask<T> `System.Threading.Tasks.Extensions` package can be downloaded from Nuget Gallery

Chapter 72: C# 6.0 Features

This sixth iteration of the C# language is provided by the Roslyn compiler. This compiler came out with version 4.6 of the .NET Framework, however it can generate code in a backward compatible manner to allow targeting earlier framework versions. C# version 6 code can be compiled in a fully backwards compatible manner to .NET 4.0. It can also be used for earlier frameworks, however some features that require additional framework support may not function correctly.

Section 72.1: Exception filters

Exception filters give developers the ability to add a condition (in the form of a `boolean` expression) to a catch block, allowing the `catch` to execute only if the condition evaluates to `true`.

Exception filters allow the propagation of debug information in the original exception, where as using an `if` statement inside a `catch` block and re-throwing the exception stops the propagation of debug information in the original exception. With exception filters, the exception continues to propagate upwards in the call stack *unless* the condition is met. As a result, exception filters make the debugging experience much easier. Instead of stopping on the `throw` statement, the debugger will stop on the statement throwing the exception, with the current state and all local variables preserved. Crash dumps are affected in a similar way.

> Exception filters have been supported by the **CLR** since the beginning and they've been accessible from VB.NET and F# for over a decade by exposing a part of the CLR's exception handling model. Only after the release of C# 6.0 has the functionality also been available for C# developers.

Using exception filters

Exception filters are utilized by appending a `when` clause to the `catch` expression. It is possible to use any expression returning a `bool` in a `when` clause (except await). The declared Exception variable `ex` is accessible from within the `when` clause:

```
var SqlErrorToIgnore = 123;
try
{
    DoSQLOperations();
}
catch (SqlException ex) when (ex.Number != SqlErrorToIgnore)
{
    throw new Exception("An error occurred accessing the database", ex);
}
```

Multiple `catch` blocks with `when` clauses may be combined. The first `when` clause returning `true` will cause the exception to be caught. Its `catch` block will be entered, while the other `catch` clauses will be ignored (their `when` clauses won't be evaluated). For example:

```
try
{ ... }
catch (Exception ex) when (someCondition) //If someCondition evaluates to true,
                                          //the rest of the catches are ignored.
{ ... }
catch (NotImplementedException ex) when (someMethod()) //someMethod() will only run if
                                                       //someCondition evaluates to false
{ ... }
```

```
catch(Exception ex) // If both when clauses evaluate to false
{ ... }
```

Risky when clause

> **Caution**
>
> It can be risky to use exception filters: when an `Exception` is thrown from within the `when` clause, the `Exception` from the `when` clause is ignored and is treated as `false`. This approach allows developers to write `when` clause without taking care of invalid cases.

The following example illustrates such a scenario:

```
public static void Main()
{
    int a = 7;
    int b = 0;
    try
    {
        DoSomethingThatMightFail();
    }
    catch (Exception ex) when (a / b == 0)
    {
        // This block is never reached because a / b throws an ignored
        // DivideByZeroException which is treated as false.
    }
    catch (Exception ex)
    {
        // This block is reached since the DivideByZeroException in the
        // previous when clause is ignored.
    }
}

public static void DoSomethingThatMightFail()
{
    // This will always throw an ArgumentNullException.
    Type.GetType(null);
}
```

View Demo

Note that exception filters avoid the confusing line number problems associated with using `throw` when failing code is within the same function. For example in this case the line number is reported as 6 instead of 3:

```
1. int a = 0, b = 0;
2. try {
3.     int c = a / b;
4. }
5. catch (DivideByZeroException) {
6.     throw;
7. }
```

The exception line number is reported as 6 because the error was caught and re-thrown with the `throw` statement on line 6.

The same does not happen with exception filters:

```
1. int a = 0, b = 0;
2. try {
3.     int c = a / b;
4. }
5. catch (DivideByZeroException) when (a != 0) {
6.     throw;
7. }
```

In this example a is 0 then `catch` clause is ignored but 3 is reported as line number. This is because they **do not unwind the stack**. More specifically, the exception *is not caught* on line 5 because a in fact does equal `0` and thus there is no opportunity for the exception to be re-thrown on line 6 because line 6 does not execute.

Logging as a side effect

Method calls in the condition can cause side effects, so exception filters can be used to run code on exceptions without catching them. A common example that takes advantage of this is a `Log` method that always returns `false`. This allows tracing log information while debugging without the need to re-throw the exception.

> **Be aware that** while this seems to be a comfortable way of logging, it can be risky, especially if 3rd party logging assemblies are used. These might throw exceptions while logging in non-obvious situations that may not be detected easily (see **Risky `when(...)` clause** above).

```
try
{
    DoSomethingThatMightFail(s);
}
catch (Exception ex)
```

(Log(ex, "An error occurred")) { // This catch block will never be reached } // ... static bool Log(Exception ex, string message, params object[] args) { Debug.Print(message, args); return false; }

View Demo

The common approach in previous versions of C# was to log and re-throw the exception.

`Version < 6.0`

```
try
{
    DoSomethingThatMightFail(s);
}
catch (Exception ex)
{
     Log(ex, "An error occurred");
     throw;
}

// ...

static void Log(Exception ex, string message, params object[] args)
{
    Debug.Print(message, args);
}
```

View Demo

The `finally` block

The `finally` block executes every time whether the exception is thrown or not. One subtlety with expressions in `when` is exception filters are executed further up the stack *before* entering the inner `finally` blocks. This can cause unexpected results and behaviors when code attempts to modify global state (like the current thread's user or culture) and set it back in a `finally` block.

Example: `finally` block

```
private static bool Flag = false;

static void Main(string[] args)
{
    Console.WriteLine("Start");
    try
    {
        SomeOperation();
    }
    catch (Exception) when (EvaluatesTo())
    {
        Console.WriteLine("Catch");
    }
    finally
    {
        Console.WriteLine("Outer Finally");
    }
}

private static bool EvaluatesTo()
{
    Console.WriteLine($"EvaluatesTo: {Flag}");
    return true;
}

private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    finally
    {
        Flag = false;
        Console.WriteLine("Inner Finally");
    }
}
```

Produced Output:

> Start
> EvaluatesTo: True
> Inner Finally
> Catch
> Outer Finally

View Demo

In the example above, if the method `SomeOperation` does not wish to "leak" the global state changes to caller's `when` clauses, it should also contain a `catch` block to modify the state. For example:

```
private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    catch
    {
       Flag = false;
       throw;
    }
    finally
    {
        Flag = false;
        Console.WriteLine("Inner Finally");
    }
}
```

It is also common to see `IDisposable` helper classes leveraging the semantics of using blocks to achieve the same goal, as `IDisposable.Dispose` will always be called before an exception called within a `using` block starts bubbling up the stack.

Section 72.2: String interpolation

String interpolation allows the developer to combine `variables` and text to form a string.

Basic Example

Two `int` variables are created: `foo` and `bar`.

```
int foo = 34;
int bar = 42;

string resultString = $"The foo is {foo}, and the bar is {bar}.";

Console.WriteLine(resultString);
```

Output:

> The foo is 34, and the bar is 42.

View Demo

Braces within strings can still be used, like this:

```
var foo = 34;
var bar = 42;

// String interpolation notation (new style)
Console.WriteLine($"The foo is
```

o}}, and the bar is **{{bar}}**.");

This produces the following output:

> The foo is {foo}, and the bar is {bar}.

Using interpolation with verbatim string literals

Using `@` before the string will cause the string to be interpreted verbatim. So, e.g. Unicode characters or line breaks will stay exactly as they've been typed. However, this will not effect the expressions in an interpolated string as shown in the following example:

```
Console.WriteLine($@"In case it wasn't clear:
\u00B9
The foo
is
```

}, and the bar is **{bar}**."); Output:

> In case it wasn't clear:
> \u00B9
> The foo
> is 34,
> and the bar
> is 42.

View Demo

Expressions

With string interpolation, *expressions* within curly braces `{}` can also be evaluated. The result will be inserted at the corresponding location within the string. For example, to calculate the maximum of `foo` and `bar` and insert it, use `Math.Max` within the curly braces:`Console.WriteLine($"And the greater one is:` th.Max(foo, bar) }");

Output:

> And the greater one is: 42

Note: Any leading or trailing whitespace (including space, tab and CRLF/newline) between the curly brace and the expression is completely ignored and not included in the output

View Demo

As another example, variables can be formatted as a currency:`Console.WriteLine($"Foo formatted as a currency to 4 decimal places:` :c4}");

Output:

> Foo formatted as a currency to 4 decimal places: $34.0000

View Demo

Or they can be formatted as dates:`Console.WriteLine($"Today is: eTime.Today:dddd, MMMM dd - yyyy}");`

Output:

> Today is: Monday, July, 20 - 2015

View Demo

Statements with a Conditional (Ternary) Operator can also be evaluated within the interpolation. However, these must be wrapped in parentheses, since the colon is otherwise used to indicate formatting as shown above:

```
Console.WriteLine($"{(foo > bar ? "Foo is larger than bar!" : "Bar is larger than foo!")}");
```

Output:

> Bar is larger than foo!

View Demo

Conditional expressions and format specifiers can be mixed:

```
Console.WriteLine($"Environment: {(Environment.Is64BitProcess ? 64 : 32):00'-bit'} process");
```

Output:

> Environment: 32-bit process

Escape sequences

Escaping backslash (\) and quote (") characters works exactly the same in interpolated strings as in non-interpolated strings, for both verbatim and non-verbatim string literals:

`Console.WriteLine($"Foo is: }. In a non-verbatim string, we need to escape \" and \\ with backslashes.");`
Console.WriteLine($@"Foo is: **{foo}**. In a verbatim string, we need to escape "" with an extra quote, but we don't need to escape \");

Output:

> Foo is 34. In a non-verbatim string, we need to escape " and \ with backslashes.
> Foo is 34. In a verbatim string, we need to escape " with an extra quote, but we don't need to escape \

To include a curly brace { or } in an interpolated string, use two curly braces {{ or }}:`$"{{foo}} is: }"`

Output:

> {foo} is: 34

View Demo

FormattableString type

The type of a `$"..."` string interpolation expression is not always a simple string. The compiler decides which type to assign depending on the context:`string s = $"hello, e}";` System.FormattableString s = $"Hello, **{name}**"; System.IFormattable s = $"Hello, **{name}**";

This is also the order of type preference when the compiler needs to choose which overloaded method is going to be called.

A new type, `System.FormattableString`, represents a composite format string, along with the arguments to be formatted. Use this to write applications that handle the interpolation arguments specifically:

```
public void AddLogItem(FormattableString formattableString)
{
    foreach (var arg in formattableString.GetArguments())
    {
        // do something to interpolation argument 'arg'
    }

    // use the standard interpolation and the current culture info
    // to get an ordinary String:
    var formatted = formattableString.ToString();

    // ...
}
```

Call the above method with:`AddLogItem($"The foo is }, and the bar is **{bar}**.");` For example, one could choose not to incur the performance cost of formatting the string if the logging level was already going to filter out the log item.

Implicit conversions

There are implicit type conversions from an interpolated string:`var s = $"Foo: }";` System.IFormattable s = $"Foo: **{foo}**"; You can also produce an `IFormattable` variable that allows you to convert the string with invariant context:`var s = $"Bar: }";` System.FormattableString s = $"Bar: **{bar}**";

Current and Invariant Culture Methods

If code analysis is turned on, interpolated strings will all produce warning CA1305 (Specify `IFormatProvider`). A static method may be used to apply current culture.

```
public static class Culture
{
    public static string Current(FormattableString formattableString)
    {
        return formattableString?.ToString(CultureInfo.CurrentCulture);
    }
    public static string Invariant(FormattableString formattableString)
    {
        return formattableString?.ToString(CultureInfo.InvariantCulture);
    }
}
```

Then, to produce a correct string for the current culture, just use the expression:`Culture.Current($"interpolated` eof(string).Name} string.") Culture.Invariant($"interpolated **{typeof(string).Name}** string.") **Note**: `Current` and

`Invariant` cannot be created as extension methods because, by default, the compiler assigns type `String` to *interpolated string expression* which causes the following code to fail to compile:

```
$"interpolated {typeof(string).Name} string.".Current();
```

`FormattableString` class already contains `Invariant()` method, so the simplest way of switching to invariant culture is by relying on `using static`:`using static System.FormattableString;`p>string invariant = Invariant($"Now = **{DateTime.Now}**"); string current = $"Now = **{DateTime.Now}**";

Behind the scenes

Interpolated strings are just a syntactic sugar for `String.Format()`. The compiler (Roslyn) will turn it into a `String.Format` behind the scenes:

```
var text = $"Hello {name + lastName}";
```

The above will be converted to something like this:

```
string text = string.Format("Hello {0}", new object[] {
    name + lastName
});
```

String Interpolation and Linq

It's possible to use interpolated strings in Linq statements to increase readability further.

```
var fooBar = (from DataRow x in fooBarTable.Rows
          select string.Format("{0}{1}", x["foo"], x["bar"])).ToList();
```

Can be re-written as:

```
var fooBar = (from DataRow x in fooBarTable.Rows
          select $"{x["foo"]}{x["bar"]}").ToList();
```

Reusable Interpolated Strings

With `string.Format`, you can create reusable format strings:

```
public const string ErrorFormat = "Exception caught:\r\n{0}";

// ...

Logger.Log(string.Format(ErrorFormat, ex));
```

Interpolated strings, however, will not compile with placeholders referring to non-existent variables. The following will not compile:

```
public const string ErrorFormat = $"Exception caught:\r\n{error}";
// CS0103: The name 'error' does not exist in the current context
```

Instead, create a `Func<>` which consumes variables and returns a `String`:

```
public static Func<Exception, string> FormatError =
    error => $"Exception caught:\r\n{error}";

// ...
```

```
Logger.Log(FormatError(ex));
```

String interpolation and localization

If you're localizing your application you may wonder if it is possible to use string interpolation along with localization. Indeed, it would be nice to have the possibility to store in resource files `String`s like:`"My name is` e} {middlename} {surname}" instead of the much less readable:

```
"My name is {0} {1} {2}"
```

`String` interpolation process occurs *at compile time*, unlike formatting string with `string.Format` which occurs *at runtime*. Expressions in an interpolated string must reference names in the current context and need to be stored in resource files. That means that if you want to use localization you have to do it like:

```
var FirstName = "John";

// method using different resource file "strings"
// for French ("strings.fr.resx"), German ("strings.de.resx"),
// and English ("strings.en.resx")
void ShowMyNameLocalized(string name, string middlename = "", string surname = "")
{
    // get localized string
    var localizedMyNameIs = Properties.strings.Hello;
    // insert spaces where necessary
    name = (string.IsNullOrWhiteSpace(name) ? "" : name + " ");
    middlename = (string.IsNullOrWhiteSpace(middlename) ? "" : middlename + " ");
    surname = (string.IsNullOrWhiteSpace(surname) ? "" : surname + " ");
    // display it
    Console.WriteLine($"{localizedMyNameIs} {name}{middlename}{surname}".Trim());
}

// switch to French and greet John
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("fr-FR");
ShowMyNameLocalized(FirstName);

// switch to German and greet John
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("de-DE");
ShowMyNameLocalized(FirstName);

// switch to US English and greet John
Thread.CurrentThread.CurrentUICulture = CultureInfo.GetCultureInfo("en-US");
ShowMyNameLocalized(FirstName);
```

If the resource strings for the languages used above are correctly stored in the individual resource files, you should get the following output:

> Bonjour, mon nom est John
> Hallo, mein Name ist John
> Hello, my name is John

Note that this implies that the name follows the localized string in every language. If that is not the case, you need to add placeholders to the resource strings and modify the function above or you need to query the culture info in the function and provide a switch case statement containing the different cases. For more details about resource files, see How to use localization in C#.

It is a good practice to use a default fallback language most people will understand, in case a translation is not available. I suggest to use English as default fallback language.

Recursive interpolation

Although not very useful, it is allowed to use an interpolated `string` recursively inside another's curly brackets:

```
Console.WriteLine($"String has {$"My class is called {nameof(MyClass)}.".Length} chars:");
Console.WriteLine($"My class is called {nameof(MyClass)}.");
```

Output:

> String has 27 chars:

> My class is called MyClass.

Section 72.3: Auto-property initializers

Introduction

Properties can be initialized with the `=` operator after the closing `}`. The `Coordinate` class below shows the available options for initializing a property:

Version ≥ 6.0

```
public class Coordinate
{
    public int X { get; set; } = 34; // get or set auto-property with initializer

    public int Y { get; } = 89;      // read-only auto-property with initializer
}
```

Accessors With Different Visibility

You can initialize auto-properties that have different visibility on their accessors. Here's an example with a protected setter:

```
    public string Name { get; protected set; } = "Cheeze";
```

The accessor can also be `internal`, `internal protected`, or `private`.

Read-Only Properties

In addition to flexibility with visibility, you can also initialize read-only auto-properties. Here's an example:

```
    public List<string> Ingredients { get; } =
        new List<string> { "dough", "sauce", "cheese" };
```

This example also shows how to initialize a property with a complex type. Also, auto-properties can't be write-only, so that also precludes write-only initialization.

Old style (pre C# 6.0)

Before C# 6, this required much more verbose code. We were using one extra variable called backing property for the property to give default value or to initialize the public property like below,

Version < 6.0

```
public class Coordinate
{
    private int _x = 34;
    public int X { get { return _x; } set { _x = value; } }

    private readonly int _y = 89;
    public int Y { get { return _y; } }

    private readonly int _z;
    public int Z { get { return _z; } }

    public Coordinate()
    {
        _z = 42;
    }
}
```

Note: *Before C# 6.0, you could still initialize read and write* ***auto implemented properties*** *(properties with a getter and a setter) from within the constructor, but you could not initialize the property inline with its declaration*

View Demo

Usage

Initializers must evaluate to static expressions, just like field initializers. If you need to reference non-static members, you can either initialize properties in constructors like before, or use expression-bodied properties. Non-static expressions, like the one below (commented out), will generate a compiler error:

```
// public decimal X { get; set; } = InitMe();  // generates compiler error

decimal InitMe() { return 4m; }
```

But static methods **can** be used to initialize auto-properties:

```
public class Rectangle
{
    public double Length { get; set; } = 1;
    public double Width { get; set; } = 1;
    public double Area { get; set; } = CalculateArea(1, 1);

    public static double CalculateArea(double length, double width)
    {
        return length * width;
    }
}
```

This method can also be applied to properties with different level of accessors:

```
public short Type { get; private set; } = 15;
```

The auto-property initializer allows assignment of properties directly within their declaration. For read-only

properties, it takes care of all the requirements required to ensure the property is immutable. Consider, for example, the `FingerPrint` class in the following example:

```
public class FingerPrint
{
  public DateTime TimeStamp { get; } = DateTime.UtcNow;

  public string User { get; } =
    System.Security.Principal.WindowsPrincipal.Current.Identity.Name;

  public string Process { get; } =
    System.Diagnostics.Process.GetCurrentProcess().ProcessName;
}
```

View Demo

Cautionary notes

Take care to not confuse auto-property or field initializers with similar-looking expression-body methods which make use of `=>` as opposed to `=`, and fields which do not include `{ get; }`.

For example, each of the following declarations are different.

```
public class UserGroupDto
{
    // Read-only auto-property with initializer:
    public ICollection<UserDto> Users1 { get; } = new HashSet<UserDto>();

    // Read-write field with initializer:
    public ICollection<UserDto> Users2 = new HashSet<UserDto>();

    // Read-only auto-property with expression body:
    public ICollection<UserDto> Users3 => new HashSet<UserDto>();
}
```

Missing `{ get; }` in the property declaration results in a public field. Both read-only auto-property `Users1` and read-write field `Users2` are initialized only once, but a public field allows changing collection instance from outside the class, which is usually undesirable. Changing a read-only auto-property with expression body to read-only property with initializer requires not only removing `>` from `=>`, but adding `{ get; }`.

The different symbol (`=>` instead of `=`) in `Users3` results in each access to the property returning a new instance of the `HashSet<UserDto>` which, while valid C# (from the compiler's point of view) is unlikely to be the desired behavior when used for a collection member.

The above code is equivalent to:

```
public class UserGroupDto
{
    // This is a property returning the same instance
    // which was created when the UserGroupDto was instantiated.
    private ICollection<UserDto> _users1 = new HashSet<UserDto>();
    public ICollection<UserDto> Users1 { get { return _users1; } }

    // This is a field returning the same instance
    // which was created when the UserGroupDto was instantiated.
    public virtual ICollection<UserDto> Users2 = new HashSet<UserDto>();
```

```
    // This is a property which returns a new HashSet<UserDto> as
    // an ICollection<UserDto> on each call to it.
    public ICollection<UserDto> Users3 { get { return new HashSet<UserDto>(); } }
}
```

Section 72.4: Null propagation

The `?.` operator and `?[...]` operator are called the null-conditional operator. It is also sometimes referred to by other names such as the safe navigation operator.

This is useful, because if the `.` (member accessor) operator is applied to an expression that evaluates to `null`, the program will throw a `NullReferenceException`. If the developer instead uses the `?.` (null-conditional) operator, the expression will evaluate to null instead of throwing an exception.

Note that if the `?.` operator is used and the expression is non-null, `?.` and `.` are equivalent.

Basics

```
var teacherName = classroom.GetTeacher().Name;
// throws NullReferenceException if GetTeacher() returns null
```

View Demo

If the `classroom` does not have a teacher, `GetTeacher()` may return `null`. When it is `null` and the `Name` property is accessed, a `NullReferenceException` will be thrown.

If we modify this statement to use the `?.` syntax, the result of the entire expression will be `null`:

```
var teacherName = classroom.GetTeacher()?.Name;
// teacherName is null if GetTeacher() returns null
```

View Demo

Subsequently, if `classroom` could also be `null`, we could also write this statement as:

```
var teacherName = classroom?.GetTeacher()?.Name;
// teacherName is null if GetTeacher() returns null OR classroom is null
```

View Demo

This is an example of short-circuiting: When any conditional access operation using the null-conditional operator evaluates to null, the entire expression evaluates to null immediately, without processing the rest of the chain.

When the terminal member of an expression containing the null-conditional operator is of a value type, the expression evaluates to a `Nullable<T>` of that type and so cannot be used as a direct replacement for the expression without `?.`.

```
bool hasCertification = classroom.GetTeacher().HasCertification;
// compiles without error but may throw a NullReferenceException at runtime

bool hasCertification = classroom?.GetTeacher()?.HasCertification;
// compile time error: implicit conversion from bool? to bool not allowed

bool? hasCertification = classroom?.GetTeacher()?.HasCertification;
// works just fine, hasCertification will be null if any part of the chain is null
```

```
bool hasCertification = classroom?.GetTeacher()?.HasCertification.GetValueOrDefault();
// must extract value from nullable to assign to a value type variable
```

Use with the Null-Coalescing Operator (??)

You can combine the null-conditional operator with the Null-coalescing Operator (`??`) to return a default value if the expression resolves to `null`. Using our example above:

```
var teacherName = classroom?.GetTeacher()?.Name ?? "No Name";
// teacherName will be "No Name" when GetTeacher()
// returns null OR classroom is null OR Name is null
```

Use with Indexers

The null-conditional operator can be used with indexers:

```
var firstStudentName = classroom?.Students?[0]?.Name;
```

In the above example:

- The first `?.` ensures that `classroom` is not `null`.
- The second `?` ensures that the entire `Students` collection is not `null`.
- The third `?.` after the indexer ensures that the `[0]` indexer did not return a `null` object. It should be noted that this operation can **still** throw an `IndexOutOfRangeException`.

Use with void Functions

Null-conditional operator can also be used with `void` functions. However in this case, the statement will not evaluate to `null`. It will just prevent a `NullReferenceException`.

```
List<string> list = null;
list?.Add("hi");          // Does not evaluate to null
```

Use with Event Invocation

Assuming the following event definition:

```
private event EventArgs OnCompleted;
```

When invoking an event, traditionally, it is best practice to check if the event is `null` in case no subscribers are present:

```
var handler = OnCompleted;
if (handler != null)
{
    handler(EventArgs.Empty);
}
```

Since the null-conditional operator has been introduced, the invocation can be reduced to a single line:

```
OnCompleted?.Invoke(EventArgs.Empty);
```

Limitations

Null-conditional operator produces rvalue, not lvalue, that is, it cannot be used for property assignment, event subscription etc. For example, the following code will not work:

```
// Error: The left-hand side of an assignment must be a variable, property or indexer
Process.GetProcessById(1337)?.EnableRaisingEvents = true;
// Error: The event can only appear on the left hand side of += or -=
Process.GetProcessById(1337)?.Exited += OnProcessExited;
```

Gotchas

Note that:

```
int? nameLength = person?.Name.Length;    // safe if 'person' is null
```

is **not** the same as:

```
int? nameLength = (person?.Name).Length;  // avoid this
```

because the former corresponds to:

```
int? nameLength = person != null ? (int?)person.Name.Length : null;
```

and the latter corresponds to:

```
int? nameLength = (person != null ? person.Name : null).Length;
```

Despite ternary operator `?:` is used here for explaining the difference between two cases, these operators are not equivalent. This can be easily demonstrated with the following example:

```
void Main()
{
    var foo = new Foo();
    Console.WriteLine("Null propagation");
    Console.WriteLine(foo.Bar?.Length);

    Console.WriteLine("Ternary");
    Console.WriteLine(foo.Bar != null ? foo.Bar.Length : (int?)null);
}

class Foo
{
    public string Bar
    {
        get
        {
            Console.WriteLine("I was read");
            return string.Empty;
        }
    }
}
```

Which outputs:

> Null propagation
> I was read
> 0

> Ternary
> I was read
> I was read
> 0

View Demo

To avoid multiple invocations equivalent would be:

```
var interimResult = foo.Bar;
Console.WriteLine(interimResult != null ? interimResult.Length : (int?)null);
```

And this difference somewhat explains why null propagation operator is not yet supported in expression trees.

Section 72.5: Expression-bodied function members

Expression-bodied function members allow the use of lambda expressions as member bodies. For simple members, it can result in cleaner and more readable code.

Expression-bodied functions can be used for properties, indexers, methods, and operators.

Properties

```
public decimal TotalPrice => BasePrice + Taxes;
```

Is equivalent to:

```
public decimal TotalPrice
{
    get
    {
        return BasePrice + Taxes;
    }
}
```

When an expression-bodied function is used with a property, the property is implemented as a getter-only property.

View Demo

Indexers

```
public object this[string key] => dictionary[key];
```

Is equivalent to:

```
public object this[string key]
{
    get
    {
        return dictionary[key];
    }
}
```

Methods

```
static int Multiply(int a, int b) => a * b;
```

Is equivalent to:

```
static int Multiply(int a, int b)
{
    return a * b;
}
```

Which can also be used with `void` methods:

```
public void Dispose() => resource?.Dispose();
```

An override of `ToString` could be added to the `Pair<T>` class:

```
public override string ToString() => $"{First}, {Second}";
```

Additionally, this simplistic approach works with the `override` keyword:

```
public class Foo
{
    public int Bar { get; }

    public string override ToString() => $"Bar: {Bar}";
}
```

Operators

This also can be used by operators:

```
public class Land
{
    public double Area { get; set; }

    public static Land operator +(Land first, Land second) =>
        new Land { Area = first.Area + second.Area };
}
```

Limitations

Expression-bodied function members have some limitations. They can't contain block statements and any other statements that contain blocks: `if`, `switch`, `for`, `foreach`, `while`, do, `try`, etc.

Some `if` statements can be replaced with ternary operators. Some `for` and `foreach` statements can be converted to LINQ queries, for example:

```
IEnumerable<string> Digits
{
    get
    {
        for (int i = 0; i < 10; i++)
            yield return i.ToString();
    }
}
```

```
IEnumerable<string> Digits => Enumerable.Range(0, 10).Select(i => i.ToString());
```

In all other cases, the old syntax for function members can be used.

Expression-bodied function members can contain `async`/`await`, but it's often redundant:

```
async Task<int> Foo() => await Bar();
```

Can be replaced with:

```
Task<int> Foo() => Bar();
```

Section 72.6: Operator nameof

The `nameof` operator returns the name of a code element as a `string`. This is useful when throwing exceptions related to method arguments and also when implementing `INotifyPropertyChanged`.

```
public string SayHello(string greeted)
{
    if (greeted == null)
        throw new ArgumentNullException(nameof(greeted));

    Console.WriteLine("Hello, " + greeted);
}
```

The `nameof` operator is evaluated at compile time and changes the expression into a string literal. This is also useful for strings that are named after their member that exposes them. Consider the following:

```
public static class Strings
{
    public const string Foo = nameof(Foo); // Rather than Foo = "Foo"
    public const string Bar = nameof(Bar); // Rather than Bar = "Bar"
}
```

Since `nameof` expressions are compile-time constants, they can be used in attributes, `case` labels, `switch` statements, and so on.

It is convenient to use `nameof` with `Enum`s. Instead of:

```
Console.WriteLine(Enum.One.ToString());
```

it is possible to use:

```
Console.WriteLine(nameof(Enum.One))
```

The output will be `One` in both cases.

The `nameof` operator can access non-static members using static-like syntax. Instead of doing:

```
string foo = "Foo";
string lengthName = nameof(foo.Length);
```

Can be replaced with:

```
string lengthName = nameof(string.Length);
```

The output will be `Length` in both examples. However, the latter prevents the creation of unnecessary instances.

Although the `nameof` operator works with most language constructs, there are some limitations. For example, you cannot use the `nameof` operator on open generic types or method return values:

```
public static int Main()
{
    Console.WriteLine(nameof(List<>)); // Compile-time error
    Console.WriteLine(nameof(Main())); // Compile-time error
}
```

Furthermore, if you apply it to a generic type, the generic type parameter will be ignored:

```
Console.WriteLine(nameof(List<int>));  // "List"
Console.WriteLine(nameof(List<bool>)); // "List"
```

For more examples, see this topic dedicated to `nameof`.

Workaround for previous versions (more detail)

Although the `nameof` operator does not exist in C# for versions prior to 6.0, similar functionality can be had by using `MemberExpression` as in the following:

`Version < 6.0`

Expression:

```
public static string NameOf<T>(Expression<Func<T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}

public static string NameOf<TObj, T>(Expression<Func<TObj, T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}
```

Usage:

```
string variableName = NameOf(() => variable);
string propertyName = NameOf((Foo o) => o.Bar);
```

Note that this approach causes an expression tree to be created on every call, so the performance is much worse compared to `nameof` operator which is evaluated at compile time and has zero overhead at runtime.

Section 72.7: Using static type

The `using static [Namespace.Type]` directive allows the importing of static members of types and enumeration values. Extension methods are imported as extension methods (from just one type), not into top-level scope.

Version ≥ 6.0

```
using static System.Console;
using static System.ConsoleColor;
using static System.Math;

class Program
{
    static void Main()
    {
        BackgroundColor = DarkBlue;
        WriteLine(Sqrt(2));
    }
}
```

Live Demo Fiddle

Version < 6.0

```
using System;

class Program
{
    static void Main()
    {
        Console.BackgroundColor = ConsoleColor.DarkBlue;
        Console.WriteLine(Math.Sqrt(2));
    }
}
```

Section 72.8: Index initializers

Index initializers make it possible to create and initialize objects with indexes at the same time.

This makes initializing Dictionaries very easy:

```
var dict = new Dictionary<string, int>()
{
    ["foo"] = 34,
    ["bar"] = 42
};
```

Any object that has an indexed getter or setter can be used with this syntax:

```
class Program
{
    public class MyClassWithIndexer
    {
        public int this[string index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
    }
```

```
    public static void Main()
    {
        var x = new MyClassWithIndexer()
        {
            ["foo"] = 34,
            ["bar"] = 42
        };

        Console.ReadKey();
    }
}
```

Output:

> Index: foo, value: 34
> Index: bar, value: 42

View Demo

If the class has multiple indexers it is possible to assign them all in a single group of statements:

```
class Program
{
    public class MyClassWithIndexer
    {
        public int this[string index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
        public string this[int index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
    }

    public static void Main()
    {
        var x = new MyClassWithIndexer()
        {
            ["foo"] = 34,
            ["bar"] = 42,
            [10] = "Ten",
            [42] = "Meaning of life"
        };
    }
}
```

Output:

> Index: foo, value: 34
> Index: bar, value: 42

> Index: 10, value: Ten
> Index: 42, value: Meaning of life

It should be noted that the indexer `set` accessor might behave differently compared to an `Add` method (used in collection initializers).

For example:

```
var d = new Dictionary<string, int>
{
    ["foo"] = 34,
    ["foo"] = 42,
}; // does not throw, second value overwrites the first one
```

versus:

```
var d = new Dictionary<string, int>
{
    { "foo", 34 },
    { "foo", 42 },
}; // run-time ArgumentException: An item with the same key has already been added.
```

Section 72.9: Improved overload resolution

Following snippet shows an example of passing a method group (as opposed to a lambda) when a delegate is expected. Overload resolution will now resolve this instead of raising an ambiguous overload error due to the ability of **C# 6** to check the return type of the method that was passed.

```
using System;
public class Program
{
    public static void Main()
    {
        Overloaded(DoSomething);
    }

    static void Overloaded(Action action)
    {
       Console.WriteLine("overload with action called");
    }

    static void Overloaded(Func<int> function)
    {
       Console.WriteLine("overload with Func<int> called");
    }

    static int DoSomething()
    {
        Console.WriteLine(0);
        return 0;
    }
}
```

Results:

```
Version = 6.0
```

Output

> overload with Func<int> called

View Demo

`Version = 5.0`

Error

> error CS0121: The call is ambiguous between the following methods or properties: 'Program.Overloaded(System.Action)' and 'Program.Overloaded(System.Func)'

C# 6 can also handle well the following case of exact matching for lambda expressions which would have resulted in an error in **C# 5**.

```
using System;

class Program
{
    static void Foo(Func<Func<long>> func) {}
    static void Foo(Func<Func<int>> func) {}

    static void Main()
    {
        Foo(() => () => 7);
    }
}
```

Section 72.10: Await in catch and finally

It is possible to use `await` expression to apply await operator to Tasks or Task(Of TResult) in the `catch` and `finally` blocks in C#6.

It was not possible to use the `await` expression in the `catch` and `finally` blocks in earlier versions due to compiler limitations. C#6 makes awaiting async tasks a lot easier by allowing the `await` expression.

```
try
{
    //since C#5
    await service.InitializeAsync();
}
catch (Exception e)
{
    //since C#6
    await logger.LogAsync(e);
}
finally
{
    //since C#6
    await service.CloseAsync();
}
```

It was required in C# 5 to use a `bool` or declare an `Exception` outside the try catch to perform async operations.

This method is shown in the following example:

```
bool error = false;
Exception ex = null;

try
{
    // Since C#5
    await service.InitializeAsync();
}
catch (Exception e)
{
    // Declare bool or place exception inside variable
    error = true;
    ex = e;
}

// If you don't use the exception
if (error)
{
    // Handle async task
}

// If want to use information from the exception
if (ex != null)
{
    await logger.LogAsync(e);
}

// Close the service, since this isn't possible in the finally
await service.CloseAsync();
```

Section 72.11: Minor changes and bugfixes

Parentheses are now forbidden around named parameters. The following compiles in C#5, but not C#6

Version ≤ 5.0

```
Console.WriteLine((value: 23));
```

Operands of `is` and `as` are no longer allowed to be method groups. The following compiles in C#5, but not C#6

Version ≤ 5.0

```
var result = "".Any is byte;
```

> The native compiler allowed this (although it did show a warning), and in fact didn't even check extension method compatibility, allowing crazy things like `1.Any is string` or `IDisposable.Dispose is object`.

See this reference for updates on changes.

Section 72.12: Using an extension method for collection initialization

Collection initialization syntax can be used when instantiating any class which implements `IEnumerable` and has a method named `Add` which takes a single parameter.

In previous versions, this `Add` method had to be an **instance** method on the class being initialized. In C#6, it can also be an extension method.

```
public class CollectionWithAdd : IEnumerable
{
    public void Add<T>(T item)
    {
        Console.WriteLine("Item added with instance add method: " + item);
    }

    public IEnumerator GetEnumerator()
    {
        // Some implementation here
    }
}

public class CollectionWithoutAdd : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        // Some implementation here
    }
}

public static class Extensions
{
    public static void Add<T>(this CollectionWithoutAdd collection, T item)
    {
        Console.WriteLine("Item added with extension add method: " + item);
    }
}

public class Program
{
    public static void Main()
    {
        var collection1 = new CollectionWithAdd{1,2,3}; // Valid in all C# versions
        var collection2 = new CollectionWithoutAdd{4,5,6}; // Valid only since C# 6
    }
}
```

This will output:

> Item added with instance add method: 1
> Item added with instance add method: 2
> Item added with instance add method: 3
> Item added with extension add method: 4
> Item added with extension add method: 5
> Item added with extension add method: 6

Section 72.13: Disable Warnings Enhancements

In C# 5.0 and earlier the developer could only suppress warnings by number. With the introduction of Roslyn Analyzers, C# needs a way to disable warnings issued from specific libraries. With C# 6.0 the pragma directive can suppress warnings by name.

Before:

```
#pragma warning disable 0501
```

C# 6.0:

```
#pragma warning disable CS0501
```

Chapter 73: C# 5.0 Features

Method/Modifier with Parameter	Details
`Type<T>`	`T` is the return type

Section 73.1: Async & Await

`async` and `await` are two operators that are intended to improve performance by freeing up Threads and waiting for operations to complete before moving forward.

Here's an example of getting a string before returning it's length:

```
//This method is async because:
//1. It has async and Task or Task<T> as modifiers
//2. It ends in "Async"
async Task<int> GetStringLengthAsync(string URL){
    HttpClient client = new HttpClient();
    //Sends a GET request and returns the response body as a string
    Task<string> getString = client.GetStringAsync(URL);
    //Waits for getString to complete before returning its length
    string contents = await getString;
    return contents.Length;
}

private async void doProcess(){
    int length = await GetStringLengthAsync("http://example.com/");
    //Waits for all the above to finish before printing the number
    Console.WriteLine(length);
}
```

Here's another example of downloading a file and handling what happens when it's progress has changed and when the download completes (there are two ways to do this):

Method 1:

```
//This one using async event handlers, but not async coupled with await
private void DownloadAndUpdateAsync(string uri, string DownloadLocation){
    WebClient web = new WebClient();
    //Assign the event handler
    web.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
    web.DownloadFileCompleted += new AsyncCompletedEventHandler(FileCompleted);
    //Download the file asynchronously
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);
}

//event called for when download progress has changed
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e){
    //example code
    int i = 0;
    i++;
    doSomething();
}

//event called for when download has finished
private void FileCompleted(object sender, AsyncCompletedEventArgs e){
    Console.WriteLine("Completed!")
}
```

Method 2:

```
//however, this one does
//Refer to first example on why this method is async
private void DownloadAndUpdateAsync(string uri, string DownloadLocation){
    WebClient web = new WebClient();
    //Assign the event handler
    web.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
    //Download the file async
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);
    //Notice how there is no complete event, instead we're using techniques from the first example
}
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e){
    int i = 0;
    i++;
    doSomething();
}
private void doProcess(){
    //Wait for the download to finish
    await DownloadAndUpdateAsync(new Uri("http://example.com/file"))
    doSomething();
}
```

Section 73.2: Caller Information Attributes

C.I.A.s are intended as a simple way of getting attributes from whatever is calling the targeted method. There is really only 1 way to use them and there are only 3 attributes.

Example:

```
//This is the "calling method": the method that is calling the target method
public void doProcess()
{
    GetMessageCallerAttributes("Show my attributes.");
}
//This is the target method
//There are only 3 caller attributes
public void GetMessageCallerAttributes(string message,
    //gets the name of what is calling this method
    [System.Runtime.CompilerServices.CallerMemberName] string memberName = "",
    //gets the path of the file in which the "calling method" is in
    [System.Runtime.CompilerServices.CallerFilePath] string sourceFilePath = "",
    //gets the line number of the "calling method"
    [System.Runtime.CompilerServices.CallerLineNumber] int sourceLineNumber = 0)
{
    //Writes lines of all the attributes
    System.Diagnostics.Trace.WriteLine("Message: " + message);
    System.Diagnostics.Trace.WriteLine("Member: " + memberName);
    System.Diagnostics.Trace.WriteLine("Source File Path: " + sourceFilePath);
    System.Diagnostics.Trace.WriteLine("Line Number: " + sourceLineNumber);
}
```

Example Output:

```
//Message: Show my attributes.
//Member: doProcess
//Source File Path: c:\Path\To\The\File
//Line Number: 13
```

Chapter 74: C# 4.0 Features

Section 74.1: Optional parameters and named arguments

We can omit the argument in the call if that argument is an Optional Argument Every Optional Argument has its own default value It will take default value if we do not supply the value A default value of a Optional Argument must be a

1. Constant expression.
2. Must be a value type such as enum or struct.
3. Must be an expression of the form default(valueType)

It must be set at the end of parameter list

Method parameters with default values:

```
public void ExampleMethod(int required, string optValue = "test", int optNum = 42)
{
    //...
}
```

As said by MSDN, A named argument ,

Enables you to pass the argument to the function by associating the parameter's name No needs for remembering the parameters position that we are not aware of always. No need to look the order of the parameters in the parameters list of called function. We can specify parameter for each arguments by its name.

Named arguments:

```
// required = 3, optValue = "test", optNum = 4
ExampleMethod(3, optNum: 4);
// required = 2, optValue = "foo", optNum = 42
ExampleMethod(2, optValue: "foo");
// required = 6, optValue = "bar", optNum = 1
ExampleMethod(optNum: 1, optValue: "bar", required: 6);
```

Limitation of using a Named Argument

Named argument specification must appear after all fixed arguments have been specified.

If you use a named argument before a fixed argument you will get a compile time error as follows.

Named argument specification must appear after all fixed arguments have been specified

Section 74.2: Variance

Generic interfaces and delegates can have their type parameters marked as *covariant* or *contravariant* using the `out` and `in` keywords respectively. These declarations are then respected for type conversions, both implicit and explicit, and both compile time and run time.

For example, the existing interface `IEnumerable<T>` has been redefined as being covariant:

```
interface IEnumerable<out T>
{
    IEnumerator<T> GetEnumerator();
}
```

The existing interface IComparer has been redefined as being contravariant:

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

Section 74.3: Dynamic member lookup

A new pseudo-type `dynamic` is introduced into the C# type system. It is treated as `System.Object`, but in addition, any member access (method call, field, property, or indexer access, or a delegate invocation) or application of an operator on a value of such type is permitted without any type checking, and its resolution is postponed until run-time. This is known as duck typing or late binding. For example:

```
// Returns the value of Length property or field of any object
int GetLength(dynamic obj)
{
    return obj.Length;
}

GetLength("Hello, world");        // a string has a Length property,
GetLength(new int[] { 1, 2, 3 }); // and so does an array,
GetLength(42);                    // but not an integer - an exception will be thrown
                                  // in GetLength method at run-time
```

In this case, dynamic type is used to avoid more verbose Reflection. It still uses Reflection under the hood, but it's usually faster thanks to caching.

This feature is primarily targeted at interoperability with dynamic languages.

```
// Initialize the engine and execute a file
var runtime = ScriptRuntime.CreateFromConfiguration();
dynamic globals = runtime.Globals;
runtime.ExecuteFile("Calc.rb");

// Use Calc type from Ruby
dynamic calc = globals.Calc.@new();
calc.valueA = 1337;
calc.valueB = 666;
dynamic answer = calc.Calculate();
```

Dynamic type has applications even in mostly statically typed code, for example it makes double dispatch posible without implementing Visitor pattern.

Section 74.4: Optional ref keyword when using COM

The ref keyword for callers of methods is now optional when calling into methods supplied by COM interfaces. Given a COM method with the signature

```
void Increment(ref int x);
```

the invocation can now be written as either

```
Increment(0); // no need for "ref" or a place holder variable any more
```

Chapter 75: C# 3.0 Features

Section 75.1: Implicitly typed variables (var)

The `var` keyword allows a programmer to implicitly type a variable at compile time. `var` declarations have the same type as explicitly declared variables.

```
var squaredNumber = 10 * 10;
var squaredNumberDouble = 10.0 * 10.0;
var builder = new StringBuilder();
var anonymousObject = new
{
    One = SquaredNumber,
    Two = SquaredNumberDouble,
    Three = Builder
}
```

The types of the above variables are `int`, `double`, `StringBuilder`, and an anonymous type respectively.

It is important to note that a `var` variable is not dynamically typed. `SquaredNumber = Builder` is not valid since you are trying to set an `int` to an instance of `StringBuilder`

Section 75.2: Language Integrated Queries (LINQ)

```
//Example 1
int[] array = { 1, 5, 2, 10, 7 };

// Select squares of all odd numbers in the array sorted in descending order
IEnumerable<int> query = from x in array
                         where x % 2 == 1
                         orderby x descending
                         select x * x;
// Result: 49, 25, 1
```

Example from wikipedia article on C# 3.0, LINQ sub-section

Example 1 uses query syntax which was designed to look similar to SQL queries.

```
//Example 2
IEnumerable<int> query = array.Where(x => x % 2 == 1)
    .OrderByDescending(x => x)
    .Select(x => x * x);
// Result: 49, 25, 1 using 'array' as defined in previous example
```

Example from wikipedia article on C# 3.0, LINQ sub-section

Example 2 uses method syntax to achieve the same outcome as example 1.

It is important to note that, in C#, LINQ query syntax is syntactic sugar for LINQ method syntax. The compiler translates the queries into method calls at compile time. Some queries have to be expressed in method syntax. From MSDN - "For example, you must use a method call to express a query that retrieves the number of elements that match a specified condition."

Section 75.3: Lambda expresions

Lambda Expresions are an extension of anonymous methods that allow for implicitly typed parameters and return values. Their syntax is less verbose than anonymous methods and follows a functional programming style.

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var numberList = new List<int> {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        var sumOfSquares = numberList.Select( number => number * number )
            .Aggregate( (int first, int second) => { return first + second; } );
        Console.WriteLine( sumOfSquares );
    }
}
```

The above code will output the sum of the squares of the numbers 1 through 10 to the console.

The first lambda expression squares the numbers in the list. Since there is only 1 parameter parenthesis may be omitted. You can include parenthesis if you wish:

```
.Select( (number) => number * number);
```

or explicitly type the parameter but then parenthesis are required:

```
.Select( (int number) => number * number);
```

The lambda body is an expression and has an implicit return. You can use a statement body if you want as well. This is useful for more complex lambdas.

```
.Select( number => { return number * number; } );
```

The select method returns a new IEnumerable with the computed values.

The second lambda expression sums the numbers in list returned from the select method. Parentheses are required as there are multiple parameters. The types of the parameters are explicitly typed but this is not necessary. The below method is equivalent.

```
.Aggregate( (first, second) => { return first + second; } );
```

As is this one:

```
.Aggregate( (int first, int second) => first + second );
```

Section 75.4: Anonymous types

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first. The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler.

You can make anonymous types by using the `new` keyword followed by a curly brace *({)*. Inside the curly braces, you

could define properties like on code below.

```
var v = new { Amount = 108, Message = "Hello" };
```

It's also possible to create an array of anonymous types. See code below:

```
var a = new[] {
    new {
        Fruit = "Apple",
        Color = "Red"
    },
    new {
        Fruit = "Banana",
        Color = "Yellow"
    }
};
```

Or use it with LINQ queries:

```
var productQuery = from prod in products
                   select new { prod.Color, prod.Price };
```

Chapter 76: Exception Handling

Section 76.1: Creating Custom Exceptions

You are allowed to implement custom exceptions that can be thrown just like any other exception. This makes sense when you want to make your exceptions distinguishable from other errors during runtime.

In this example we will create a custom exception for clear handling of problems the application may have while parsing a complex input.

Creating Custom Exception Class

To create a custom exception create a sub-class of `Exception`:

```
public class ParserException : Exception
{
    public ParserException() :
      base("The parsing went wrong and we have no additional information.") { }
}
```

Custom exception become very useful when you want to provide additional information to the catcher:

```
public class ParserException : Exception
{
    public ParserException(string fileName, int lineNumber) :
      base($"Parser error in {fileName}:{lineNumber}")
    {
      FileName = fileName;
      LineNumber = lineNumber;
    }
    public string FileName {get; private set;}
    public int LineNumber {get; private set;}
}
```

Now, when you `catch(ParserException x)` you will have additional semantics to fine-tune exception handling.

Custom classes can implement the following features to support additional scenarios.

re-throwing

During the parsing process, the original exception is still of interest. In this example it is a `FormatException` because the code attempts to parse a piece of string, which is expected to be a number. In this case the custom exception should support the inclusion of the '**InnerException**':

```
//new constructor:
ParserException(string msg, Exception inner) : base(msg, inner) {
}
```

serialization

In some cases your exceptions may have to cross AppDomain boundaries. This is the case if your parser is running in its own AppDomain to support hot reloading of new parser configurations. In Visual Studio, you can use `Exception` template to generate code like this.

```
[Serializable]
public class ParserException : Exception
{
```

```
    // Constructor without arguments allows throwing your exception without
    // providing any information, including error message. Should be included
    // if your exception is meaningful without any additional details. Should
    // set message by calling base constructor (default message is not helpful).
    public ParserException()
        : base("Parser failure.")
    {}

    // Constructor with message argument allows overriding default error message.
    // Should be included if users can provide more helpful messages than
    // generic automatically generated messages.
    public ParserException(string message)
        : base(message)
    {}

    // Constructor for serialization support. If your exception contains custom
    // properties, read their values here.
    protected ParserException(SerializationInfo info, StreamingContext context)
        : base(info, context)
    {}
}
```

Using the ParserException

```
try
{
    Process.StartRun(fileName)
}
catch (ParserException ex)
{
    Console.WriteLine($"{ex.Message} in ${ex.FileName}:${ex.LineNumber}");
}
catch (PostProcessException x)
{
    ...
}
```

You may also use custom exceptions for catching and wrapping exceptions. This way many different errors can be converted into a single error type that is more useful to the application:

```
try
{
    int foo = int.Parse(token);
}
catch (FormatException ex)
{
    //Assuming you added this constructor
    throw new ParserException(
      $"Failed to read {token} as number.",
      FileName,
      LineNumber,
      ex);
}
```

When handling exceptions by raising your own custom exceptions, you should generally include a reference the original exception in the `InnerException` property, as shown above.

Security Concerns

If exposing the reason for the exception might compromise security by allowing users to see the inner workings of your application it can be a bad idea to wrap the inner exception. This might apply if you are creating a class library

that will be used by others.

Here is how you could raise a custom exception without wrapping the inner exception:

```
try
{
  // ...
}
catch (SomeStandardException ex)
{
  // ...
  throw new MyCustomException(someMessage);
}
```

Conclusion

When raising a custom exception (either with wrapping or with an unwrapped new exception), you should raise an exception that is meaningful to the caller. For instance, a user of a class library may not know much about how that library does its internal work. The exceptions that are thrown by the dependencies of the class library are not meaningful. Rather, the user wants an exception that is relevant to how the class library is using those dependencies in an erroneous way.

```
try
{
  // ...
}
catch (IOException ex)
{
  // ...
  throw new StorageServiceException(@"The Storage Service encountered a problem saving
your data. Please consult the inner exception for technical details.
If you are not able to resolve the problem, please call 555-555-1234 for technical
assistance.", ex);
}
```

Section 76.2: Finally block

```
try
{
    /* code that could throw an exception */
}
catch (Exception)
{
    /* handle the exception */
}
finally
{
    /* Code that will be executed, regardless if an exception was thrown / caught or not */
}
```

The `try / catch / finally` block can be very handy when reading from files.
For example:

```
FileStream f = null;

try
{
    f = File.OpenRead("file.txt");
    /* process the file here */
```

```
}
finally
{
    f?.Close(); // f may be null, so use the null conditional operator.
}
```

A try block must be followed by either a `catch` or a `finally` block. However, since there is no catch block, the execution will cause termination. Before termination, the statements inside the finally block will be executed.

In the file-reading we could have used a `using` block as `FileStream` (what `OpenRead` returns) implements `IDisposable`.

Even if there is a `return` statement in `try` block, the `finally` block will usually execute; there are a few cases where it will not:

- When a StackOverflow occurs.
- `Environment.FailFast`
- The application process is killed, usually by an external source.

Section 76.3: Best Practices

Cheatsheet

DO	DON'T
Control flow with control statements	Control flow with exceptions
Keep track of ignored (absorbed) exception by logging	Ignore exception
Repeat exception by using `throw`	Re-throw exception - `throw new ArgumentNullException()` or `throw ex`
Throw predefined system exceptions	Throw custom exceptions similar to predefined system exceptions
Throw custom/predefined exception if it is crucial to application logic	Throw custom/predefined exceptions to state a warning in flow
Catch exceptions that you want to handle	Catch every exception

DO NOT manage business logic with exceptions.

Flow control should NOT be done by exceptions. Use conditional statements instead. If a control can be done with `if-else` statement clearly, don't use exceptions because it reduces readability and performance.

Consider the following snippet by Mr. Bad Practices:

```
// This is a snippet example for DO NOT
object myObject;
void DoingSomethingWithMyObject()
{
    Console.WriteLine(myObject.ToString());
}
```

When execution reaches `Console.WriteLine(myObject.ToString());` application will throw an NullReferenceException. Mr. Bad Practices realized that `myObject` is null and edited his snippet to catch & handle `NullReferenceException`:

```
// This is a snippet example for DO NOT
object myObject;
void DoingSomethingWithMyObject()
{
```

```
    try
    {
        Console.WriteLine(myObject.ToString());
    }
    catch(NullReferenceException ex)
    {
        // Hmmm, if I create a new instance of object and assign it to myObject:
        myObject = new object();
        // Nice, now I can continue to work with myObject
        DoSomethingElseWithMyObject();
    }
}
```

Since previous snippet only covers logic of exception, what should I do if `myObject` is not null at this point? Where should I cover this part of logic? Right after `Console.WriteLine(myObject.ToString());`? How about after the `try...catch` block?

How about Mr. Best Practices? How would he handle this?

```
// This is a snippet example for DO
object myObject;
void DoingSomethingWithMyObject()
{
    if(myObject == null)
        myObject = new object();

    // When execution reaches this point, we are sure that myObject is not null
    DoSomethingElseWithMyObject();
}
```

Mr. Best Practices achieved same logic with fewer code and a clear & understandable logic.

DO NOT re-throw Exceptions

Re-throwing exceptions is expensive. It negatively impact performance. For code that routinely fails, you can use design patterns to minimize performance issues. This topic describes two design patterns that are useful when exceptions might significantly impact performance.

DO NOT absorb exceptions with no logging

```
try
{
    //Some code that might throw an exception
}
catch(Exception ex)
{
    //empty catch block, bad practice
}
```

Never swallow exceptions. Ignoring exceptions will save that moment but will create a chaos for maintainability later. When logging exceptions, you should always log the exception instance so that the complete stack trace is logged and not the exception message only.

```
try
{
    //Some code that might throw an exception
}
catch(NullException ex)
{
```

```
    LogManager.Log(ex.ToString());
}
```

Do not catch exceptions that you cannot handle

Many resources, such as this one, strongly urge you to consider why you are catching an exception in the place that you are catching it. You should only catch an exception if you can handle it at that location. If you can do something there to help mitigate the problem, such as trying an alternative algorithm, connecting to a backup database, trying another filename, waiting 30 seconds and trying again, or notifying an administrator, you can catch the error and do that. If there is nothing that you can plausibly and reasonably do, just "let it go" and let the exception be handled at a higher level. If the exception is sufficiently catastrophic and there is no reasonable option other than for the entire program to crash because of the severity of the problem, then let it crash.

```
try
{
    //Try to save the data to the main database.
}
catch(SqlException ex)
{
    //Try to save the data to the alternative database.
}
//If anything other than a SqlException is thrown, there is nothing we can do here. Let the exception
bubble up to a level where it can be handled.
```

Section 76.4: Exception Anti-patterns

Swallowing Exceptions

One should always re-throw exception in the following way:

```
try
{
    ...
}
catch (Exception ex)
{
    ...
    throw;
}
```

Re-throwing an exception like below will obfuscate the original exception and will lose the original stack trace. One should never do this! The stack trace prior to the catch and rethrow will be lost.

```
try
{
    ...
}
catch (Exception ex)
{
    ...
    throw ex;
}
```

Baseball Exception Handling

One should not use exceptions as a substitute for normal flow control constructs like if-then statements and while loops. This anti-pattern is sometimes called Baseball Exception Handling.

Here is an example of the anti-pattern:

```
try
{
    while (AccountManager.HasMoreAccounts())
    {
        account = AccountManager.GetNextAccount();
        if (account.Name == userName)
        {
            //We found it
            throw new AccountFoundException(account);
        }
    }
}
catch (AccountFoundException found)
{
    Console.Write("Here are your account details: " + found.Account.Details.ToString());
}
```

Here is a better way to do it:

```
Account found = null;
while (AccountManager.HasMoreAccounts() && (found==null))
{
    account = AccountManager.GetNextAccount();
    if (account.Name == userName)
    {
        //We found it
        found = account;
    }
}
Console.Write("Here are your account details: " + found.Details.ToString());
```

catch (Exception)

There are almost no (some say none!) reasons to catch the generic exception type in your code. You should catch only the exception types you expect to happen, because you hide bugs in your code otherwise.

```
try
{
     var f = File.Open(myfile);
     // do something
}
catch (Exception x)
{
     // Assume file not found
     Console.Write("Could not open file");
     // but maybe the error was a NullReferenceException because of a bug in the file handling code?
}
```

Better do:

```
try
{
     var f = File.Open(myfile);
     // do something which should normally not throw exceptions
}
catch (IOException)
{
     Console.Write("File not found");
```

```
}
// Unfortunatelly, this one does not derive from the above, so declare separatelly
catch (UnauthorizedAccessException)
{
     Console.Write("Insufficient rights");
}
```

If any other exception happens, we purposedly let the application crash, so it directly steps in the debugger and we can fix the problem. We mustn't ship a program where any other exceptions than these happen anyway, so it's not a problem to have a crash.

The following is a bad example, too, because it uses exceptions to work around a programming error. That's not what they're designed for.

```
public void DoSomething(String s)
{
     if (s == null)
         throw new ArgumentNullException(nameof(s));
     // Implementation goes here
}

try
{
     DoSomething(myString);
}
catch(ArgumentNullException x)
{
    // if this happens, we have a programming error and we should check
    // why myString was null in the first place.
}
```

Section 76.5: Basic Exception Handling

```
try
{
    /* code that could throw an exception */
}
catch (Exception ex)
{
    /* handle the exception */
}
```

Note that handling all exceptions with the same code is often not the best approach.
This is commonly used when any inner exception handling routines fail, as a last resort.

Section 76.6: Handling specific exception types

```
try
{
    /* code to open a file */
}
catch (System.IO.FileNotFoundException)
{
    /* code to handle the file being not found */
}
catch (System.IO.UnauthorizedAccessException)
{
    /* code to handle not being allowed access to the file */
```

```
}
catch (System.IO.IOException)
{
    /* code to handle IOException or it's descendant other than the previous two */
}
catch (System.Exception)
{
    /* code to handle other errors */
}
```

Be careful that exceptions are evaluated in order and inheritance is applied. So you need to start with the most specific ones and end with their ancestor. At any given point, only one catch block will get executed.

Section 76.7: Aggregate exceptions / multiple exceptions from one method

Who says you cannot throw multiple exceptions in one method. If you are not used to playing around with AggregateExceptions you may be tempted to create your own data-structure to represent many things going wrong. There are of course were another data-structure that is not an exception would be more ideal such as the results of a validation. Even if you do play with AggregateExceptions you may be on the receiving side and always handling them not realizing they can be of use to you.

It is quite plausible to have a method execute and even though it will be a failure as a whole you will want to highlight multiple things that went wrong in the exceptions that are thrown. As an example this behavior can be seen with how Parallel methods work were a task broken into multiple threads and any number of them could throw exceptions and this needs to be reported. Here is a silly example of how you could benefit from this:

```
    public void Run()
    {
        try
        {
            this.SillyMethod(1, 2);
        }
        catch (AggregateException ex)
        {
            Console.WriteLine(ex.Message);
            foreach (Exception innerException in ex.InnerExceptions)
            {
                Console.WriteLine(innerException.Message);
            }
        }
    }

    private void SillyMethod(int input1, int input2)
    {
        var exceptions = new List<Exception>();

        if (input1 == 1)
        {
            exceptions.Add(new ArgumentException("I do not like ones"));
        }
        if (input2 == 2)
        {
            exceptions.Add(new ArgumentException("I do not like twos"));
        }
        if (exceptions.Any())
        {
            throw new AggregateException("Funny stuff happended during execution", exceptions);
        }
```

```
    }
```

Section 76.8: Throwing an exception

Your code can, and often should, throw an exception when something unusual has happened.

```
public void WalkInto(Destination destination)
{
    if (destination.Name == "Mordor")
    {
        throw new InvalidOperationException("One does not simply walk into Mordor.");
    }
    // ... Implement your normal walking code here.
}
```

Section 76.9: Unhandled and Thread Exception

AppDomain.UnhandledException This event provides notification of uncaught exceptions.It allows the application to log information about the exception before the system default handler reports the exception to the user and terminates the application.If sufficient information about the state of the application is available, other actions may be undertaken — such as saving program data for later recovery.Caution is advised, because program data can become corrupted when exceptions are not handled.

```
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    private static void Main(string[] args)
    {
        AppDomain.CurrentDomain.UnhandledException += new
UnhandledExceptionEventHandler(UnhandledException);
    }
```

Application.ThreadException This event allows your Windows Forms application to handle otherwise unhandled exceptions that occur in Windows Forms threads. Attach your event handlers to the ThreadException event to deal with these exceptions, which will leave your application in an unknown state. Where possible, exceptions should be handled by a structured exception handling block.

```
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    private static void Main(string[] args)
    {
        AppDomain.CurrentDomain.UnhandledException += new
UnhandledExceptionEventHandler(UnhandledException);
        Application.ThreadException += new ThreadExceptionEventHandler(ThreadException);
    }
```

And finally exception handling

```
static void UnhandledException(object sender, UnhandledExceptionEventArgs e)
    {
        Exception ex = (Exception)e.ExceptionObject;
        // your code
    }
```

```
static void ThreadException(object sender, ThreadExceptionEventArgs e)
    {
        Exception ex = e.Exception;
        // your code
    }
```

Section 76.10: Implementing IErrorHandler for WCF Services

Implementing IErrorHandler for WCF services is a great way to centralize error handling and logging. The implementation shown here should catch any unhandled exception that is thrown as a result of a call to one of your WCF services. Also shown in this example is how to return a custom object, and how to return JSON rather than the default XML.

Implement IErrorHandler:

```
using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;
using System.Runtime.Serialization.Json;
using System.ServiceModel;
using System.ServiceModel.Web;

namespace BehaviorsAndInspectors
{
    public class ErrorHandler : IErrorHandler
    {

        public bool HandleError(Exception ex)
        {
            // Log exceptions here

            return true;

        } // end

        public void ProvideFault(Exception ex, MessageVersion version, ref Message fault)
        {
            // Get the outgoing response portion of the current context
            var response = WebOperationContext.Current.OutgoingResponse;

            // Set the default http status code
            response.StatusCode = HttpStatusCode.InternalServerError;

            // Add ContentType header that specifies we are using JSON
            response.ContentType = new MediaTypeHeaderValue("application/json").ToString();

            // Create the fault message that is returned (note the ref parameter) with
BaseDataResponseContract
            fault = Message.CreateMessage(
                version,
                string.Empty,
                new CustomReturnType { ErrorMessage = "An unhandled exception occurred!" },
                new DataContractJsonSerializer(typeof(BaseDataResponseContract), new List<Type> {
typeof(BaseDataResponseContract) }));

            if (ex.GetType() == typeof(VariousExceptionTypes))
            {
                 // You might want to catch different types of exceptions here and process them
differently
            }
```

```
            // Tell WCF to use JSON encoding rather than default XML
            var webBodyFormatMessageProperty = new
WebBodyFormatMessageProperty(WebContentFormat.Json);
            fault.Properties.Add(WebBodyFormatMessageProperty.Name, webBodyFormatMessageProperty);

        } // end

    } // end class

} // end namespace
```

In this example we attach the handler to the service behavior. You could also attach this to IEndpointBehavior, IContractBehavior, or IOperationBehavior in a similar way.

Attach to Service Behaviors:

```
using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Configuration;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;

namespace BehaviorsAndInspectors
{
    public class ErrorHandlerExtension : BehaviorExtensionElement, IServiceBehavior
    {
        public override Type BehaviorType
        {
            get { return GetType(); }
        }

        protected override object CreateBehavior()
        {
            return this;
        }

        private IErrorHandler GetInstance()
        {
            return new ErrorHandler();
        }

        void IServiceBehavior.AddBindingParameters(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints, BindingParameterCollection
bindingParameters) { } // end

        void IServiceBehavior.ApplyDispatchBehavior(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase)
        {
            var errorHandlerInstance = GetInstance();

            foreach (ChannelDispatcher dispatcher in serviceHostBase.ChannelDispatchers)
            {
                dispatcher.ErrorHandlers.Add(errorHandlerInstance);
            }
        }

        void IServiceBehavior.Validate(ServiceDescription serviceDescription, ServiceHostBase
serviceHostBase) { } // end
```

```
    } // end class

} // end namespace
```

Configs in Web.config:

```
...
<system.serviceModel>

    <services>
      <service name="WebServices.MyService">
        <endpoint binding="webHttpBinding" contract="WebServices.IMyService" />
      </service>
    </services>

    <extensions>
      <behaviorExtensions>
        <!-- This extension if for the WCF Error Handling-->
        <add name="ErrorHandlerBehavior"
type="WebServices.BehaviorsAndInspectors.ErrorHandlerExtensionBehavior, WebServices,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
      </behaviorExtensions>
    </extensions>

    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceMetadata httpGetEnabled="true"/>
          <serviceDebug includeExceptionDetailInFaults="true"/>
          <ErrorHandlerBehavior />
        </behavior>
      </serviceBehaviors>
    </behaviors>

    ....
</system.serviceModel>
...
```

Here are a few links that may be helpful on this topic:

https://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.ierrorhandler(v=vs.100).aspx

http://www.brainthud.com/cards/5218/25441/which-four-behavior-interfaces-exist-for-interacting-with-a-service-or-client-description-what-methods-do-they-implement-and

Other Examples:

IErrorHandler returning wrong message body when HTTP status code is 401 Unauthorized

IErrorHandler doesn't seem to be handling my errors in WCF .. any ideas?

How to make custom WCF error handler return JSON response with non-OK http code?

How do you set the Content-Type header for an HttpClient request?

Section 76.11: Using the exception object

You are allowed to create and throw exceptions in your own code. Instantiating an exception is done the same way that any other C# object.

```
Exception ex = new Exception();

// constructor with an overload that takes a message string
Exception ex = new Exception("Error message");
```

You can then use the `throw` keyword to raise the exception:

```
try
{
    throw new Exception("Error");
}
catch (Exception ex)
{
    Console.Write(ex.Message); // Logs 'Error' to the output window
}
```

Note: If you're throwing a new exception inside a catch block, ensure that the original exception is passed as "inner exception", e.g.

```
void DoSomething()
{
    int b=1; int c=5;
    try
    {
        var a = 1;
        b = a - 1;
        c = a / b;
        a = a / c;
    }
    catch (DivideByZeroException dEx) when (b==0)
    {
        // we're throwing the same kind of exception
        throw new DivideByZeroException("Cannot divide by b because it is zero", dEx);
    }
    catch (DivideByZeroException dEx) when (c==0)
    {
        // we're throwing the same kind of exception
        throw new DivideByZeroException("Cannot divide by c because it is zero", dEx);
    }
}

void Main()
{
    try
    {
        DoSomething();
    }
    catch (Exception ex)
    {
        // Logs full error information (incl. inner exception)
        Console.Write(ex.ToString());
    }
}
```

In this case it is assumed that the exception cannot be handled but some useful information is added to the message (and the original exception can still be accessed via `ex.InnerException` by an outer exception block).

It will show something like:

> System.DivideByZeroException: Cannot divide by b because it is zero ---> System.DivideByZeroException: Attempted to divide by zero.
> at UserQuery.g__DoSomething0_0() in C:[...]\LINQPadQuery.cs:line 36
> --- End of inner exception stack trace ---
> at UserQuery.g__DoSomething0_0() in C:[...]\LINQPadQuery.cs:line 42
> at UserQuery.Main() in C:[...]\LINQPadQuery.cs:line 55

If you're trying this example in LinqPad, you'll notice that the line numbers aren't very meaningful (they don't always help you). But passing a helpful error text as suggested above oftentimes significantly reduces the time to track down the location of the error, which is in this example clearly the line

> c = a / b;

in function `DoSomething()`.

[Try it in .NET Fiddle]

Section 76.12: Nesting of Exceptions & try catch blocks

One is able to nest one exception / `try` `catch` block inside the other.

This way one can manage small blocks of code which are capable of working without disrupting your whole mechanism.

```
try
{
//some code here
    try
    {
        //some thing which throws an exception. For Eg : divide by 0
    }
    catch (DivideByZeroException dzEx)
    {
        //handle here only this exception
        //throw from here will be passed on to the parent catch block
    }
    finally
    {
        //any thing to do after it is done.
    }
 //resume from here & proceed as normal;
}
catch(Exception e)
{
    //handle here
}
```

Note: Avoid Swallowing Exceptions when throwing to the parent catch block

Chapter 77: NullReferenceException

Section 77.1: NullReferenceException explained

A `NullReferenceException` is thrown when you try to access a non-static member (property, method, field or event) of a reference object but it is null.

```
Car myFirstCar = new Car();
Car mySecondCar = null;
Color myFirstColor = myFirstCar.Color; // No problem as myFirstCar exists / is not null
Color mySecondColor = mySecondCar.Color; // Throws a NullReferenceException
// as mySecondCar is null and yet we try to access its color.
```

To debug such an exception, it's quite easy: on the line where the exception is thrown, you just have to look before every '.' or '[', or on rare occasions '('.

```
myGarage.CarCollection[currentIndex.Value].Color = theCarInTheStreet.Color;
```

Where does my exception come from? Either:

- `myGarage` is `null`
- `myGarage.CarCollection` is `null`
- `currentIndex` is `null`
- `myGarage.CarCollection[currentIndex.Value]` is `null`
- `theCarInTheStreet` is `null`

In debug mode, you only have to put your mouse cursor on every of these elements and you will find your null reference. Then, what you have to do is understand why it doesn't have a value. The correction totally depends on the goal of your method.

Have you forgotten to instantiate/initialize it?

```
myGarage.CarCollection = new Car[10];
```

Are you supposed to do something different if the object is null?

```
if (myGarage == null)
{
    Console.WriteLine("Maybe you should buy a garage first!");
}
```

Or maybe someone gave you a null argument, and was not supposed to:

```
if (theCarInTheStreet == null)
{
    throw new ArgumentNullException("theCarInTheStreet");
}
```

In any case, remember that a method should never throw a NullReferenceException. If it does, that means you have forgotten to check something.

Chapter 78: Handling FormatException when converting string to other types

Section 78.1: Converting string to integer

There are various methods available for explicitly converting a `string` to an `integer`, such as:

1. `Convert.ToInt16();`
2. `Convert.ToInt32();`
3. `Convert.ToInt64();`
4. `int.Parse();`

But all these methods will throw a `FormatException`, if the input string contains non-numeric characters. For this, we need to write an additional exception handling(`try..catch`) to deal them in such cases.

Explanation with Examples:

So, let our input be:

```
string inputString = "10.2";
```

Example 1: `Convert.ToInt32()`

```
int convertedInt = Convert.ToInt32(inputString); // Failed to Convert
// Throws an Exception "Input string was not in a correct format."
```

Note: *Same goes for the other mentioned methods namely -* `Convert.ToInt16();` *and* `Convert.ToInt64();`

Example 2: `int.Parse()`

```
int convertedInt = int.Parse(inputString); // Same result "Input string was not in a correct format.
```

How do we circumvent this?

As told earlier, for handling the exceptions we usually need a `try..catch` as shown below:

```
try
{
    string inputString = "10.2";
    int convertedInt = int.Parse(inputString);
}
catch (Exception Ex)
{
    //Display some message, that the conversion has failed.
}
```

But, using the `try..catch` everywhere will not be a good practice, and there may be some scenarios where we wanted to give `0` if the input is wrong, *(If we follow the above method we need to assign* `0` *to* `convertedInt` *from the catch block).* To handle such scenarios we can make use of a special method called `.TryParse()`.

The `.TryParse()` method having an internal Exception handling, which will give you the output to the `out` parameter, and returns a Boolean value indicating the conversion status *(`true` if the conversion was successful; `false` if it failed).* Based on the return value we can determine the conversion status. Lets see one Example:

Usage 1: Store the return value in a Boolean variable

```
int convertedInt; // Be the required integer
bool isSuccessConversion = int.TryParse(inputString, out convertedInt);
```

We can check The variable `isSuccessConversion` after the Execution to check the conversion status. If it is false then the value of `convertedInt` will be `0`*(no need to check the return value if you want `0` for conversion failure).*

Usage 2: Check the return value with `if`

```
if (int.TryParse(inputString, out convertedInt))
{
    // convertedInt will have the converted value
    // Proceed with that
}
else
{
 // Display an error message
}
```

Usage 3: Without checking the return value you can use the following, if you don't care about the return value *(converted or not, `0` will be ok)*

```
int.TryParse(inputString, out convertedInt);
// use the value of convertedInt
// But it will be 0 if not converted
```

Chapter 79: Read & Understand Stacktraces

A stack trace is a great aid when debugging a program. You will get a stack trace when your program throws an Exception, and sometimes when the program terminates abnormally.

Section 79.1: Stack trace for a simple NullReferenceException in Windows Forms

Let's create a small piece of code that throws an exception:

```
private void button1_Click(object sender, EventArgs e)
{
    string msg = null;
    msg.ToCharArray();
}
```

If we execute this, we get the following Exception and stack trace:

```
System.NullReferenceException: "Object reference not set to an instance of an object."
   at WindowsFormsApplication1.Form1.button1_Click(Object sender, EventArgs e) in
F:\WindowsFormsApplication1\WindowsFormsApplication1\Form1.cs:line 29
   at System.Windows.Forms.Control.OnClick(EventArgs e)
   at System.Windows.Forms.Button.OnClick(EventArgs e)
   at System.Windows.Forms.Button.OnMouseUp(MouseEventArgs mevent)
```

The stack trace goes on like that, but this part will suffice for our purposes.

At the top of the stack trace we see the line:

> at WindowsFormsApplication1.Form1.button1_Click(Object sender, EventArgs e) in F:\WindowsFormsApplication1\WindowsFormsApplication1\Form1.cs:line 29

This is the most important part. It tells us the *exact* line where the Exception occurred: line 29 in Form1.cs .
So, this is where you begin your search.

The second line is

> at System.Windows.Forms.Control.OnClick(EventArgs e)

This is the method that called `button1_Click`. So now we know that `button1_Click`, where the error occurred, was called from `System.Windows.Forms.Control.OnClick`.

We can continue like this; the third line is

> at System.Windows.Forms.Button.OnClick(EventArgs e)

This is, in turn, the code that called `System.windows.Forms.Control.OnClick`.

The stack trace is the list of functions that was called until your code encountered the Exception. And by following

this, you can figure out which execution path your code followed until it ran into trouble!

Note that the stack trace includes calls from the .Net system; you don't normally need to follow all Microsofts `System.Windows.Forms` code to find out what went wrong, only the code that belongs to your own application.

So, why is this called a "stack trace"?
Because, every time a program calls a method, it keeps track of where it was. It has a data structure called the "stack", where it dumps its last location.
If it is done executing the method, it looks on the stack to see where it was before it called the method - and continues from there.

So the stack lets the computer know where it left off, before calling a new method.

But it also serves as a debugging help. Like a detective tracing the steps that a criminal took when committing their crime, a programmer can use the stack to trace the steps a program took before it crashed.

Chapter 80: Diagnostics

Section 80.1: Redirecting log output with TraceListeners

You can redirect the debug output to a text file by adding a TextWriterTraceListener to the Debug.Listeners collection.

```
public static void Main(string[] args)
{
    TextWriterTraceListener myWriter = new TextWriterTraceListener(@"debug.txt");
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");

    myWriter.Flush();
}
```

You can redirect the debug output to a console application's out stream using a ConsoleTraceListener.

```
public static void Main(string[] args)
{
    ConsoleTraceListener myWriter = new ConsoleTraceListener();
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");
}
```

Section 80.2: Debug.WriteLine

Writes to the trace listeners in the Listeners collection when the application is compiled in debug configuration.

```
public static void Main(string[] args)
{
    Debug.WriteLine("Hello");
}
```

In Visual Studio or Xamarin Studio this will appear in the Application Output window. This is due to the presence of the default trace listener in the TraceListenerCollection.

Chapter 81: Overflow

Section 81.1: Integer overflow

There is a maximum capacity an integer can store. And when you go over that limit, it will loop back to the negative side. For `int`, it is `2147483647`

```
int x = int.MaxValue;                //MaxValue is 2147483647
x = unchecked(x + 1);                //make operation explicitly unchecked so that the example also
works when the check for arithmetic overflow/underflow is enabled in the project settings
Console.WriteLine(x);                //Will print -2147483648
Console.WriteLine(int.MinValue);     //Same as Min value
```

For any integers out of this range use namespace System.Numerics which has datatype BigInteger. Check below link for more information https://msdn.microsoft.com/en-us/library/system.numerics.biginteger(v=vs.110).aspx

Section 81.2: Overflow during operation

Overflow also happens during the operation. In the following example, x is an `int`, 1 is an `int` by default. Therefore addition is an `int` addition. And the result will be an `int`. And it will overflow.

```
int x = int.MaxValue;                //MaxValue is 2147483647
long y = x + 1;                      //It will be overflown
Console.WriteLine(y);                //Will print -2147483648
Console.WriteLine(int.MinValue);     //Same as Min value
```

You can prevent that by using 1L. Now 1 will be a `long` and addition will be a `long` addition

```
int x = int.MaxValue;                //MaxValue is 2147483647
long y = x + 1L;                     //It will be OK
Console.WriteLine(y);                //Will print 2147483648
```

Section 81.3: Ordering matters

There is overflow in the following code

```
int x = int.MaxValue;
Console.WriteLine(x + x + 1L);  //prints -1
```

Whereas in the following code there is no overflow

```
int x = int.MaxValue;
Console.WriteLine(x + 1L + x);  //prints 4294967295
```

This is due to the left-to-right ordering of the operations. In the first code fragment `x + x` overflows and after that it becomes a `long`. On the other hand `x + 1L` becomes `long` and after that `x` is added to this value.

Chapter 82: Getting Started: Json with C#

The following topic will introduce a way to work with Json using C# language and concepts of Serialization and Deserialization.

Section 82.1: Simple Json Example

```
{
    "id": 89,
    "name": "Aldous Huxley",
    "type": "Author",
    "books":[{
                "name": "Brave New World",
                "date": 1932
             },
             {
               "name": "Eyeless in Gaza",
               "date": 1936
             },
             {
               "name": "The Genius and the Goddess",
               "date": 1955
             }]
}
```

If you are new into Json, here is an exemplified tutorial.

Section 82.2: First things First: Library to work with Json

To work with Json using C#, it is need to use Newtonsoft (.net library). This library provides methods that allows the programmer serialize and deserialize objects and more. There is a tutorial if you want to know details about its methods and usages.

If you use Visual Studio, go to *Tools/Nuget Package Manager/Manage Package to Solution/* and type "Newtonsoft" into the search bar and install the package. If you don't have NuGet, this detailed tutorial might help you.

Section 82.3: C# Implementation

Before reading some code, it is important to undersand the main concepts that will help to program applications using json.

> **Serialization**: Process of converting a object into a stream of bytes that can be sent through applications. The following code can be serialized and converted into the previous json.

> **Deserialization**: Process of converting a json/stream of bytes into an object. Its exactly the opposite process of serialization. The previous json can be deserialized into an C# object as demonstrated in examples below.

To work this out, it is important to turn the json structure into classes in order to use processes already described. If you use Visual Studio, you can turn a json into a class automatically just by selecting *"Edit/Paste Special/Paste JSON as Classes"* and pasting the json structure.

```
using Newtonsoft.Json;

  class Author
{
    [JsonProperty("id")] // Set the variable below to represent the json attribute
    public int id;       //"id"
    [JsonProperty("name")]
    public string name;
    [JsonProperty("type")]
    public string type;
    [JsonProperty("books")]
    public Book[] books;

    public Author(int id, string name, string type, Book[] books) {
        this.id = id;
        this.name = name;
        this.type= type;
        this.books = books;
    }
}

 class Book
{
   [JsonProperty("name")]
   public string name;
   [JsonProperty("date")]
   public DateTime date;
}
```

Section 82.4: Serialization

```
 static void Main(string[] args)
    {
        Book[] books = new Book[3];
        Author author = new Author(89,"Aldous Huxley","Author",books);
        string objectDeserialized = JsonConvert.SerializeObject(author);
        //Converting author into json
    }
```

The method ".SerializeObject" receives as parameter a *type object*, so you can put anything into it.

Section 82.5: Deserialization

You can receive a json from anywhere, a file or even a server so it is not included in the following code.

```
static void Main(string[] args)
{
    string jsonExample; // Has the previous json
    Author author = JsonConvert.DeserializeObject<Author>(jsonExample);
}
```

The method ".DeserializeObject" deserializes '*jsonExample*' into an "*Author*" object. This is why it is important to set the json variables in the classes definition, so the method access it in order to fill it.

Section 82.6: Serialization & De-Serialization Common Utilities function

This sample used to common function for all type object serialization and deserialization.

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Xml.Serialization;

namespace Framework
{
public static class IGUtilities
{
        public static string Serialization(this T obj)
        {
        string data = JsonConvert.SerializeObject(obj);
        return data;
        }

        public static T Deserialization(this string JsonData)
        {
        T copy = JsonConvert.DeserializeObject(JsonData);
        return copy;
        }

         public static T Clone(this T obj)
        {
            string data = JsonConvert.SerializeObject(obj);
            T copy = JsonConvert.DeserializeObject(data);
            return copy;
        }
}
}
```

Chapter 83: Using json.net

Using JSON.net JsonConverter class.

Section 83.1: Using JsonConverter on simple values

Example using JsonCoverter to deserialize the runtime property from the api response into a Timespan Object in the Movies model

JSON (http://www.omdbapi.com/?i=tt1663662)

```
{
    Title: "Pacific Rim",
    Year: "2013",
    Rated: "PG-13",
    Released: "12 Jul 2013",
    Runtime: "131 min",
    Genre: "Action, Adventure, Sci-Fi",
    Director: "Guillermo del Toro",
    Writer: "Travis Beacham (screenplay), Guillermo del Toro (screenplay), Travis Beacham (story)",
    Actors: "Charlie Hunnam, Diego Klattenhoff, Idris Elba, Rinko Kikuchi",
    Plot: "As a war between humankind and monstrous sea creatures wages on, a former pilot and a
trainee are paired up to drive a seemingly obsolete special weapon in a desperate effort to save
the world from the apocalypse.",
    Language: "English, Japanese, Cantonese, Mandarin",
    Country: "USA",
    Awards: "Nominated for 1 BAFTA Film Award. Another 6 wins & 46 nominations.",
    Poster:
"https://images-na.ssl-images-amazon.com/images/M/MV5BMTY3MTI5NjQ4Nl5BMl5BanBnXkFtZTcwOTU1OTU0OQ@@.
_V1_SX300.jpg",
    Ratings: [{
            Source: "Internet Movie Database",
            Value: "7.0/10"
        },
        {
            Source: "Rotten Tomatoes",
            Value: "71%"
        },
        {
            Source: "Metacritic",
            Value: "64/100"
        }
    ],
    Metascore: "64",
    imdbRating: "7.0",
    imdbVotes: "398,198",
    imdbID: "tt1663662",
    Type: "movie",
    DVD: "15 Oct 2013",
    BoxOffice: "$101,785,482.00",
    Production: "Warner Bros. Pictures",
    Website: "http://pacificrimmovie.com",
    Response: "True"
}
```

Movie Model

```
using Project.Serializers;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Runtime.Serialization;
using System.Threading.Tasks;

namespace Project.Models
{
    [DataContract]
    public class Movie
    {
        public Movie() { }

        [DataMember]
        public int Id { get; set; }

        [DataMember]
        public string ImdbId { get; set; }

        [DataMember]
        public string Title { get; set; }

        [DataMember]
        public DateTime Released { get; set; }

        [DataMember]
        [JsonConverter(typeof(RuntimeSerializer))]
        public TimeSpan Runtime { get; set; }

    }
}
```

RuntimeSerializer

```
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.RegularExpressions;
using System.Threading.Tasks;

namespace Project.Serializers
{
    public class RuntimeSerializer : JsonConverter
    {
        public override bool CanConvert(Type objectType)
        {
            return objectType == typeof(TimeSpan);
        }

        public override object ReadJson(JsonReader reader, Type objectType, object existingValue,
JsonSerializer serializer)
        {
            if (reader.TokenType == JsonToken.Null)
                return null;

            JToken jt = JToken.Load(reader);
            String value = jt.Value<String>();

            Regex rx = new Regex("(\\s*)min$");
            value = rx.Replace(value, (m) => "");

            int timespanMin;
            if(!Int32.TryParse(value, out timespanMin))
```

```
            {
                throw new NotSupportedException();
            }

            return new TimeSpan(0, timespanMin, 0);
        }

        public override void WriteJson(JsonWriter writer, object value, JsonSerializer serializer)
        {
            serializer.Serialize(writer, value);
        }
    }
}
```

Calling It

```
Movie m = JsonConvert.DeserializeObject<Movie>(apiResponse));
```

Section 83.2: Collect all fields of JSON object

```
using Newtonsoft.Json.Linq;
using System.Collections.Generic;

public class JsonFieldsCollector
{
    private readonly Dictionary<string, JValue> fields;

    public JsonFieldsCollector(JToken token)
    {
        fields = new Dictionary<string, JValue>();
        CollectFields(token);
    }

    private void CollectFields(JToken jToken)
    {
        switch (jToken.Type)
        {
            case JTokenType.Object:
                foreach (var child in jToken.Children<JProperty>())
                    CollectFields(child);
                break;
            case JTokenType.Array:
                foreach (var child in jToken.Children())
                    CollectFields(child);
                break;
            case JTokenType.Property:
                CollectFields(((JProperty) jToken).Value);
                break;
            default:
                fields.Add(jToken.Path, (JValue)jToken);
                break;
        }
    }

    public IEnumerable<KeyValuePair<string, JValue>> GetAllFields() => fields;
}
```

Usage:

```
var json = JToken.Parse(/* JSON string */);
var fieldsCollector = new JsonFieldsCollector(json);
```

```
var fields = fieldsCollector.GetAllFields();

foreach (var field in fields)
    Console.WriteLine($"{field.Key}: '{field.Value}'");
```

Demo

For this JSON object

```
{
  "User": "John",
  "Workdays": {
    "Monday": true,
    "Tuesday": true,
    "Friday": false
  },
  "Age": 42
}
```

expected output will be:

```
User: 'John'
Workdays.Monday: 'True'
Workdays.Tuesday: 'True'
Workdays.Friday: 'False'
Age: '42'
```

Chapter 84: Lambda expressions

Section 84.1: Lambda Expressions as Shorthand for Delegate Initialization

```
public delegate int ModifyInt(int input);
ModifyInt multiplyByTwo = x => x * 2;
```

The above Lambda expression syntax is equivalent to the following verbose code:

```
public delegate int ModifyInt(int input);

ModifyInt multiplyByTwo = delegate(int x){
    return x * 2;
};
```

Section 84.2: Lambda Expression as an Event Handler

Lambda expressions can be used to handle events, which is useful when:

- The handler is short.
- The handler never needs to be unsubscribed.

A good situation in which a lambda event handler might be used is given below:

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
```

If unsubscribing a registered event handler at some future point in the code is necessary, the event handler expression should be saved to a variable, and the registration/unregistration done through that variable:

```
EventHandler handler = (sender, args) => Console.WriteLine("Email sent");

smtpClient.SendCompleted += handler;
smtpClient.SendCompleted -= handler;
```

The reason that this is done rather than simply retyping the lambda expression verbatim to unsubscribe it (`-=`) is that the C# compiler won't necessarily consider the two expressions equal:

```
EventHandler handlerA = (sender, args) => Console.WriteLine("Email sent");
EventHandler handlerB = (sender, args) => Console.WriteLine("Email sent");
Console.WriteLine(handlerA.Equals(handlerB)); // May return "False"
```

Note that if additional statements are added to the lambda expression, then the required surrounding curly braces may be accidentally omitted, without causing compile-time error. For example:

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
emailSendButton.Enabled = true;
```

This will compile, but will result in adding the lambda expression `(sender, args) => Console.WriteLine("Email sent");` as an event handler, and executing the statement `emailSendButton.Enabled = true;` immediately. To fix this, the contents of the lambda must be surrounded in curly braces. This can be avoided by using curly braces from the start, being cautious when adding additional statements to a lambda-event-handler, or surrounding the lambda in round brackets from the start:

```
smtpClient.SendCompleted += ((sender, args) => Console.WriteLine("Email sent"));
//Adding an extra statement will result in a compile-time error
```

Section 84.3: Lambda Expressions with Multiple Parameters or No Parameters

Use parentheses around the expression to the left of the `=>` operator to indicate multiple parameters.

```
delegate int ModifyInt(int input1, int input2);
ModifyInt multiplyTwoInts = (x,y) => x * y;
```

Similarly, an empty set of parentheses indicates that the function does not accept parameters.

```
delegate string ReturnString();
ReturnString getGreeting = () => "Hello world.";
```

Section 84.4: Lambdas can be emitted both as `Func` and `Expression`

Assuming the following `Person` class:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

The following lambda:

```
p => p.Age > 18
```

Can be passed as an argument to both methods:

```
public void AsFunc(Func<Person, bool> func)
public void AsExpression(Expression<Func<Person, bool>> expr)
```

Because the compiler is capable of transforming lambdas both to delegates and `Expressions`.

Obviously, LINQ providers rely heavily on `Expressions` (exposed mainly through the `IQueryable<T>` interface) in order to be able to parse queries and translate them to store queries.

Section 84.5: Put Multiple Statements in a Statement Lambda

Unlike an expression lambda, a statement lambda can contain multiple statements separated by semicolons.

```
delegate void ModifyInt(int input);

ModifyInt addOneAndTellMe = x =>
{
    int result = x + 1;
    Console.WriteLine(result);
};
```

Note that the statements are enclosed in braces `{}`.

Remember that statement lambdas cannot be used to create expression trees.

Section 84.6: Lambdas for both `Func` and `Action`

Typically lambdas are used for defining simple *functions* (generally in the context of a linq expression):

```
var incremented = myEnumerable.Select(x => x + 1);
```

Here the `return` is implicit.

However, it is also possible to pass *actions* as lambdas:

```
myObservable.Do(x => Console.WriteLine(x));
```

Section 84.7: Using lambda syntax to create a closure

See remarks for discussion of closures. Suppose we have an interface:

```
public interface IMachine<TState, TInput>
{
    TState State { get; }
    public void Input(TInput input);
}
```

and then the following is executed:

```
IMachine<int, int> machine = ...;
Func<int, int> machineClosure = i => {
    machine.Input(i);
    return machine.State;
};
```

Now `machineClosure` refers to a function from `int` to `int`, which behind the scenes uses the `IMachine` instance which `machine` refers to in order to carry out the computation. Even if the reference `machine` goes out of scope, as long as the `machineClosure` object is maintained, the original `IMachine` instance will be retained as part of a 'closure', automatically defined by the compiler.

Warning: this can mean that the same function call returns different values at different times (e.g. In this example if the machine keeps a sum of its inputs). In lots of cases, this may be unexpected and is to be avoided for any code in a functional style - accidental and unexpected closures can be a source of bugs.

Section 84.8: Passing a Lambda Expression as a Parameter to a Method

```
List<int> l2 = l1.FindAll(x => x > 6);
```

Here `x => x > 6` is a lambda expression acting as a predicate that makes sure that only elements above 6 are returned.

Section 84.9: Basic lambda expressions

```
Func<int, int> add1 = i => i + 1;

Func<int, int, int> add = (i, j) => i + j;
```

```
// Behaviourally equivalent to:

int Add1(int i)
{
    return i + 1;
}

int Add(int i, int j)
{
    return i + j;
}

...

Console.WriteLine(add1(42)); //43
Console.WriteLine(Add1(42)); //43
Console.WriteLine(add(100, 250)); //350
Console.WriteLine(Add(100, 250)); //350
```

Section 84.10: Basic lambda expressions with LINQ

```
// assume source is {0, 1, 2, ..., 10}

var evens = source.Where(n => n%2 == 0);
// evens = {0, 2, 4, ... 10}

var strings = source.Select(n => n.ToString());
// strings = {"0", "1", ..., "10"}
```

Section 84.11: Lambda syntax with statement block body

```
Func<int, string> doubleThenAddElevenThenQuote = i => {
    var doubled = 2 * i;
    var addedEleven = 11 + doubled;
    return $"'{addedEleven}'";
};
```

Section 84.12: Lambda expressions with System.Linq.Expressions

```
Expression<Func<int, bool>> checkEvenExpression = i => i%2 == 0;
// lambda expression is automatically converted to an Expression<Func<int, bool>>
```

Chapter 85: Generic Lambda Query Builder

Section 85.1: QueryFilter class

This class holds predicate filters values.

```
public class QueryFilter
{
    public string PropertyName { get; set; }
    public string Value { get; set; }
    public Operator Operator { get; set; }

    // In the query {a => a.Name.Equals("Pedro")}
    // Property name to filter - propertyName = "Name"
    // Filter value - value = "Pedro"
    // Operation to perform - operation = enum Operator.Equals
    public QueryFilter(string propertyName, string value, Operator operatorValue)
    {
        PropertyName = propertyName;
        Value = value;
        Operator = operatorValue;
    }
}
```

Enum to hold the operations values:

```
 public enum Operator
{
    Contains,
    GreaterThan,
    GreaterThanOrEqual,
    LessThan,
    LessThanOrEqualTo,
    StartsWith,
    EndsWith,
    Equals,
    NotEqual
}
```

Section 85.2: GetExpression Method

```
public static Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)
{
    Expression exp = null;

    // Represents a named parameter expression. {parm => parm.Name.Equals()}, it is the param part
    // To create a ParameterExpression need the type of the entity that the query is against an a
name
    // The type is possible to find with the generic T and the name is fixed parm
    ParameterExpression param = Expression.Parameter(typeof(T), "parm");

    // It is good parctice never trust in the client, so it is wise to validate.
    if (filters.Count == 0)
        return null;

    // The expression creation differ if there is one, two or more filters.
    if (filters.Count != 1)
    {
```

```
        if (filters.Count == 2)
            // It is result from direct call.
            // For simplicity sake the private overloads will be explained in another example.
            exp = GetExpression<T>(param, filters[0], filters[1]);
        else
        {
            // As there is no method for more than two filters,
            // I iterate through all the filters and put I in the query two at a time
            while (filters.Count > 0)
            {
                // Retrieve the first two filters
                var f1 = filters[0];
                var f2 = filters[1];

                // To build a expression with a conditional AND operation that evaluates
                // the second operand only if the first operand evaluates to true.
                // It needed to use the BinaryExpression a Expression derived class
                // That has the AndAlso method that join two expression together
                exp = exp == null ? GetExpression<T>(param, filters[0], filters[1]) :
Expression.AndAlso(exp, GetExpression<T>(param, filters[0], filters[1]));

                // Remove the two just used filters, for the method in the next iteration finds the
next filters
                filters.Remove(f1);
                filters.Remove(f2);

                // If it is that last filter, add the last one and remove it
                if (filters.Count == 1)
                {
                    exp = Expression.AndAlso(exp, GetExpression<T>(param, filters[0]));
                    filters.RemoveAt(0);
                }
            }
        }
    }
    else
        // It is result from direct call.
        exp = GetExpression<T>(param, filters[0]);

           // converts the Expression into Lambda and retuns the query
    return Expression.Lambda<Func<T, bool>>(exp, param);
}
```

Section 85.3: GetExpression Private overload

For one filter:

Here is where the query is created, it receives a expression parameter and a filter.

```
private static Expression GetExpression<T>(ParameterExpression param, QueryFilter queryFilter)
{
    // Represents accessing a field or property, so here we are accessing for example:
    // the property "Name" of the entity
    MemberExpression member = Expression.Property(param, queryFilter.PropertyName);

    //Represents an expression that has a constant value, so here we are accessing for example:
    // the values of the Property "Name".
    // Also for clarity sake the GetConstant will be explained in another example.
    ConstantExpression constant = GetConstant(member.Type, queryFilter.Value);

    // With these two, now I can build the expression
```

```
    // every operator has it one way to call, so the switch will do.
    switch (queryFilter.Operator)
    {
        case Operator.Equals:
            return Expression.Equal(member, constant);

        case Operator.Contains:
            return Expression.Call(member, ContainsMethod, constant);

        case Operator.GreaterThan:
            return Expression.GreaterThan(member, constant);

        case Operator.GreaterThanOrEqual:
            return Expression.GreaterThanOrEqual(member, constant);

        case Operator.LessThan:
            return Expression.LessThan(member, constant);

        case Operator.LessThanOrEqualTo:
            return Expression.LessThanOrEqual(member, constant);

        case Operator.StartsWith:
            return Expression.Call(member, StartsWithMethod, constant);

        case Operator.EndsWith:
            return Expression.Call(member, EndsWithMethod, constant);
    }

    return null;
}
```

For two filters:

It returns the BinaryExpresion instance instead of the simple Expression.

```
private static BinaryExpression GetExpression<T>(ParameterExpression param, QueryFilter filter1,
QueryFilter filter2)
{
    // Built two separated expression and join them after.
    Expression result1 = GetExpression<T>(param, filter1);
    Expression result2 = GetExpression<T>(param, filter2);
    return Expression.AndAlso(result1, result2);
}
```

Section 85.4: ConstantExpression Method

`ConstantExpression` must be the same type of the `MemberExpression`. The value in this example is a string, which is converted before creating the `ConstantExpression` instance.

```
private static ConstantExpression GetConstant(Type type, string value)
{
    // Discover the type, convert it, and create ConstantExpression
    ConstantExpression constant = null;
    if (type == typeof(int))
    {
        int num;
        int.TryParse(value, out num);
        constant = Expression.Constant(num);
    }
    else if(type == typeof(string))
    {
```

```
        constant = Expression.Constant(value);
    }
    else if (type == typeof(DateTime))
    {
        DateTime date;
        DateTime.TryParse(value, out date);
        constant = Expression.Constant(date);
    }
    else if (type == typeof(bool))
    {
        bool flag;
        if (bool.TryParse(value, out flag))
        {
            flag = true;
        }
        constant = Expression.Constant(flag);
    }
    else if (type == typeof(decimal))
    {
        decimal number;
        decimal.TryParse(value, out number);
        constant = Expression.Constant(number);
    }
    return constant;
}
```

Section 85.5: Usage

Collection filters = new List(); QueryFilter filter = new QueryFilter("Name", "Burger", Operator.StartsWith); filters.Add(filter);

```
Expression<Func<Food, bool>> query = ExpressionBuilder.GetExpression<Food>(filters);
```

In this case, it is a query against the Food entity, that want to find all foods that start with "Burger" in the name.

Output:

```
query = {parm => a.parm.StartsWith("Burger")}


Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)
```

Chapter 86: Properties

Section 86.1: Auto-implemented properties

Auto-implemented properties were introduced in C# 3.
An auto-implemented property is declared with an empty getter and setter (accessors):

```
public bool IsValid { get; set; }
```

When an auto-implemented property is written in your code, the compiler creates a private anonymous field that can only be accessed through the property's accessors.

The above auto-implemented property statement is equivalent to writing this lengthy code:

```
private bool _isValid;
public bool IsValid
{
    get { return _isValid; }
    set { _isValid = value; }
}
```

Auto-implemented properties cannot have any logic in their accessors, for example:

```
public bool IsValid { get; set { PropertyChanged("IsValid"); } } // Invalid code
```

An auto-implemented property *can* however have different access modifiers for its accessors:

```
public bool IsValid { get; private set; }
```

C# 6 allows auto-implemented properties to have no setter at all (making it immutable, since its value can be set only inside the constructor or hard coded):

```
public bool IsValid { get; }
public bool IsValid { get; } = true;
```

For more information on initializing auto-implemented properties, read the Auto-property initializers documentation.

Section 86.2: Default Values for Properties

Setting a default value can be done by using Initializers (C#6)

```
public class Name
{
    public string First { get; set; } = "James";
    public string Last { get; set; } = "Smith";
}
```

If it is read only you can return values like this:

```
  public class Name
  {
      public string First => "James";
      public string Last => "Smith";
```

```
}
```

Section 86.3: Public Get

Getters are used to expose values from classes.

```
string name;
public string Name
{
    get { return this.name; }
}
```

Section 86.4: Public Set

Setters are used to assign values to properties.

```
string name;
public string Name
{
    set { this.name = value; }
}
```

Section 86.5: Accessing Properties

```
class Program
{
    public static void Main(string[] args)
    {
        Person aPerson = new Person("Ann Xena Sample", new DateTime(1984, 10, 22));
        //example of accessing properties (Id, Name & DOB)
        Console.WriteLine("Id is:  \t{0}\nName is:\t'{1}'.\nDOB is: \t{2:yyyy-MM-dd}.\nAge is:
\t{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());
        //example of setting properties

        aPerson.Name = "   Hans Trimmer  ";
        aPerson.DOB = new DateTime(1961, 11, 11);
        //aPerson.Id = 5; //this won't compile as Id's SET method is private; so only accessible
within the Person class.
        //aPerson.DOB = DateTime.UtcNow.AddYears(1); //this would throw a runtime error as there's
validation to ensure the DOB is in past.

        //see how our changes above take effect; note that the Name has been trimmed
        Console.WriteLine("Id is:  \t{0}\nName is:\t'{1}'.\nDOB is: \t{2:yyyy-MM-dd}.\nAge is:
\t{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());

        Console.WriteLine("Press any key to continue");
        Console.Read();
    }
}

public class Person
{
    private static int nextId = 0;
    private string name;
    private DateTime dob; //dates are held in UTC; i.e. we disregard timezones
    public Person(string name, DateTime dob)
    {
        this.Id = ++Person.nextId;
        this.Name = name;
```

```
        this.DOB = dob;
    }
    public int Id
    {
        get;
        private set;
    }
    public string Name
    {
        get { return this.name; }
        set
        {
            if (string.IsNullOrWhiteSpace(value)) throw new InvalidNameException(value);
            this.name = value.Trim();
        }
    }
    public DateTime DOB
    {
        get { return this.dob; }
        set
        {
            if (value < DateTime.UtcNow.AddYears(-200) || value > DateTime.UtcNow) throw new
InvalidDobException(value);
            this.dob = value;
        }
    }
    public int GetAgeInYears()
    {
        DateTime today = DateTime.UtcNow;
        int offset = HasHadBirthdayThisYear() ? 0 : -1;
        return today.Year - this.dob.Year + offset;
    }
    private bool HasHadBirthdayThisYear()
    {
        bool hasHadBirthdayThisYear = true;
        DateTime today = DateTime.UtcNow;
        if (today.Month > this.dob.Month)
        {
            hasHadBirthdayThisYear = true;
        }
        else
        {
            if (today.Month == this.dob.Month)
            {
                hasHadBirthdayThisYear = today.Day > this.dob.Day;
            }
            else
            {
                hasHadBirthdayThisYear = false;
            }
        }
        return hasHadBirthdayThisYear;
    }
}

public class InvalidNameException : ApplicationException
{
    const string InvalidNameExceptionMessage = "'{0}' is an invalid name.";
    public InvalidNameException(string value):
base(string.Format(InvalidNameExceptionMessage,value)){}
}
public class InvalidDobException : ApplicationException
```

```
{
    const string InvalidDobExceptionMessage = "'{0:yyyy-MM-dd}' is an invalid DOB.  The date must
not be in the future, or over 200 years in the past.";
    public InvalidDobException(DateTime value):
base(string.Format(InvalidDobExceptionMessage,value)){}
}
```

Section 86.6: Read-only properties

Declaration

A common misunderstanding, especially beginners, have is read-only property is the one marked with `readonly` keyword. That's not correct and in fact *following is a compile time error*:

```
public readonly string SomeProp { get; set; }
```

A property is read-only when it only has a getter.

```
public string SomeProp { get; }
```

Using read-only properties to create immutable classes

```
public Address
{
    public string ZipCode { get; }
    public string City { get; }
    public string StreetAddress { get; }

    public Address(
        string zipCode,
        string city,
        string streetAddress)
    {
        if (zipCode == null)
            throw new ArgumentNullException(nameof(zipCode));
        if (city == null)
            throw new ArgumentNullException(nameof(city));
        if (streetAddress == null)
            throw new ArgumentNullException(nameof(streetAddress));

        ZipCode = zipCode;
        City = city;
        StreetAddress = streetAddress;
    }
}
```

Section 86.7: Various Properties in Context

```
public class Person
{
    //Id property can be read by other classes, but only set by the Person class
    public int Id {get; private set;}
    //Name property can be retrieved or assigned
    public string Name {get; set;}

    private DateTime dob;
    //Date of Birth property is stored in a private variable, but retrieved or assigned through the
public property.
    public DateTime DOB
    {
```

```
        get { return this.dob; }
        set { this.dob = value; }
    }
    //Age property can only be retrieved; it's value is derived from the date of birth
    public int Age
    {
        get
        {
            int offset = HasHadBirthdayThisYear() ? 0 : -1;
            return DateTime.UtcNow.Year - this.dob.Year + offset;
        }
    }

    //this is not a property but a method; though it could be rewritten as a property if desired.
    private bool HasHadBirthdayThisYear()
    {
        bool hasHadBirthdayThisYear = true;
        DateTime today = DateTime.UtcNow;
        if (today.Month > this.dob.Month)
        {
            hasHadBirthdayThisYear = true;
        }
        else
        {
            if (today.Month == this.dob.Month)
            {
                hasHadBirthdayThisYear = today.Day > this.dob.Day;
            }
            else
            {
                hasHadBirthdayThisYear = false;
            }
        }
        return hasHadBirthdayThisYear;
    }
}
```

Chapter 87: Initializing Properties

Section 87.1: C# 6.0: Initialize an Auto-Implemented Property

Create a property with getter and/or setter and initialize all in one line:

```
public string Foobar { get; set; } = "xyz";
```

Section 87.2: Initializing Property with a Backing Field

```
public string Foobar {
    get { return _foobar; }
    set { _foobar = value; }
}
private string _foobar = "xyz";
```

Section 87.3: Property Initialization during object instantiation

Properties can be set when an object is instantiated.

```
var redCar = new Car
{
    Wheels = 2,
    Year = 2016,
    Color = Color.Red
};
```

Section 87.4: Initializing Property in Constructor

```
class Example
{
    public string Foobar { get; set; }
    public List<string> Names { get; set; }
    public Example()
    {
        Foobar = "xyz";
        Names = new List<string>(){"carrot","fox","ball"};
    }
}
```

Chapter 88: INotifyPropertyChanged interface

Section 88.1: Implementing INotifyPropertyChanged in C# 6

The implementation of `INotifyPropertyChange` can be error-prone, as the interface requires specifying property name as a string. In order to make the implementation more robust, an attribute `CallerMemberName` can be used.

```
class C : INotifyPropertyChanged
{
    // backing field
    int offset;
    // property
    public int Offset
    {
        get
        {
            return offset;
        }
        set
        {
            if (offset == value)
                return;
            offset = value;
            RaisePropertyChanged();
        }
    }

    // helper method for raising PropertyChanged event
    void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    // interface implemetation
    public event PropertyChangedEventHandler PropertyChanged;
}
```

If you have several classes implementing `INotifyPropertyChanged`, you may find it useful to refactor out the interface implementation and the helper method to the common base class:

```
class NotifyPropertyChangedImpl : INotifyPropertyChanged
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    // interface implemetation
    public event PropertyChangedEventHandler PropertyChanged;
}

class C : NotifyPropertyChangedImpl
{
    int offset;
    public int Offset
    {
        get { return offset; }
        set { if (offset != value) { offset = value; RaisePropertyChanged(); } }
    }
}
```

Section 88.2: INotifyPropertyChanged With Generic Set Method

The `NotifyPropertyChangedBase`class below defines a generic Set method that can be called from any derived type.

```
public class NotifyPropertyChangedBase : INotifyPropertyChanged
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    public event PropertyChangedEventHandler PropertyChanged;

    public virtual bool Set<T>(ref T field, T value, [CallerMemberName] string propertyName = null)
    {
        if (Equals(field, value))
            return false;
        storage = value;
        RaisePropertyChanged(propertyName);
        return true;
    }
}
```

To use this generic Set method, you simply need to create a class that derives from NotifyPropertyChangedBase.

```
public class SomeViewModel : NotifyPropertyChangedBase
{
    private string _foo;
    private int _bar;

    public string Foo
    {
        get { return _foo; }
        set { Set(ref _foo, value); }
    }

    public int Bar
    {
        get { return _bar; }
        set { Set(ref _bar, value); }
    }
}
```

As shown above, you can call `Set(ref _fieldName, value);` in a property's setter and it will automatically raise a PropertyChanged event if it is needed.

You can then register to the PropertyChanged event from another class that needs to handle property changes.

```
public class SomeListener
{
    public SomeListener()
    {
        _vm = new SomeViewModel();
        _vm.PropertyChanged += OnViewModelPropertyChanged;
    }

    private void OnViewModelPropertyChanged(object sender, PropertyChangedEventArgs e)
    {
        Console.WriteLine($"Property {e.PropertyName} was changed.");
```

```
    }

    private readonly SomeViewModel _vm;

}
```

Chapter 89: Events

Parameter	Details
EventArgsT	The type that derives from EventArgs and contains the event parameters.
EventName	The name of the event.
HandlerName	The name of the event handler.
SenderObject	The object that's invoking the event.
EventArguments	An instance of the EventArgsT type that contains the event parameters.

An event is a notification that something has occurred (such as a mouse click) or, in some cases, is about to occur (such as a price change).

Classes can define events and their instances (objects) may raise these events. For instance, a Button may contain a Click event that gets raised when a user has clicked it.

Event handlers are then methods that get called when their corresponding event is raised. A form may contain a Clicked event handler for every Button it contains, for instance.

Section 89.1: Declaring and Raising Events

Declaring an Event

You can declare an event on any `class` or `struct` using the following syntax:

```
public class MyClass
{
    // Declares the event for MyClass
    public event EventHandler MyEvent;

    // Raises the MyEvent event
    public void RaiseEvent()
    {
        OnMyEvent();
    }
}
```

There is an expanded syntax for declaring events, where you hold a private instance of the event, and define a public instance using `add` and `set` accessors. The syntax is very similar to C# properties. In all cases, the syntax demonstrated above should be preferred, because the compiler emits code to help ensure that multiple threads can safely add and remove event handlers to the event on your class.

Raising the Event

Version ≥ 6.0

```
private void OnMyEvent()
{
    EventName?.Invoke(this, EventArgs.Empty);
}
```

Version < 6.0

```
private void OnMyEvent()
{
    // Use a local for EventName, because another thread can modify the
    // public EventName between when we check it for null, and when we
    // raise the event.
    var eventName = EventName;
```

```
    // If eventName == null, then it means there are no event-subscribers,
    // and therefore, we cannot raise the event.
    if(eventName != null)
        eventName(this, EventArgs.Empty);

}
```

Note that events can only be raised by the declaring type. Clients can only subscribe/unsubscribe.

For C# versions before 6.0, where `EventName?.Invoke` is not supported, it is a good practice to assign the event to a temporary variable before invocation, as shown in the example, which ensures thread-safety in cases where multiple threads execute the same code. Failing to do so may cause a `NullReferenceException` to be thrown in certain cases where multiple threads are using the same object instance. In C# 6.0, the compiler emits code similar to that shown in the code example for C# 6.

Section 89.2: Event Properties

If a class raises a large the number of events, the storage cost of one field per delegate may not be acceptable. The .NET Framework provides event properties for these cases. This way you can use another data structure like `EventHandlerList` to store event delegates:

```
public class SampleClass
{
    // Define the delegate collection.
    protected EventHandlerList eventDelegates = new EventHandlerList();

    // Define a unique key for each event.
    static readonly object someEventKey = new object();

    // Define the SomeEvent event property.
    public event EventHandler SomeEvent
    {
        add
        {
            // Add the input delegate to the collection.
            eventDelegates.AddHandler(someEventKey, value);
        }
        remove
        {
            // Remove the input delegate from the collection.
            eventDelegates.RemoveHandler(someEventKey, value);
        }
    }

    // Raise the event with the delegate specified by someEventKey
    protected void OnSomeEvent(EventArgs e)
    {
        var handler = (EventHandler)eventDelegates[someEventKey];
        if (handler != null)
            handler(this, e);
    }
}
```

This approach is widely used in GUI frameworks like WinForms where controls can have dozens and even hundreds of events.

Note that `EventHandlerList` is not thread-safe, so if you expect your class to be used from multiple threads, you will need to add lock statements or other synchronization mechanism (or use a storage that provides thread

safety).

Section 89.3: Creating cancelable event

A cancelable event can be raised by a class when it is about to perform an action that can be canceled, such as the `FormClosing` event of a `Form`.

To create such event:

- Create a new event arg deriving from `CancelEventArgs` and add additional properties for event data.
- Create an event using `EventHandler<T>` and use the new cancel event arg class which you created.

Example

In the below example, we create a `PriceChangingEventArgs` event for `Price` property of a class. The event data class contains a `Value` which let the consumer know about the new . The event raises when you assign a new value to `Price` property and lets the consumer know the value is changing and let them to cancel the event. If the consumer cancels the event, the previous value for `Price` will be used:

PriceChangingEventArgs

```
public class PriceChangingEventArgs : CancelEventArgs
{
    int value;
    public int Value
    {
        get { return value; }
    }
    public PriceChangingEventArgs(int value)
    {
        this.value = value;
    }
}
```

Product

```
public class Product
{
    int price;
    public int Price
    {
        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(value);
            OnPriceChanging(e);
            if (!e.Cancel)
                price = value;
        }
    }

    public event EventHandler<PriceChangingEventArgs> PropertyChanging;
    protected void OnPriceChanging(PriceChangingEventArgs e)
    {
        var handler = PropertyChanging;
        if (handler != null)
            PropertyChanging(this, e);
    }
```

```
}
```

Section 89.4: Standard Event Declaration

Event declaration:

```
public event EventHandler<EventArgsT> EventName;
```

Event handler declaration:

```
public void HandlerName(object sender, EventArgsT args) { /* Handler logic */ }
```

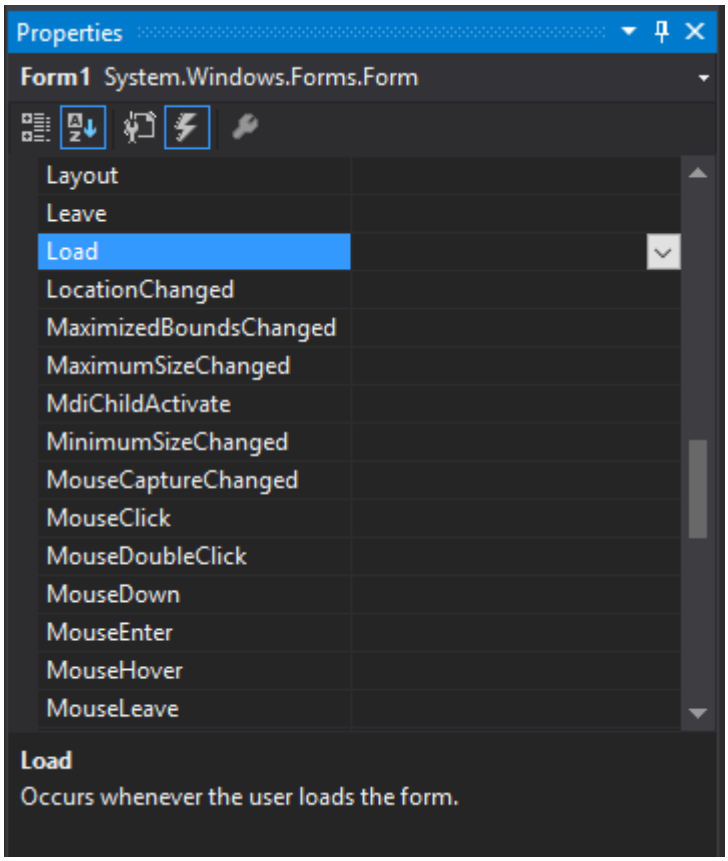
Subscribing to the event:

Dynamically:

```
EventName += HandlerName;
```

Through the Designer:

1. Click the Events button on the control's properties window (Lightening bolt)
2. Double-click the Event name:



3. Visual Studio will generate the event code:

```
private void Form1_Load(object sender, EventArgs e)
{

}
```

Invoking the method:

```
EventName(SenderObject, EventArguments);
```

Section 89.5: Anonymous Event Handler Declaration

Event declaration:

```
public event EventHandler<EventArgsType> EventName;
```

Event handler declaration using lambda operator => and subscribing to the event:

```
EventName += (obj, eventArgs) => { /* Handler logic */ };
```

Event handler declaration using delegate anonymous method syntax:

```
EventName += delegate(object obj, EventArgsType eventArgs) { /* Handler Logic */ };
```

Declaration & subscription of an event handler that does not use the event's parameter, and so can use the above syntax without needing to specify parameters:

```
EventName += delegate { /* Handler Logic */ }
```

Invoking the event:

```
EventName?.Invoke(SenderObject, EventArguments);
```

Section 89.6: Non-Standard Event Declaration

Events can be of any delegate type, not just `EventHandler` and `EventHandler<T>`. For example:

```
//Declaring an event
public event Action<Param1Type, Param2Type, ...> EventName;
```

This is used similarly to standard `EventHandler` events:

```
//Adding a named event handler
public void HandlerName(Param1Type parameter1, Param2Type parameter2, ...) {
    /* Handler logic */
}
EventName += HandlerName;

//Adding an anonymous event handler
EventName += (parameter1, parameter2, ...) => { /* Handler Logic */ };

//Invoking the event
EventName(parameter1, parameter2, ...);
```

It is possible to declare multiple events of the same type in a single statement, similar to with fields and local variables (though this may often be a bad idea):

```
public event EventHandler Event1, Event2, Event3;
```

This declares three separate events (`Event1`, `Event2`, and `Event3`) all of type `EventHandler`.
Note: Although some compilers may accept this syntax in interfaces as well as classes, the C# specification (v5.0 §13.2.3) provides grammar for interfaces that does not allow it, so using this in interfaces may be unreliable with different

compilers.

Section 89.7: Creating custom EventArgs containing additional data

Custom events usually need custom event arguments containing information about the event. For example `MouseEventArgs` which is used by mouse events like `MouseDown` or `MouseUp` events, contains information about `Location` or `Buttons` which used to generate the event.

When creating new events, to create a custom event arg:

- Create a class deriving from `EventArgs` and define properties for necessary data.
- As a convention, the name of the class should ends with `EventArgs`.

Example

In the below example, we create a `PriceChangingEventArgs` event for `Price` property of a class. The event data class contains a `CurrentPrice` and a `NewPrice`. The event raises when you assign a new value to `Price` property and lets the consumer know the value is changing and let them to know about current price and new price:

PriceChangingEventArgs

```
public class PriceChangingEventArgs : EventArgs
{
    public PriceChangingEventArgs(int currentPrice, int newPrice)
    {
        this.CurrentPrice = currentPrice;
        this.NewPrice = newPrice;
    }

    public int CurrentPrice { get; private set; }
    public int NewPrice { get; private set; }
}
```

Product

```
public class Product
{
    public event EventHandler<PriceChangingEventArgs> PriceChanging;

    int price;
    public int Price
    {
        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(price, value);
            OnPriceChanging(e);
            price = value;
        }
    }

    protected void OnPriceChanging(PriceChangingEventArgs e)
    {
        var handler = PriceChanging;
        if (handler != null)
            handler(this, e);
    }
```

```
}
```

You can enhance the example by allowing the consumer to change the new value and then the value will be used for property. To do so it's enough to apply these changes in classes.

Change the definition of `NewPrice` to be settable:

```
public int NewPrice { get; set; }
```

Change the definition of `Price` to use `e.NewPrice` as value of property, after calling `OnPriceChanging` :

```
int price;
public int Price
{
    get { return price; }
    set
    {
        var e = new PriceChangingEventArgs(price, value);
        OnPriceChanging(e);
        price = e.NewPrice;
    }
}
```

Chapter 90: Expression Trees

Parameter	Details
TDelegate	The delegate type to be used for the expression
lambdaExpression	The lambda expression (ex. `num => num < 5`)

Expression Trees are Expressions arranged in a treelike data structure. Each node in the tree is a representation of an expression, an expression being code. An In-Memory representation of a Lambda expression would be an Expression tree, which holds the actual elements (i.e. code) of the query, but not its result. Expression trees make the structure of a lambda expression transparent and explicit.

Section 90.1: Create Expression Trees with a lambda expression

Following is most basic expression tree that is created by lambda.

```
Expression<Func<int, bool>> lambda = num => num == 42;
```

To create expression trees 'by hand', one should use `Expression` class.

Expression above would be equivalent to:

```
ParameterExpression parameter = Expression.Parameter(typeof(int), "num"); // num argument
ConstantExpression constant = Expression.Constant(42, typeof(int)); // 42 constant
BinaryExpression equality = Expression.Equals(parameter, constant); // equality of two expressions
(num == 42)
Expression<Func<int, bool>> lambda = Expression.Lambda<Func<int, bool>>(equality, parameter);
```

Section 90.2: Creating Expression Trees by Using the API

```
using System.Linq.Expressions;

// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

Section 90.3: Compiling Expression Trees

```
// Define an expression tree, taking an integer, returning a bool.
Expression<Func<int, bool>> expr = num => num < 5;

// Call the Compile method on the expression tree to return a delegate that can be called.
Func<int, bool> result = expr.Compile();

// Invoke the delegate and write the result to the console.
Console.WriteLine(result(4)); // Prints true

// Prints True.

// You can also combine the compile step with the call/invoke step as below:
```

```
Console.WriteLine(expr.Compile()(4));
```

Section 90.4: Parsing Expression Trees

```
using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
                  param.Name, left.Name, operation.NodeType, right.Value);

// Decomposed expression: num => num LessThan 5
```

Section 90.5: Expression Tree Basic

Expression trees represent code in a tree-like data structure, where each node is an expression

Expression Trees enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. You can compile and run code represented by expression trees.

These are also used in the dynamic language run-time (DLR) to provide interoperability between dynamic languages and the .NET Framework and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL).

Expression Trees can be created Via

1. Anonymous lambda expression,
2. Manually by using the System.Linq.Expressions namespace.

Expression Trees from Lambda Expressions

When a lambda expression is assigned to Expression type variable , the compiler emits code to build an expression tree that represents the lambda expression.

The following code examples shows how to have the C# compiler create an expression tree that represents the lambda expression num => num < 5.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

Expression Trees by Using the API

Expression Trees also created using the **Expression** Class. This class contains static factory methods that create expression tree nodes of specific types.

Below are few type of Tree nodes.

1. ParameterExpression
2. MethodCallExpression

The following code example shows how to create an expression tree that represents the lambda expression num => num < 5 by using the API.

```
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 = Expression.Lambda<Func<int, bool>>(numLessThanFive,new
ParameterExpression[] { numParam });
```

Section 90.6: Examining the Structure of an Expression using Visitor

Define a new visitor class by overriding some of the methods of ExpressionVisitor:

```
class PrintingVisitor : ExpressionVisitor {
    protected override Expression VisitConstant(ConstantExpression node) {
        Console.WriteLine("Constant: {0}", node);
        return base.VisitConstant(node);
    }
    protected override Expression VisitParameter(ParameterExpression node) {
        Console.WriteLine("Parameter: {0}", node);
        return base.VisitParameter(node);
    }
    protected override Expression VisitBinary(BinaryExpression node) {
        Console.WriteLine("Binary with operator {0}", node.NodeType);
        return base.VisitBinary(node);
    }
}
```

Call `Visit` to use this visitor on an existing expression:

```
Expression<Func<int,bool>> isBig = a => a > 1000000;
var visitor = new PrintingVisitor();
visitor.Visit(isBig);
```

Section 90.7: Understanding the expressions API

We're going to use the expression tree API to create a `CalculateSalesTax` tree. In plain English, here's a summary of the steps it takes to create the tree.

1. Check if the product is taxable
2. If it is, multiply the line total by the applicable tax rate and return that amount
3. Otherwise return 0

```
//For reference, we're using the API to build this lambda expression
    orderLine => orderLine.IsTaxable ? orderLine.Total * orderLine.Order.TaxRate : 0;

//The orderLine parameter we pass in to the method.  We specify it's type (OrderLine) and the name of
the parameter.
    ParameterExpression orderLine = Expression.Parameter(typeof(OrderLine), "orderLine");

//Check if the parameter is taxable;  First we need to access the is taxable property, then check if
it's true
    PropertyInfo isTaxableAccessor = typeof(OrderLine).GetProperty("IsTaxable");
    MemberExpression getIsTaxable = Expression.MakeMemberAccess(orderLine, isTaxableAccessor);
    UnaryExpression isLineTaxable = Expression.IsTrue(getIsTaxable);
```

```
//Before creating the if, we need to create the braches
    //If the line is taxable, we'll return the total times the tax rate; get the total and tax rate,
then multiply
    //Get the total
    PropertyInfo totalAccessor = typeof(OrderLine).GetProperty("Total");
    MemberExpression getTotal = Expression.MakeMemberAccess(orderLine, totalAccessor);

    //Get the order
    PropertyInfo orderAccessor = typeof(OrderLine).GetProperty("Order");
    MemberExpression getOrder = Expression.MakeMemberAccess(orderLine, orderAccessor);

    //Get the tax rate - notice that we pass the getOrder expression directly to the member access
    PropertyInfo taxRateAccessor = typeof(Order).GetProperty("TaxRate");
    MemberExpression getTaxRate = Expression.MakeMemberAccess(getOrder, taxRateAccessor);

    //Multiply the two - notice we pass the two operand expressions directly to multiply
    BinaryExpression multiplyTotalByRate = Expression.Multiply(getTotal, getTaxRate);

//If the line is not taxable, we'll return a constant value - 0.0 (decimal)
    ConstantExpression zero = Expression.Constant(0M);

//Create the actual if check and branches
    ConditionalExpression ifTaxableTernary = Expression.Condition(isLineTaxable,
multiplyTotalByRate, zero);

//Wrap the whole thing up in a "method" - a LambdaExpression
    Expression<Func<OrderLine, decimal>> method = Expression.Lambda<Func<OrderLine,
decimal>>(ifTaxableTernary, orderLine);
```

Chapter 91: Overload Resolution

Section 91.1: Basic Overloading Example

This code contains an overloaded method named **Hello**:

```
class Example
{
    public static void Hello(int arg)
    {
        Console.WriteLine("int");
    }

    public static void Hello(double arg)
    {
        Console.WriteLine("double");
    }

    public static void Main(string[] args)
    {
        Hello(0);
        Hello(0.0);
    }
}
```

When the **Main** method is called, it will print

```
int
double
```

At compile-time, when the compiler finds the method call `Hello(0)`, it finds all methods with the name `Hello`. In this case, it finds two of them. It then tries to determine which of the methods is *better*. The algorithm for determining which method is better is complex, but it usually boils down to "make as few implicit conversions as possible".

Thus, in the case of `Hello(0)`, no conversion is needed for the method `Hello(int)` but an implicit numeric conversion is needed for the method `Hello(double)`. Thus, the first method is chosen by the compiler.

In the case of `Hello(0.0)`, there is no way to convert `0.0` to an `int` implicitly, so the method `Hello(int)` is not even considered for overload resolution. Only method remains and so it is chosen by the compiler.

Section 91.2: "params" is not expanded, unless necessary

The following program:

```
class Program
{
    static void Method(params Object[] objects)
    {
        System.Console.WriteLine(objects.Length);
    }
    static void Method(Object a, Object b)
    {
        System.Console.WriteLine("two");
    }
    static void Main(string[] args)
```

```
    {
        object[] objectArray = new object[5];

        Method(objectArray);
        Method(objectArray, objectArray);
        Method(objectArray, objectArray, objectArray);
    }
}
```

will print:

```
5
two
3
```

The call expression `Method(objectArray)` could be interpreted in two ways: a single `Object` argument that happens to be an array (so the program would output `1` because that would be the number of arguments, or as an array of arguments, given in the normal form, as though the method `Method` did not have the keyword `params`. In these situations, the normal, non-expanded form always takes precedence. So, the program outputs `5`.

In the second expression, `Method(objectArray, objectArray)`, both the expanded form of the first method and the traditional second method are applicable. In this case also, non-expanded forms take precedence, so the program prints `two`.

In the third expression, `Method(objectArray, objectArray, objectArray)`, the only option is to use the expanded form of the first method, and so the program prints `3`.

Section 91.3: Passing null as one of the arguments

If you have

```
void F1(MyType1 x) {
    // do something
}

void F1(MyType2 x) {
    // do something else
}
```

and for some reason you need to call the first overload of `F1` but with `x = null`, then doing simply

```
F1(null);
```

will not compile as the call is ambiguous. To counter this you can do

```
F1(null as MyType1);
```

Chapter 92: BindingList<T>

Section 92.1: Add item to list

```
BindingList<string> listOfUIItems = new BindingList<string>();
listOfUIItems.Add("Alice");
listOfUIItems.Add("Bob");
```

Section 92.2: Avoiding N*2 iteration

This is placed in a Windows Forms event handler

```
var nameList = new BindingList<string>();
ComboBox1.DataSource = nameList;
for(long i = 0; i < 10000; i++ ) {
    nameList.AddRange(new [] {"Alice", "Bob", "Carol" });
}
```

This takes a long time to execute, to fix, do the below:

```
var nameList = new BindingList<string>();
ComboBox1.DataSource = nameList;
nameList.RaiseListChangedEvents = false;
for(long i = 0; i < 10000; i++ ) {
    nameList.AddRange(new [] {"Alice", "Bob", "Carol" });
}
nameList.RaiseListChangedEvents = true;
nameList.ResetBindings();
```

Chapter 93: Preprocessor directives

Section 93.1: Conditional Expressions

When the following is compiled, it will return a different value depending on which directives are defined.

```
// Compile with /d:A or /d:B to see the difference
string SomeFunction()
{
#if A
    return "A";
#elif B
    return "B";
#else
    return "C";
#endif
}
```

Conditional expressions are typically used to log additional information for debug builds.

```
void SomeFunc()
{
    try
    {
        SomeRiskyMethod();
    }
    catch (ArgumentException ex)
    {
        #if DEBUG
        log.Error("SomeFunc", ex);
        #endif

        HandleException(ex);
    }
}
```

Section 93.2: Other Compiler Instructions

Line

`#line` controls the line number and filename reported by the compiler when outputting warnings and errors.

```
void Test()
{
    #line 42 "Answer"
    #line filename "SomeFile.cs"
    int life; // compiler warning CS0168 in "SomeFile.cs" at Line 42
    #line default
    // compiler warnings reset to default
}
```

Pragma Checksum

`#pragma checksum` allows the specification of a specific checksum for a generated program database (PDB) for debugging.

```
#pragma checksum "MyCode.cs" "{00000000-0000-0000-0000-000000000000}" "{0123456789A}"
```

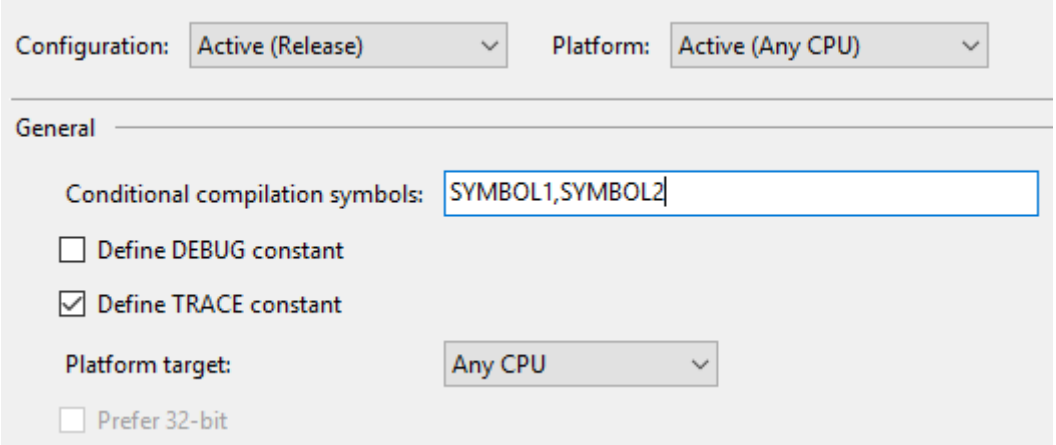
Section 93.3: Defining and Undefining Symbols

A compiler symbol is a keyword that is defined at compile-time that can be checked for to conditionally execute specific sections of code.

There are three ways to define a compiler symbol. They can be defined via code:

```
#define MYSYMBOL
```

They can be defined in Visual Studio, under Project Properties > Build > Conditional Compilation Symbols:



(Note that `DEBUG` and `TRACE` have their own checkboxes and do not need to be specified explicitly.)

Or they can be defined at compile-time using the `/define:[name]` switch on the C# compiler, `csc.exe`.

You can also undefined symbols using the `#undefine` directive.

The most prevalent example of this is the `DEBUG` symbol, which gets defined by Visual Studio when an application is compiled in Debug mode (versus Release mode).

```
public void DoBusinessLogic()
{
    try
    {
        AuthenticateUser();
        LoadAccount();
        ProcessAccount();
        FinalizeTransaction();
    }
    catch (Exception ex)
    {
#if DEBUG
        System.Diagnostics.Trace.WriteLine("Unhandled exception!");
        System.Diagnostics.Trace.WriteLine(ex);
        throw;
#else
        LoggingFramework.LogError(ex);
        DisplayFriendlyErrorMessage();
#endif
    }
}
```

In the example above, when an error occurs in the business logic of the application, if the application is compiled in Debug mode (and the `DEBUG` symbol is set), the error will be written to the trace log, and the exception will be re-

thrown for debugging. However, if the application is compiled in Release mode (and no `DEBUG` symbol is set), a logging framework is used to quietly log the error, and a friendly error message is displayed to the end user.

Section 93.4: Region Blocks

Use `#region` and `#endregion` to define a collapsible code region.

```
#region Event Handlers

public void Button_Click(object s, EventArgs e)
{
    // ...
}

public void DropDown_SelectedIndexChanged(object s, EventArgs e)
{
    // ...
}

#endregion
```

These directives are only beneficial when an IDE that supports collapsible regions (such as Visual Studio) is used to edit the code.

Section 93.5: Disabling and Restoring Compiler Warnings

You can disable compiler warnings using `#pragma warning disable` and restore them using `#pragma warning restore`:

```
#pragma warning disable CS0168

// Will not generate the "unused variable" compiler warning since it was disabled
var x = 5;

#pragma warning restore CS0168

// Will generate a compiler warning since the warning was just restored
var y = 8;
```

Comma-separated warning numbers are allowed:

```
#pragma warning disable CS0168, CS0219
```

The `CS` prefix is optional, and can even be intermixed (though this is not a best practice):

```
#pragma warning disable 0168, 0219, CS0414
```

Section 93.6: Generating Compiler Warnings and Errors

Compiler warnings can be generated using the `#warning` directive, and errors can likewise be generated using the `#error` directive.

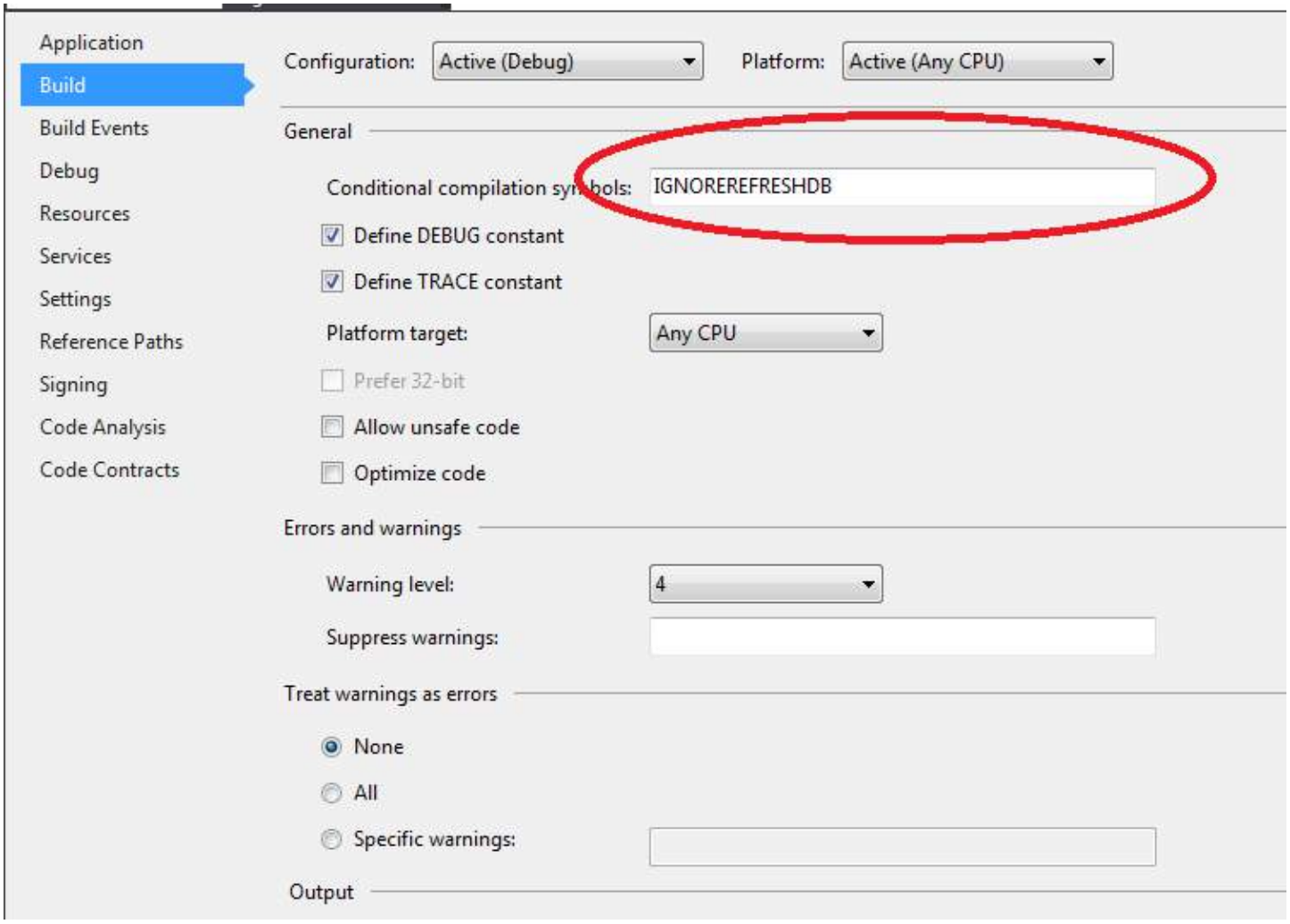
```
#if SOME_SYMBOL
#error This is a compiler Error.
#elif SOME_OTHER_SYMBOL
#warning This is a compiler Warning.
```

```
#endif
```

Section 93.7: Custom Preprocessors at project level

It is convenient to set custom conditional preprocessing at project level when some actions need to be skipped lets say for tests.

Go to `Solution Explorer` -> Click `Right Mouse` on project you want to set variable to -> `Properties` -> `Build` -> In General find field `Conditional compilation symbols` and enter your conditional variable here



Code example that will skip some code:

```
public void Init()
{
    #if !IGNOREREFRESHDB
    // will skip code here
     db.Initialize();
    #endif
}
```

Section 93.8: Using the Conditional attribute

Adding a `Conditional` attribute from `System.Diagnostics` namespace to a method is a clean way to control which methods are called in your builds and which are not.

```
#define EXAMPLE_A
```

```
using System.Diagnostics;
class Program
{
    static void Main()
    {
        ExampleA(); // This method will be called
        ExampleB(); // This method will not be called
    }

    [Conditional("EXAMPLE_A")]
    static void ExampleA() {...}

    [Conditional("EXAMPLE_B")]
    static void ExampleB() {...}
}
```

Chapter 94: Structs

Section 94.1: Declaring a struct

```
public struct Vector
{
    public int X;
    public int Y;
    public int Z;
}

public struct Point
{
    public decimal x, y;

    public Point(decimal pointX, decimal pointY)
    {
        x = pointX;
        y = pointY;
    }
}
```

- `struct` instance fields can be set via a parametrized constructor or individually after `struct` construction.
- Private members can only be initialized by the constructor.
- `struct` defines a sealed type that implicitly inherits from System.ValueType.
- Structs cannot inherit from any other type, but they can implement interfaces.
- Structs are copied on assignment, meaning all data is copied to the new instance and changes to one of them are not reflected by the other.
- A struct cannot be `null`, although it *can* used as a nullable type:

```
Vector v1 = null; //illegal
Vector? v2 = null; //OK
Nullable<Vector> v3 = null // OK
```

- Structs can be instantiated with or without using the `new` operator.

```
//Both of these are acceptable
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Vector v2;
v2.X = 1;
v2.Y = 2;
v2.Z = 3;
```

However, the `new` operator must be used in order to use an initializer:

```
Vector v1 = new MyStruct { X=1, Y=2, Z=3 }; // OK
Vector v2 { X=1, Y=2, Z=3 }; // illegal
```

A struct can declare everything a class can declare, with a few exceptions:

- A struct cannot declare a parameterless constructor. `struct` instance fields can be set via a parameterized constructor or individually after `struct` construction. Private members can only be initialized by the constructor.
- A struct cannot declare members as protected, since it is implicitly sealed.
- Struct fields can only be initialized if they are const or static.

Section 94.2: Struct usage

With constructor:

```
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
// Output X=1,Y=2,Z=3

Vector v1 = new Vector();
//v1.X is not assigned
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
// Output X=0,Y=2,Z=3

Point point1 = new Point();
point1.x = 0.5;
point1.y = 0.6;

Point point2 = new Point(0.5, 0.6);
```

Without constructor:

```
Vector v1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
//Output ERROR "Use of possibly unassigned field 'X'

Vector v1;
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}",v1.X,v1.Y,v1.Z);
// Output X=1,Y=2,Z=3

Point point3;
point3.x = 0.5;
point3.y = 0.6;
```

If we use a struct with its constructor, we aren't going to have problems with unassigned field (each unassigned field has null value).

Unlike classes, a struct doesn't have to be constructed, i.e. there is no need to use the new keyword, unless you need to call one of the constructors. A struct does not require the new keyword because is a value-type and thus cannot be null.

Section 94.3: Structs are copied on assignment

Sinse structs are value types all the data is *copied* on assignment, and any modification to the new copy does not change the data for the original copy. The code snippet below shows that `p1` is *copied* to `p2` and changes made on `p1` does not affect `p2` instance.

```
var p1 = new Point {
    x = 1,
    y = 2
};

Console.WriteLine($"{p1.x} {p1.y}"); // 1 2

var p2 = p1;
Console.WriteLine($"{p2.x} {p2.y}"); // Same output: 1 2

p1.x = 3;
Console.WriteLine($"{p1.x} {p1.y}"); // 3 2
Console.WriteLine($"{p2.x} {p2.y}"); // p2 remain the same: 1 2
```

Section 94.4: Struct implementing interface

```
public interface IShape
{
    decimal Area();
}

public struct Rectangle : IShape
{
    public decimal Length { get; set; }
    public decimal Width { get; set; }

    public decimal Area()
    {
        return Length * Width;
    }
}
```

Chapter 95: Attributes

Section 95.1: Creating a custom attribute

```
//1) All attributes should be inherited from System.Attribute
//2) You can customize your attribute usage (e.g. place restrictions) by using System.AttributeUsage
Attribute
//3) You can use this attribute only via reflection in the way it is supposed to be used
//4) MethodMetadataAttribute is just a name. You can use it without "Attribute" postfix - e.g.
[MethodMetadata("This text could be retrieved via reflection")].
//5) You can overload an attribute constructors
[System.AttributeUsage(System.AttributeTargets.Method | System.AttributeTargets.Class)]
public class MethodMetadataAttribute : System.Attribute
{
    //this is custom field given just for an example
    //you can create attribute without any fields
    //even an empty attribute can be used - as marker
    public string Text { get; set; }

    //this constructor could be used as [MethodMetadata]
    public MethodMetadataAttribute ()
    {
    }

    //This constructor could be used as [MethodMetadata("String")]
    public MethodMetadataAttribute (string text)
    {
        Text = text;
    }
}
```

Section 95.2: Reading an attribute

Method `GetCustomAttributes` returns an array of custom attributes applied to the member. After retrieving this array you can search for one or more specific attributes.

```
var attribute = typeof(MyClass).GetCustomAttributes().OfType<MyCustomAttribute>().Single();
```

Or iterate through them

```
foreach(var attribute in typeof(MyClass).GetCustomAttributes()) {
    Console.WriteLine(attribute.GetType());
}
```

`GetCustomAttribute` extension method from `System.Reflection.CustomAttributeExtensions` retrieves a custom attribute of a specified type, it can be applied to any `MemberInfo`.

```
var attribute = (MyCustomAttribute) typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute));
```

`GetCustomAttribute` also has generic signature to specify type of attribute to search for.

```
var attribute = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>();
```

Boolean argument `inherit` can be passed to both of those methods. If this value set to `true` the ancestors of element would be also to inspected.

Section 95.3: Using an attribute

```
[StackDemo(Text = "Hello, World!")]
public class MyClass
{
    [StackDemo("Hello, World!")]
    static void MyMethod()
    {
    }
}
```

Section 95.4: DebuggerDisplay Attribute

Adding the `DebuggerDisplay` Attribute will change the way the debugger displays the class when it is hovered over.

Expressions that are wrapped in `{}` will be evaluated by the debugger. This can be a simple property like in the following sample or more complex logic.

```
[DebuggerDisplay("{StringProperty} - {IntProperty}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }
}
```

Adding `,nq` before the closing bracket removes the quotes when outputting a string.

```
[DebuggerDisplay("{StringProperty,nq} - {IntProperty}")]
```

Even though general expressions are allowed in the `{}` they are not recommended. The `DebuggerDisplay` attribute will be written into the assembly metadata as a string. Expressions in `{}` are not checked for validity. So a `DebuggerDisplay` attribute containing more complex logic than i.e. some simple arithmetic might work fine in C#, but the same expression evaluated in VB.NET will probably not be syntactically valid and produce an error while debugging.

A way to make `DebuggerDisplay` more language agnostic is to write the expression in a method or property and call it instead.

```
[DebuggerDisplay("{DebuggerDisplay(),nq}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }

   private string DebuggerDisplay()
    {
```

```
        return $"{StringProperty} - {IntProperty}"";
    }
}
```

One might want `DebuggerDisplay`to output all or just some of the properties and when debugging and inspecting also the type of the object.
The example below also surrounds the helper method with `#if DEBUG` as `DebuggerDisplay` is used in debugging environments.

```
[DebuggerDisplay("{DebuggerDisplay(),nq}")]
public class AnObject
{
   public int ObjectId { get; set; }
   public string StringProperty { get; set; }
   public int IntProperty { get; set; }

#if DEBUG
   private string DebuggerDisplay()
    {
        return
            $"ObjectId:{this.ObjectId}, StringProperty:{this.StringProperty},
Type:{this.GetType()}";
    }
    #endif
}
```

Section 95.5: Caller info attributes

Caller info attributes can be used to pass down information about the invoker to the invoked method. The declaration looks like this:

```
using System.Runtime.CompilerServices;

public void LogException(Exception ex,
                         [CallerMemberName]string callerMemberName = "",
                         [CallerLineNumber]int callerLineNumber = 0,
                         [CallerFilePath]string callerFilePath = "")
{
    //perform logging
}
```

And the invocation looks like this:

```
public void Save(DBContext context)
{
    try
    {
        context.SaveChanges();
    }
    catch (Exception ex)
    {
        LogException(ex);
    }
}
```

Notice that only the first parameter is passed explicitly to the `LogException` method whereas the rest of them will be provided at compile time with the relevant values.

The `callerMemberName` parameter will receive the value `"Save"` - the name of the calling method.

The `callerLineNumber` parameter will receive the number of whichever line the `LogException` method call is written on.

And the 'callerFilePath' parameter will receive the full path of the file `Save` method is declared in.

Section 95.6: Obsolete Attribute

System.Obsolete is an attribute that is used to mark a type or a member that has a better version, and thus should not be used.

```
[Obsolete("This class is obsolete. Use SomeOtherClass instead.")]
class SomeClass
{
    //
}
```

In case the class above is used, the compiler will give the warning "This class is obsolete. Use SomeOtherClass instead."

Section 95.7: Reading an attribute from interface

There is no simple way to obtain attributes from an interface, since classes does not inherit attributes from an interface. Whenever implementing an interface or overriding members in a derived class, you need to re-declare the attributes. So in the example below output would be `True` in all three cases.

```
using System;
using System.Linq;
using System.Reflection;

namespace InterfaceAttributesDemo {

    [AttributeUsage(AttributeTargets.Interface, Inherited = true)]
    class MyCustomAttribute : Attribute {
        public string Text { get; set; }
    }

    [MyCustomAttribute(Text = "Hello from interface attribute")]
    interface IMyClass {
        void MyMethod();
    }

    class MyClass : IMyClass {
        public void MyMethod() { }
    }

    public class Program {
        public static void Main(string[] args) {
            GetInterfaceAttributeDemo();
        }

        private static void GetInterfaceAttributeDemo() {
            var attribute1 = (MyCustomAttribute)
typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute), true);
            Console.WriteLine(attribute1 == null); // True

            var attribute2 =
```

```
typeof(MyClass).GetCustomAttributes(true).OfType<MyCustomAttribute>().SingleOrDefault();
            Console.WriteLine(attribute2 == null); // True

            var attribute3 = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>(true);
            Console.WriteLine(attribute3 == null); // True
        }
    }
}
```

One way to retrieve interface attributes is to search for them through all the interfaces implemented by a class.

```
var attribute = typeof(MyClass).GetInterfaces().SelectMany(x =>
x.GetCustomAttributes().OfType<MyCustomAttribute>()).SingleOrDefault();
Console.WriteLine(attribute == null); // False
Console.WriteLine(attribute.Text); // Hello from interface attribute
```

Chapter 96: Delegates

Section 96.1: Declaring a delegate type

The following syntax creates a `delegate` type with name `NumberInOutDelegate`, representing a method which takes an `int` and returns an `int`.

```
public delegate int NumberInOutDelegate(int input);
```

This can be used as follows:

```
public static class Program
{
    static void Main()
    {
        NumberInOutDelegate square = MathDelegates.Square;
        int answer1 = square(4);
        Console.WriteLine(answer1); // Will output 16

        NumberInOutDelegate cube = MathDelegates.Cube;
        int answer2 = cube(4);
        Console.WriteLine(answer2); // Will output 64
    }
}

public static class MathDelegates
{
    static int Square (int x)
    {
        return x*x;
    }

    static int Cube (int x)
    {
        return x*x*x;
    }
}
```

The `example` delegate instance is executed in the same way as the `Square` method. A delegate instance literally acts as a delegate for the caller: the caller invokes the delegate, and then the delegate calls the target method. This indirection decouples the caller from the target method.

You can declare a **generic** delegate type, and in that case you may specify that the type is covariant (`out`) or contravariant (`in`) in some of the type arguments. For example:

```
public delegate TTo Converter<in TFrom, out TTo>(TFrom input);
```

Like other generic types, generic delegate types can have constraints, such as `where TFrom : struct`, `IConvertible where TTo : new()`.

Avoid co- and contravariance for delegate types that are meant to be used for multicast delegates, such as event handler types. This is because concatenation (`+`) can fail if the run-time type is different from the compile-time type because of the variance. For example, avoid:

```
public delegate void EventHandler<in TEventArgs>(object sender, TEventArgs e);
```

Instead, use an invariant generic type:

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

Also supported are delegates where some parameters are modified by `ref` or `out`, as in:

```
public delegate bool TryParser<T>(string input, out T result);
```

(sample use `TryParser<decimal> example = decimal.TryParse;`), or delegates where the last parameter has the `params` modifier. Delegate types can have optional parameters (supply default values). Delegate types can use pointer types like `int*` or `char*` in their signatures or return types (use `unsafe` keyword). A delegate type and its parameters can carry custom attributes.

Section 96.2: The Func<T, TResult>, Action<T> and Predicate<T> delegate types

The System namespace contains `Func<..., TResult>` delegate types with between 0 and 15 generic parameters, returning type `TResult`.

```
private void UseFunc(Func<string> func)
{
    string output = func(); // Func with a single generic type parameter returns that type
    Console.WriteLine(output);
}

private void UseFunc(Func<int, int, string> func)
{
    string output = func(4, 2); // Func with multiple generic type parameters takes all but the
first as parameters of that type
    Console.WriteLine(output);
}
```

The System namespace also contains `Action<...>` delegate types with different number of generic parameters (from 0 to 16). It is similar to `Func<T1, .., Tn>`, but it always returns `void`.

```
private void UseAction(Action action)
{
    action(); // The non-generic Action has no parameters
}

private void UseAction(Action<int, string> action)
{
    action(4, "two"); // The generic action is invoked with parameters matching its type arguments
}
```

`Predicate<T>` is also a form of `Func` but it will always return `bool`. A predicate is a way of specifying a custom criteria. Depending on the value of the input and the logic defined within the predicate, it will return either `true` or `false`. `Predicate<T>` therefore behaves in the same way as `Func<T, bool>` and both can be initialized and used in the same way.

```
Predicate<string> predicate = s => s.StartsWith("a");
Func<string, bool> func = s => s.StartsWith("a");

// Both of these return true
var predicateReturnsTrue = predicate("abc");
var funcReturnsTrue = func("abc");
```

```
// Both of these return false
var predicateReturnsFalse = predicate("xyz");
var funcReturnsFalse = func("xyz");
```

The choice of whether to use `Predicate<T>` or `Func<T, bool>` is really a matter of opinion. `Predicate<T>` is arguably more expressive of the author's intent, while `Func<T, bool>` is likely to be familiar to a greater proportion of C# developers.

In addition to that, there are some cases where only one of the options is available, especially when interacting with another API. For example `List<T>` and `Array<T>` generally take `Predicate<T>` for their methods, while most LINQ extensions only accept `Func<T, bool>`.

Section 96.3: Combine Delegates (Multicast Delegates)

Addition `+` and subtraction `-` operations can be used to combine delegate instances. The delegate contains a list of the assigned delegates.

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace DelegatesExample {
    class MainClass {
        private delegate void MyDelegate(int a);

        private static void PrintInt(int a) {
            Console.WriteLine(a);
        }

        private static void PrintType<T>(T a) {
            Console.WriteLine(a.GetType());
        }

        public static void Main (string[] args)
        {
            MyDelegate d1 = PrintInt;
            MyDelegate d2 = PrintType;

            // Output:
            // 1
            d1(1);

            // Output:
            // System.Int32
            d2(1);

            MyDelegate d3 = d1 + d2;
            // Output:
            // 1
            // System.Int32
            d3(1);

            MyDelegate d4 = d3 - d2;
            // Output:
            // 1
            d4(1);

            // Output:
            // True
            Console.WriteLine(d1 == d4);
```

```
        }
    }
}
```

In this example `d3` is a combination of `d1` and `d2` delegates, so when called the program outputs both `1` and `System.Int32` strings.

Combining delegates with **non void** return types:

If a multicast delegate has a `nonvoid` return type, the caller receives the return value from the last method to be invoked. The preceding methods are still called, but their return values are discarded.

```
    class Program
    {
        public delegate int Transformer(int x);

        static void Main(string[] args)
        {
            Transformer t = Square;
            t += Cube;
            Console.WriteLine(t(2));  // O/P 8
        }

        static int Square(int x) { return x * x; }

        static int Cube(int x) { return x*x*x; }
    }
```

`t(2)` will call first `Square` and then `Cube`. The return value of Square is discarded and return value of the last method i.e. `Cube` is retained.

Section 96.4: Safe invoke multicast delegate

Ever wanted to call a multicast delegate but you want the entire invokation list to be called even if an exception occurs in any in the chain. Then you are in luck, I have created an extension method that does just that, throwing an `AggregateException` only after execution of the entire list completes:

```
public static class DelegateExtensions
{
    public static void SafeInvoke(this Delegate del,params object[] args)
    {
        var exceptions = new List<Exception>();

        foreach (var handler in del.GetInvocationList())
        {
            try
            {
                handler.Method.Invoke(handler.Target, args);
            }
            catch (Exception ex)
            {
                exceptions.Add(ex);
            }
        }

        if(exceptions.Any())
        {
            throw new AggregateException(exceptions);
        }
```

```
    }
}

public class Test
{
    public delegate void SampleDelegate();

    public void Run()
    {
        SampleDelegate delegateInstance = this.Target2;
        delegateInstance += this.Target1;

        try
        {
            delegateInstance.SafeInvoke();
        }
        catch(AggregateException ex)
        {
            // Do any exception handling here
        }
    }

    private void Target1()
    {
        Console.WriteLine("Target 1 executed");
    }

    private void Target2()
    {
        Console.WriteLine("Target 2 executed");
        throw new Exception();
    }
}
```

This outputs:

```
Target 2 executed
Target 1 executed
```

Invoking directly, without `SaveInvoke`, would only execute Target 2.

Section 96.5: Delegate Equality

Calling `.Equals()` on a delegate compares by reference equality:

```
Action action1 = () => Console.WriteLine("Hello delegates");
Action action2 = () => Console.WriteLine("Hello delegates");
Action action1Again = action1;

Console.WriteLine(action1.Equals(action1)) // True
Console.WriteLine(action1.Equals(action2)) // False
Console.WriteLine(action1Again.Equals(action1)) // True
```

These rules also apply when doing `+=` or `-=` on a multicast delegate, for example when subscribing and unsubscribing from events.

Section 96.6: Underlying references of named method

delegates

When assigning named methods to delegates, they will refer to the same underlying object if:

- They are the same instance method, on the same instance of a class
- They are the same static method on a class

```
public class Greeter
{
    public void WriteInstance()
    {
        Console.WriteLine("Instance");
    }

    public static void WriteStatic()
    {
        Console.WriteLine("Static");
    }
}

// ...

Greeter greeter1 = new Greeter();
Greeter greeter2 = new Greeter();

Action instance1 = greeter1.WriteInstance;
Action instance2 = greeter2.WriteInstance;
Action instance1Again = greeter1.WriteInstance;

Console.WriteLine(instance1.Equals(instance2)); // False
Console.WriteLine(instance1.Equals(instance1Again)); // True

Action @static = Greeter.WriteStatic;
Action staticAgain = Greeter.WriteStatic;

Console.WriteLine(@static.Equals(staticAgain)); // True
```

Section 96.7: Assigning a named method to a delegate

Named methods can be assigned to delegates with matching signatures:

```
public static class Example
{
    public static int AddOne(int input)
    {
        return input + 1;
    }
}


Func<int,int> addOne = Example.AddOne
```

`Example.AddOne` takes an `int` and returns an `int`, its signature matches the delegate `Func<int,int>`. `Example.AddOne` can be directly assigned to `addOne` because they have matching signatures.

Section 96.8: Assigning to a delegate by lambda

Lambdas can be used to create anonymous methods to assign to a delegate:

```
Func<int,int> addOne = x => x+1;
```

Note that the explicit declaration of type is required when creating a variable this way:

```
var addOne = x => x+1; // Does not work
```

Section 96.9: Encapsulating transformations in funcs

```
public class MyObject{
    public DateTime? TestDate { get; set; }

    public Func<MyObject, bool> DateIsValid = myObject => myObject.TestDate.HasValue &&
myObject.TestDate > DateTime.Now;

    public void DoSomething(){
        //We can do this:
        if(this.TestDate.HasValue && this.TestDate > DateTime.Now){
            CallAnotherMethod();
        }

        //or this:
        if(DateIsValid(this)){
            CallAnotherMethod();
        }
    }
}
```

In the spirit of clean coding, encapsulating checks and transformations like the one above as a Func can make your code easier to read and understand. While the above example is very simple, what if there were multiple DateTime properties each with their own differing validation rules and we wanted to check different combinations? Simple, one-line Funcs that each have established return logic can be both readable and reduce the apparent complexity of your code. Consider the below Func calls and imagine how much more code would be cluttering up the method:

```
public void CheckForIntegrity(){
    if(ShipDateIsValid(this) && TestResultsHaveBeenIssued(this) && !TestResultsFail(this)){
        SendPassingTestNotification();
    }
}
```

Section 96.10: Passing delegates as parameters

Delegates can be used as typed function pointers:

```
class FuncAsParameters
{
  public void Run()
  {
    DoSomething(ErrorHandler1);
    DoSomething(ErrorHandler2);
  }

  public bool ErrorHandler1(string message)
  {
```

```
    Console.WriteLine(message);
    var shouldWeContinue = ...
    return shouldWeContinue;
  }

  public bool ErrorHandler2(string message)
  {
    // ...Write message to file...
    var shouldWeContinue = ...
    return shouldWeContinue;
  }

  public void DoSomething(Func<string, bool> errorHandler)
  {
    // In here, we don't care what handler we got passed!
    ...
    if (...error...)
    {
      if (!errorHandler("Some error occurred!"))
      {
        // The handler decided we can't continue
        return;
      }
    }
  }
}
```

Section 96.11: Closure inside a delegate

Closures are inline anonymous methods that have the ability to use `Parent` method variables and other anonymous methods which are defined in the parent's scope.

> In essence, a closure is a block of code which can be executed at a later time, but which maintains the environment in which it was first created - i.e. it can still use the local variables etc of the method which created it, even after that method has finished executing. **-- Jon Skeet**

```
delegate int testDel();
static void Main(string[] args)
{
    int foo = 4;
    testDel myClosure = delegate()
    {
        return foo;
    };
    int bar = myClosure();

}
```

Example taken from Closures in .NET.

Chapter 97: File and Stream I/O

Parameter	Details
path	The location of the file.
append	If the file exist, true will add data to the end of the file (append), false will overwrite the file.
text	Text to be written or stored.
contents	A collection of strings to be written.
source	The location of the file you want to use.
dest	The location you want a file to go to.

Manages files.

Section 97.1: Reading from a file using the System.IO.File class

You can use the **System.IO.File.ReadAllText** function to read the entire contents of a file into a string.

```
string text = System.IO.File.ReadAllText(@"C:\MyFolder\MyTextFile.txt");
```

You can also read a file as an array of lines using the **System.IO.File.ReadAllLines** function:

```
string[] lines = System.IO.File.ReadAllLines(@"C:\MyFolder\MyTextFile.txt");
```

Section 97.2: Lazily reading a file line-by-line via an IEnumerable

When working with large files, you can use the `System.IO.File.ReadLines` method to read all lines from a file into an `IEnumerable<string>`. This is similar to `System.IO.File.ReadAllLines`, except that it doesn't load the whole file into memory at once, making it more efficient when working with large files.

```
IEnumerable<string> AllLines = File.ReadLines("file_name.txt", Encoding.Default);
```

The second parameter of File.ReadLines is optional. You may use it when it is required to specify encoding.

It is important to note that calling `ToArray`, `ToList` or another similar function will force all of the lines to be loaded at once, meaning that the benefit of using `ReadLines` is nullified. It is best to enumerate over the `IEnumerable` using a `foreach` loop or LINQ if using this method.

Section 97.3: Async write text to a file using StreamWriter

```
// filename is a string with the full path
// true is to append
using (System.IO.StreamWriter file = new System.IO.StreamWriter(filename, true))
{
   // Can write either a string or char array
   await file.WriteAsync(text);
}
```

Section 97.4: Copy File

File static class

`File` static class can be easily used for this purpose.

```
File.Copy(@"sourcePath\abc.txt", @"destinationPath\abc.txt");
File.Copy(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

Remark: By this method, file is copied, meaning that it will be read from the source and then written to the destination path. This is a resource consuming process, it would take relative time to the file size, and can cause your program to freeze if you don't utilize threads.

Section 97.5: Writing lines to a file using the System.IO.StreamWriter class

The **System.IO.StreamWriter** class:

> Implements a TextWriter for writing characters to a stream in a particular encoding.

Using the `WriteLine` method, you can write content line-by-line to a file.

Notice the use of the `using` keyword which makes sure the StreamWriter object is disposed as soon as it goes out of scope and thus the file is closed.

```
string[] lines = { "My first string", "My second string", "and even a third string" };
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt"))
{
    foreach (string line in lines)
    {
        sw.WriteLine(line);
    }
}
```

Note that the StreamWriter can receive a second `bool` parameter in it's constructor, allowing to `Append` to a file instead of overwriting the file:

```
bool appendExistingFile = true;
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt",
appendExistingFile ))
{
    sw.WriteLine("This line will be appended to the existing file");
}
```

Section 97.6: Writing to a file using the System.IO.File class

You can use the **System.IO.File.WriteAllText** function to write a string to a file.

```
string text = "String that will be stored in the file";
System.IO.File.WriteAllText(@"C:\MyFolder\OutputFile.txt", text);
```

You can also use the **System.IO.File.WriteAllLines** function which receives an `IEnumerable<String>` as the second parameter (as opposed to a single string in the previous example). This lets you write content from an array of lines.

```
string[] lines = { "My first string", "My second string", "and even a third string" };
System.IO.File.WriteAllLines(@"C:\MyFolder\OutputFile.txt", lines);
```

Section 97.7: Create File

File static class

By using `Create` method of the `File` static class we can create files. Method creates the file at the given path, at the same time it opens the file and gives us the `FileStream` of the file. Make sure you close the file after you are done with it.

ex1:

```
var fileStream1 = File.Create("samplePath");
/// you can write to the fileStream1
fileStream1.Close();
```

ex2:

```
using(var fileStream1 = File.Create("samplePath"))
{
    /// you can write to the fileStream1
}
```

ex3:

```
File.Create("samplePath").Close();
```

FileStream class

There are many overloads of this classes constructor which is actually well documented here. Below example is for the one that covers most used functionalities of this class.

```
var fileStream2 = new FileStream("samplePath", FileMode.OpenOrCreate, FileAccess.ReadWrite,
FileShare.None);
```

You can check the enums for FileMode, FileAccess, and FileShare from those links. What they basically means are as follows:

FileMode: Answers "Should file be created? opened? create if not exist then open?" kinda questions.

FileAccess: Answers "Should I be able to read the file, write to the file or both?" kinda questions.

FileShare: Answers "Should other users be able to read, write etc. to the file while I am using it simultaneously?" kinda questions.

Section 97.8: Move File

File static class

File static class can easily be used for this purpose.

```
File.Move(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

Remark1: Only changes the index of the file (if the file is moved in the same volume). This operation does not take relative time to the file size.

Remark2: Cannot override an existing file on destination path.

Section 97.9: Delete File

```
string path = @"c:\path\to\file.txt";
File.Delete(path);
```

While `Delete` does not throw exception if file doesn't exist, it will throw exception e.g. if specified path is invalid or caller does not have the required permissions. You should always wrap calls to `Delete` inside try-catch block and handle all expected exceptions. In case of possible race conditions, wrap logic inside lock statement.

Section 97.10: Files and Directories

Get all files in Directory

```
var FileSearchRes = Directory.GetFiles(@Path, "*.*", SearchOption.AllDirectories);
```

Returns an array of `FileInfo`, representing all the files in the specified directory.

Get Files with specific extension

```
var FileSearchRes = Directory.GetFiles(@Path, "*.pdf", SearchOption.AllDirectories);
```

Returns an array of `FileInfo`, representing all the files in the specified directory with the specified extension.

Chapter 98: Networking

Section 98.1: Basic TCP Communication Client

This code example creates a TCP client, sends "Hello World" over the socket connection, and then writes the server response to the console before closing the connection.

```
// Declare Variables
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;

// Create TCP client and connect
using (var _client = new TcpClient(host, port))
using (var _netStream = _client.GetStream())
{
    _netStream.ReadTimeout = timeout;

    // Write a message over the socket
    string message = "Hello World!";
    byte[] dataToSend = System.Text.Encoding.ASCII.GetBytes(message);
    _netStream.Write(dataToSend, 0, dataToSend.Length);

    // Read server response
    byte[] recvData = new byte[256];
    int bytes = _netStream.Read(recvData, 0, recvData.Length);
    message = System.Text.Encoding.ASCII.GetString(recvData, 0, bytes);
    Console.WriteLine(string.Format("Server: {0}", message));
};// The client and stream will close as control exits the using block (Equivilent but safer than
calling Close();
```

Section 98.2: Download a file from a web server

Downloading a file from the internet is a very common task required by almost every application your likely to build.

To accomplish this, you can use the "System.Net.WebClient" class.

The simplest use of this, using the "using" pattern, is shown below:

```
using (var webClient = new WebClient())
{
    webClient.DownloadFile("http://www.server.com/file.txt", "C:\\file.txt");
}
```

What this example does is it uses "using" to make sure that your web client is cleaned up correctly when finished, and simply transfers the named resource from the URL in the first parameter, to the named file on your local hard drive in the second parameter.

The first parameter is of type "System.Uri", the second parameter is of type "System.String"

You can also use this function is an async form, so that it goes off and performs the download in the background, while your application get's on with something else, using the call in this way is of major importance in modern applications, as it helps to keep your user interface responsive.

When you use the Async methods, you can hook up event handlers that allow you to monitor the progress, so that

you could for example, update a progress bar, something like the following:

```
var webClient = new WebClient())
webClient.DownloadFileCompleted += new AsyncCompletedEventHandler(Completed);
webClient.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
webClient.DownloadFileAsync("http://www.server.com/file.txt", "C:\\file.txt");
```

One important point to remember if you use the Async versions however, and that's "Be very carefull about using them in a 'using' syntax".

The reason for this is quite simple. Once you call the download file method, it will return immediately. If you have this in a using block, you will return then exit that block, and immediately dispose the class object, and thus cancel your download in progress.

If you use the 'using' way of performing an Async transfer, then be sure to stay inside the enclosing block until the transfer completes.

Section 98.3: Async TCP Client

Using `async`/`await` in C# applications simplifies multi-threading. This is how you can use `async`/`await` in conjunction with a TcpClient.

```
// Declare Variables
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;

// Create TCP client and connect
// Then get the netstream and pass it
// To our StreamWriter and StreamReader
using (var client = new TcpClient())
using (var netstream = client.GetStream())
using (var writer = new StreamWriter(netstream))
using (var reader = new StreamReader(netstream))
{
    // Asynchronsly attempt to connect to server
    await client.ConnectAsync(host, port);

    // AutoFlush the StreamWriter
    // so we don't go over the buffer
    writer.AutoFlush = true;

    // Optionally set a timeout
    netstream.ReadTimeout = timeout;

    // Write a message over the TCP Connection
    string message = "Hello World!";
    await writer.WriteLineAsync(message);

    // Read server response
    string response = await reader.ReadLineAsync();
    Console.WriteLine(string.Format($"Server: {response}"));
}
// The client and stream will close as control exits
// the using block (Equivilent but safer than calling Close();
```

Section 98.4: Basic UDP Client

This code example creates a UDP client then sends "Hello World" across the network to the intended recipient. A listener does not have to be active, as UDP Is connectionless and will broadcast the message regardless. Once the message is sent, the clients work is done.

```
byte[] data = Encoding.ASCII.GetBytes("Hello World");
string ipAddress = "192.168.1.141";
string sendPort = 55600;
try
{
     using (var client = new UdpClient())
     {
         IPEndPoint ep = new IPEndPoint(IPAddress.Parse(ipAddress), sendPort);
         client.Connect(ep);
         client.Send(data, data.Length);
     }
}
catch (Exception ex)
{
     Console.WriteLine(ex.ToString());
}
```

Below is an example of a UDP listener to complement the above client. It will constantly sit and listen for traffic on a given port and simply write that data to the console. This example contains a control flag '`done`' that is not set internally and relies on something to set this to allow for ending the listener and exiting.

```
bool done = false;
int listenPort = 55600;
using(UdpClinet listener = new UdpClient(listenPort))
{
    IPEndPoint listenEndPoint = new IPEndPoint(IPAddress.Any, listenPort);
    while(!done)
    {
        byte[] receivedData = listener.Receive(ref listenPort);

        Console.WriteLine("Received broadcast message from client {0}", listenEndPoint.ToString());

        Console.WriteLine("Decoded data is:");
        Console.WriteLine(Encoding.ASCII.GetString(receivedData)); //should be "Hello World" sent
from above client
    }
}
```

Chapter 99: Performing HTTP requests

Section 99.1: Creating and sending an HTTP POST request

```
using System.Net;
using System.IO;

...

string requestUrl = "https://www.example.com/submit.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);
request.Method = "POST";

// Optionally, set properties of the HttpWebRequest, such as:
request.AutomaticDecompression = DecompressionMethods.Deflate | DecompressionMethods.GZip;
request.ContentType = "application/x-www-form-urlencoded";
// Could also set other HTTP headers such as Request.UserAgent, Request.Referer,
// Request.Accept, or other headers via the Request.Headers collection.

// Set the POST request body data. In this example, the POST data is in
// application/x-www-form-urlencoded format.
string postData = "myparam1=myvalue1&myparam2=myvalue2";
using (var writer = new StreamWriter(request.GetRequestStream()))
{
    writer.Write(postData);
}

// Submit the request, and get the response body from the remote server.
string responseFromRemoteServer;
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseFromRemoteServer = reader.ReadToEnd();
    }
}
```

Section 99.2: Creating and sending an HTTP GET request

```
using System.Net;
using System.IO;

...

string requestUrl = "https://www.example.com/page.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);

// Optionally, set properties of the HttpWebRequest, such as:
request.AutomaticDecompression = DecompressionMethods.GZip | DecompressionMethods.Deflate;
request.Timeout = 2 * 60 * 1000; // 2 minutes, in milliseconds

// Submit the request, and get the response body.
string responseBodyFromRemoteServer;
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseBodyFromRemoteServer = reader.ReadToEnd();
    }
```

```
}
```

Section 99.3: Error handling of specific HTTP response codes (such as 404 Not Found)

```
using System.Net;

...

string serverResponse;
try
{
    // Call a method that performs an HTTP request (per the above examples).
    serverResponse = PerformHttpRequest();
}
catch (WebException ex)
{
    if (ex.Status == WebExceptionStatus.ProtocolError)
    {
        HttpWebResponse response = ex.Response as HttpWebResponse;
        if (response != null)
        {
            if ((int)response.StatusCode == 404) // Not Found
            {
                // Handle the 404 Not Found error
                // ...
            }
            else
            {
                // Could handle other response.StatusCode values here.
                // ...
            }
        }
    }
    else
    {
        // Could handle other error conditions here, such as WebExceptionStatus.ConnectFailure.
        // ...
    }
}
```

Section 99.4: Retrieve HTML for Web Page (Simple)

```
string contents = "";
string url = "http://msdn.microsoft.com";

using (System.Net.WebClient client = new System.Net.WebClient())
{
    contents = client.DownloadString(url);
}

Console.WriteLine(contents);
```

Section 99.5: Sending asynchronous HTTP POST request with JSON body

```
public static async Task PostAsync(this Uri uri, object value)
{
```

```
    var content = new ObjectContext(value.GetType(), value, new JsonMediaTypeFormatter());

    using (var client = new HttpClient())
    {
        return await client.PostAsync(uri, content);
    }
}

. . .

var uri = new Uri("http://stackoverflow.com/documentation/c%23/1971/performing-http-requests");
await uri.PostAsync(new { foo = 123.45, bar = "Richard Feynman" });
```

Chapter 100: Reading and writing .zip files

Parameter	Details
archiveFileName	The path to the archive to open, specified as a relative or absolute path. A relative path is interpreted as relative to the current working directory.

Section 100.1: Writing to a zip file

To write a new .zip file:

```
System.IO.Compression
System.IO.Compression.FileSystem

using (FileStream zipToOpen = new FileStream(@"C:\temp", FileMode.Open))
{
    using (ZipArchive archive = new ZipArchive(zipToOpen, ZipArchiveMode.Update))
    {
        ZipArchiveEntry readmeEntry = archive.CreateEntry("Readme.txt");
        using (StreamWriter writer = new StreamWriter(readmeEntry.Open()))
        {
            writer.WriteLine("Information about this package.");
            writer.WriteLine("========================");
        }
    }
}
```

Section 100.2: Writing Zip Files in-memory

The following example will return the `byte[]` data of a zipped file containing the files provided to it, without needing access to the file system.

```
public static byte[] ZipFiles(Dictionary<string, byte[]> files)
{
    using (MemoryStream ms = new MemoryStream())
    {
        using (ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Update))
        {
            foreach (var file in files)
            {
                ZipArchiveEntry orderEntry = archive.CreateEntry(file.Key); //create a file with
this name
                using (BinaryWriter writer = new BinaryWriter(orderEntry.Open()))
                {
                    writer.Write(file.Value); //write the binary data
                }
            }
        }
        //ZipArchive must be disposed before the MemoryStream has data
        return ms.ToArray();
    }
}
```

Section 100.3: Get files from a Zip file

This example gets a listing of files from the provided zip archive binary data:

```
public static Dictionary<string, byte[]> GetFiles(byte[] zippedFile)
```

```
{
    using (MemoryStream ms = new MemoryStream(zippedFile))
    using (ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Read))
    {
        return archive.Entries.ToDictionary(x => x.FullName, x => ReadStream(x.Open()));
    }
}

private static byte[] ReadStream(Stream stream)
{
    using (var ms = new MemoryStream())
    {
        stream.CopyTo(ms);
        return ms.ToArray();
    }
}
```

Section 100.4: The following example shows how to open a zip archive and extract all .txt files to a folder

```
using System;
using System.IO;
using System.IO.Compression;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string zipPath = @"c:\example\start.zip";
            string extractPath = @"c:\example\extract";

            using (ZipArchive archive = ZipFile.OpenRead(zipPath))
            {
                foreach (ZipArchiveEntry entry in archive.Entries)
                {
                    if (entry.FullName.EndsWith(".txt", StringComparison.OrdinalIgnoreCase))
                    {
                        entry.ExtractToFile(Path.Combine(extractPath, entry.FullName));
                    }
                }
            }
        }
    }
}
```

Chapter 101: FileSystemWatcher

path	filter
The directory to monitor, in standard or Universal Naming Convention (UNC) notation.	The type of files to watch. For example, "*.txt" watches for changes to all text files.

Section 101.1: IsFileReady

A common mistake a lot of people starting out with FileSystemWatcher does is not taking into account That the FileWatcher event is raised as soon as the file is created. However, it may take some time for the file to be finished .

Example:

Take a file size of 1 GB for example . The file apr ask created by another program (Explorer.exe copying it from somewhere) but it will take minutes to finish that process. The event is raised that creation time and you need to wait for the file to be ready to be copied.

This is a method for checking if the file is ready.

```
 public static bool IsFileReady(String sFilename)
{
    // If the file can be opened for exclusive access it means that the file
    // is no longer locked by another process.
    try
    {
        using (FileStream inputStream = File.Open(sFilename, FileMode.Open, FileAccess.Read,
FileShare.None))
        {
            if (inputStream.Length > 0)
            {
                return true;
            }
            else
            {
                return false;
            }

        }
    }
    catch (Exception)
    {
        return false;
    }
}
```

Section 101.2: Basic FileWatcher

The following example creates a `FileSystemWatcher` to watch the directory specified at run time. The component is set to watch for changes in **LastWrite** and **LastAccess** time, the creation, deletion, or renaming of text files in the directory. If a file is changed, created, or deleted, the path to the file prints to the console. When a file is renamed, the old and new paths print to the console.

Use the System.Diagnostics and System.IO namespaces for this example.

```
FileSystemWatcher watcher;
```

```
private void watch()
{
  // Create a new FileSystemWatcher and set its properties.
  watcher = new FileSystemWatcher();
  watcher.Path = path;

 /* Watch for changes in LastAccess and LastWrite times, and
       the renaming of files or directories. */
  watcher.NotifyFilter = NotifyFilters.LastAccess | NotifyFilters.LastWrite
                         | NotifyFilters.FileName | NotifyFilters.DirectoryName;

  // Only watch text files.
  watcher.Filter = "*.txt*";

  // Add event handler.
  watcher.Changed += new FileSystemEventHandler(OnChanged);
  // Begin watching.
  watcher.EnableRaisingEvents = true;
}

// Define the event handler.
private void OnChanged(object source, FileSystemEventArgs e)
{
  //Copies file to another directory or another action.
  Console.WriteLine("File: " +  e.FullPath + " " + e.ChangeType);
}
```

Chapter 102: Access network shared folder with username and password

Accessing network share file using PInvoke.

Section 102.1: Code to access network shared file

```
public class NetworkConnection : IDisposable
    {
        string _networkName;

        public NetworkConnection(string networkName,
            NetworkCredential credentials)
        {
            _networkName = networkName;

            var netResource = new NetResource()
            {
                Scope = ResourceScope.GlobalNetwork,
                ResourceType = ResourceType.Disk,
                DisplayType = ResourceDisplaytype.Share,
                RemoteName = networkName
            };

            var userName = string.IsNullOrEmpty(credentials.Domain)
                ? credentials.UserName
                : string.Format(@"{0}\{1}", credentials.Domain, credentials.UserName);

            var result = WNetAddConnection2(
                netResource,
                credentials.Password,
                userName,
                0);

            if (result != 0)
            {
                throw new Win32Exception(result);
            }
        }

    ~NetworkConnection()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        WNetCancelConnection2(_networkName, 0, true);
    }

    [DllImport("mpr.dll")]
    private static extern int WNetAddConnection2(NetResource netResource,
        string password, string username, int flags);
```

```
    [DllImport("mpr.dll")]
    private static extern int WNetCancelConnection2(string name, int flags,
        bool force);
}

[StructLayout(LayoutKind.Sequential)]
public class NetResource
{
    public ResourceScope Scope;
    public ResourceType ResourceType;
    public ResourceDisplaytype DisplayType;
    public int Usage;
    public string LocalName;
    public string RemoteName;
    public string Comment;
    public string Provider;
}

public enum ResourceScope : int
{
    Connected = 1,
    GlobalNetwork,
    Remembered,
    Recent,
    Context
};

public enum ResourceType : int
{
    Any = 0,
    Disk = 1,
    Print = 2,
    Reserved = 8,
}

public enum ResourceDisplaytype : int
{
    Generic = 0x0,
    Domain = 0x01,
    Server = 0x02,
    Share = 0x03,
    File = 0x04,
    Group = 0x05,
    Network = 0x06,
    Root = 0x07,
    Shareadmin = 0x08,
    Directory = 0x09,
    Tree = 0x0a,
    Ndscontainer = 0x0b
}
```

Chapter 103: Asynchronous Socket

By using asynchronous sockets a server can listening for incoming connections and do some other logic in the mean time in contrast to synchronous socket when they are listening they block the main thread and the application is becoming unresponsive an will freeze until a client connects.

Section 103.1: Asynchronous Socket (Client / Server) example

> **Server Side example**

Create Listener for server

Start of with creating an server that will handle clients that connect, and requests that will be send. So create an Listener Class that will handle this.

```
class Listener
{
    public Socket ListenerSocket; //This is the socket that will listen to any incoming connections
    public short Port = 1234; // on this port we will listen

    public Listener()
    {
        ListenerSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
    }
 }
```

First we need to initialize the Listener socket where we can listen on for any connections. We are going to use an Tcp Socket that is why we use SocketType.Stream. Also we specify to witch port the server should listen to

Then we start listening for any incoming connections.

The tree methods we use here are:

1. ListenerSocket.Bind();

   This method binds the socket to an IPEndPoint. This class contains the host and local or remote port information needed by an application to connect to a service on a host.

2. ListenerSocket.Listen(10);

   The backlog parameter specifies the number of incoming connections that can be queued for acceptance.

3. ListenerSocket.BeginAccept();

   The server will start listening for incoming connections and will go on with other logic. When there is an connection the server switches back to this method and will run the AcceptCallBack methodt

```
    public void StartListening()
    {
```

```
        try
        {
                MessageBox.Show($"Listening started port:{Port} protocol type:
{ProtocolType.Tcp}");
                ListenerSocket.Bind(new IPEndPoint(IPAddress.Any, Port));
                ListenerSocket.Listen(10);
                ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
        }
        catch(Exception ex)
        {
            throw new Exception("listening error" + ex);
        }
    }
```

So when a client connects we can accept them by this method:

Three methods wee use here are:

1. ListenerSocket.EndAccept()

   We started the callback with `Listener.BeginAccept()` end now we have to end that call back. `The EndAccept()` method accepts an IAsyncResult parameter, this will store the state of the asynchronous method, From this state we can extract the socket where the incoming connection was coming from.

2. `ClientController.AddClient()`

   With the socket we got from `EndAccept()` we create an Client with an own made method *(code ClientController below server example)*.

3. ListenerSocket.BeginAccept()

   We need to start listening again when the socket is done with handling the new connection. Pass in the method who will catch this callback. And also pass int the Listener socket so we can reuse this socket for upcoming connections.

```
    public void AcceptCallback(IAsyncResult ar)
    {
        try
        {
            Console.WriteLine($"Accept CallBack port:{Port} protocol type: {ProtocolType.Tcp}");
            Socket acceptedSocket = ListenerSocket.EndAccept(ar);
            ClientController.AddClient(acceptedSocket);

            ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
        }
        catch (Exception ex)
        {
            throw new Exception("Base Accept error"+ ex);
        }
    }
```

Now we have an Listening Socket but how do we receive data send by the client that is what the next code is showing.

Create Server Receiver for each client

First of create a receive class with a constructor that takes in a Socket as parameter:

```
    public class ReceivePacket
    {
        private byte[] _buffer;
        private Socket _receiveSocket;

        public ReceivePacket(Socket receiveSocket)
        {
           _receiveSocket = receiveSocket;
        }
    }
```

In the next method we first start off with giving the buffer a size of 4 bytes (Int32) or package contains to parts {lenght, actual data}. So the first 4 bytes we reserve for the lenght of the data the rest for the actual data.

Next we use BeginReceive() method. This method is used to start receiving from connected clients and when it will receive data it will run the `ReceiveCallback` function.

```
    public void StartReceiving()
    {
        try
        {
            _buffer = new byte[4];
            _receiveSocket.BeginReceive(_buffer, 0, _buffer.Length, SocketFlags.None,
ReceiveCallback, null);
        }
        catch {}
    }

    private void ReceiveCallback(IAsyncResult AR)
    {
        try
        {
            // if bytes are less than 1 takes place when a client disconnect from the server.
            // So we run the Disconnect function on the current client
            if (_receiveSocket.EndReceive(AR) > 1)
            {
                // Convert the first 4 bytes (int 32) that we received and convert it to an Int32
(this is the size for the coming data).
                _buffer = new byte[BitConverter.ToInt32(_buffer, 0)];
                // Next receive this data into the buffer with size that we did receive before
                _receiveSocket.Receive(_buffer, _buffer.Length, SocketFlags.None);
                // When we received everything its onto you to convert it into the data that you've
send.
                // For example string, int etc... in this example I only use the implementation for
sending and receiving a string.

                // Convert the bytes to string and output it in a message box
                string data = Encoding.Default.GetString(_buffer);
                MessageBox.Show(data);
                // Now we have to start all over again with waiting for a data to come from the
socket.
                StartReceiving();
            }
            else
            {
                Disconnect();
```

```
            }
        }
        catch
        {
            // if exeption is throw check if socket is connected because than you can startreive
again else Dissconect
            if (!_receiveSocket.Connected)
            {
                Disconnect();
            }
            else
            {
                StartReceiving();
            }
        }
    }

    private void Disconnect()
    {
        // Close connection
        _receiveSocket.Disconnect(true);
        // Next line only apply for the server side receive
        ClientController.RemoveClient(_clientId);
        // Next line only apply on the Client Side receive
        Here you want to run the method TryToConnect()
    }
```

So we've setup a server that can receive and listen for incoming connections. When a clients connect it will be added to a list of clients and every client has his own receive class. To make the server listen:

```
Listener listener = new Listener();
listener.StartListening();
```

Some Classes I use in this example

```
    class Client
    {
        public Socket _socket { get; set; }
        public ReceivePacket Receive { get; set; }
        public int Id { get; set; }

        public Client(Socket socket, int id)
        {
            Receive = new ReceivePacket(socket, id);
            Receive.StartReceiving();
            _socket = socket;
            Id = id;
        }
    }

     static class ClientController
     {
          public static List<Client> Clients = new List<Client>();

          public static void AddClient(Socket socket)
          {
              Clients.Add(new Client(socket,Clients.Count));
          }

          public static void RemoveClient(int id)
          {
```

```
            Clients.RemoveAt(Clients.FindIndex(x => x.Id == id));
        }
    }
```

> **Client Side example**

Connecting to server

First of all we want to create a class what connects to the server te name we give it is: Connector:

```
class Connector
{
    private Socket _connectingSocket;
}
```

Next Method for this class is TryToConnect()

This method goth a few interestin things:

1. Create the socket;
2. Next I loop until the socket is connected
3. Every loop it is just holding the Thread for 1 second we don't want to DOS the server XD
4. With Connect() it will try to connect to the server. If it fails it will throw an exception but the wile will keep the program connecting to the server. You can use a Connect CallBack method for this, but I'll just go for calling a method when the Socket is connected.
5. Notice the Client is now trying to connect to your local pc on port 1234.

```
public void TryToConnect()
{
    _connectingSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);

     while (!_connectingSocket.Connected)
     {
         Thread.Sleep(1000);

         try
         {
             _connectingSocket.Connect(new IPEndPoint(IPAddress.Parse("127.0.0.1"), 1234));
         }
         catch { }
     }
     SetupForReceiveing();
    }
}

private void SetupForReceiveing()
{
   // View Client Class bottom of Client Example
    Client.SetClient(_connectingSocket);
    Client.StartReceiving();
}
```

Sending a message to the server

So now we have an almost finish or Socket application. The only thing that we don't have jet is a Class for sending a message to the server.

```
public class SendPacket
{
    private Socket _sendSocked;

    public SendPacket(Socket sendSocket)
    {
        _sendSocked = sendSocket;
    }

    public void Send(string data)
    {
        try
        {
            /* what hapends here:
                 1. Create a list of bytes
                 2. Add the length of the string to the list.
                    So if this message arrives at the server we can easily read the length of the
coming message.
                 3. Add the message(string) bytes
            */

            var fullPacket = new List<byte>();
            fullPacket.AddRange(BitConverter.GetBytes(data.Length));
            fullPacket.AddRange(Encoding.Default.GetBytes(data));

            /* Send the message to the server we are currently connected to.
            Or package stucture is {length of data 4 bytes (int32), actual data}*/
            _sendSocked.Send(fullPacket.ToArray());
        }
        catch (Exception ex)
        {
            throw new Exception();
        }
    }
```

Finaly crate two buttons one for connect and the other for sending a message:

```
    private void ConnectClick(object sender, EventArgs e)
    {
        Connector tpp = new Connector();
        tpp.TryToConnect();
    }

    private void SendClick(object sender, EventArgs e)
    {
        Client.SendString("Test data from client");
    }
```

The client class I used in this example

```
    public static void SetClient(Socket socket)
    {
        Id = 1;
        Socket = socket;
        Receive = new ReceivePacket(socket, Id);
```

```
        SendPacket = new SendPacket(socket);
    }
```

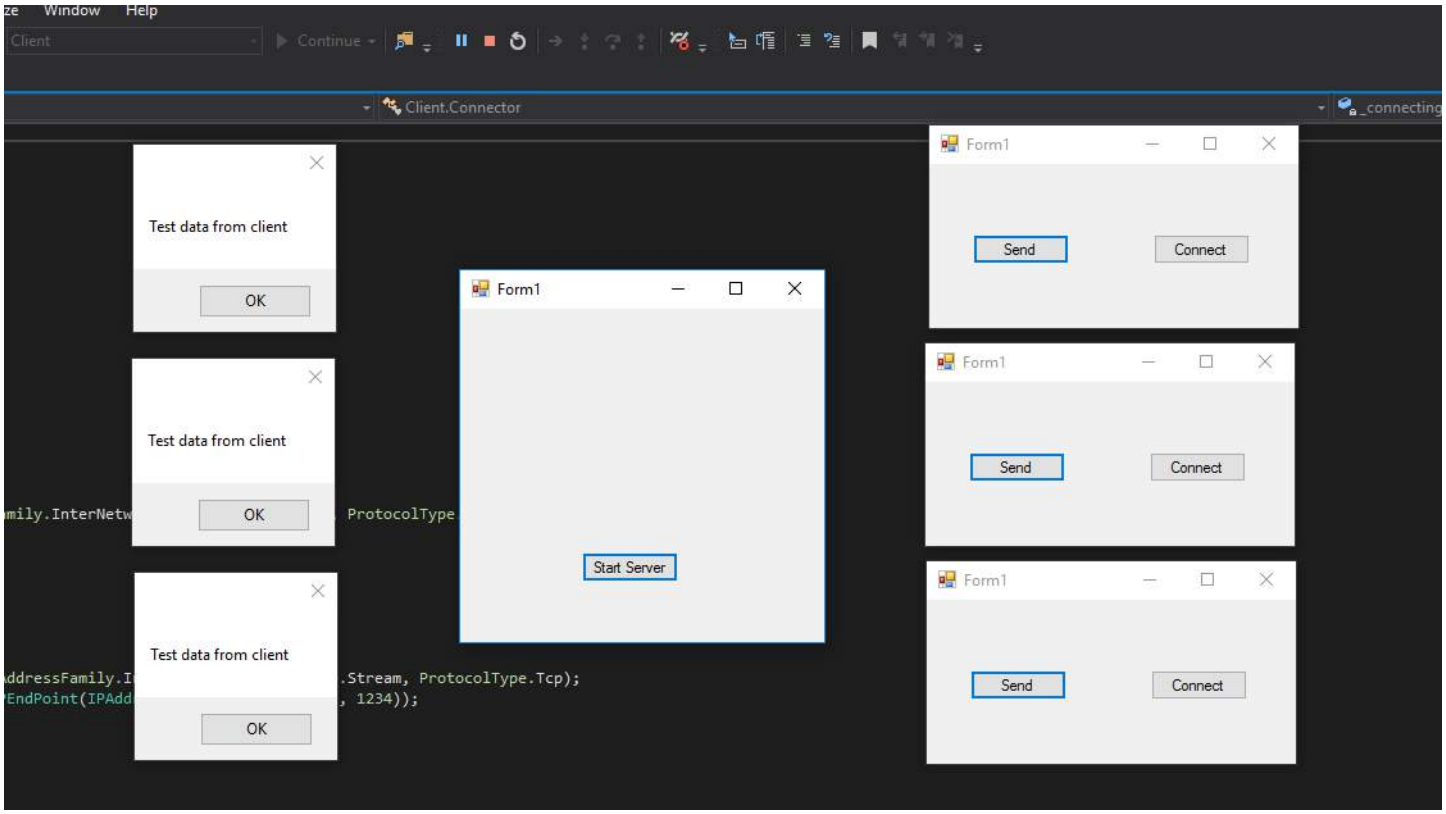
Notice

The Receive Class from the server is the same as the receive class from the client.

> **Conclusion**

You now have a server and a client. You can work this basic example out. For example make it that the server also can receive files or other tings. Or send a message to the client. In the server you got a list of client so when you receive something you will know from with client it came from.

Final result:



Chapter 104: Action Filters

Section 104.1: Custom Action Filters

We write custom action filters for various reasons. We may have a custom action filter for logging, or for saving data to database before any action execution. We could also have one for fetching data from the database and setting it as the global values of the application.

To create a custom action filter, we need to perform the following tasks:

1. Create a class
2. Inherit it from ActionFilterAttribute class

Override at least one of the following methods:

OnActionExecuting – This method is called before a controller action is executed.

OnActionExecuted – This method is called after a controller action is executed.

OnResultExecuting – This method is called before a controller action result is executed.

OnResultExecuted – This method is called after a controller action result is executed.

The filter can be created as shown in the listing below:

```
    using System;

    using System.Diagnostics;

    using System.Web.Mvc;



    namespace WebApplication1
    {

        public class MyFirstCustomFilter : ActionFilterAttribute
        {
            public override void OnResultExecuting(ResultExecutingContext filterContext)
            {
                //You may fetch data from database here
                filterContext.Controller.ViewBag.GreetMesssage = "Hello Foo";
                base.OnResultExecuting(filterContext);
            }

            public override void OnActionExecuting(ActionExecutingContext filterContext)
            {
                var controllerName = filterContext.RouteData.Values["controller"];
                var actionName = filterContext.RouteData.Values["action"];
                var message = String.Format("{0} controller:{1} action:{2}", "onactionexecuting",
controllerName, actionName);
                Debug.WriteLine(message, "Action Filter Log");
                base.OnActionExecuting(filterContext);
            }
        }
    }
```

Chapter 105: Polymorphism

Section 105.1: Types of Polymorphism

Polymorphism means that a operation can also be applied to values of some other types.

There are multiple types of Polymorphism:

- **Ad hoc polymorphism:**
  contains `function overloading`. The target is that a Method can be used with different types without the need of being generic.
- **Parametric polymorphism:**
  is the use of generic types. See Generics
- **Subtyping:**
  has the target inherit of a class to generalize a similar functionality

Ad hoc polymorphism

The target of `Ad hoc polymorphism` is to create a method, that can be called by different datatypes without a need of type-conversion in the function call or generics. The following method(s) `sumInt(par1, par2)` can be called with different datatypes and has for each combination of types a own implementation:

```
public static int sumInt( int a, int b)
{
    return a + b;
}

public static int sumInt( string a, string b)
{
    int _a, _b;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    if(!Int32.TryParse(b, out _b))
        _b = 0;

    return _a + _b;
}

public static int sumInt(string a, int b)
{
    int _a;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    return _a + b;
}

public static int sumInt(int a, string b)
{
    return sumInt(b,a);
}
```

Here's a example call:

```
public static void Main()
{
    Console.WriteLine(sumInt( 1 , 2 ));  //  3
    Console.WriteLine(sumInt("3","4"));  //  7
    Console.WriteLine(sumInt("5", 6 ));  // 11
    Console.WriteLine(sumInt( 7 ,"8"));  // 15
}
```

Subtyping

Subtyping is the use of inherit from a base class to generalize a similar behavior:

```
public interface Car{
    void refuel();
}

public class NormalCar : Car
{
    public void refuel()
    {
        Console.WriteLine("Refueling with petrol");
    }
}

public class ElectricCar : Car
{
    public void refuel()
    {
        Console.WriteLine("Charging battery");
    }
}
```

Both classes `NormalCar` and `ElectricCar` now have a method to refuel, but their own implementation. Here's a Example:

```
public static void Main()
{
    List<Car> cars = new List<Car>(){
        new NormalCar(),
        new ElectricCar()
    };

    cars.ForEach(x => x.refuel());
}
```

The output will be was following:

> Refueling with petrol
> Charging battery

Section 105.2: Another Polymorphism Example

Polymorphism is one of the pillar of OOP. Poly derives from a Greek term which means 'multiple forms'.

Below is an example which exhibits Polymorphism. The class `Vehicle` takes multiple forms as a base class.

The Derived classes `Ducati` and `Lamborghini` inherits from `Vehicle` and overrides the base class's `Display()` method, to display its own `NumberOfWheels`.

```
public class Vehicle
{
    protected int NumberOfWheels { get; set; } = 0;
    public Vehicle()
    {
    }

    public virtual void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Vehicle)} is {NumberOfWheels}");
    }
}

public class Ducati : Vehicle
{
    public Ducati()
    {
        NoOfWheels = 2;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Ducati)} is {NumberOfWheels}");
    }
}

public class Lamborghini : Vehicle
{
    public Lamborghini()
    {
        NoOfWheels = 4;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Lamborghini)} is
{NumberOfWheels}");
    }
}
```

Below is the code snippet where Polymorphism is exhibited. The object is created for the base type `Vehicle` using a variable `vehicle` at Line 1. It calls the base class method `Display()` at Line 2 and display the output as shown.

```
static void Main(string[] args)
{
    Vehicle vehicle = new Vehicle();    //Line 1
    vehicle.Display();                  //Line 2
    vehicle = new Ducati();             //Line 3
    vehicle.Display();                  //Line 4
    vehicle = new Lamborghini();        //Line 5
    vehicle.Display();                  //Line 6
}
```

At Line 3, the `vehicle` object is pointed to the derived class `Ducati` and calls its `Display()` method, which displays the output as shown. Here comes the polymorphic behavior, even though the object `vehicle` is of type `Vehicle`, it calls the derived class method `Display()` as the type `Ducati` overrides the base class `Display()` method, since the

`vehicle` object is pointed towards `Ducati`.

The same explanation is applicable when it invokes the `Lamborghini` type's `Display()` method.

The Output is shown below

```
The number of wheels for the Vehicle is 0        // Line 2
The number of wheels for the Ducati is 2         // Line 4
The number of wheels for the Lamborghini is 4    // Line 6
```

Chapter 106: Immutability

Section 106.1: System.String class

In C# (and .NET) a string is represented by class System.String. The `string` keyword is an alias for this class.

The System.String class is immutable, i.e once created its state cannot be altered.

So all the operations you perform on a string like Substring, Remove, Replace, concatenation using `+` operator etc will create a new string and return it.

See the following program for demonstration -

```
string str = "mystring";
string newString = str.Substring(3);
Console.WriteLine(newString);
Console.WriteLine(str);
```

This will print `string` and `mystring` respectively.

Section 106.2: Strings and immutability

Immutable types are types that when changed create a new version of the object in memory, rather than changing the existing object in memory. The simplest example of this is the built-in `string` type.

Taking the following code, that appends " world" onto the word "Hello"

```
string myString = "hello";
myString += " world";
```

What is happening in memory in this case is that a new object is created when you append to the `string` in the second line. If you do this as part of a large loop, there is the potential for this to cause performance issues in your application.

The mutable equivalent for a `string` is a `StringBuilder`

Taking the following code

```
StringBuilder myStringBuilder = new StringBuilder("hello");
myStringBuilder.append(" world");
```

When you run this, you are modifying the `StringBuilder` object itself in memory.

Chapter 107: Indexer

Section 107.1: A simple indexer

```
class Foo
{
    private string[] cities = new[] { "Paris", "London", "Berlin" };

    public string this[int index]
    {
        get {
            return cities[index];
        }
        set {
            cities[index] = value;
        }
    }
}
```

Usage:

```
    var foo = new Foo();

    // access a value
    string berlin = foo[2];

    // assign a value
    foo[0] = "Rome";
```

View Demo

Section 107.2: Overloading the indexer to create a SparseArray

By overloading the indexer you can create a class that looks and feels like an array but isn't. It will have O(1) get and set methods, can access an element at index 100, and yet still have the size of the elements inside of it. The SparseArray class

```
class SparseArray
    {
        Dictionary<int, string> array = new Dictionary<int, string>();

        public string this[int i]
        {
            get
            {
                if(!array.ContainsKey(i))
                {
                    return null;
                }
                return array[i];
            }
            set
            {
                if(!array.ContainsKey(i))
                    array.Add(i, value);
            }
```

```
        }
    }
```

Section 107.3: Indexer with 2 arguments and interface

```
interface ITable {
    // an indexer can be declared in an interface
    object this[int x, int y] { get; set; }
}

class DataTable : ITable
{
    private object[,] cells = new object[10, 10];

    /// <summary>
    /// implementation of the indexer declared in the interface
    /// </summary>
    /// <param name="x">X-Index</param>
    /// <param name="y">Y-Index</param>
    /// <returns>Content of this cell</returns>
    public object this[int x, int y]
    {
        get
        {
            return cells[x, y];
        }
        set
        {
            cells[x, y] = value;
        }
    }
}
```

Chapter 108: Checked and Unchecked

Section 108.1: Checked and Unchecked

C# statements executes in either checked or unchecked context. In a checked context, arithmetic overflow raises an exception. In an unchecked context, arithmetic overflow is ignored and the result is truncated.

```
short m = 32767;
short n = 32767;
int result1 =  checked((short)(m + n));   //will throw an OverflowException
int result2 =  unchecked((short)(m + n)); // will return -2
```

If neither of these are specified then the default context will rely on other factors, such as compiler options.

Section 108.2: Checked and Unchecked as a scope

The keywords can also create scopes in order to (un)check multiple operations.

```
short m = 32767;
short n = 32767;
checked
{
    int result1 = (short)(m + n); //will throw an OverflowException
}
unchecked
{
    int result2 = (short)(m + n); // will return -2
}
```

Chapter 109: Stream

Section 109.1: Using Streams

A stream is an object that provides a low-level means to transfer data. They themselves do not act as data containers.

The data that we deal with is in form of byte array(`byte []`). The functions for reading and writing are all byte orientated, e.g. `WriteByte()`.

There are no functions for dealing with integers, strings etc. This makes the stream very general-purpose, but less simple to work with if, say, you just want to transfer text. Streams can be particularly very helpful when you are dealing with large amount of data.

We will need to use different type of Stream based where it needs to be written/read from (i.e. the backing store). For example, if the source is a file, we need to use `FileStream`:

```
string filePath = @"c:\Users\exampleuser\Documents\userinputlog.txt";
using (FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read,
FileShare.ReadWrite))
{
    // do stuff here...

    fs.Close();
}
```

Similarly, `MemoryStream` is used if the backing store is memory:

```
// Read all bytes in from a file on the disk.
byte[] file = File.ReadAllBytes(“C:\\file.txt”);

// Create a memory stream from those bytes.
using (MemoryStream memory = new MemoryStream(file))
{
   // do stuff here...
}
```

Similarly, `System.Net.Sockets.NetworkStream` is used for network access.

All Streams are derived from the generic class `System.IO.Stream`. Data cannot be directly read or written from streams. The .NET Framework provides helper classes such as `StreamReader`, `StreamWriter`, `BinaryReader` and `BinaryWriter` that convert between native types and the low-level stream interface, and transfer the data to or from the stream for you.

Reading and writing to streams can be done via `StreamReader` and `StreamWriter`. One should be careful when closing these. By default, closing will also close contained stream as well and make it unusable for further uses. This default behaviour can be change by using a constructor which has `bool leaveOpen` parameter and setting its value as `true`.

`StreamWriter`:

```
FileStream fs = new FileStream("sample.txt", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
string NextLine = "This is the appended line.";
sw.Write(NextLine);
```

```
sw.Close();
//fs.Close(); There is no need to close fs. Closing sw will also close the stream it contains.
```

`StreamReader`:

```
using (var ms = new MemoryStream())
{
    StreamWriter sw = new StreamWriter(ms);
    sw.Write(123);
    //sw.Close();     This will close ms and when we try to use ms later it will cause an exception
    sw.Flush();     //You can send the remaining data to stream. Closing will do this automatically
    // We need to set the position to 0 in order to read
    // from the beginning.
    ms.Position = 0;
    StreamReader sr = new StreamReader(ms);
    var myStr = sr.ReadToEnd();
    sr.Close();
    ms.Close();
}
```

Since Classes `Stream`, `StreamReader`, `StreamWriter`, etc. implement the `IDisposable` interface, we can call the `Dispose()` method on objects of these classes.

Chapter 110: Timers

Section 110.1: Multithreaded Timers

`System.Threading.Timer` - Simplest multithreaded timer. Contains two methods and one constructor.

Example: A timer calls the DataWrite method, which writes "multithread executed..." after five seconds have elapsed, and then every second after that until the user presses Enter:

```
using System;
using System.Threading;
class Program
{
  static void Main()
  {
    // First interval = 5000ms; subsequent intervals = 1000ms
    Timer timer = new Timer (DataWrite, "multithread executed...", 5000, 1000);
    Console.ReadLine();
    timer.Dispose(); // This both stops the timer and cleans up.
  }

  static void DataWrite (object data)
  {
    // This runs on a pooled thread
    Console.WriteLine (data); // Writes "multithread executed..."
  }
}
```

Note : Will post a separate section for disposing multithreaded timers.

`Change` - This method can be called when you would like change the timer interval.

`Timeout.Infinite` - If you want to fire just once. Specify this in the last argument of the constructor.

`System.Timers` - Another timer class provided by .NET Framework. It wraps the `System.Threading.Timer`.

Features:

- `IComponent` - Allowing it to be sited in the Visual Studio's Designer's component tray
- `Interval` property instead of a `Change` method
- `Elapsed` `event` instead of a callback `delegate`
- `Enabled` property to start and stop the timer (`default value = false`)
- `Start` & `Stop` methods in case if you get confused by `Enabled` property (above point)
- `AutoReset` - for indicating a recurring event (`default value = true`)
- `SynchronizingObject` property with `Invoke` and `BeginInvoke` methods for safely calling methods on WPF elements and Windows Forms controls

Example representing all the above features:

```
using System;
using System.Timers; // Timers namespace rather than Threading
class SystemTimer
{
  static void Main()
  {
    Timer timer = new Timer(); // Doesn't require any args
    timer.Interval = 500;
```

```
    timer.Elapsed += timer_Elapsed; // Uses an event instead of a delegate
    timer.Start(); // Start the timer
    Console.ReadLine();
    timer.Stop(); // Stop the timer
    Console.ReadLine();
    timer.Start(); // Restart the timer
    Console.ReadLine();
    timer.Dispose(); // Permanently stop the timer
 }

 static void timer_Elapsed(object sender, EventArgs e)
 {
   Console.WriteLine ("Tick");
 }
}
```

`Multithreaded timers` - use the thread pool to allow a few threads to serve many timers. It means that callback method or `Elapsed` event may trigger on a different thread each time it is called.

`Elapsed` - this event always fires on time—regardless of whether the previous `Elapsed` event finished executing. Because of this, callbacks or event handlers must be thread-safe. The accuracy of multithreaded timers depends on the OS, and is typically in the 10–20 ms.

`interop` - when ever you need greater accuracy use this and call the Windows multimedia timer. This has accuracy down to 1 ms and it is defined in `winmm.dll`.

`timeBeginPeriod` - Call this first to inform OS that you need high timing accuracy

`timeSetEvent` - call this after `timeBeginPeriod` to start a multimedia timer.

`timeKillEvent` - call this when you are done, this stops the timer

`timeEndPeriod` - Call this to inform the OS that you no longer need high timing accuracy.

You can find complete examples on the Internet that use the multimedia timer by searching for the keywords `dllimport` `winmm.dll` `timesetevent`.

Section 110.2: Creating an Instance of a Timer

Timers are used to perform tasks at specific intervals of time (Do X every Y seconds) Below is an example of creating a new instance of a Timer.

NOTE: This applies to Timers using WinForms. If using WPF, you may want to look into `DispatcherTimer`

```
    using System.Windows.Forms; //Timers use the Windows.Forms namespace

    public partial class Form1 : Form
    {

        Timer myTimer = new Timer(); //create an instance of Timer named myTimer


        public Form1()
        {
            InitializeComponent();
        }

    }
```

Section 110.3: Assigning the "Tick" event handler to a Timer

All actions performed in a timer are handled in the "Tick" event.

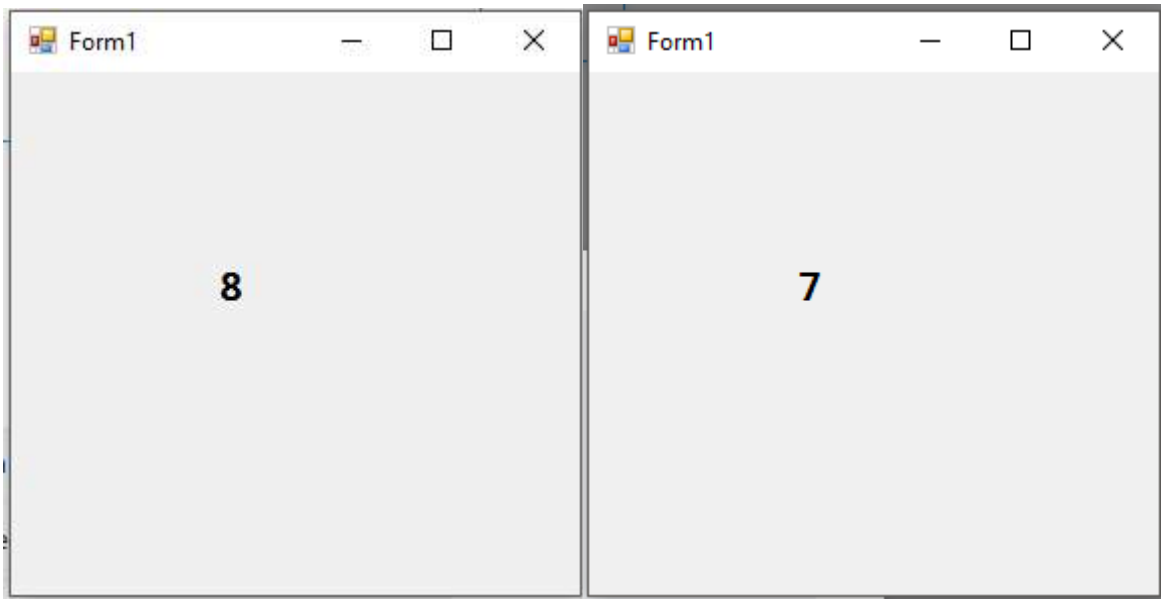
```
public partial class Form1 : Form
{

    Timer myTimer = new Timer();


    public Form1()
    {
        InitializeComponent();

        myTimer.Tick += myTimer_Tick; //assign the event handler named "myTimer_Tick"
    }

    private void myTimer_Tick(object sender, EventArgs e)
    {
        // Perform your actions here.
    }
}
```

Section 110.4: Example: Using a Timer to perform a simple countdown

```
    public partial class Form1 : Form
    {

    Timer myTimer = new Timer();
    int timeLeft = 10;

        public Form1()
        {
            InitializeComponent();

            //set properties for the Timer
            myTimer.Interval = 1000;
            myTimer.Enabled = true;

            //Set the event handler for the timer, named "myTimer_Tick"
            myTimer.Tick += myTimer_Tick;

            //Start the timer as soon as the form is loaded
            myTimer.Start();

            //Show the time set in the "timeLeft" variable
            lblCountDown.Text = timeLeft.ToString();

        }

        private void myTimer_Tick(object sender, EventArgs e)
        {
            //perform these actions at the interval set in the properties.
            lblCountDown.Text = timeLeft.ToString();
            timeLeft -= 1;

            if (timeLeft < 0)
            {
                myTimer.Stop();
```

```
            }
        }
    }
```

Results in...



And so on...

Chapter 111: Stopwatches

Section 111.1: IsHighResolution

- The IsHighResolution property indicates whether the timer is based on a high-resolution performance counter or based on the DateTime class.
- This field is read-only.

```
// Display the timer frequency and resolution.
if (Stopwatch.IsHighResolution)
{
    Console.WriteLine("Operations timed using the system's high-resolution performance counter.");
}
else
{
    Console.WriteLine("Operations timed using the DateTime class.");
}

long frequency = Stopwatch.Frequency;
Console.WriteLine("  Timer frequency in ticks per second = {0}",
    frequency);
long nanosecPerTick = (1000L*1000L*1000L) / frequency;
Console.WriteLine("  Timer is accurate within {0} nanoseconds",
    nanosecPerTick);
}
```

https://dotnetfiddle.net/ckrWUo

The timer used by the Stopwatch class depends on the system hardware and operating system. IsHighResolution is true if the Stopwatch timer is based on a high-resolution performance counter. Otherwise, IsHighResolution is false, which indicates that the Stopwatch timer is based on the system timer.

Ticks in Stopwatch are machine/OS dependent, thus you should never count on the ration of Stopwatch ticks to seconds to be the same between two systems, and possibly even on the same system after a reboot. Thus, you can never count on Stopwatch ticks to be the same interval as DateTime/TimeSpan ticks.

To get system-independent time, make sure to use the Stopwatch's Elapsed or ElapsedMilliseconds properties, which already take the Stopwatch.Frequency (ticks per second) into account.

Stopwatch should always be used over Datetime for timing processes as it is more lightweight and uses Dateime if it cant use a high-resolution performance counter.

Source

Section 111.2: Creating an Instance of a Stopwatch

A Stopwatch instance can measure elapsed time over several intervals with the total elapsed time being all individual intervals added together. This gives a reliable method of measuring elapsed time between two or more events.

```
Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();

double d = 0;
for (int i = 0; i < 1000 * 1000 * 1000; i++)
{
```

```
    d += 1;
}

stopWatch.Stop();
Console.WriteLine("Time elapsed: {0:hh\\:mm\\:ss\\.fffffff}", stopWatch.Elapsed);
```

`Stopwach` is in `System.Diagnostics` so you need to add `using System.Diagnostics;` to your file.

Chapter 112: Threading

Section 112.1: Avoiding Reading and Writing Data Simultaneously

Sometimes, you want your threads to simultaneously share data. When this happens it is important to be aware of the code and lock any parts that could go wrong. A simple example of two threads counting is shown below.

Here is some dangerous (incorrect) code:

```
using System.Threading;

class MainClass
{
    static int count { get; set; }

    static void Main()
    {
        for (int i = 1; i <= 2; i++)
        {
            var thread = new Thread(ThreadMethod);
            thread.Start(i);
            Thread.Sleep(500);
        }
    }

    static void ThreadMethod(object threadNumber)
    {
        while (true)
        {
            var temp = count;
            System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of count.");
            Thread.Sleep(1000);
            count = temp + 1;
            System.Console.WriteLine("Thread " + threadNumber + ": Incrementing the value of count
to:" + count);
            Thread.Sleep(1000);
        }
    }
}
```

You'll notice, instead of counting 1,2,3,4,5... we count 1,1,2,2,3...

To fix this problem, we need to **lock** the value of count, so that multiple different threads cannot read and write to it at the same time. With the addition of a lock and a key, we can prevent the threads from accessing the data simultaneously.

```
using System.Threading;

class MainClass
{

    static int count { get; set; }
    static readonly object key = new object();

    static void Main()
    {
        for (int i = 1; i <= 2; i++)
```

```
        {
            var thread = new Thread(ThreadMethod);
            thread.Start(i);
            Thread.Sleep(500);
        }
    }

    static void ThreadMethod(object threadNumber)
    {
        while (true)
        {
            lock (key)
            {
                var temp = count;
                System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of
count.");
                Thread.Sleep(1000);
                count = temp + 1;
                System.Console.WriteLine("Thread " + threadNumber + ": Incrementing the value of
count to:" + count);
            }
            Thread.Sleep(1000);
        }
    }
}
```

Section 112.2: Creating and Starting a Second Thread

If you're doing multiple long calculations, you can run them at the same time on different threads on your computer. To do this, we make a new **Thread** and have it point to a different method.

```
using System.Threading;

class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start();
    }

    static void Secondary() {
        System.Console.WriteLine("Hello World!");
    }
}
```

Section 112.3: Parallel.ForEach Loop

If you have a foreach loop that you want to speed up and you don't mind what order the output is in, you can convert it to a parallel foreach loop by doing the following:

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class MainClass {

    public static void Main() {
        int[] Numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        // Single-threaded
        Console.WriteLine("Normal foreach loop: ");
```

```
        foreach (var number in Numbers) {
            Console.WriteLine(longCalculation(number));
        }
        // This is the Parallel (Multi-threaded solution)
        Console.WriteLine("Parallel foreach loop: ");
        Parallel.ForEach(Numbers, number => {
            Console.WriteLine(longCalculation(number));
        });
    }

    private static int longCalculation(int number) {
        Thread.Sleep(1000); // Sleep to simulate a long calculation
        return number * number;
    }
}
```

Section 112.4: Deadlocks (hold resource and wait)

A deadlock is what occurs when two or more threads are waiting for each other to complete or to release a resource in such a way that they wait forever.

If thread1 holds a lock on resource A and is waiting for resource B to be released while thread2 holds resource B and is waiting for resource A to be released, they are deadlocked.

Clicking button1 for the following example code will cause your application to get into aforementioned deadlocked state and hang

```
private void button_Click(object sender, EventArgs e)
{
    DeadlockWorkers workers = new DeadlockWorkers();
    workers.StartThreads();
    textBox.Text = workers.GetResult();
}

private class DeadlockWorkers
{
    Thread thread1, thread2;

    object resourceA = new object();
    object resourceB = new object();

    string output;

    public void StartThreads()
    {
        thread1 = new Thread(Thread1DoWork);
        thread2 = new Thread(Thread2DoWork);
        thread1.Start();
        thread2.Start();
    }

    public string GetResult()
    {
        thread1.Join();
        thread2.Join();
        return output;
    }

    public void Thread1DoWork()
    {
```

```
        Thread.Sleep(100);
        lock (resourceA)
        {
            Thread.Sleep(100);
            lock (resourceB)
            {
                output += "T1#";
            }
        }
    }

    public void Thread2DoWork()
    {
        Thread.Sleep(100);
        lock (resourceB)
        {
            Thread.Sleep(100);
            lock (resourceA)
            {
                output += "T2#";
            }
        }
    }
}
```

To avoid being deadlocked this way, one can use Monitor.TryEnter(lock_object, timeout_in_milliseconds) to check if a lock is held on an object already. If Monitor.TryEnter does not succeed in acquiring a lock on lock_object before timeout_in_milliseconds, it returns false, giving the thread a chance to release other held resources and yielding, thus giving other threads a chance to complete as in this slightly modified version of the above:

```
private void button_Click(object sender, EventArgs e)
{
    MonitorWorkers workers = new MonitorWorkers();
    workers.StartThreads();
    textBox.Text = workers.GetResult();
}

private class MonitorWorkers
{
    Thread thread1, thread2;

    object resourceA = new object();
    object resourceB = new object();

    string output;

    public void StartThreads()
    {
        thread1 = new Thread(Thread1DoWork);
        thread2 = new Thread(Thread2DoWork);
        thread1.Start();
        thread2.Start();
    }

    public string GetResult()
    {
        thread1.Join();
        thread2.Join();
        return output;
    }
```

```
    public void Thread1DoWork()
    {
        bool mustDoWork = true;
        Thread.Sleep(100);
        while (mustDoWork)
        {
            lock (resourceA)
            {
                Thread.Sleep(100);
                if (Monitor.TryEnter(resourceB, 0))
                {
                    output += "T1#";
                    mustDoWork = false;
                    Monitor.Exit(resourceB);
                }
            }
            if (mustDoWork) Thread.Yield();
        }
    }

    public void Thread2DoWork()
    {
        Thread.Sleep(100);
        lock (resourceB)
        {
            Thread.Sleep(100);
            lock (resourceA)
            {
                output += "T2#";
            }
        }
    }
}
```

Note that this workaround relies on thread2 being stubborn about its locks and thread1 being willing to yield, such that thread2 always take precedence. Also note that thread1 has to redo the work it did after locking resource A, when it yields. Therefore be careful when implementing this approach with more than one yielding thread, as you'll then run the risk of entering a so-called livelock - a state which would occur if two threads kept doing the first bit of their work and then yield mutually, starting over repeatedly.

Section 112.5: Simple Complete Threading Demo

```
class Program
{
    static void Main(string[] args)
    {
        // Create 2 thread objects.  We're using delegates because we need to pass
        // parameters to the threads.
        var thread1 = new Thread(new ThreadStart(() => PerformAction(1)));
        var thread2 = new Thread(new ThreadStart(() => PerformAction(2)));

        // Start the threads running
        thread1.Start();
        // NB: as soon as the above line kicks off the thread, the next line starts;
        // even if thread1 is still processing.
        thread2.Start();

        // Wait for thread1 to complete before continuing
        thread1.Join();
        // Wait for thread2 to complete before continuing
```

```
        thread2.Join();

        Console.WriteLine("Done");
        Console.ReadKey();
    }

    // Simple method to help demonstrate the threads running in parallel.
    static void PerformAction(int id)
    {
        var rnd = new Random(id);
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine("Thread: {0}: {1}", id, i);
            Thread.Sleep(rnd.Next(0, 1000));
        }
    }
}
```

Section 112.6: Creating One Thread Per Processor

> `Environment.ProcessorCount` Gets the number of **logical** processors on the current machine.

The CLR will then schedule each thread to a logical processor, this theoretically could mean each thread on a different logical processor, all threads on a single logical processor or some other combination.

```
using System;
using System.Threading;

class MainClass {
    static void Main() {
        for (int i = 0; i < Environment.ProcessorCount; i++) {
            var thread = new Thread(Secondary);
            thread.Start(i);
        }

    }

    static void Secondary(object threadNumber) {
        System.Console.WriteLine("Hello World from thread: " + threadNumber);
    }
}
```

Section 112.7: Simple Complete Threading Demo using Tasks

```
class Program
{
    static void Main(string[] args)
    {
        // Run 2 Tasks.
        var task1 = Task.Run(() => PerformAction(1));
        var task2 = Task.Run(() => PerformAction(2));

        // Wait (i.e. block this thread) until both Tasks are complete.
        Task.WaitAll(new [] { task1, task2 });

        Console.WriteLine("Done");
        Console.ReadKey();
    }
```

```
    // Simple method to help demonstrate the threads running in parallel.
    static void PerformAction(int id)
    {
        var rnd = new Random(id);
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine("Task: {0}: {1}", id, i);
            Thread.Sleep(rnd.Next(0, 1000));
        }
    }
}
```

Section 112.8: Deadlocks (two threads waiting on each other)

A deadlock is what occurs when two or more threads are waiting for each other to complete or to release a resource in such a way that they wait forever.

A typical scenario of two threads waiting on each other to complete is when a Windows Forms GUI thread waits for a worker thread and the worker thread attempts to invoke an object managed by the GUI thread. Observe that with this code example, clicking button1 will cause the program to hang.

```
private void button1_Click(object sender, EventArgs e)
{
    Thread workerthread= new Thread(dowork);
    workerthread.Start();
    workerthread.Join();
    // Do something after
}

private void dowork()
{
    // Do something before
    textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
    // Do something after
}
```

`workerthread.Join()` is a call that blocks the calling thread until workerthread completes.
`textBox1.Invoke(invoke_delegate)` is a call that blocks the calling thread until the GUI thread has processed invoke_delegate, but this call causes deadlocks if the GUI thread is already waiting for the calling thread to complete.

To get around this, one can use a non-blocking way of invoking the textbox instead:

```
private void dowork()
{
    // Do work
    textBox1.BeginInvoke(new Action(() => textBox1.Text = "Some Text"));
    // Do work that is not dependent on textBox1 being updated first
}
```

However, this will cause trouble if you need to run code that is dependent on the textbox being updated first. In that case, run that as part of the invoke, but be aware that this will make it run on the GUI thread.

```
private void dowork()
{
    // Do work
    textBox1.BeginInvoke(new Action(() => {
        textBox1.Text = "Some Text";
```

```
        // Do work dependent on textBox1 being updated first,
        // start another worker thread or raise an event
    }));
    // Do work that is not dependent on textBox1 being updated first
}
```

Alternatively start af whole new thread and let that one do the waiting on the GUI thread, so that workerthread might complete.

```
private void dowork()
{
    // Do work
    Thread workerthread2 = new Thread(() =>
    {
        textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
        // Do work dependent on textBox1 being updated first,
        // start another worker thread or raise an event
    });
    workerthread2.Start();
    // Do work that is not dependent on textBox1 being updated first
}
```

To minimize the risk of running into a deadlock of mutual waiting, always avoid circular references between threads when possible. A hierarchy of threads where lower-ranking threads only leave messages for higher-ranking threads and never waiting on them will not run into this kind of issue. However, it would still be vulnerable to deadlocks based on resource locking.

Section 112.9: Explicit Task Parallism

```
    private static void explicitTaskParallism()
    {
        Thread.CurrentThread.Name = "Main";

        // Create a task and supply a user delegate by using a lambda expression.
        Task taskA = new Task(() => Console.WriteLine($"Hello from task {nameof(taskA)}."));
        Task taskB = new Task(() => Console.WriteLine($"Hello from task {nameof(taskB)}."));

        // Start the task.
        taskA.Start();
        taskB.Start();

        // Output a message from the calling thread.
        Console.WriteLine("Hello from thread '{0}'.",
                          Thread.CurrentThread.Name);
        taskA.Wait();
        taskB.Wait();
        Console.Read();
    }
```

Section 112.10: Implicit Task Parallelism

```
    private static void Main(string[] args)
    {
        var a = new A();
        var b = new B();
        //implicit task parallelism
        Parallel.Invoke(
            () => a.DoSomeWork(),
```

```
            () => b.DoSomeOtherWork()
            );

    }
```

Section 112.11: Starting a thread with parameters

using System.Threading;

```
class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start("SecondThread");
    }

    static void Secondary(object threadName) {
        System.Console.WriteLine("Hello World from thread: " + threadName);
    }
}
```

Chapter 113: Async/await, Backgroundworker, Task and Thread Examples

Section 113.1: ASP.NET Configure Await

When ASP.NET handles a request, a thread is assigned from the thread pool and a **request context** is created. The request context contains information about the current request which can be accessed through the static `HttpContext.Current` property. The request context for the request is then assigned to the thread handling the request.

A given request context **may only be active on one thread at a time**.

When execution reaches `await`, the thread handling a request is returned to the thread pool while the asynchronous method runs and the request context is free for another thread to use.

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    var products = await dbContext.Products.ToListAsync();

    // Execution resumes on a "random" thread from the pool
    // Execution continues using the original request context.
    return View(products);
}
```

When the task completes the thread pool assigns another thread to continue execution of the request. The request context is then assigned to this thread. This may or may not be the original thread.

Blocking

When the result of an `async` method call is waited for **synchronously** deadlocks can arise. For example the following code will result in a deadlock when `IndexSync()` is called:

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    List<Product> products = await dbContext.Products.ToListAsync();

    // Execution resumes on a "random" thread from the pool
    return View(products);
}

public ActionResult IndexSync()
{
    Task<ActionResult> task = Index();

    // Block waiting for the result synchronously
    ActionResult result = Task.Result;

    return result;
}
```

This is because, by default the awaited task, in this case `db.Products.ToListAsync()` will capture the context (in the case of ASP.NET the request context) and try to use it once it has completed.

When the entire call stack is asynchronous there is no problem because, once `await` is reached the original thread is release, freeing the request context.

When we block synchronously using `Task.Result` or `Task.Wait()` (or other blocking methods) the original thread is still active and retains the request context. The awaited method still operates asynchronously and once the callback tries to run, i.e. once the awaited task has returned, it attempts to obtain the request context.

Therefore the deadlock arises because while the blocking thread with the request context is waiting for the asynchronous operation to complete, the asynchronous operation is trying to obtain the request context in order to complete.

ConfigureAwait

By default calls to an awaited task will capture the current context and attempt to resume execution on the context once complete.

By using `ConfigureAwait(false)` this can be prevented and deadlocks can be avoided.

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    List<Product> products = await dbContext.Products.ToListAsync().ConfigureAwait(false);

    // Execution resumes on a "random" thread from the pool without the original request context
    return View(products);
}

public ActionResult IndexSync()
{
    Task<ActionResult> task = Index();

    // Block waiting for the result synchronously
    ActionResult result = Task.Result;

    return result;
}
```

This can avoid deadlocks when it is necessary to block on asynchronous code, however this comes at the cost of losing the context in the continuation (code after the call to await).

In ASP.NET this means that if your code following a call to `await someTask.ConfigureAwait(false);` attempts to access information from the context, for example `HttpContext.Current.User` then the information has been lost. In this case the `HttpContext.Current` is null. For example:

```
public async Task<ActionResult> Index()
{
    // Contains information about the user sending the request
    var user = System.Web.HttpContext.Current.User;

    using (var client = new HttpClient())
    {
        await client.GetAsync("http://google.com").ConfigureAwait(false);
    }

    // Null Reference Exception, Current is null
    var user2 = System.Web.HttpContext.Current.User;

    return View();
```

```
}
```

If `ConfigureAwait(true)` is used (equivalent to having no ConfigureAwait at all) then both `user` and `user2` are populated with the same data.

For this reason it is often recommended to use `ConfigureAwait(false)` in library code where the context is no longer used.

Section 113.2: Task "run and forget" extension

In certain cases (e.g. logging) it might be useful to run task and do not await for the result. The following extension allows to run task and continue execution of the rest code:

```
public static class TaskExtensions
{
    public static async void RunAndForget(
        this Task task, Action<Exception> onException = null)
    {
        try
        {
            await task;
        }
        catch (Exception ex)
        {
            onException?.Invoke(ex);
        }
    }
}
```

The result is awaited only inside the extension method. Since `async`/`await` is used, it is possible to catch an exception and call an optional method for handling it.

An example how to use the extension:

```
var task = Task.FromResult(0); // Or any other task from e.g. external lib.
task.RunAndForget(
    e =>
    {
        // Something went wrong, handle it.
    });
```

Section 113.3: Async/await

See below for a simple example of how to use async/await to do some time intensive stuff in a background process while maintaining the option of doing some other stuff that do not need to wait on the time intensive stuff to complete.

However, if you need to work with the result of the time intensive method later, you can do this by awaiting the execution.

```
public async Task ProcessDataAsync()
{
    // Start the time intensive method
    Task<int> task = TimeintensiveMethod(@"PATH_TO_SOME_FILE");

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");
```

```
    // Wait for TimeintensiveMethod to complete and get its result
    int x = await task;
    Console.WriteLine("Count: " + x);
}

private async Task<int> TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = await reader.ReadToEndAsync();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}
```

Section 113.4: BackgroundWorker

See below for a simple example of how to use a `BackgroundWorker` object to perform time-intensive operations in a background thread.

You need to:

1. Define a worker method that does the time-intensive work and call it from an event handler for the `DoWork` event of a `BackgroundWorker`.
2. Start the execution with `RunWorkerAsync`. Any argument required by the worker method attached to `DoWork` can be passed in via the `DoWorkEventArgs` parameter to `RunWorkerAsync`.

In addition to the `DoWork` event the `BackgroundWorker` class also defines two events that should be used for interacting with the user interface. These are optional.

- The `RunWorkerCompleted` event is triggered when the `DoWork` handlers have completed.
- The `ProgressChanged` event is triggered when the `ReportProgress` method is called.

```
public void ProcessDataAsync()
{
    // Start the time intensive method
    BackgroundWorker bw = new BackgroundWorker();
    bw.DoWork += BwDoWork;
    bw.RunWorkerCompleted += BwRunWorkerCompleted;
    bw.RunWorkerAsync(@"PATH_TO_SOME_FILE");

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");
}

// Method that will be called after BwDoWork exits
private void BwRunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    // we can access possible return values of our Method via the Parameter e
    Console.WriteLine("Count: " + e.Result);
}
```

```
// execution of our time intensive Method
private void BwDoWork(object sender, DoWorkEventArgs e)
{
    e.Result = TimeintensiveMethod(e.Argument);
}

private int TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = reader.ReadToEnd();

       for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}
```

Section 113.5: Task

See below for a simple example of how to use a `Task` to do some time intensive stuff in a background process.

All you need to do is wrap your time intensive method in a `Task.Run()` call.

```
public void ProcessDataAsync()
{
    // Start the time intensive method
    Task<int> t = Task.Run(() => TimeintensiveMethod(@"PATH_TO_SOME_FILE"));

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");

    Console.WriteLine("Count: " + t.Result);
}

private int TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}
```

Section 113.6: Thread

See below for a simple example of how to use a `Thread` to do some time intensive stuff in a background process.

```
public async void ProcessDataAsync()
{
    // Start the time intensive method
    Thread t = new Thread(TimeintensiveMethod);

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");
}

private void TimeintensiveMethod()
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(@"PATH_TO_SOME_FILE"))
    {
        string v = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            v.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");
}
```

As you can see we can not return a value from our `TimeIntensiveMethod` because `Thread` expects a void Method as its parameter.

To get a return value from a `Thread` use either an event or the following:

```
int ret;
Thread t= new Thread(() =>
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file))
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something to demonstrate the coolness of await-async
    ret = new Random().Next(100);
});

t.Start();
t.Join(1000);
Console.Writeline("Count: " + ret);
```

Chapter 114: Async-Await

In C#, a method declared `async` won't block within a synchronous process, in case of you're using I/O based operations (e.g. web access, working with files, ...). The result of such async marked methods may be awaited via the use of the `await`keyword.

Section 114.1: Await operator and async keyword

`await` operator and `async` keyword come together:

> The asynchronous method in which **await** is used must be modified by the **async** keyword.

The opposite is not always true: you can mark a method as `async` without using `await` in its body.

What `await` actually does is to suspend execution of the code until the awaited task completes; any task can be awaited.

Note: you cannot await for async method which returns nothing (void).

Actually, the word 'suspends' is a bit misleading because not only the execution stops, but the thread may become free for executing other operations. Under the hood, `await` is implemented by a bit of compiler magic: it splits a method into two parts - before and after `await`. The latter part is executed when the awaited task completes.

If we ignore some important details, the compiler roughly does this for you:

```
public async Task<TResult> DoIt()
{
    // do something and acquire someTask of type Task<TSomeResult>
    var awaitedResult = await someTask;
    // ... do something more and produce result of type TResult
    return result;
}
```

becomes:

```
public Task<TResult> DoIt()
{
    // ...
    return someTask.ContinueWith(task => {
        var result = ((Task<TSomeResult>)task).Result;
        return DoIt_Continuation(result);
    });
}

private TResult DoIt_Continuation(TSomeResult awaitedResult)
{
    // ...
}
```

Any usual method can be turned into async in the following way:

```
await Task.Run(() => YourSyncMethod());
```

This can be advantageous when you need to execute a long running method on the UI thread without freezing the

UI.

But there is a very important remark here: **Asynchronous does not always mean concurrent (parallel or even multi-threaded).** Even on a single thread, `async-await` still allows for asynchronous code. For example, see this custom task scheduler. Such a 'crazy' task scheduler can simply turn tasks into functions which are called within message loop processing.

We need to ask ourselves: What thread will execute the continuation of our method `DoIt_Continuation`?

By default the `await` operator schedules the execution of continuation with the current Synchronization context. It means that by default for WinForms and WPF continuation runs in the UI thread. If, for some reason, you need to change this behavior, use method `Task.ConfigureAwait()`:

```
await Task.Run(() => YourSyncMethod()).ConfigureAwait(continueOnCapturedContext: false);
```

Section 114.2: Concurrent calls

It is possible to await multiple calls concurrently by first invoking the awaitable tasks and *then* awaiting them.

```
public async Task RunConcurrentTasks()
{
    var firstTask = DoSomethingAsync();
    var secondTask = DoSomethingElseAsync();

    await firstTask;
    await secondTask;
}
```

Alternatively, `Task.WhenAll` can be used to group multiple tasks into a single `Task`, which completes when all of its passed tasks are complete.

```
public async Task RunConcurrentTasks()
{
    var firstTask = DoSomethingAsync();
    var secondTask = DoSomethingElseAsync();

    await Task.WhenAll(firstTask, secondTask);
}
```

You can also do this inside a loop, for example:

```
List<Task> tasks = new List<Task>();
while (something) {
    // do stuff
    Task someAsyncTask = someAsyncMethod();
    tasks.Add(someAsyncTask);
}

await Task.WhenAll(tasks);
```

To get results from a task after awaiting multiple tasks with Task.WhenAll, simply await the task again. Since the task is already completed it will just return the result back

```
var task1 = SomeOpAsync();
var task2 = SomeOtherOpAsync();
```

```
await Task.WhenAll(task1, task2);

var result = await task2;
```

Also, the `Task.WhenAny` can be used to execute multiple tasks in parallel, like the `Task.WhenAll` above, with the difference that this method will complete when *any* of the supplied tasks will be completed.

```
public async Task RunConcurrentTasksWhenAny()
{
    var firstTask = TaskOperation("#firstTask executed");
    var secondTask = TaskOperation("#secondTask executed");
    var thirdTask = TaskOperation("#thirdTask executed");
    await Task.WhenAny(firstTask, secondTask, thirdTask);
}
```

The `Task` returned by `RunConcurrentTasksWhenAny` will complete when any of `firstTask`, `secondTask`, or `thirdTask` completes.

Section 114.3: Try/Catch/Finally

`Version ≥ 6.0`

As of C# 6.0, the `await` keyword can now be used within a `catch` and `finally` block.

```
try {
   var client = new AsyncClient();
   await client.DoSomething();
} catch (MyException ex) {
   await client.LogExceptionAsync();
   throw;
} finally {
   await client.CloseAsync();
}
```

`Version ≥ 5.0` `Version < 6.0`

Prior to C# 6.0, you would need to do something along the lines of the following. Note that 6.0 also cleaned up the null checks with the Null Propagating operator.

```
AsynClient client;
MyException caughtException;
try {
     client = new AsyncClient();
     await client.DoSomething();
} catch (MyException ex) {
     caughtException = ex;
}

if (client != null) {
    if (caughtException != null) {
       await client.LogExceptionAsync();
    }
    await client.CloseAsync();
    if (caughtException != null) throw caughtException;
}
```

Please note that if you await a task not created by `async` (e.g. a task created by `Task.Run`), some debuggers may break on exceptions thrown by the task even when it is seemingly handled by the surrounding try/catch. This happens because the debugger considers it to be unhandled with respect to user code. In Visual Studio, there is an

option called "Just My Code", which can be disabled to prevent the debugger from breaking in such situations.

Section 114.4: Returning a Task without await

Methods that perform asynchronous operations don't need to use `await` if:

- There is only one asynchronous call inside the method
- The asynchronous call is at the end of the method
- Catching/handling exception that may happen within the Task is not necessary

Consider this method that returns a `Task`:

```
public async Task<User> GetUserAsync(int id)
{
    var lookupKey = "Users" + id;

    return await dataStore.GetByKeyAsync(lookupKey);
}
```

If `GetByKeyAsync` has the same signature as `GetUserAsync` (returning a `Task<User>`), the method can be simplified:

```
public Task<User> GetUserAsync(int id)
{
    var lookupKey = "Users" + id;

    return dataStore.GetByKeyAsync(lookupKey);
}
```

In this case, the method doesn't need to be marked `async`, even though it's preforming an asynchronous operation. The Task returned by `GetByKeyAsync` is passed directly to the calling method, where it will be `await`ed.

Important: Returning the `Task` instead of awaiting it, changes the exception behavior of the method, as it won't throw the exception inside the method which starts the task but in the method which awaits it.

```
public Task SaveAsync()
{
    try {
        return dataStore.SaveChangesAsync();
    }
    catch(Exception ex)
    {
        // this will never be called
        logger.LogException(ex);
    }
}

// Some other code calling SaveAsync()

// If exception happens, it will be thrown here, not inside SaveAsync()
await SaveAsync();
```

This will improve performance as it will save the compiler the generation of an extra **async** state machine.

Section 114.5: Async/await will only improve performance if it allows the machine to do additional work

Consider the following code:

```
public async Task MethodA()
{
     await MethodB();
     // Do other work
}

public async Task MethodB()
{
     await MethodC();
     // Do other work
}

public async Task MethodC()
{
     // Or await some other async work
     await Task.Delay(100);
}
```

This will not perform any better than

```
public void MethodA()
{
     MethodB();
     // Do other work
}

public void MethodB()
{
     MethodC();
     // Do other work
}

public void MethodC()
{
     Thread.Sleep(100);
}
```

The primary purpose of async/await is to allow the machine to do additional work - for example, to allow the calling thread to do other work while it's waiting for a result from some I/O operation. In this case, the calling thread is never allowed to do more work than it would have been able to do otherwise, so there's no performance gain over simply calling `MethodA()`, `MethodB()`, and `MethodC()` synchronously.

Section 114.6: Web.config setup to target 4.5 for correct async behaviour

The web.config system.web.httpRuntime must target 4.5 to ensure the thread will renter the request context before resuming your async method.

```
<httpRuntime targetFramework="4.5" />
```

Async and await have undefined behavior on ASP.NET prior to 4.5. Async / await will resume on an arbitrary thread that may not have the request context. Applications under load will randomly fail with null reference exceptions accessing the HttpContext after the await. Using HttpContext.Current in WebApi is dangerous because of async

Section 114.7: Simple consecutive calls

```
public async Task<JobResult> GetDataFromWebAsync()
```

```
{
  var nextJob = await _database.GetNextJobAsync();
  var response = await _httpClient.GetAsync(nextJob.Uri);
  var pageContents = await response.Content.ReadAsStringAsync();
  return await _database.SaveJobResultAsync(pageContents);
}
```

The main thing to note here is that while every `await`-ed method is called asynchronously - and for the time of that call the control is yielded back to the system - the flow inside the method is linear and does not require any special treatment due to asynchrony. If any of the methods called fail, the exception will be processed "as expected", which in this case means that the method execution will be aborted and the exception will be going up the stack.

Section 114.8: Blocking on async code can cause deadlocks

It is a bad practice to block on async calls as it can cause deadlocks in environments that have a synchronization context. The best practice is to use async/await "all the way down." For example, the following Windows Forms code causes a deadlock:

```
private async Task<bool> TryThis()
{
    Trace.TraceInformation("Starting TryThis");
    await Task.Run(() =>
    {
        Trace.TraceInformation("In TryThis task");
        for (int i = 0; i < 100; i++)
        {
            // This runs successfully - the loop runs to completion
            Trace.TraceInformation("For loop " + i);
            System.Threading.Thread.Sleep(10);
        }
    });

    // This never happens due to the deadlock
    Trace.TraceInformation("About to return");
    return true;
}

// Button click event handler
private void button1_Click(object sender, EventArgs e)
{
    // .Result causes this to block on the asynchronous call
    bool result = TryThis().Result;
    // Never actually gets here
    Trace.TraceInformation("Done with result");
}
```

Essentially, once the async call completes, it waits for the synchronization context to become available. However, the event handler "holds on" to the synchronization context while it's waiting for the `TryThis()` method to complete, thus causing a circular wait.

To fix this, code should be modified to

```
private async void button1_Click(object sender, EventArgs e)
{
  bool result = await TryThis();
  Trace.TraceInformation("Done with result");
}
```

Note: event handlers are the only place where `async void` should be used (because you can't await an `async void` method).

Chapter 115: Synchronization Context in Async-Await

Section 115.1: Pseudocode for async/await keywords

Consider a simple asynchronous method:

```
async Task Foo()
{
    Bar();
    await Baz();
    Qux();
}
```

Simplifying, we can say that this code actually means the following:

```
Task Foo()
{
    Bar();
    Task t = Baz();
    var context = SynchronizationContext.Current;
    t.ContinueWith(task) =>
    {
        if (context == null)
            Qux();
        else
            context.Post((obj) => Qux(), null);
    }, TaskScheduler.Current);

    return t;
}
```

It means that `async`/`await` keywords use current synchronization context if it exists. I.e. you can write library code that would work correctly in UI, Web, and Console applications.

Source article.

Section 115.2: Disabling synchronization context

To disable synchronization context you should call the `ConfigureAwait` method:

```
async Task() Foo()
{
    await Task.Run(() => Console.WriteLine("Test"));
}

. . .

Foo().ConfigureAwait(false);
```

> ConfigureAwait provides a means to avoid the default SynchronizationContext capturing behavior; passing false for the flowContext parameter prevents the SynchronizationContext from being used to resume execution after the await.

Quote from It's All About the SynchronizationContext.

Section 115.3: Why SynchronizationContext is so important?

Consider this example:

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = RunTooLong();
}
```

This method will freeze UI application until the `RunTooLong` will be completed. The application will be unresponsive.

You can try run inner code asynchronously:

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() => label1.Text = RunTooLong());
}
```

But this code won't execute because inner body may be run on non-UI thread and it shouldn't change UI properties directly:

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
        var label1Text = RunTooLong();

        if (label1.InvokeRequired)
            lable1.BeginInvoke((Action) delegate() { label1.Text = label1Text; });
        else
            label1.Text = label1Text;
    });
}
```

Now don't forget always to use this pattern. Or, try `SynchronizationContext.Post` that will make it for you:

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
        var label1Text = RunTooLong();
        SynchronizationContext.Current.Post((obj) =>
        {
            label1.Text = label1    Text);
        }, null);
    });
}
```

Chapter 116: BackgroundWorker

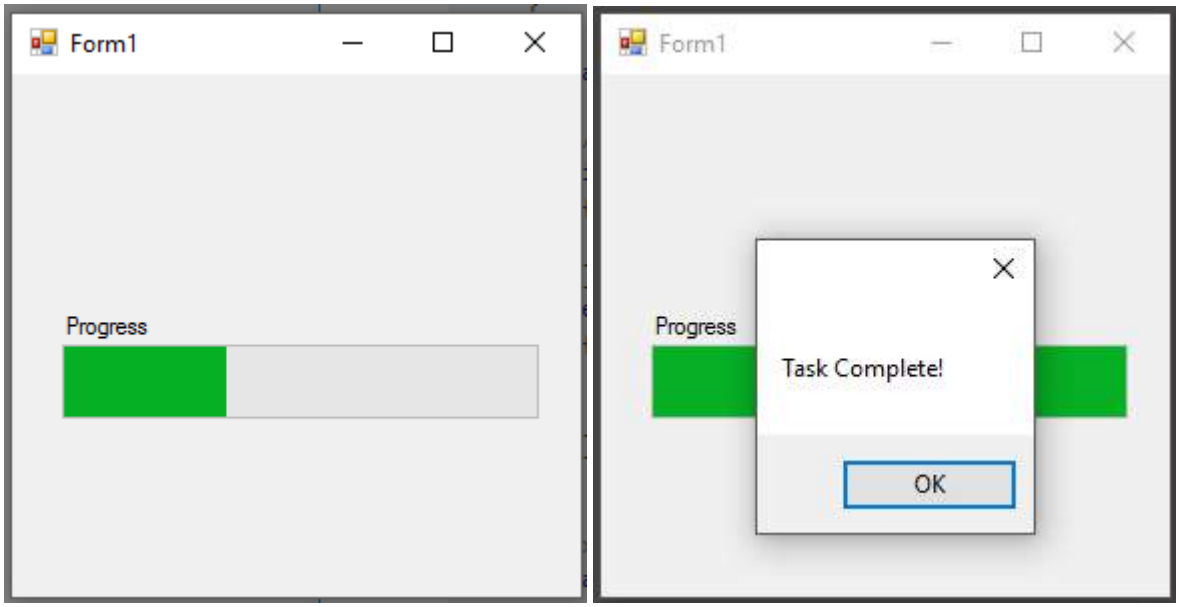
Section 116.1: Using a BackgroundWorker to complete a task

The following example demonstrates the use of a BackgroundWorker to update a WinForms ProgressBar. The backgroundWorker will update the value of the progress bar without blocking the UI thread, thus showing a reactive UI while work is done in the background.

```
namespace BgWorkerExample
{
    public partial class Form1 : Form
{

    //a new instance of a backgroundWorker is created.
    BackgroundWorker bgWorker = new BackgroundWorker();

    public Form1()
    {
        InitializeComponent();

        prgProgressBar.Step = 1;

        //this assigns event handlers for the backgroundWorker
        bgWorker.DoWork += bgWorker_DoWork;
        bgWorker.RunWorkerCompleted += bgWorker_WorkComplete;

        //tell the backgroundWorker to raise the "DoWork" event, thus starting it.
        //Check to make sure the background worker is not already running.
        if(!bgWorker.IsBusy)
            bgWorker.RunWorkerAsync();

    }

    private void bgWorker_DoWork(object sender, DoWorkEventArgs e)
    {
        //this is the method that the backgroundworker will perform on in the background thread.
        /* One thing to note! A try catch is not necessary as any exceptions will terminate the
backgroundWorker and report
          the error to the "RunWorkerCompleted" event */
        CountToY();
    }

    private void bgWorker_WorkComplete(object sender, RunWorkerCompletedEventArgs e)
    {
        //e.Error will contain any exceptions caught by the backgroundWorker
        if (e.Error != null)
        {
            MessageBox.Show(e.Error.Message);
        }
        else
        {
            MessageBox.Show("Task Complete!");
            prgProgressBar.Value = 0;
        }
    }

    // example method to perform a "long" running task.
    private void CountToY()
    {
        int x = 0;
```

```
        int maxProgress = 100;
        prgProgressBar.Maximum = maxProgress;


        while (x < maxProgress)
        {
            System.Threading.Thread.Sleep(50);
            Invoke(new Action(() => { prgProgressBar.PerformStep(); }));
            x += 1;
        }
    }


}
```

The result is the following...



Section 116.2: Assigning Event Handlers to a BackgroundWorker

Once the instance of the BackgroundWorker has been declared, it must be given properties and event handlers for the tasks it performs.

```
    /* This is the backgroundworker's "DoWork" event handler. This
       method is what will contain all the work you
       wish to have your program perform without blocking the UI. */

    bgWorker.DoWork += bgWorker_DoWork;


    /*This is how the DoWork event method signature looks like:*/
    private void bgWorker_DoWork(object sender, DoWorkEventArgs e)
    {
        // Work to be done here
        // ...
        // To get a reference to the current Backgroundworker:
        BackgroundWorker worker = sender as BackgroundWorker;
        // The reference to the BackgroundWorker is often used to report progress
        worker.ReportProgress(...);
    }
```

```
    /*This is the method that will be run once the BackgroundWorker has completed its tasks */

    bgWorker.RunWorkerCompleted += bgWorker_CompletedWork;

    /*This is how the RunWorkerCompletedEvent event method signature looks like:*/
    private void bgWorker_CompletedWork(object sender, RunWorkerCompletedEventArgs e)
    {
        // Things to be done after the backgroundworker has finished
    }

   /* When you wish to have something occur when a change in progress
     occurs, (like the completion of a specific task) the "ProgressChanged"
     event handler is used. Note that ProgressChanged events may be invoked
     by calls to bgWorker.ReportProgress(...) only if bgWorker.WorkerReportsProgress
     is set to true.  */

     bgWorker.ProgressChanged += bgWorker_ProgressChanged;

    /*This is how the ProgressChanged event method signature looks like:*/
    private void bgWorker_ProgressChanged(object sender, ProgressChangedEventArgs e)
    {
        // Things to be done when a progress change has been reported

        /* The ProgressChangedEventArgs gives access to a percentage,
         allowing for easy reporting of how far along a process is*/
        int progress = e.ProgressPercentage;
    }
```

Section 116.3: Creating a new BackgroundWorker instance

A BackgroundWorker is commonly used to perform tasks, sometimes time consuming, without blocking the UI thread.

```
// BackgroundWorker is part of the ComponentModel namespace.
using System.ComponentModel;

namespace BGWorkerExample
{
     public partial class ExampleForm : Form
     {

      // the following creates an instance of the BackgroundWorker named "bgWorker"
      BackgroundWorker bgWorker = new BackgroundWorker();

      public ExampleForm() { ...
```

Section 116.4: Assigning Properties to a BackgroundWorker

This allows the BackgroundWorker to be cancelled in between tasks

```
bgWorker.WorkerSupportsCancellation = true;
```

This allows the worker to report progress between completion of tasks...

```
bgWorker.WorkerReportsProgress = true;

//this must also be used in conjunction with the ProgressChanged event
```

Chapter 117: Task Parallel Library

Section 117.1: Parallel.ForEach

An example that uses Parallel.ForEach loop to ping a given array of website urls.

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.ForEach(urls, url =>
    {
        var ping = new System.Net.NetworkInformation.Ping();

        var result = ping.Send(url);

        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            Console.WriteLine(string.Format("{0} is online", url));
        }
    });
}
```

Section 117.2: Parallel.For

An example that uses Parallel.For loop to ping a given array of website urls.

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.For(0, urls.Length, i =>
    {
        var ping = new System.Net.NetworkInformation.Ping();

        var result = ping.Send(urls[i]);

        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            Console.WriteLine(string.Format("{0} is online", urls[i]));
        }
    });
}
```

Section 117.3: Parallel.Invoke

Invoking methods or actions in parallel (Parallel region)

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.Invoke(
        () => PingUrl(urls[0]),
        () => PingUrl(urls[1]),
        () => PingUrl(urls[2]),
        () => PingUrl(urls[3])
    );
}

void PingUrl(string url)
{
    var ping = new System.Net.NetworkInformation.Ping();

    var result = ping.Send(url);

    if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
    {
        Console.WriteLine(string.Format("{0} is online", url));
    }
}
```

Chapter 118: Making a variable thread safe

Section 118.1: Controlling access to a variable in a Parallel.For loop

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static void Main( string[] args )
    {
        object sync = new object();
        int sum = 0;
        Parallel.For( 1, 1000, ( i ) => {
            lock( sync ) sum = sum + i; // lock is necessary

            // As a practical matter, ensure this `parallel for` executes
            // on multiple threads by simulating a lengthy operation.
            Thread.Sleep( 1 );
        } );
        Console.WriteLine( "Correct answer should be 499500.  sum is: {0}", sum );
    }
}
```

It is not sufficient to just do `sum = sum + i` without the lock because the read-modify-write operation is not atomic. A thread will overwrite any external modifications to `sum` that occur after it has read the current value of `sum`, but before it stores the modified value of `sum + i` back into `sum`.

Chapter 119: Lock Statement

Section 119.1: Throwing exception in a lock statement

Following code will release the lock. There will be no problem. Behind the scenes lock statement works as `try finally`

```
lock(locker)
{
    throw new Exception();
}
```

More can be seen in the C# 5.0 Specification:

A `lock` statement of the form

```
lock (x) ...
```

where `x` is an expression of a *reference-type*, is precisely equivalent to

```
bool __lockWasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockWasTaken);
    ...
}
finally {
    if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

except that `x` is only evaluated once.

Section 119.2: Simple usage

Common usage of `lock` is a critical section.

In the following example `ReserveRoom` is supposed to be called from different threads. Synchronization with `lock` is the simplest way to prevent race condition here. Method body is surrounded with `lock` which ensures that two or more threads cannot execute it simultaneously.

```
public class Hotel
{
    private readonly object _roomLock = new object();

    public void ReserveRoom(int roomNumber)
    {
        // lock keyword ensures that only one thread executes critical section at once
        // in this case, reserves a hotel room of given number
        // preventing double bookings
        lock (_roomLock)
        {
            // reserve room logic goes here
        }
    }
}
```

If a thread reaches `lock`-ed block while another thread is running within it, the former will wait another to exit the

block.

> Best practice is to define a private object to lock on, or a private static object variable to protect data common to all instances.

Section 119.3: Return in a lock statement

Following code will release lock.

```
lock(locker)
{
    return 5;
}
```

For a detailed explanation, this SO answer is recommended.

Section 119.4: Anti-Patterns and gotchas

Locking on an stack-allocated / local variable

One of the fallacies while using `lock` is the usage of local objects as locker in a function. Since these local object instances will differ on each call of the function, `lock` will not perform as expected.

```
List<string> stringList = new List<string>();

public void AddToListNotThreadSafe(string something)
{
    // DO NOT do this, as each call to this method
    // will lock on a different instance of an Object.
    // This provides no thread safety, it only degrades performance.
    var localLock = new Object();
    lock(localLock)
    {
        stringList.Add(something);
    }
}

// Define object that can be used for thread safety in the AddToList method
readonly object classLock = new object();

public void AddToList(List<string> stringList, string something)
{
    // USE THE classLock instance field to achieve a
    // thread-safe lock before adding to stringList
    lock(classLock)
    {
        stringList.Add(something);
    }
}
```

Assuming that locking restricts access to the synchronizing object itself

If one thread calls: `lock(obj)` and another thread calls `obj.ToString()` second thread is not going to be blocked.

```
object obj = new Object();

public void SomeMethod()
```

```
{
     lock(obj)
    {
       //do dangerous stuff
    }
 }

 //Meanwhile on other tread
 public void SomeOtherMethod()
 {
   var objInString = obj.ToString(); //this does not block
 }
```

Expecting subclasses to know when to lock

Sometimes base classes are designed such that their subclasses are required to use a lock when accessing certain protected fields:

```
public abstract class Base
{
    protected readonly object padlock;
    protected readonly List<string> list;

    public Base()
    {
        this.padlock = new object();
        this.list = new List<string>();
    }

    public abstract void Do();
}

public class Derived1 : Base
{
    public override void Do()
    {
        lock (this.padlock)
        {
            this.list.Add("Derived1");
        }
    }
}

public class Derived2 : Base
{
    public override void Do()
    {
        this.list.Add("Derived2"); // OOPS! I forgot to lock!
    }
}
```

It is much safer to *encapsulate locking* by using a Template Method:

```
public abstract class Base
{
    private readonly object padlock; // This is now private
    protected readonly List<string> list;

    public Base()
    {
        this.padlock = new object();
```

```
        this.list = new List<string>();
    }

    public void Do()
    {
        lock (this.padlock) {
            this.DoInternal();
        }
    }

    protected abstract void DoInternal();
}

public class Derived1 : Base
{
    protected override void DoInternal()
    {
        this.list.Add("Derived1"); // Yay! No need to lock
    }
}
```

Locking on a boxed ValueType variable does not synchronize

In the following example, a private variable is implicitly boxed as it's supplied as an `object` argument to a function, expecting a monitor resource to lock at. The boxing occurs just prior to calling the IncInSync function, so the boxed instance corresponds to a different heap object each time the function is called.

```
public int Count { get; private set; }

private readonly int counterLock = 1;

public void Inc()
{
    IncInSync(counterLock);
}

private void IncInSync(object monitorResource)
{
    lock (monitorResource)
    {
        Count++;
    }
}
```

Boxing occurs in the `Inc` function:

```
BulemicCounter.Inc:
IL_0000:  nop
IL_0001:  ldarg.0
IL_0002:  ldarg.0
IL_0003:  ldfld       UserQuery+BulemicCounter.counterLock
IL_0008:  box         System.Int32**
IL_000D:  call        UserQuery+BulemicCounter.IncInSync
IL_0012:  nop
IL_0013:  ret
```

It does not mean that a boxed ValueType can't be used for monitor locking at all:

```
private readonly object counterLock = 1;
```

Now boxing occurs in constructor, which is fine for locking:

```
IL_0001:  ldc.i4.1
IL_0002:  box         System.Int32
IL_0007:  stfld       UserQuery+BulemicCounter.counterLock
```

Using locks unnecessarily when a safer alternative exists

A very common pattern is to use a private `List` or `Dictionary` in a thread safe class and lock every time it is accessed:

```
public class Cache
{
    private readonly object padlock;
    private readonly Dictionary<string, object> values;

    public WordStats()
    {
        this.padlock = new object();
        this.values = new Dictionary<string, object>();
    }

    public void Add(string key, object value)
    {
        lock (this.padlock)
        {
            this.values.Add(key, value);
        }
    }

    /* rest of class omitted */
}
```

If there are multiple methods accessing the `values` dictionary, the code can get very long and, more importantly, locking all the time obscures its *intent*. Locking is also very easy to forget and lack of proper locking can cause very hard to find bugs.

By using a `ConcurrentDictionary`, we can avoid locking completely:

```
public class Cache
{
    private readonly ConcurrentDictionary<string, object> values;

    public WordStats()
    {
        this.values = new ConcurrentDictionary<string, object>();
    }

    public void Add(string key, object value)
    {
        this.values.Add(key, value);
    }

    /* rest of class omitted */
}
```

Using concurrent collections also improves performance because all of them employ lock-free techniques to some extent.

Section 119.5: Using instances of Object for lock

When using C#'s inbuilt `lock` statement an instance of some type is needed, but its state does not matter. An instance of `object` is perfect for this:

```
public class ThreadSafe {
  private static readonly object locker = new object();


  public void SomeThreadSafeMethod() {
    lock (locker) {
      // Only one thread can be here at a time.
    }
  }
}
```

NB. instances of `Type` should not be used for this (in the code above `typeof(ThreadSafe)`) because instances of `Type` are shared across AppDomains and thus the extent of the lock can expectedly include code it shouldn't (eg. if `ThreadSafe` is loaded into two AppDomains in the same process then locking on its `Type` instance would mutually lock).

Chapter 120: Yield Keyword

When you use the yield keyword in a statement, you indicate that the method, operator, or get accessor in which it appears is an iterator. Using yield to define an iterator removes the need for an explicit extra class (the class that holds the state for an enumeration) when you implement the IEnumerable and IEnumerator pattern for a custom collection type.

Section 120.1: Simple Usage

The `yield` keyword is used to define a function which returns an `IEnumerable` or `IEnumerator` (as well as their derived generic variants) whose values are generated lazily as a caller iterates over the returned collection. Read more about the purpose in the remarks section.

The following example has a yield return statement that's inside a `for` loop.

```
public static IEnumerable<int> Count(int start, int count)
{
    for (int i = 0; i <= count; i++)
    {
        yield return start + i;
    }
}
```

Then you can call it:

```
foreach (int value in Count(start: 4, count: 10))
{
    Console.WriteLine(value);
}
```

Console Output

```
4
5
6
...
14
```

Live Demo on .NET Fiddle

Each iteration of the `foreach` statement body creates a call to the `Count` iterator function. Each call to the iterator function proceeds to the next execution of the `yield return` statement, which occurs during the next iteration of the `for` loop.

Section 120.2: Correctly checking arguments

An iterator method is not executed until the return value is enumerated. It's therefore advantageous to assert preconditions outside of the iterator.

```
public static IEnumerable<int> Count(int start, int count)
{
    // The exception will throw when the method is called, not when the result is iterated
    if (count < 0)
        throw new ArgumentOutOfRangeException(nameof(count));
```

```
    return CountCore(start, count);
}

private static IEnumerable<int> CountCore(int start, int count)
{
    // If the exception was thrown here it would be raised during the first MoveNext()
    // call on the IEnumerator, potentially at a point in the code far away from where
    // an incorrect value was passed.
    for (int i = 0; i < count; i++)
    {
        yield return start + i;
    }
}
```

Calling Side Code (Usage):

```
// Get the count
var count = Count(1,10);
// Iterate the results
foreach(var x in count)
{
    Console.WriteLine(x);
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Live Demo on .NET Fiddle

When a method uses `yield` to generate an enumerable the compiler creates a state machine that when iterated over will run code up to a `yield`. It then returns the yielded item, and saves its state.

This means you won't find out about invalid arguments (passing `null` etc.) when you first call the method (because that creates the state machine), only when you try and access the first element (because only then does the code within the method get ran by the state machine). By wrapping it in a normal method that first checks arguments you can check them when the method is called. This is an example of failing fast.

When using C# 7+, the `CountCore` function can be conveniently hidden into the `Count` function as a *local function*. See example here.

Section 120.3: Early Termination

You can extend the functionality of existing `yield` methods by passing in one or more values or elements that could define a terminating condition within the function by calling a `yield break` to stop the inner loop from executing.

```
public static IEnumerable<int> CountUntilAny(int start, HashSet<int> earlyTerminationSet)
{
    int curr = start;

    while (true)
    {
        if (earlyTerminationSet.Contains(curr))
        {
            // we've hit one of the ending values
            yield break;
        }

        yield return curr;

        if (curr == Int32.MaxValue)
        {
            // don't overflow if we get all the way to the end; just stop
            yield break;
        }

        curr++;
    }
}
```

The above method would iterate from a given `start` position until one of the values within the `earlyTerminationSet` was encountered.

```
// Iterate from a starting point until you encounter any elements defined as
// terminating elements
var terminatingElements = new HashSet<int>{ 7, 9, 11 };
// This will iterate from 1 until one of the terminating elements is encountered (7)
foreach(var x in CountUntilAny(1,terminatingElements))
{
    // This will write out the results from 1 until 7 (which will trigger terminating)
    Console.WriteLine(x);
}
```

Output:

```
1
2
3
4
5
6
```

Live Demo on .NET Fiddle

Section 120.4: More Pertinent Usage

```
public IEnumerable<User> SelectUsers()
{
    // Execute an SQL query on a database.
    using (IDataReader reader = this.Database.ExecuteReader(CommandType.Text, "SELECT Id, Name FROM
Users"))
    {
```

```
        while (reader.Read())
        {
            int id = reader.GetInt32(0);
            string name = reader.GetString(1);
            yield return new User(id, name);
        }
    }
}
```

There are other ways of getting an `IEnumerable<User>` from an SQL database, of course -- this just demonstrates that you can use `yield` to turn anything that has "sequence of elements" semantics into an `IEnumerable<T>` that someone can iterate over.

Section 120.5: Lazy Evaluation

Only when the `foreach` statement moves to the next item does the iterator block evaluate up to the next `yield` statement.

Consider the following example:

```
private IEnumerable<int> Integers()
{
    var i = 0;
    while(true)
    {
        Console.WriteLine("Inside iterator: " + i);
        yield return i;
        i++;
    }
}

private void PrintNumbers()
{
    var numbers = Integers().Take(3);
    Console.WriteLine("Starting iteration");

    foreach(var number in numbers)
    {
        Console.WriteLine("Inside foreach: " + number);
    }
}
```

This will output:

> Starting iteration
> Inside iterator: 0
> Inside foreach: 0
> Inside iterator: 1
> Inside foreach: 1
> Inside iterator: 2
> Inside foreach: 2

View Demo

As a consequence:

- "Starting iteration" is printed first even though the iterator method was called before the line printing it because the line `Integers().Take(3);` does not actually starts iteration (no call to `IEnumerator.MoveNext()` was made)
- The lines printing to console alternate between the one inside the iterator method and the one inside the `foreach`, rather than all the ones inside the iterator method evaluating first
- This program terminates due to the `.Take()` method, even though the iterator method has a `while true` which it never breaks out of.

Section 120.6: Try...finally

If an iterator method has a yield inside a `try...finally`, then the returned `IEnumerator` will execute the `finally` statement when `Dispose` is called on it, as long as the current point of evaluation is inside the `try` block.

Given the function:

```
private IEnumerable<int> Numbers()
{
    yield return 1;
    try
    {
        yield return 2;
        yield return 3;
    }
    finally
    {
        Console.WriteLine("Finally executed");
    }
}
```

When calling:

```
private void DisposeOutsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.Dispose();
}
```

Then it prints:

> 1

View Demo

When calling:

```
private void DisposeInsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
```

```
    enumerator.Dispose();
}
```

Then it prints:

> 1
> 2
> Finally executed

View Demo

Section 120.7: Eager evaluation

The `yield` keyword allows lazy-evaluation of the collection. Forcibly loading the whole collection into memory is called **eager evaluation**.

The following code shows this:

```
IEnumerable<int> myMethod()
{
    for(int i=0; i <= 8675309; i++)
    {
        yield return i;
    }
}
...
// define the iterator
var it = myMethod.Take(3);
// force its immediate evaluation
// list will contain 0, 1, 2
var list = it.ToList();
```

Calling `ToList`, `ToDictionary` or `ToArray` will force the immediate evaluation of the enumeration, retrieving all the elements into a collection.

Section 120.8: Using yield to create an IEnumerator<T> when implementing IEnumerable<T>

The `IEnumerable<T>` interface has a single method, `GetEnumerator()`, which returns an `IEnumerator<T>`.

While the `yield` keyword can be used to directly create an `IEnumerable<T>`, it can *also* be used in exactly the same way to create an `IEnumerator<T>`. The only thing that changes is the return type of the method.

This can be useful if we want to create our own class which implements `IEnumerable<T>`:

```
public class PrintingEnumerable<T> : IEnumerable<T>
{
    private IEnumerable<T> _wrapped;

    public PrintingEnumerable(IEnumerable<T> wrapped)
    {
        _wrapped = wrapped;
    }

    // This method returns an IEnumerator<T>, rather than an IEnumerable<T>
```

```
    // But the yield syntax and usage is identical.
    public IEnumerator<T> GetEnumerator()
    {
        foreach(var item in _wrapped)
        {
            Console.WriteLine("Yielding: " + item);
            yield return item;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

(Note that this particular example is just illustrative, and could be more cleanly implemented with a single iterator method returning an `IEnumerable<T>`.)

Section 120.9: Lazy Evaluation Example: Fibonacci Numbers

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Numerics; // also add reference to System.Numberics

namespace ConsoleApplication33
{
    class Program
    {
        private static IEnumerable<BigInteger> Fibonacci()
        {
            BigInteger prev = 0;
            BigInteger current = 1;
            while (true)
            {
                yield return current;
                var next = prev + current;
                prev = current;
                current = next;
            }
        }

        static void Main()
        {
            // print Fibonacci numbers from 10001 to 10010
            var numbers = Fibonacci().Skip(10000).Take(10).ToArray();
            Console.WriteLine(string.Join(Environment.NewLine, numbers));
        }
    }
}
```

How it works under the hood (I recommend to decompile resulting .exe file in IL Disaambler tool):

1. C# compiler generates a class implementing `IEnumerable<BigInteger>` and `IEnumerator<BigInteger>` (`<Fibonacci>d__0` in ildasm).
2. This class implements a state machine. State consists of current position in method and values of local variables.
3. The most interesting code are in `bool IEnumerator.MoveNext()` method. Basically, what `MoveNext()` do:

- Restores current state. Variables like `prev` and `current` become fields in our class (`<current>5__2` and `<prev>5__1` in ildasm). In our method we have two positions (`<>1__state`): first at the opening curly brace, second at `yield return`.
- Executes code until next `yield return` or `yield break`/}.
- For `yield return` resulting value is saved, so `Current` property can return it. `true` is returned. At this point current state is saved again for the next `MoveNext` invocation.
- For `yield break`/} method just returns `false` meaning iteration is done.

Also note, that 10001th number is 468 bytes long. State machine only saves `current` and `prev` variables as fields. While if we would like to save all numbers in the sequence from the first to the 10000th, the consumed memory size will be over 4 megabytes. So lazy evaluation, if properly used, can reduce memory footprint in some cases.

Section 120.10: The difference between break and yield break

Using `yield break` as opposed to `break` might not be as obvious as one may think. There are lot of bad examples on the Internet where the usage of the two is interchangeable and doesn't really demonstrate the difference.

The confusing part is that both of the keywords (or key phrases) make sense only within loops (`foreach`, `while`...) So when to choose one over the other?

It's important to realize that once you use the `yield` keyword in a method you effectively turn the method into an iterator. The only purpose of the such method is then to iterate over a finite or infinite collection and yield (output) its elements. Once the purpose is fulfilled, there's no reason to continue method's execution. Sometimes, it happens naturally with the last closing bracket of the method `}`. But sometimes, you want to end the method prematurely. In a normal (non-iterating) method you would use the `return` keyword. But you can't use `return` in an iterator, you have to use `yield break`. In other words, `yield break` for an iterator is the same as `return` for a standard method. Whereas, the `break` statement just terminates the closest loop.

Let's see some examples:

```
/// <summary>
/// Yields numbers from 0 to 9
/// </summary>
/// <returns>{0,1,2,3,4,5,6,7,8,9}</returns>
public static IEnumerable<int> YieldBreak()
{
    for (int i = 0; ; i++)
    {
        if (i < 10)
        {
            // Yields a number
            yield return i;
        }
        else
        {
            // Indicates that the iteration has ended, everything
            // from this line on will be ignored
            yield break;
        }
    }
    yield return 10; // This will never get executed
}
```

```
/// <summary>
/// Yields numbers from 0 to 10
/// </summary>
/// <returns>{0,1,2,3,4,5,6,7,8,9,10}</returns>
```

```
    public static IEnumerable<int> Break()
    {
        for (int i = 0; ; i++)
        {
            if (i < 10)
            {
                // Yields a number
                yield return i;
            }
            else
            {
                // Terminates just the loop
                break;
            }
        }
        // Execution continues
        yield return 10;
    }
```

Section 120.11: Return another Enumerable within a method returning Enumerable

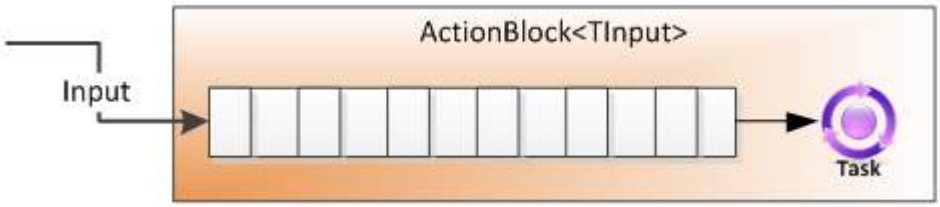
```
public IEnumerable<int> F1()
{
    for (int i = 0; i < 3; i++)
        yield return i;

    //return F2(); // Compile Error!!
    foreach (var element in F2())
        yield return element;
}

public int[] F2()
{
    return new[] { 3, 4, 5 };
}
```

Chapter 121: Task Parallel Library (TPL) Dataflow Constructs

Section 121.1: ActionBlock<T>

(foreach)

This class can be thought of logically as a buffer for data to be processed combined with tasks for processing that data, with the "dataflow block" managing both. In its most basic usage, we can instantiate an ActionBlock and "post" data to it; the delegate provided at the ActionBlock's construction will be executed asynchronously for every piece of data posted.



Synchronous Computation

```
var ab = new ActionBlock<TInput>(i =>
{
    Compute(i);
});
...
ab.Post(1);
ab.Post(2);
ab.Post(3);
```

Throttling Asynchronous Downloads to at most 5 concurrently

```
var downloader = new ActionBlock<string>(async url =>
{
    byte [] imageData = await DownloadAsync(url);
    Process(imageData);
}, new DataflowBlockOptions { MaxDegreeOfParallelism = 5 });

downloader.Post("http://website.com/path/to/images");
downloader.Post("http://another-website.com/path/to/images");
```

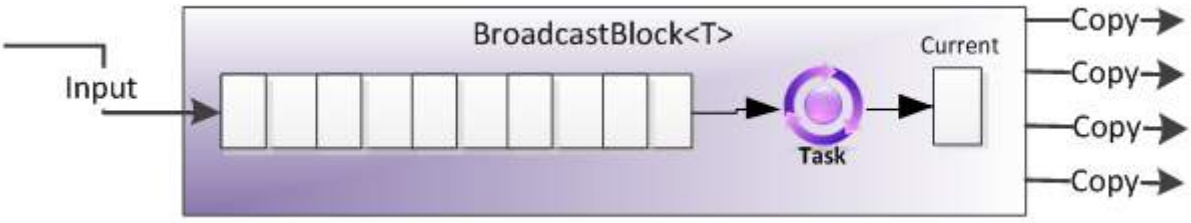
Introduction to TPL Dataflow by Stephen Toub

Section 121.2: BroadcastBlock<T>

(Copy an item and send the copies to every block that it's linked to)

Unlike BufferBlock, BroadcastBlock's mission in life is to enable all targets linked from the block to get a copy of every element published, continually overwriting the "current" value with those propagated to it.

Additionally, unlike BufferBlock, BroadcastBlock doesn't hold on to data unnecessarily. After a particular datum has been offered to all targets, that element will be overwritten by whatever piece of data is next in line (as with all dataflow blocks, messages are handled in FIFO order). That element will be offered to all targets, and so on.



Asynchronous Producer/Consumer with a Throttled Producer

```
var ui = TaskScheduler.FromCurrentSynchronizationContext();
var bb = new BroadcastBlock<ImageData>(i => i);

var saveToDiskBlock = new ActionBlock<ImageData>(item =>
    item.Image.Save(item.Path)
);

var showInUiBlock = new ActionBlock<ImageData>(item =>
    imagePanel.AddImage(item.Image),
    new DataflowBlockOptions { TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext() }
);

bb.LinkTo(saveToDiskBlock);
bb.LinkTo(showInUiBlock);
```

Exposing Status from an Agent

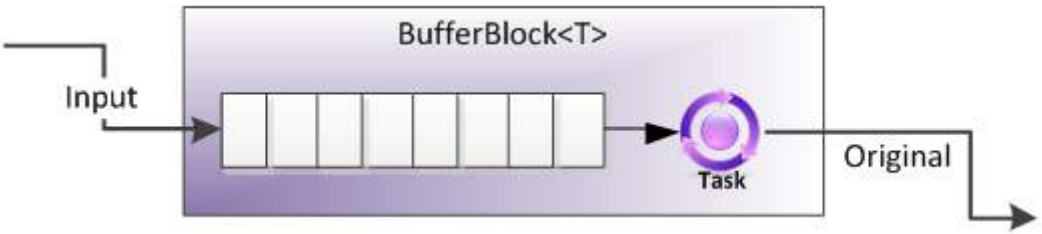
```
public class MyAgent
{
    public ISourceBlock<string> Status { get; private set; }

    public MyAgent()
    {
        Status = new BroadcastBlock<string>();
        Run();
    }

    private void Run()
    {
        Status.Post("Starting");
        Status.Post("Doing cool stuff");
        ...
        Status.Post("Done");
    }
}
```

Introduction to TPL Dataflow by Stephen Toub

Section 121.3: BufferBlock<T>

(FIFO Queue: The data that comes in is the data that goes out)

In short, BufferBlock provides an unbounded or bounded buffer for storing instances of T.
You can “post” instances of T to the block, which cause the data being posted to be stored in a first-in-first-out (FIFO) order by the block.
You can “receive” from the block, which allows you to synchronously or asynchronously obtain instances of T previously stored or available in the future (again, FIFO).



Asynchronous Producer/Consumer with a Throttled Producer

```
// Hand-off through a bounded BufferBlock<T>
private static BufferBlock<int> _Buffer = new BufferBlock<int>(
    new DataflowBlockOptions { BoundedCapacity = 10 });

// Producer
private static async void Producer()
{
    while(true)
    {
        await _Buffer.SendAsync(Produce());
    }
}

// Consumer
private static async Task Consumer()
{
    while(true)
    {
        Process(await _Buffer.ReceiveAsync());
    }
}

// Start the Producer and Consumer
private static async Task Run()
{
    await Task.WhenAll(Producer(), Consumer());
}
```

Introduction to TPL Dataflow by Stephen Toub

Section 121.4: JoinBlock<T1, T2,…>

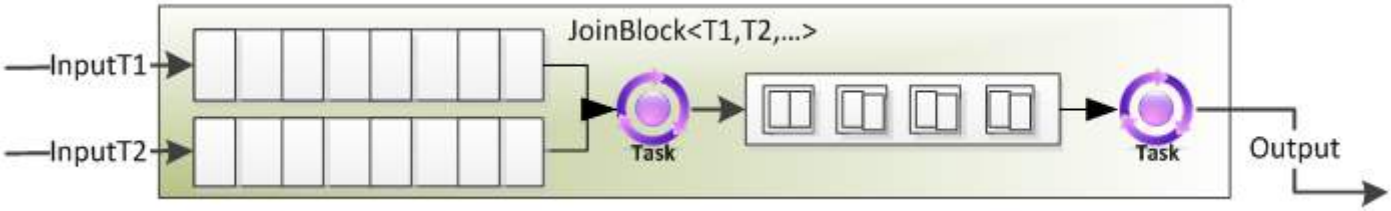
(Collects 2-3 inputs and combines them into a Tuple)

Like BatchBlock, JoinBlock<T1, T2, …> is able to group data from multiple data sources. In fact, that's JoinBlock<T1, T2, …>'s primary purpose.

For example, a JoinBlock<string, double, int> is an ISourceBlock<Tuple<string, double, int>>.

As with BatchBlock, JoinBlock<T1, T2,…> is capable of operating in both greedy and non-greedy mode.

- In the default greedy mode, all data offered to targets are accepted, even if the other target doesn't have the necessary data with which to form a tuple.
- In non-greedy mode, the block's targets will postpone data until all targets have been offered the necessary data to create a tuple, at which point the block will engage in a two-phase commit protocol to atomically retrieve all necessary items from the sources. This postponement makes it possible for another entity to consume the data in the meantime so as to allow the overall system to make forward progress.



Processing Requests with a Limited Number of Pooled Objects

```
var throttle = new JoinBlock<ExpensiveObject, Request>();
for(int i=0; i<10; i++)
{
    requestProcessor.Target1.Post(new ExpensiveObject());
}

var processor = new Transform<Tuple<ExpensiveObject, Request>, ExpensiveObject>(pair =>
{
    var resource = pair.Item1;
    var request = pair.Item2;

    request.ProcessWith(resource);

    return resource;
});

throttle.LinkTo(processor);
processor.LinkTo(throttle.Target1);
```

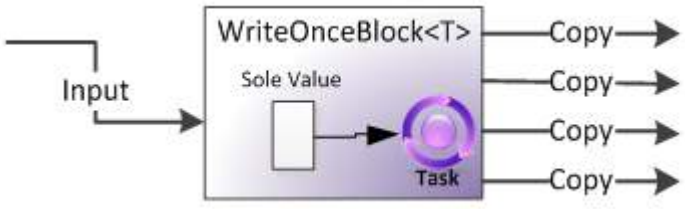
Introduction to TPL Dataflow by Stephen Toub

Section 121.5: WriteOnceBlock<T>

(Readonly variable: Memorizes its first data item and passes out copies of it as its output. Ignores all other data items)

If BufferBlock is the most fundamental block in TPL Dataflow, WriteOnceBlock is the simplest.
It stores at most one value, and once that value has been set, it will never be replaced or overwritten.

You can think of WriteOnceBlock in as being similar to a readonly member variable in C#, except instead of only being settable in a constructor and then being immutable, it's only settable once and is then immutable.



Splitting a Task's Potential Outputs

```
public static async void SplitIntoBlocks(this Task<T> task,
    out IPropagatorBlock<T> result,
    out IPropagatorBlock<Exception> exception)
{
    result = new WriteOnceBlock<T>(i => i);
    exception = new WriteOnceBlock<Exception>(i => i);
```

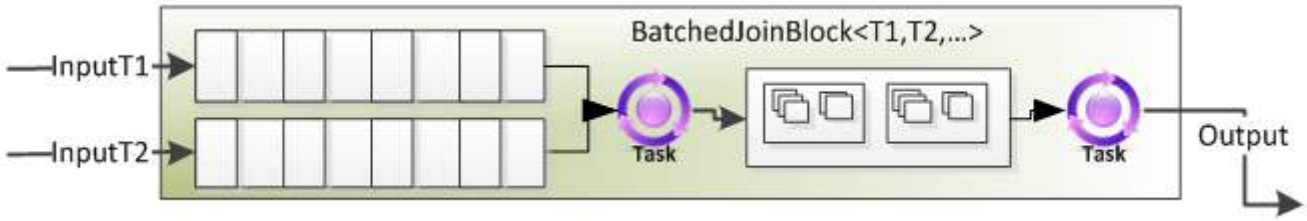
```
    try
    {
        result.Post(await task);
    }
    catch(Exception ex)
    {
        exception.Post(ex);
    }
}
```

Introduction to TPL Dataflow by Stephen Toub

Section 121.6: BatchedJoinBlock<T1, T2,…>

(Collects a certain number of total items from 2-3 inputs and groups them into a Tuple of collections of data items)

BatchedJoinBlock<T1, T2,…> is in a sense a combination of BatchBlock and JoinBlock<T1, T2,…>.
Whereas JoinBlock<T1, T2,…> is used to aggregate one input from each target into a tuple, and BatchBlock is used to aggregate N inputs into a collection, BatchedJoinBlock<T1, T2,…> is used to gather N inputs from across all of the targets into tuples of collections.



Scatter/Gather

Consider a scatter/gather problem where N operations are launched, some of which may succeed and produce string outputs, and others of which may fail and produce Exceptions.

```
var batchedJoin = new BatchedJoinBlock<string, Exception>(10);

for (int i=0; i<10; i++)
{
    Task.Factory.StartNew(() => {
        try { batchedJoin.Target1.Post(DoWork()); }
        catch(Exception ex) { batchJoin.Target2.Post(ex); }
    });
}

var results = await batchedJoin.ReceiveAsync();

foreach(string s in results.Item1)
{
    Console.WriteLine(s);
}

foreach(Exception e in results.Item2)
{
    Console.WriteLine(e);
}
```

Introduction to TPL Dataflow by Stephen Toub

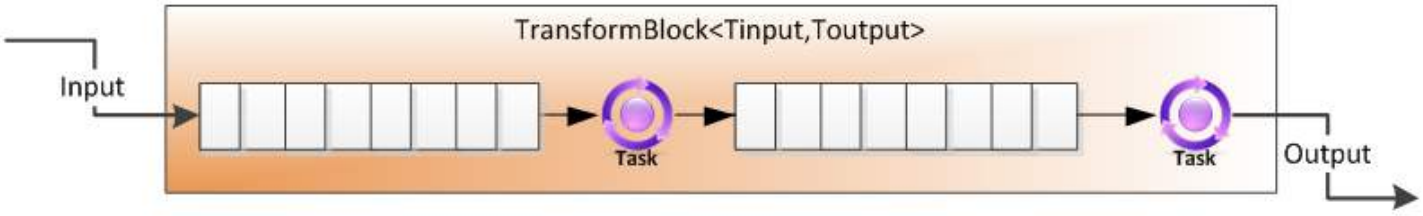
Section 121.7: TransformBlock<TInput, TOutput>

(Select, one-to-one)

As with ActionBlock, TransformBlock<TInput, TOutput> enables the execution of a delegate to perform some action for each input datum; **unlike with ActionBlock, this processing has an output.** This delegate can be a Func<TInput, TOutput>, in which case processing of that element is considered completed when the delegate returns, or it can be a Func<TInput,Task>, in which case processing of that element is considered completed not when the delegate returns but when the returned Task completes. For those familiar with LINQ, it's somewhat similar to Select() in that it takes an input, transforms that input in some manner, and then produces an output.

By default, TransformBlock<TInput, TOutput> processes its data sequentially with a MaxDegreeOfParallelism equal to 1. In addition to receiving buffered input and processing it, this block will take all of its processed output and buffer that as well (data that has not been processed, and data that has been processed).

It has 2 tasks: One to process the data, and one to push data to the next block.



A Concurrent Pipeline

```
var compressor = new TransformBlock<byte[], byte[]>(input => Compress(input));
var encryptor = new TransformBlock<byte[], byte[]>(input => Encrypt(input));

compressor.LinkTo(Encryptor);
```

Introduction to TPL Dataflow by Stephen Toub

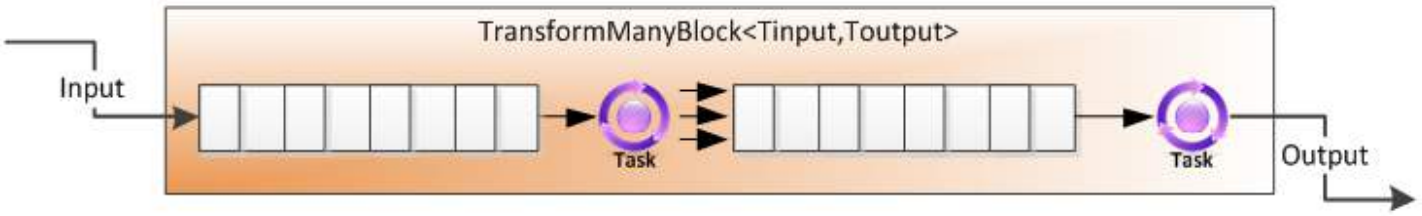
Section 121.8: TransformManyBlock<TInput, TOutput>

(SelectMany, 1-m: The results of this mapping are "flattened", just like LINQ's SelectMany)

TransformManyBlock<TInput, TOutput> is very similar to TransformBlock<TInput, TOutput>.
The key difference is that whereas a TransformBlock<TInput, TOutput> produces one and only one output for each input, TransformManyBlock<TInput, TOutput> produces any number (zero or more) outputs for each input. As with ActionBlock and TransformBlock<TInput, TOutput>, this processing may be specified using delegates, both for synchronous and asynchronous processing.

A Func<TInput, IEnumerable> is used for synchronous, and a Func<TInput, Task<IEnumerable>> is used for asynchronous. As with both ActionBlock and TransformBlock<TInput, TOutput>, TransformManyBlock<TInput, TOutput> defaults to sequential processing, but may be configured otherwise.

The mapping delegate retuns a collection of items, which are inserted individually into the output buffer.



Asynchronous Web Crawler

```
var downloader = new TransformManyBlock<string, string>(async url =>
{
    Console.WriteLine("Downloading " + url);
    try
    {
        return ParseLinks(await DownloadContents(url));
    }
    catch{}

    return Enumerable.Empty<string>();
});
downloader.LinkTo(downloader);
```

Expanding an Enumerable Into Its Constituent Elements

```
var expanded = new TransformManyBlock<T[], T>(array => array);
```

Filtering by going from 1 to 0 or 1 elements

```
public IPropagatorBlock<T> CreateFilteredBuffer<T>(Predicate<T> filter)
{
    return new TransformManyBlock<T, T>(item =>
        filter(item) ? new [] { item } : Enumerable.Empty<T>());
}
```

Introduction to TPL Dataflow by Stephen Toub

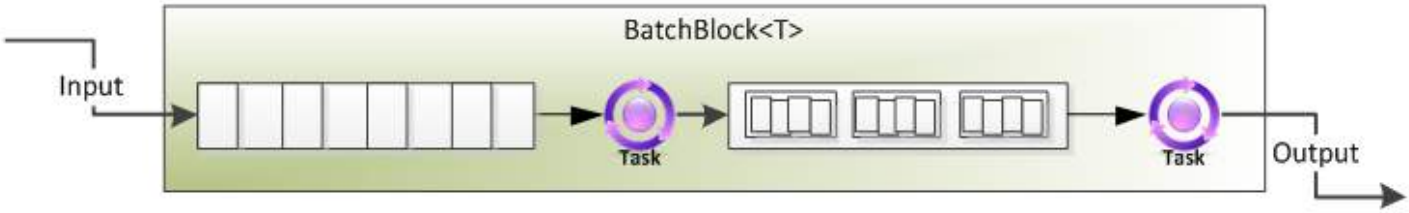
Section 121.9: BatchBlock<T>

(Groups a certain number of sequential data items into collections of data items)

BatchBlock combines N single items into one batch item, represented as an array of elements. An instance is created with a specific batch size, and the block then creates a batch as soon as it's received that number of elements, asynchronously outputting the batch to the output buffer.

BatchBlock is capable of executing in both greedy and non-greedy modes.

- In the default greedy mode, all messages offered to the block from any number of sources are accepted and buffered to be converted into batches.
    - In non-greedy mode, all messages are postponed from sources until enough sources have offered messages to the block to create a batch. Thus, a BatchBlock can be used to receive 1 element from each of N sources, N elements from 1 source, and a myriad of options in between.



Batching Requests into groups of 100 to Submit to a Database

```
var batchRequests = new BatchBlock<Request>(batchSize:100);
var sendToDb = new ActionBlock<Request[]>(reqs => SubmitToDatabase(reqs));

batchRequests.LinkTo(sendToDb);
```

Creating a batch once a second

```
var batch = new BatchBlock<T>(batchSize:Int32.MaxValue);
new Timer(() => { batch.TriggerBatch(); }).Change(1000, 1000);
```

Introduction to TPL Dataflow by Stephen Toub

Chapter 122: Functional Programming

Section 122.1: Func and Action

Func provides a holder for parameterised anonymous functions. The leading types are the inputs and the last type is always the return value.

```
// square a number.
Func<double, double> square = (x) => { return x * x; };

// get the square root.
// note how the signature matches the built in method.
Func<double, double> squareroot = Math.Sqrt;

// provide your workings.
Func<double, double, string> workings = (x, y) =>
    string.Format("The square of {0} is {1}.", x, square(y))
```

Action objects are like void methods so they only have an input type. No result is placed on the evaluation stack.

```
// right-angled triangle.
class Triangle
{
    public double a;
    public double b;
    public double h;
}

// Pythagorean theorem.
Action<Triangle> pythagoras = (x) =>
    x.h = squareroot(square(x.a) + square(x.b));

Triangle t = new Triangle { a = 3, b = 4 };
pythagoras(t);
Console.WriteLine(t.h); // 5.
```

Section 122.2: Higher-Order Functions

A higher-order function is one that takes another function as an argument or returns a function (or both).

This is commonly done with lambdas, for example when passing a predicate to a LINQ Where clause:

```
var results = data.Where(p => p.Items == 0);
```

The Where() clause could receive many different predicates which gives it considerable flexibility.

Passing a method into another method is also seen when implementing the Strategy design pattern. For example, various sorting methods could be chosen from and passed into a Sort method on an object depending on the requirements at run-time.

Section 122.3: Avoid Null References

C# developers get a lot of null reference exceptions to deal with. F# developers don't because they have the Option type. An Option<> type (some prefer Maybe<> as a name) provides a Some and a None return type. It makes it explicit that a method may be about to return a null record.

For instance, you can't read the following and know if you will have to deal with a null value.

```
var user = _repository.GetUser(id);
```

If you do know about the possible null you can introduce some boilerplate code to deal with it.

```
var username = user != null ? user.Name : string.Empty;
```

What if we have an Option<> returned instead?

```
Option<User> maybeUser = _repository.GetUser(id);
```

The code now makes it explicit that we may have a None record returned and the boilerplate code to check for Some or None is required:

```
var username = maybeUser.HasValue ? maybeUser.Value.Name : string.Empty;
```

The following method shows how to return an Option<>

```
public Option<User> GetUser(int id)
{
    var users = new List<User>
    {
        new User { Id = 1, Name = "Joe Bloggs" },
        new User { Id = 2, Name = "John Smith" }
    };

    var user = users.FirstOrDefault(user => user.Id == id);

    return user != null ? new Option<User>(user) : new Option<User>();
}
```

Here is a minimal implementation of Option<>.

```
public struct Option<T>
{
    private readonly T _value;

    public T Value
    {
        get
        {
            if (!HasValue)
                throw new InvalidOperationException();

            return _value;
        }
    }

    public bool HasValue
    {
        get { return _value != null; }
    }

    public Option(T value)
    {
        _value = value;
    }
```

```
    public static implicit operator Option<T>(T value)
    {
        return new Option<T>(value);
    }
}
```

To demonstrate the above avoidNull.csx can be run with the C# REPL.

As stated, this is a minimal implementation. A search for "Maybe" NuGet packages will turn up a number of good libraries.

Section 122.4: Immutability

Immutability is common in functional programming and rare in object oriented programming.

Create, for example, an address type with mutable state:

```
public class Address ()
{
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string City  { get; set; }
}
```

Any piece of code could alter any property in the above object.

Now create the immutable address type:

```
public class Address ()
{
    public readonly string Line1;
    public readonly string Line2;
    public readonly string City;

    public Address(string line1, string line2, string city)
    {
        Line1 = line1;
        Line2 = line2;
        City  = city;
    }
}
```

Bear in mind that having read-only collections does not respect immutability. For example,

```
public class Classroom
{
    public readonly List<Student> Students;

    public Classroom(List<Student> students)
    {
        Students = students;
    }
}
```

is not immutable, as the user of the object can alter the collection (add or remove elements from it). In order to make it immutable, one has either to use an interface like IEnumerable, which does not expose methods to add, or to make it a ReadOnlyCollection.

```
public class Classroom
{
    public readonly ReadOnlyCollection<Student> Students;

    public Classroom(ReadOnlyCollection<Student> students)
    {
        Students = students;
    }
}

List<Students> list = new List<Student>();
// add students
Classroom c = new Classroom(list.AsReadOnly());
```

With the immutable object we have the following benefits:

- It will be in a known state (other code can't change it).
- It is thread safe.
- The constructor offers a single place for validation.
- Knowing that the object cannot be altered makes the code easier to understand.

Section 122.5: Immutable collections

The `System.Collections.Immutable` NuGet package provides immutable collection classes.

Creating and adding items

```
var stack = ImmutableStack.Create<int>();
var stack2 = stack.Push(1); // stack is still empty, stack2 contains 1
var stack3 = stack.Push(2); // stack2 still contains only one, stack3 has 2, 1
```

Creating using the builder

Certain immutable collections have a `Builder` inner class that can be used to cheaply build large immutable instances:

```
var builder = ImmutableList.CreateBuilder<int>(); // returns ImmutableList.Builder
builder.Add(1);
builder.Add(2);
var list = builder.ToImmutable();
```

Creating from an existing IEnumerable

```
var numbers = Enumerable.Range(1, 5);
var list = ImmutableList.CreateRange<int>(numbers);
```

List of all immutable collection types:

- `System.Collections.Immutable.ImmutableArray<T>`
- `System.Collections.Immutable.ImmutableDictionary<TKey,TValue>`
- `System.Collections.Immutable.ImmutableHashSet<T>`
- `System.Collections.Immutable.ImmutableList<T>`
- `System.Collections.Immutable.ImmutableQueue<T>`
- `System.Collections.Immutable.ImmutableSortedDictionary<TKey,TValue>`
- `System.Collections.Immutable.ImmutableSortedSet<T>`
- `System.Collections.Immutable.ImmutableStack<T>`

Chapter 123: Func delegates

Parameter	Details
`arg` or `arg1`	the (first) parameter of the method
`arg2`	the second parameter of the method
`arg3`	the third parameter of the method
`arg4`	the fourth parameter of the method
`T` or `T1`	the type of the (first) parameter of the method
`T2`	the type of the second parameter of the method
`T3`	the type of the third parameter of the method
`T4`	the type of the fourth parameter of the method
`TResult`	the return type of the method

Section 123.1: Without parameters

This example shows how to create a delegate that encapsulates the method that returns the current time

```
static DateTime UTCNow()
{
    return DateTime.UtcNow;
}

static DateTime LocalNow()
{
    return DateTime.Now;
}

static void Main(string[] args)
{
    Func<DateTime> method = UTCNow;
    // method points to the UTCNow method
    // that retuns current UTC time
    DateTime utcNow = method();

    method = LocalNow;
    // now method points to the LocalNow method
    // that returns local time

    DateTime localNow = method();
}
```

Section 123.2: With multiple variables

```
static int Sum(int a, int b)
{
    return a + b;
}

static int Multiplication(int a, int b)
{
    return a * b;
}

static void Main(string[] args)
{
```

```
    Func<int, int, int> method = Sum;
    // method points to the Sum method
    // that retuns 1 int variable and takes 2 int variables
    int sum = method(1, 1);

    method = Multiplication;
    // now method points to the Multiplication method

    int multiplication = method(1, 1);
}
```

Section 123.3: Lambda & anonymous methods

An anonymous method can be assigned wherever a delegate is expected:

```
Func<int, int> square = delegate (int x) { return x * x; }
```

Lambda expressions can be used to express the same thing:

```
Func<int, int> square = x => x * x;
```

In either case, we can now invoke the method stored inside `square` like this:

```
var sq = square.Invoke(2);
```

Or as a shorthand:

```
var sq = square(2);
```

Notice that for the assignment to be type-safe, the parameter types and return type of the anonymous method must match those of the delegate type:

```
Func<int, int> sum = delegate (int x, int y) { return x + y; } // error
Func<int, int> sum = (x, y) => x + y; // error
```

Section 123.4: Covariant & Contravariant Type Parameters

`Func` also supports Covariant & Contravariant

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }

class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;
```

```
        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;

    }
}
```

Chapter 124: Function with multiple return values

Section 124.1: "anonymous object" + "dynamic keyword" solution

You can return an anonymous object from your function

```
public static object FunctionWithUnknowReturnValues ()
{
    /// anonymous object
    return new { a = 1, b = 2 };
}
```

And assign the result to a dynamic object and read the values in it.

```
/// dynamic object
dynamic x = FunctionWithUnknowReturnValues();

Console.WriteLine(x.a);
Console.WriteLine(x.b);
```

Section 124.2: Tuple solution

You can return an instance of `Tuple` class from your function with two template parameters as `Tuple<string, MyClass>`:

```
public Tuple<string, MyClass> FunctionWith2ReturnValues ()
{
    return Tuple.Create("abc", new MyClass());
}
```

And read the values like below:

```
Console.WriteLine(x.Item1);
Console.WriteLine(x.Item2);
```

Section 124.3: Ref and Out Parameters

The `ref` keyword is used to pass an Argument as Reference. `out` will do the same as `ref` but it does not require an assigned value by the caller prior to calling the function.

Ref Parameter: If you want to pass a variable as ref parameter then you need to initialize it before you pass it as ref parameter to method.

Out Parameter: If you want to pass a variable as out parameter you don't need to initialize it before you pass it as out parameter to method.

```
static void Main(string[] args)
{
    int a = 2;
    int b = 3;
    int add = 0;
    int mult= 0;
```

```
    AddOrMult(a, b, ref add, ref mult); //AddOrMult(a, b, out add, out mult);
    Console.WriteLine(add); //5
    Console.WriteLine(mult); //6
}

private static void AddOrMult(int a, int b, ref int add, ref int mult) //AddOrMult(int a, int b,
out int add, out int mult)
{
    add = a + b;
    mult = a * b;
}
```

Chapter 125: Binary Serialization

Section 125.1: Controlling serialization behavior with attributes

If you use the `[NonSerialized]` attribute, then that member will always have its default value after deserialization (ex. 0 for an `int`, null for `string`, false for a `bool`, etc.), regardless of any initialization done in the object itself (constructors, declarations, etc.). To compensate, the attributes `[OnDeserializing]` (called just BEFORE deserializing) and `[OnDeserialized]` (called just AFTER deserializing) together with their counterparts, `[OnSerializing]` and `[OnSerialized]` are provided.

Assume we want to add a "Rating" to our Vector and we want to make sure the value always starts at 1. The way it is written below, it will be 0 after being deserialized:

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal Rating = 1M;

    public Vector()
    {
        Rating = 1M;
    }

    public Vector(decimal initialRating)
    {
        Rating = initialRating;
    }
}
```

To fix this problem, we can simply add the following method inside of the class to set it to 1:

```
[OnDeserializing]
void OnDeserializing(StreamingContext context)
{
    Rating = 1M;
}
```

Or, if we want to set it to a calculated value, we can wait for it to be finished deserializing and then set it:

```
[OnDeserialized]
void OnDeserialized(StreamingContext context)
{
    Rating = 1 + ((X+Y+Z)/3);
}
```

Similarly, we can control how things are written out by using `[OnSerializing]` and `[OnSerialized]`.

Section 125.2: Serialization Binder

The binder gives you an opportunity to inspect what types are being loaded in your application domain

Create a class inherited from SerializationBinder

```
class MyBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        if (typeName.Equals("BinarySerializationExample.Item"))
            return typeof(Item);
        return null;
    }
}
```

Now we can check what types are loading and on this basis to decide what we really want to receive

For using a binder, you must add it to the BinaryFormatter.

```
object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    binaryFormatter.Binder = new MyBinder();

    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

The complete solution

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class MyBinder : SerializationBinder
    {
        public override Type BindToType(string assemblyName, string typeName)
        {
            if (typeName.Equals("BinarySerializationExample.Item"))
                return typeof(Item);
            return null;
        }
    }

    [Serializable]
    public class Item
    {
        private string _name;

        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
```

```
            var item = new Item
            {
                Name = "Orange"
            };

            var bytes = SerializeData(item);
            var deserializedData = (Item)DeserializeData(bytes);
        }

        private static byte[] SerializeData(object obj)
        {
            var binaryFormatter = new BinaryFormatter();
            using (var memoryStream = new MemoryStream())
            {
                binaryFormatter.Serialize(memoryStream, obj);
                return memoryStream.ToArray();
            }
        }

        private static object DeserializeData(byte[] bytes)
        {
            var binaryFormatter = new BinaryFormatter
            {
                Binder = new MyBinder()
            };

            using (var memoryStream = new MemoryStream(bytes))
                return binaryFormatter.Deserialize(memoryStream);
        }
    }
}
```

Section 125.3: Some gotchas in backward compatibility

This small example shows how you can lose backward compatibility in your programs if you do not take care in advance about this. And ways to get more control of serialization process

At first, we will write an example of the first version of the program:

Version 1

```
[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }
}
```

And now, let us assume that in the second version of the program added a new class. And we need to store it in an array.

Now code will look like this:

Version 2

```
[Serializable]
class NewItem
{
    [OptionalField]
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}

[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }

    [OptionalField]
    private List<NewItem> _newItems;

    public List<NewItem> NewItems
    {
        get { return _newItems; }
        set { _newItems = value; }
    }
}
```

And code for serialize and deserialize

```
private static byte[] SerializeData(object obj)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream())
    {
        binaryFormatter.Serialize(memoryStream, obj);
        return memoryStream.ToArray();
    }
}

private static object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

And so, what would happen when you serialize the data in the program of v2 and will try to deserialize them in the program of v1?

You get an exception:

```
System.Runtime.Serialization.SerializationException was unhandled
Message=The ObjectManager found an invalid number of fixups. This usually indicates a problem in
the Formatter.Source=mscorlib
StackTrace:
at System.Runtime.Serialization.ObjectManager.DoFixups()
at System.Runtime.Serialization.Formatters.Binary.ObjectReader.Deserialize(HeaderHandler handler,
__BinaryParser serParser, Boolean fCheck, Boolean isCrossAppDomain, IMethodCallMessage
methodCallMessage)
at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream, HeaderHandler handler, Boolean fCheck, Boolean isCrossAppDomain,
IMethodCallMessage methodCallMessage)
at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream)
at Microsoft.Samples.TestV1.Main(String[] args) in c:\Users\andrew\Documents\Visual Studio
2013\Projects\vts\CS\V1 Application\TestV1Part2\TestV1Part2.cs:line 29
at System.AppDomain._nExecuteAssembly(Assembly assembly, String[] args)
at Microsoft.VisualStudio.HostingProcess.HostProc.RunUsersAssembly()
at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback
callback, Object state)
at System.Threading.ThreadHelper.ThreadStart()
```

Why?

The ObjectManager has a different logic to resolve dependencies for arrays and for reference and value types. We added an array of new the reference type which is absent in our assembly.

When ObjectManager attempts to resolve dependencies it builds the graph. When it sees the array, it can not fix it immediately, so that it creates a dummy reference and then fixes the array later.

And since this type is not in the assembly and dependencies can't be fixed. For some reason, it does not remove the array from the list of elements for the fixes and at the end, it throws an exception "IncorrectNumberOfFixups".

It is some 'gotchas' in the process of serialization. For some reason, it does not work correctly only for arrays of new reference types.

```
A Note:
Similar code will work correctly if you do not use arrays with new classes
```

And the first way to fix it and maintain compatibility?

- Use a collection of new structures rather than classes or use a dictionary(possible classes), because a dictionary it's a collection of keyvaluepair(it's structure)
- Use ISerializable, if you can't change the old code

Section 125.4: Making an object serializable

Add the `[Serializable]` attribute to mark an entire object for binary serialization:

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal DontSerializeThis;
```

```
    [OptionalField]
    public string Name;
}
```

All members will be serialized unless we explicitly opt-out using the `[NonSerialized]` attribute. In our example, `X`, `Y`, `Z`, and `Name` are all serialized.

All members are required to be present on deserialization unless marked with `[NonSerialized]` or `[OptionalField]`. In our example, `X`, `Y`, and `Z` are all required and deserialization will fail if they are not present in the stream. `DontSerializeThis` will always be set to `default(decimal)` (which is 0). If `Name` is present in the stream, then it will be set to that value, otherwise it will be set to `default(string)` (which is null). The purpose of `[OptionalField]` is to provide a bit of version tolerance.

Section 125.5: Serialization surrogates (Implementing ISerializationSurrogate)

Implements a serialization surrogate selector that allows one object to perform serialization and deserialization of another

As well allows to properly serialize or deserialize a class that is not itself serializable

Implement ISerializationSurrogate interface

```
public class ItemSurrogate : ISerializationSurrogate
{
    public void GetObjectData(object obj, SerializationInfo info, StreamingContext context)
    {
        var item = (Item)obj;
        info.AddValue("_name", item.Name);
    }

    public object SetObjectData(object obj, SerializationInfo info, StreamingContext context,
ISurrogateSelector selector)
    {
        var item = (Item)obj;
        item.Name = (string)info.GetValue("_name", typeof(string));
        return item;
    }
}
```

Then you need to let your IFormatter know about the surrogates by defining and initializing a SurrogateSelector and assigning it to your IFormatter

```
var surrogateSelector = new SurrogateSelector();
surrogateSelector.AddSurrogate(typeof(Item), new StreamingContext(StreamingContextStates.All), new
ItemSurrogate());
var binaryFormatter = new BinaryFormatter
{
    SurrogateSelector = surrogateSelector
};
```

Even if the class is not marked serializable.

```
//this class is not serializable
public class Item
{
    private string _name;
```

```
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}
```

The complete solution

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class Item
    {
        private string _name;

        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }
    }

    class ItemSurrogate : ISerializationSurrogate
    {
        public void GetObjectData(object obj, SerializationInfo info, StreamingContext context)
        {
            var item = (Item)obj;
            info.AddValue("_name", item.Name);
        }

        public object SetObjectData(object obj, SerializationInfo info, StreamingContext context,
ISurrogateSelector selector)
        {
            var item = (Item)obj;
            item.Name = (string)info.GetValue("_name", typeof(string));
            return item;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var item = new Item
            {
                Name = "Orange"
            };

            var bytes = SerializeData(item);
            var deserializedData = (Item)DeserializeData(bytes);
        }

        private static byte[] SerializeData(object obj)
        {
            var surrogateSelector = new SurrogateSelector();
            surrogateSelector.AddSurrogate(typeof(Item), new
```

```
StreamingContext(StreamingContextStates.All), new ItemSurrogate());

            var binaryFormatter = new BinaryFormatter
            {
                SurrogateSelector = surrogateSelector
            };

            using (var memoryStream = new MemoryStream())
            {
                binaryFormatter.Serialize(memoryStream, obj);
                return memoryStream.ToArray();
            }
        }

        private static object DeserializeData(byte[] bytes)
        {
            var surrogateSelector = new SurrogateSelector();
            surrogateSelector.AddSurrogate(typeof(Item), new
StreamingContext(StreamingContextStates.All), new ItemSurrogate());

            var binaryFormatter = new BinaryFormatter
            {
                SurrogateSelector = surrogateSelector
            };

            using (var memoryStream = new MemoryStream(bytes))
                return binaryFormatter.Deserialize(memoryStream);
        }
    }
}
```

Section 125.6: Adding more control by implementing ISerializable

That would get more control over serialization, how to save and load types

Implement ISerializable interface and create an empty constructor to compile

```
[Serializable]
public class Item : ISerializable
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public Item ()
    {

    }

    protected Item (SerializationInfo info, StreamingContext context)
    {
        _name = (string)info.GetValue("_name", typeof(string));
    }

    public void GetObjectData(SerializationInfo info, StreamingContext context)
```

```
    {
        info.AddValue("_name", _name, typeof(string));
    }
}
```

For data serialization, you can specify the desired name and the desired type

```
info.AddValue("_name", _name, typeof(string));
```

When the data is deserialized, you will be able to read the desired type

```
_name = (string)info.GetValue("_name", typeof(string));
```

Chapter 126: ICloneable

Section 126.1: Implementing ICloneable in a class

Implement `ICloneable` in a class with a twist. Expose a public type safe `Clone()` and implement `object Clone()` privately.

```
public class Person : ICloneable
{
    // Contents of class
    public string Name { get; set; }
    public int Age { get; set; }
    // Constructor
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }
    // Copy Constructor
    public Person(Person other)
    {
        this.Name=other.Name;
        this.Age=other.Age;
    }

    #region ICloneable Members
    // Type safe Clone
    public Person Clone() { return new Person(this); }
    // ICloneable implementation
    object ICloneable.Clone()
    {
        return Clone();
    }
    #endregion
}
```

Later to be used as follows:

```
{
    Person bob=new Person("Bob", 25);
    Person bob_clone=bob.Clone();
    Debug.Assert(bob_clone.Name==bob.Name);

    bob.Age=56;
    Debug.Assert(bob.Age!=bob.Age);
}
```

Notice that changing the age of `bob` does not change the age of `bob_clone`. This is because the design uses cloning instead of assigning of (reference) variables.

Section 126.2: Implementing ICloneable in a struct

The implementation of ICloneable for a struct is not generally needed because structs do a memberwise copy with the assignment operator `=`. But the design might require the implementation of another interface that inherits from `ICloneable`.

Another reason would be if the struct contains a reference type (or an array) which would need copying also.

```
// Structs are recommended to be immutable objects
[ImmutableObject(true)]
public struct Person : ICloneable
{
    // Contents of class
    public string Name { get; private set; }
    public int Age { get; private set; }
    // Constructor
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }
    // Copy Constructor
    public Person(Person other)
    {
        // The assignment operator copies all members
        this=other;
    }

    #region ICloneable Members
    // Type safe Clone
    public Person Clone() { return new Person(this); }
    // ICloneable implementation
    object ICloneable.Clone()
    {
        return Clone();
    }
    #endregion
}
```

Later to be used as follows:

```
static void Main(string[] args)
{
    Person bob=new Person("Bob", 25);
    Person bob_clone=bob.Clone();
    Debug.Assert(bob_clone.Name==bob.Name);
}
```

Chapter 127: IComparable

Section 127.1: Sort versions

Class:

```
public class Version : IComparable<Version>
{
    public int[] Parts { get; }

    public Version(string value)
    {
        if (value == null)
            throw new ArgumentNullException();
        if (!Regex.IsMatch(value, @"^[0-9]+(\.[0-9]+)*$"))
            throw new ArgumentException("Invalid format");
        var parts = value.Split('.');
        Parts = new int[parts.Length];
        for (var i = 0; i < parts.Length; i++)
            Parts[i] = int.Parse(parts[i]);
    }

    public override string ToString()
    {
        return string.Join(".", Parts);
    }

    public int CompareTo(Version that)
    {
        if (that == null) return 1;
        var thisLength = this.Parts.Length;
        var thatLength = that.Parts.Length;
        var maxLength = Math.Max(thisLength, thatLength);
        for (var i = 0; i < maxLength; i++)
        {
            var thisPart = i < thisLength ? this.Parts[i] : 0;
            var thatPart = i < thatLength ? that.Parts[i] : 0;
            if (thisPart < thatPart) return -1;
            if (thisPart > thatPart) return 1;
        }
        return 0;
    }
}
```

Test:

```
Version a, b;

a = new Version("4.2.1");
b = new Version("4.2.6");
a.CompareTo(b); // a < b : -1

a = new Version("2.8.4");
b = new Version("2.8.0");
a.CompareTo(b); // a > b : 1

a = new Version("5.2");
b = null;
a.CompareTo(b); // a > b : 1
```

```
a = new Version("3");
b = new Version("3.6");
a.CompareTo(b); // a < b : -1

var versions = new List<Version>
{
    new Version("2.0"),
    new Version("1.1.5"),
    new Version("3.0.10"),
    new Version("1"),
    null,
    new Version("1.0.1")
};

versions.Sort();

foreach (var version in versions)
    Console.WriteLine(version?.ToString() ?? "NULL");
```

Output:

> NULL
> 1
> 1.0.1
> 1.1.5
> 2.0
> 3.0.10

Demo:

Live demo on Ideone

Chapter 128: Accessing Databases

Section 128.1: Connection Strings

A Connection String is a string that specifies information about a particular data source and how to go about connecting to it by storing credentials, locations, and other information.

```
Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;
```

Storing Your Connection String

Typically, a connection string will be stored within a configuration file (such as an `app.config` or `web.config` within ASP.NET applications). The following is an example of what a local connection might look like within one of these files :

```
<connectionStrings>
   <add name="WidgetsContext" providerName="System.Data.SqlClient"
 connectionString="Server=.\SQLEXPRESS;Database=Widgets;Integrated Security=True;"/>
</connectionStrings>

<connectionStrings>
   <add name="WidgetsContext" providerName="System.Data.SqlClient"
 connectionString="Server=.\SQLEXPRESS;Database=Widgets;Integrated Security=SSPI;"/>
</connectionStrings>
```

This will allow your application to access the connection string programmatically through `WidgetsContext`. Although both `Integrated Security=SSPI` and `Integrated Security=True` perform the same function;`Integrated Security=SSPI` is preferred since works with both SQLClient & OleDB provider where as `Integrated Security=true` throws an exception when used with the OleDb provider.

Different Connections for Different Providers

Each data provider (SQL Server, MySQL, Azure, etc.) all feature their own flavor of syntax for their connection strings and expose different available properties. ConnectionStrings.com is an incredibly useful resource if you are unsure about what yours should look like.

Section 128.2: Entity Framework Connections

Entity Framework exposes abstraction classes that are used to interact with underlying databases in the form of classes like `DbContext`. These contexts generally consist of `DbSet<T>` properties that expose the available collections that can be queried :

```
public class ExampleContext: DbContext
{
    public virtual DbSet<Widgets> Widgets { get; set; }
}
```

The `DbContext` itself will handle making the connections with the databases and will generally read the appropriate Connection String data from a configuration to determine how to establish the connections :

```
public class ExampleContext: DbContext
{
    // The parameter being passed in to the base constructor indicates the name of the
    // connection string
```

```
    public ExampleContext() : base("ExampleContextEntities")
    {
    }

    public virtual DbSet<Widgets> Widgets { get; set; }
}
```

Executing Entity Framework Queries

Actually executing an Entity Framework query can be quite easy and simply requires you to create an instance of the context and then use the available properties on it to pull or access your data

```
using(var context = new ExampleContext())
{
      // Retrieve all of the Widgets in your database
      var data = context.Widgets.ToList();
}
```

Entity Framework also provides an extensive change-tracking system that can be used to handle updating entries within your database by simply calling the `SaveChanges()` method to push changes to the database :

```
using(var context = new ExampleContext())
{
      // Grab the widget you wish to update
      var widget = context.Widgets.Find(w => w.Id == id);
      // If it exists, update it
      if(widget != null)
      {
            // Update your widget and save your changes
            widget.Updated = DateTime.UtcNow;
            context.SaveChanges();
      }
}
```

Section 128.3: ADO.NET Connections

ADO.NET Connections are one of the simplest ways to connect to a database from a C# application. They rely on the use of a provider and a connection string that points to your database to perform queries against.

Common Data Provider Classes

Many of the following are classes that are commonly used to query databases and their related namespaces :

- `SqlConnection`,`SqlCommand`,`SqlDataReader` from `System.Data.SqlClient`
- `OleDbConnection`,`OleDbCommand`,`OleDbDataReader` from `System.Data.OleDb`
- `MySqlConnection`, `MySqlCommand`, `MySqlDbDataReader` from `MySql.Data`

All of these are commonly used to access data through C# and will be commonly encountered throughout building data-centric applications. Many other classes that are not mentioned that implement the same `FooConnection`,`FooCommand`,`FooDataReader` classes can be expected to behave the same way.

Common Access Pattern for ADO.NET Connections

A common pattern that can be used when accessing your data through an ADO.NET connection might look as follows :

```
// This scopes the connection (your specific class may vary)
using(var connection = new SqlConnection("{your-connection-string}")
{
    // Build your query
    var query = "SELECT * FROM YourTable WHERE Property = @property");
    // Scope your command to execute
    using(var command = new SqlCommand(query, connection))
    {
         // Open your connection
         connection.Open();

         // Add your parameters here if necessary

         // Execute your query as a reader (again scoped with a using statement)
         using(var reader = command.ExecuteReader())
         {
               // Iterate through your results here
         }
    }
}
```

Or if you were just performing a simple update and didn't require a reader, the same basic concept would apply :

```
using(var connection = new SqlConnection("{your-connection-string}"))
{
     var query = "UPDATE YourTable SET Property = Value WHERE Foo = @foo";
     using(var command = new SqlCommand(query,connection))
     {
          connection.Open();

          // Add parameters here

          // Perform your update
          command.ExecuteNonQuery();
     }
}
```

You can even program against a set of common interfaces and not have to worry about the provider specific classes. The core interfaces provided by ADO.NET are:

- IDbConnection - for managing database connections
- IDbCommand - for running SQL commands
- IDbTransaction - for managing transactions
- IDataReader - for reading data returned by a command
- IDataAdapter - for channeling data to and from datasets

```
var connectionString = "{your-connection-string}";
var providerName = "{System.Data.SqlClient}"; //for Oracle use "Oracle.ManagedDataAccess.Client"
//most likely you will get the above two from ConnectionStringSettings object

var factory = DbProviderFactories.GetFactory(providerName);

using(var connection = new factory.CreateConnection()) {
    connection.ConnectionString = connectionString;
    connection.Open();

    using(var command = new connection.CreateCommand()) {
        command.CommandText = "{sql-query}";    //this needs to be tailored for each database system

        using(var reader = command.ExecuteReader()) {
```

```
            while(reader.Read()) {
                ...
            }
        }
    }
}
```

Chapter 129: Using SQLite in C#

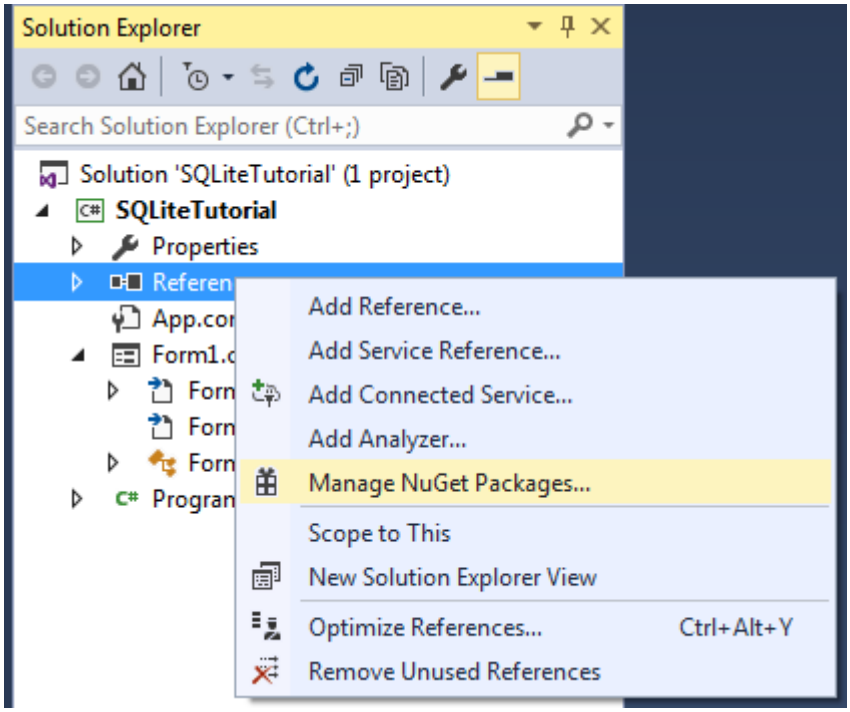
Section 129.1: Creating simple CRUD using SQLite in C#

First of all we need to add SQLite support to our application. There are two ways of doing that

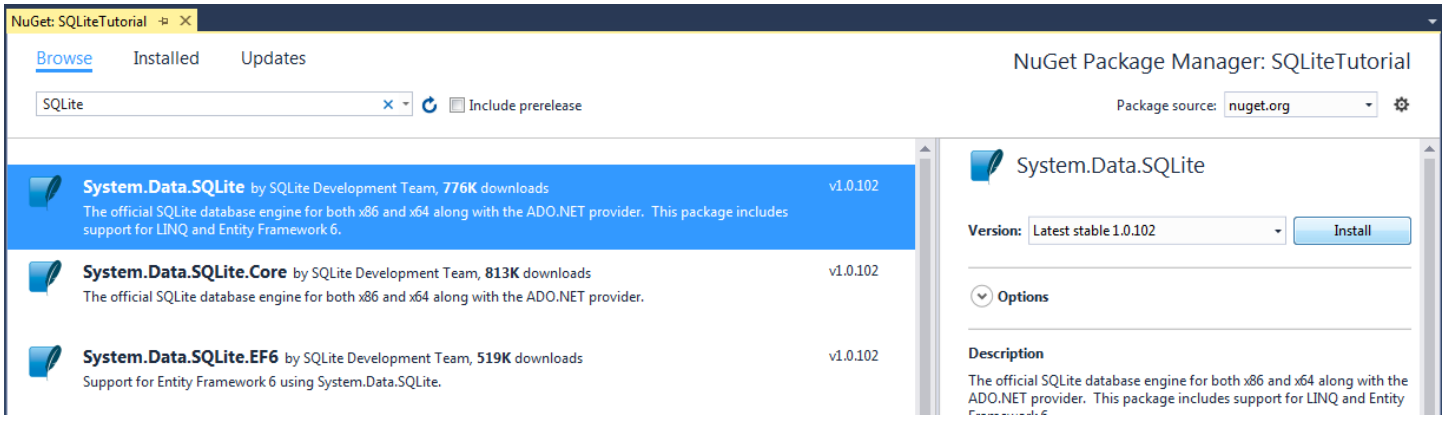
- Download DLL suiting your system from SQLite download page and then add to the project manually
- Add SQLite dependency via NuGet

We'll do it the second way

First open the NuGet menu



and search for **System.Data.SQLite**, select it and hit **Install**



Installation can also be done from Package Manager Console with

```
PM> Install-Package System.Data.SQLite
```

Or for only core features

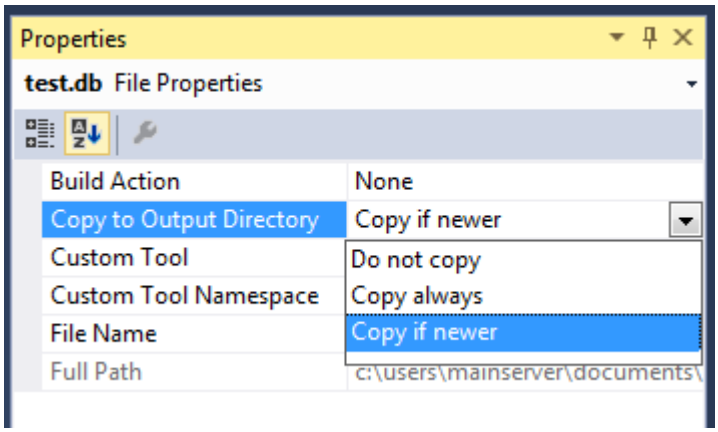
```
PM> Install-Package System.Data.SQLite.Core
```

That's it for the download, so we can go right into coding.

First create a simple SQLite database with this table and add it as a file to the project

```
CREATE TABLE User(
  Id INTEGER PRIMARY KEY AUTOINCREMENT,
  FirstName TEXT NOT NULL,
  LastName TEXT NOT NULL
);
```

Also do not forget to set the **Copy to Output Directory** property of the file to **Copy if newer** of **Copy always**, based on your needs



Create a class called User, which will be the base entity for our database

```
private class User
{
    public string FirstName { get; set; }
    public string Lastname { get; set; }
}
```

We'll write two methods for query execution, first one for inserting, updating or removing from database

```
private int ExecuteWrite(string query, Dictionary<string, object> args)
{
    int numberOfRowsAffected;

    //setup the connection to the database
    using (var con = new SQLiteConnection("Data Source=test.db"))
    {
        con.Open();

        //open a new command
        using (var cmd = new SQLiteCommand(query, con))
        {
            //set the arguments given in the query
            foreach (var pair in args)
            {
                cmd.Parameters.AddWithValue(pair.Key, pair.Value);
            }

            //execute the query and get the number of row affected
            numberOfRowsAffected = cmd.ExecuteNonQuery();
        }

        return numberOfRowsAffected;
```

```
    }
}
```

and the second one for reading from database

```
private DataTable Execute(string query)
{
    if (string.IsNullOrEmpty(query.Trim()))
        return null;

    using (var con = new SQLiteConnection("Data Source=test.db"))
    {
        con.Open();
        using (var cmd = new SQLiteCommand(query, con))
        {
            foreach (KeyValuePair<string, object> entry in args)
            {
                cmd.Parameters.AddWithValue(entry.Key, entry.Value);
            }

            var da = new SQLiteDataAdapter(cmd);

            var dt = new DataTable();
            da.Fill(dt);

            da.Dispose();
            return dt;
        }
    }
}
```

Now lets get into our **CRUD** methods

Adding user

```
private int AddUser(User user)
{
    const string query = "INSERT INTO User(FirstName, LastName) VALUES(@firstName, @lastName)";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@firstName", user.FirstName},
        {"@lastName", user.Lastname}
    };

    return ExecuteWrite(query, args);
}
```

Editing user

```
private int EditUser(User user)
{
    const string query = "UPDATE User SET FirstName = @firstName, LastName = @lastName WHERE Id =
@id";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
```

```
    {
        {"@id", user.Id},
        {"@firstName", user.FirstName},
        {"@lastName", user.Lastname}
    };

    return ExecuteWrite(query, args);
}
```

Deleting user

```
private int DeleteUser(User user)
{
    const string query = "Delete from User WHERE Id = @id";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@id", user.Id}
    };

    return ExecuteWrite(query, args);
}
```

Getting user by `Id`

```
private User GetUserById(int id)
{
    var query = "SELECT * FROM User WHERE Id = @id";

    var args = new Dictionary<string, object>
    {
        {"@id", id}
    };

    DataTable dt = ExecuteRead(query, args);

    if (dt == null || dt.Rows.Count == 0)
    {
        return null;
    }

    var user = new User
    {
        Id = Convert.ToInt32(dt.Rows[0]["Id"]),
        FirstName = Convert.ToString(dt.Rows[0]["FirstName"]),
        Lastname = Convert.ToString(dt.Rows[0]["LastName"])
    };

    return user;
}
```

Section 129.2: Executing Query

```
using (SQLiteConnection conn = new SQLiteConnection(@"Data
Source=data.db;Pooling=true;FailIfMissing=false"))
{
    conn.Open();
    using (SQLiteCommand cmd = new SQLiteCommand(conn))
```

```
    {
        cmd.CommandText = "query";
        using (SqlDataReader dr = cmd.ExecuteReader())
        {
            while(dr.Read())
            {
                //do stuff
            }
        }
    }
}
```

Note: Setting `FailIfMissing` to true creates the file `data.db` if missing. However, the file will be empty. So, any required tables have to be recreated.

Chapter 130: Caching

Section 130.1: MemoryCache

```
//Get instance of cache
using System.Runtime.Caching;

var cache = MemoryCache.Default;

//Check if cache contains an item with
cache.Contains("CacheKey");

//get item from cache
var item = cache.Get("CacheKey");

//get item from cache or add item if not existing
object list = MemoryCache.Default.AddOrGetExisting("CacheKey", "object to be stored",
DateTime.Now.AddHours(12));

//note if item not existing the item is added by this method
//but the method returns null
```

Chapter 131: Code Contracts

Section 131.1: Postconditions

```
public double GetPaymentsTotal(string name)
{
    Contract.Ensures(Contract.Result<double>() >= 0);

    double total = 0.0;

    foreach (var payment in this._payments) {
        if (string.Equals(payment.Name, name)) {
            total += payment.Amount;
        }
    }

    return total;
}
```

Section 131.2: Invariants

```
namespace CodeContractsDemo
{
    using System;
    using System.Diagnostics.Contracts;

    public class Point
    {
        public int X { get; set; }
        public int Y { get; set; }

        public Point()
        {
        }

        public Point(int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public void Set(int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public void Test(int x, int y)
        {
            for (int dx = -x; dx <= x; dx++) {
                this.X = dx;
                Console.WriteLine("Current X = {0}", this.X);
            }

            for (int dy = -y; dy <= y; dy++) {
                this.Y = dy;
                Console.WriteLine("Current Y = {0}", this.Y);
            }
```

```
            Console.WriteLine("X = {0}", this.X);
            Console.WriteLine("Y = {0}", this.Y);
        }

        [ContractInvariantMethod]
        private void ValidateCoordinates()
        {
            Contract.Invariant(this.X >= 0);
            Contract.Invariant(this.Y >= 0);
        }
    }
}
```

Section 131.3: Defining Contracts on Interface

```
[ContractClass(typeof(ValidationContract))]
interface IValidation
{
    string CustomerID{get;set;}
    string Password{get;set;}
}

[ContractClassFor(typeof(IValidation))]
sealed class ValidationContract:IValidation
{
    string IValidation.CustomerID
    {
        [Pure]
        get
        {
            return Contract.Result<string>();
        }
        set
        {
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "Customer ID
cannot be null!!");
        }
    }

    string IValidation.Password
    {
        [Pure]
        get
        {
            return Contract.Result<string>();
        }
        set
        {
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "Password cannot
be null!!");
        }
    }
}

class Validation:IValidation
{
    public string GetCustomerPassword(string customerID)
    {
        Contract.Requires(!string.IsNullOrEmpty(customerID),"Customer ID cannot be Null");
        Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(customerID), "Exception!!");
        Contract.Ensures(Contract.Result<string>() != null);
```

```
        string password="AAA@1234";
        if (customerID!=null)
        {
            return password;
        }
        else
        {
            return null;
        }

    }

    private string m_custID, m_PWD;

    public string CustomerID
    {
        get
        {
            return m_custID;
        }
        set
        {
            m_custID = value;
        }
    }

    public string Password
    {
        get
        {
            return m_PWD;
        }
        set
        {
            m_PWD = value;
        }
    }
}
```

In the above code, we have defined an interface called `IValidation` with an attribute `[ContractClass]`. This attribute takes an address of a class where we have implemented a contract for an Interface. The class `ValidationContract` makes use of properties defined in the interface and checks for the null values using `Contract.Requires<T>`. `T` is an exception class.

We have also marked the get accessor with an attribute `[Pure]`. The pure attribute ensures that the method or a property does not change the instance state of a class in which `IValidation` interface is implemented.

Section 131.4: Preconditions

```
namespace CodeContractsDemo
{
    using System;
    using System.Collections.Generic;
    using System.Diagnostics.Contracts;

    public class PaymentProcessor
    {
        private List<Payment> _payments = new List<Payment>();

        public void Add(Payment payment)
```

```
        {
            Contract.Requires(payment != null);
            Contract.Requires(!string.IsNullOrEmpty(payment.Name));
            Contract.Requires(payment.Date <= DateTime.Now);
            Contract.Requires(payment.Amount > 0);

            this._payments.Add(payment);
        }
    }
}
```

Chapter 132: Code Contracts and Assertions

Section 132.1: Assertions to check logic should always be true

Assertions are used not to perform testing of input parameters, but to verify that program flow is corect -- i.e., that you can make certain assumptions about your code at a certain point in time. In other words: a test done with `Debug.Assert` should *always* assume that the value tested is `true`.

Debug.Assert only executes in DEBUG builds; it is filtered out of RELEASE builds. It must be considered a debugging tool in addition to unit testing and not as a replacement of code contracts or input validation methods.

For instance, this is a good assertion:

```
var systemData = RetrieveSystemConfiguration();
Debug.Assert(systemData != null);
```

Here assert is a good choice because we can assume that RetrieveSystemConfiguration() will return a valid value and will never return null.

Here is another good example:

```
UserData user = RetrieveUserData();
Debug.Assert(user != null);
Debug.Assert(user.Age > 0);
int year = DateTime.Today.Year - user.Age;
```

First, we may assume that RetrieveUserData() will return a valid value. Then, before using the Age property, we verify the assumption (which should always be true) that the age of the user is strictly positive.

This is a bad example of assert:

```
string input = Console.ReadLine();
int age = Convert.ToInt32(input);
Debug.Assert(age > 16);
Console.WriteLine("Great, you are over 16");
```

Assert is not for input validation because it is incorrect to assume that this assertion will always be true. You must use input validation methods for that. In the case above, you should also verify that the input value is a number in the first place.

Chapter 133: Structural Design Patterns

Structural design patterns are patterns that describe how objects and classes can be combined and form a large structure and that ease design by identifying a simple way to realize relationships between entities. There are seven structural patterns described. They are as follows: Adapter, Bridge, Composite, Decorator, Facade, Flyweight and Proxy

Section 133.1: Adapter Design Pattern

> "**Adapter**" as the name suggests is the object which lets two mutually incompatible interfaces communicate with each other.

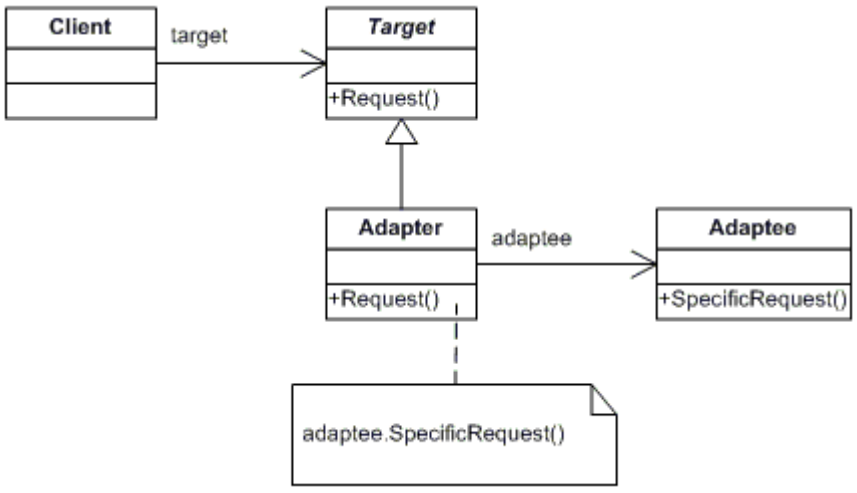
> **For example:** if you buy a Iphone 8 (or any other Apple product) you need alot of adapters. Because the default interface does not support audio jac or USB. With these adapters you can use earphones with wires or you can use a normal Ethernet cable. So *"two mutually incompatible interfaces communicate with each other"*.

> **So in technical terms this means:** Convert the interface of a class into another interface that a clients expect. Adapter let classes work together that couldn't otherwise because of incompatible interfaces. The classes and objects participating in this pattern are:

The adapter pattern exits out 4 elements

> 1. **ITarget:** This is the interface which is used by the client to achieve functionality.
> 2. **Adaptee:** This is the functionality which the client desires but its interface is not compatible with the client.
> 3. **Client:** This is the class which wants to achieve some functionality by using the adaptee's code.
> 4. **Adapter:** This is the class which would implement ITarget and would call the Adaptee code which the client wants to call.

UML



First code Example (Theoretical example).

```
public interface ITarget
{
    void MethodA();
}

public class Adaptee
{
    public void MethodB()
    {
        Console.WriteLine("MethodB() is called");
    }
}

public class Client
{
    private ITarget target;

    public Client(ITarget target)
    {
        this.target = target;
    }

    public void MakeRequest()
    {
        target.MethodA();
    }
}

public class Adapter : Adaptee, ITarget
{
    public void MethodA()
    {
        MethodB();
    }
}
```

Second code example (Real world imlementation)

```
/// <summary>
///  Interface: This is the interface which is used by the client to achieve functionality.
/// </summary>
public interface ITarget
{
    List<string> GetEmployeeList();
}

/// <summary>
/// Adaptee: This is the functionality which the client desires but its interface is not compatible
with the client.
/// </summary>
public class CompanyEmplyees
{
    public string[][] GetEmployees()
    {
        string[][] employees = new string[4][];

        employees[0] = new string[] { "100", "Deepak", "Team Leader" };
        employees[1] = new string[] { "101", "Rohit", "Developer" };
        employees[2] = new string[] { "102", "Gautam", "Developer" };
        employees[3] = new string[] { "103", "Dev", "Tester" };
```

```
        return employees;
    }
}

/// <summary>
/// Client: This is the class which wants to achieve some functionality by using the adaptee's code
(list of employees).
/// </summary>
public class ThirdPartyBillingSystem
{
    /*
     * This class is from a thirt party and you do'n have any control over it.
     * But it requires a Emplyee list to do its work
     */

    private ITarget employeeSource;

    public ThirdPartyBillingSystem(ITarget employeeSource)
    {
        this.employeeSource = employeeSource;
    }

    public void ShowEmployeeList()
    {
        // call the clietn list in the interface
        List<string> employee = employeeSource.GetEmployeeList();

        Console.WriteLine("######### Employee List ##########");
        foreach (var item in employee)
        {
            Console.Write(item);
        }

    }
}

/// <summary>
/// Adapter: This is the class which would implement ITarget and would call the Adaptee code which
the client wants to call.
/// </summary>
public class EmployeeAdapter : CompanyEmplyees, ITarget
{
    public List<string> GetEmployeeList()
    {
        List<string> employeeList = new List<string>();
        string[][] employees = GetEmployees();
        foreach (string[] employee in employees)
        {
            employeeList.Add(employee[0]);
            employeeList.Add(",");
            employeeList.Add(employee[1]);
            employeeList.Add(",");
            employeeList.Add(employee[2]);
            employeeList.Add("\n");
        }

        return employeeList;
    }
}

///
/// Demo
```

```
///
class Programs
{
    static void Main(string[] args)
    {
        ITarget Itarget = new EmployeeAdapter();
        ThirdPartyBillingSystem client = new ThirdPartyBillingSystem(Itarget);
        client.ShowEmployeeList();
        Console.ReadKey();
    }
}
```

When to use

- Allow a system to use classes of another system that is incompatible with it.
- Allow communication between new and already existing system which are independent to each other
- Ado.Net SqlAdapter, OracleAdapter, MySqlAdapter are best example of Adapter Pattern.

Chapter 134: Creational Design Patterns

Section 134.1: Singleton Pattern

The Singleton pattern is designed to restrict creation of a class to exactly one single instance.

This pattern is used in a scenario where it makes sense to have only one of something, such as:

- a single class that orchestrates other objects' interactions, ex. Manager class
- or one class that represents a unique, single resource, ex. Logging component

One of the most common ways to implement the Singleton pattern is via a static **factory method** such as a `CreateInstance()` or `GetInstance()` (or a static property in C#, `Instance`), which is then designed to always return the same instance.

The first call to the method or property creates and returns the Singleton instance. Thereafter, the method always returns the same instance. This way, there is only ever one instance of the singleton object.

Preventing creation of instances via `new` can be accomplished by making the class constructor(s) `private`.

Here is a typical code example for implementing a Singleton pattern in C#:

```
class Singleton
{
    // Because the _instance member is made private, the only way to get the single
    // instance is via the static Instance property below. This can also be similarly
    // achieved with a GetInstance() method instead of the property.
    private static Singleton _instance = null;

    // Making the constructor private prevents other instances from being
    // created via something like Singleton s = new Singleton(), protecting
    // against unintentional misuse.
    private Singleton()
    {
    }

    public static Singleton Instance
    {
        get
        {
            // The first call will create the one and only instance.
            if (_instance == null)
            {
                _instance = new Singleton();
            }

            // Every call afterwards will return the single instance created above.
            return _instance;
        }
    }
}
```

To illustrate this pattern further, the code below checks whether an identical instance of the Singleton is returned when the Instance property is called more than once.

```
class Program
{
```

```
    static void Main(string[] args)
    {
        Singleton s1 = Singleton.Instance;
        Singleton s2 = Singleton.Instance;

        // Both Singleton objects above should now reference the same Singleton instance.
        if (Object.ReferenceEquals(s1, s2))
        {
            Console.WriteLine("Singleton is working");
        }
        else
        {
            // Otherwise, the Singleton Instance property is returning something
            // other than the unique, single instance when called.
            Console.WriteLine("Singleton is broken");
        }
    }
}
```

Note: this implementation is not thread safe.

To see more examples, including how to make this thread-safe, visit: Singleton Implementation

Singletons are conceptually similar to a global value, and cause similar design flaws and concerns. Because of this, the Singleton pattern is widely regarded as an anti-pattern.

Visit "What is so bad about Singletons?" for more information on the problems that arise with their use.

In C#, you have the ability to make a class `static`, which makes all members static, and the class cannot be instantiated. Given this, it is common to see static classes used in place of the Singleton pattern.

For key differences between the two, visit C# Singleton Pattern Versus Static Class.

Section 134.2: Factory Method pattern

Factory Method is one of creational design patterns. It is used to deal with the problem of creating objects without specifying exact result type. This document will teach you how to use Factory Method DP properly.

Let me explain the idea of it to you on a simple example. Imagine you're working in a factory that produces three types of devices - Ammeter, Voltmeter and resistance meter. You are writing a program for a central computer that will create selected device, but you don't know final decision of your boss on what to produce.

Let's create an interface `IDevice` with some common functions that all devices have:

```
public interface IDevice
{
    int Measure();
    void TurnOff();
    void TurnOn();
}
```

Now, we can create classes that represent our devices. Those classes must implement `IDevice` interface:

```
public class AmMeter : IDevice
{
    private Random r = null;
    public AmMeter()
    {
```

```
        r = new Random();
    }
    public int Measure() { return r.Next(-25, 60); }
    public void TurnOff() { Console.WriteLine("AmMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("AmMeter turns on..."); }
}
public class OhmMeter : IDevice
{
    private Random r = null;
    public OhmMeter()
    {
        r = new Random();
    }
    public int Measure() { return r.Next(0, 1000000); }
    public void TurnOff() { Console.WriteLine("OhmMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("OhmMeter turns on..."); }
}
public class VoltMeter : IDevice
{
    private Random r = null;
    public VoltMeter()
    {
        r = new Random();
    }
    public int Measure() { return r.Next(-230, 230); }
    public void TurnOff() { Console.WriteLine("VoltMeter flashes lights saying good bye!"); }
    public void TurnOn() { Console.WriteLine("VoltMeter turns on..."); }
}
```

Now we have to define factory method. Let's create `DeviceFactory` class with static method inside:

```
public enum Device
{
    AM,
    VOLT,
    OHM
}
public class DeviceFactory
{
    public static IDevice CreateDevice(Device d)
    {
        switch(d)
        {
            case Device.AM: return new AmMeter();
            case Device.VOLT: return new VoltMeter();
            case Device.OHM: return new OhmMeter();
            default: return new AmMeter();
        }
    }
}
```

Great! Let's test our code:

```
public class Program
{
    static void Main(string[] args)
    {
        IDevice device = DeviceFactory.CreateDevice(Device.AM);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
```

```
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        device.TurnOff();
        Console.WriteLine();

        device = DeviceFactory.CreateDevice(Device.VOLT);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        device.TurnOff();
        Console.WriteLine();

        device = DeviceFactory.CreateDevice(Device.OHM);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
        device.TurnOff();
        Console.WriteLine();
    }
}
```

This is the example output you might see after running this code:

> AmMeter turns on...
>
> 36
>
> 6
>
> 33
>
> 43
>
> 24
>
> AmMeter flashes lights saying good bye!
>
> VoltMeter turns on...
>
> 102
>
> -61
>
> 85
>
> 138
>
> 36
>
> VoltMeter flashes lights saying good bye!
>
> OhmMeter turns on...

> 723828
>
> 368536
>
> 685412
>
> 800266
>
> 578595
>
> OhmMeter flashes lights saying good bye!

Section 134.3: Abstract Factory Pattern

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

In this example demonstrates the creation of different animal worlds for a computer game using different factories. Although the animals created by the Continent factories are different, the interactions among the animals remain the same.

```
using System;

namespace GangOfFour.AbstractFactory
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Abstract Factory Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
      // Create and run the African animal world
      ContinentFactory africa = new AfricaFactory();
      AnimalWorld world = new AnimalWorld(africa);
      world.RunFoodChain();

      // Create and run the American animal world
      ContinentFactory america = new AmericaFactory();
      world = new AnimalWorld(america);
      world.RunFoodChain();

      // Wait for user input
      Console.ReadKey();
    }
  }


  /// <summary>
  /// The 'AbstractFactory' abstract class
  /// </summary>
  abstract class ContinentFactory
  {
    public abstract Herbivore CreateHerbivore();
    public abstract Carnivore CreateCarnivore();
  }
```

```
/// <summary>
/// The 'ConcreteFactory1' class
/// </summary>
class AfricaFactory : ContinentFactory
{
  public override Herbivore CreateHerbivore()
  {
    return new Wildebeest();
  }
  public override Carnivore CreateCarnivore()
  {
    return new Lion();
  }
}

/// <summary>
/// The 'ConcreteFactory2' class
/// </summary>
class AmericaFactory : ContinentFactory
{
  public override Herbivore CreateHerbivore()
  {
    return new Bison();
  }
  public override Carnivore CreateCarnivore()
  {
    return new Wolf();
  }
}

/// <summary>
/// The 'AbstractProductA' abstract class
/// </summary>
abstract class Herbivore
{
}

/// <summary>
/// The 'AbstractProductB' abstract class
/// </summary>
abstract class Carnivore
{
  public abstract void Eat(Herbivore h);
}

/// <summary>
/// The 'ProductA1' class
/// </summary>
class Wildebeest : Herbivore
{
}

/// <summary>
/// The 'ProductB1' class
/// </summary>
class Lion : Carnivore
{
  public override void Eat(Herbivore h)
  {
    // Eat Wildebeest
    Console.WriteLine(this.GetType().Name +
```

```
        " eats " + h.GetType().Name);
    }
  }

  /// <summary>
  /// The 'ProductA2' class
  /// </summary>
  class Bison : Herbivore
  {
  }

  /// <summary>
  /// The 'ProductB2' class
  /// </summary>
  class Wolf : Carnivore
  {
    public override void Eat(Herbivore h)
    {
      // Eat Bison
      Console.WriteLine(this.GetType().Name +
        " eats " + h.GetType().Name);
    }
  }

  /// <summary>
  /// The 'Client' class
  /// </summary>
  class AnimalWorld
  {
    private Herbivore _herbivore;
    private Carnivore _carnivore;

    // Constructor
    public AnimalWorld(ContinentFactory factory)
    {
      _carnivore = factory.CreateCarnivore();
      _herbivore = factory.CreateHerbivore();
    }

    public void RunFoodChain()
    {
      _carnivore.Eat(_herbivore);
    }
  }
}
```

Output:

> Lion eats Wildebeest
>
> Wolf eats Bison

Section 134.4: Builder Pattern

Separate the construction of a complex object from its representation so that the same construction process can create different representations and and provides a high level of control over the assembly of the objects.

In this example demonstrates the Builder pattern in which different vehicles are assembled in a step-by-step

fashion. The Shop uses VehicleBuilders to construct a variety of Vehicles in a series of sequential steps.

```
using System;
using System.Collections.Generic;

namespace GangOfFour.Builder
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Builder Design Pattern.
  /// </summary>
  public class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
      VehicleBuilder builder;

      // Create shop with vehicle builders
      Shop shop = new Shop();

      // Construct and display vehicles
      builder = new ScooterBuilder();
      shop.Construct(builder);
      builder.Vehicle.Show();

      builder = new CarBuilder();
      shop.Construct(builder);
      builder.Vehicle.Show();

      builder = new MotorCycleBuilder();
      shop.Construct(builder);
      builder.Vehicle.Show();

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Director' class
  /// </summary>
  class Shop
  {
    // Builder uses a complex series of steps
    public void Construct(VehicleBuilder vehicleBuilder)
    {
      vehicleBuilder.BuildFrame();
      vehicleBuilder.BuildEngine();
      vehicleBuilder.BuildWheels();
      vehicleBuilder.BuildDoors();
    }
  }

  /// <summary>
  /// The 'Builder' abstract class
  /// </summary>
  abstract class VehicleBuilder
  {
    protected Vehicle vehicle;
```

```
  // Gets vehicle instance
  public Vehicle Vehicle
  {
    get { return vehicle; }
  }

  // Abstract build methods
  public abstract void BuildFrame();
  public abstract void BuildEngine();
  public abstract void BuildWheels();
  public abstract void BuildDoors();
}

/// <summary>
/// The 'ConcreteBuilder1' class
/// </summary>
class MotorCycleBuilder : VehicleBuilder
{
  public MotorCycleBuilder()
  {
    vehicle = new Vehicle("MotorCycle");
  }

  public override void BuildFrame()
  {
    vehicle["frame"] = "MotorCycle Frame";
  }

  public override void BuildEngine()
  {
    vehicle["engine"] = "500 cc";
  }

  public override void BuildWheels()
  {
    vehicle["wheels"] = "2";
  }

  public override void BuildDoors()
  {
    vehicle["doors"] = "0";
  }
}


/// <summary>
/// The 'ConcreteBuilder2' class
/// </summary>
class CarBuilder : VehicleBuilder
{
  public CarBuilder()
  {
    vehicle = new Vehicle("Car");
  }

  public override void BuildFrame()
  {
    vehicle["frame"] = "Car Frame";
  }

  public override void BuildEngine()
```

```
    {
      vehicle["engine"] = "2500 cc";
    }

    public override void BuildWheels()
    {
      vehicle["wheels"] = "4";
    }

    public override void BuildDoors()
    {
      vehicle["doors"] = "4";
    }
  }

  /// <summary>
  /// The 'ConcreteBuilder3' class
  /// </summary>
  class ScooterBuilder : VehicleBuilder
  {
    public ScooterBuilder()
    {
      vehicle = new Vehicle("Scooter");
    }

    public override void BuildFrame()
    {
      vehicle["frame"] = "Scooter Frame";
    }

    public override void BuildEngine()
    {
      vehicle["engine"] = "50 cc";
    }

    public override void BuildWheels()
    {
      vehicle["wheels"] = "2";
    }

    public override void BuildDoors()
    {
      vehicle["doors"] = "0";
    }
  }

  /// <summary>
  /// The 'Product' class
  /// </summary>
  class Vehicle
  {
    private string _vehicleType;
    private Dictionary<string,string> _parts =
      new Dictionary<string,string>();

    // Constructor
    public Vehicle(string vehicleType)
    {
      this._vehicleType = vehicleType;
    }

    // Indexer
```

```
    public string this[string key]
    {
      get { return _parts[key]; }
      set { _parts[key] = value; }
    }

    public void Show()
    {
      Console.WriteLine("\n---------------------------");
      Console.WriteLine("Vehicle Type: {0}", _vehicleType);
      Console.WriteLine(" Frame : {0}", _parts["frame"]);
      Console.WriteLine(" Engine : {0}", _parts["engine"]);
      Console.WriteLine(" #Wheels: {0}", _parts["wheels"]);
      Console.WriteLine(" #Doors : {0}", _parts["doors"]);
    }
  }
}
```

Output

> Vehicle Type: Scooter Frame : Scooter Frame
> Engine : none
> #Wheels: 2
> #Doors : 0
>
> Vehicle Type: Car
> Frame : Car Frame
> Engine : 2500 cc
> #Wheels: 4
> #Doors : 4
>
> Vehicle Type: MotorCycle
> Frame : MotorCycle Frame
> Engine : 500 cc
> #Wheels: 2
> #Doors : 0

Section 134.5: Prototype Pattern

Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.

In this example demonstrates the Prototype pattern in which new Color objects are created by copying pre-existing, user-defined Colors of the same type.

```
using System;
using System.Collections.Generic;

namespace GangOfFour.Prototype
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Prototype Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
```

```
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      ColorManager colormanager = new ColorManager();

      // Initialize with standard colors
      colormanager["red"] = new Color(255, 0, 0);
      colormanager["green"] = new Color(0, 255, 0);
      colormanager["blue"] = new Color(0, 0, 255);

      // User adds personalized colors
      colormanager["angry"] = new Color(255, 54, 0);
      colormanager["peace"] = new Color(128, 211, 128);
      colormanager["flame"] = new Color(211, 34, 20);

      // User clones selected colors
      Color color1 = colormanager["red"].Clone() as Color;
      Color color2 = colormanager["peace"].Clone() as Color;
      Color color3 = colormanager["flame"].Clone() as Color;

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Prototype' abstract class
  /// </summary>
  abstract class ColorPrototype
  {
    public abstract ColorPrototype Clone();
  }

  /// <summary>
  /// The 'ConcretePrototype' class
  /// </summary>
  class Color : ColorPrototype
  {
    private int _red;
    private int _green;
    private int _blue;

    // Constructor
    public Color(int red, int green, int blue)
    {
      this._red = red;
      this._green = green;
      this._blue = blue;
    }

    // Create a shallow copy
    public override ColorPrototype Clone()
    {
      Console.WriteLine(
        "Cloning color RGB: {0,3},{1,3},{2,3}",
        _red, _green, _blue);

      return this.MemberwiseClone() as ColorPrototype;
    }
  }
```

```
  /// <summary>
  /// Prototype manager
  /// </summary>
  class ColorManager
  {
    private Dictionary<string, ColorPrototype> _colors =
      new Dictionary<string, ColorPrototype>();

    // Indexer
    public ColorPrototype this[string key]
    {
      get { return _colors[key]; }
      set { _colors.Add(key, value); }
    }
  }
}
```

Output:

> Cloning color RGB: 255, 0, 0
>
> Cloning color RGB: 128,211,128
>
> Cloning color RGB: 211, 34, 20

Chapter 135: Implementing Decorator Design Pattern

Section 135.1: Simulating cafeteria

Decorator is one of structural design patterns. It is used to add, remove or change behaviour of object. This document will teach you how to use Decorator DP properly.

Let me explain the idea of it to you on a simple example. Imagine you're now in Starbobs, famous coffee company. You can place an order for any coffee you want - with cream and sugar, with cream and topping and much more combinations! But, the base of all drinks is coffee - dark, bitter drink, you can modify. Let's write a simple program that simulates coffee machine.

First, we need to create and abstract class that describes our base drink:

```
public abstract class AbstractCoffee
{
    protected AbstractCoffee k = null;

    public AbstractCoffee(AbstractCoffee k)
    {
        this.k = k;
    }

    public abstract string ShowCoffee();
}
```

Now, let's create some extras, like sugar, milk and topping. Created classes must implement `AbstractCoffee` - they will decorate it:

```
public class Milk : AbstractCoffee
{
    public Milk(AbstractCoffee c) : base(c) { }
    public override string ShowCoffee()
    {
        if (k != null)
            return k.ShowCoffee() + " with Milk";
        else return "Milk";
    }
}
public class Sugar : AbstractCoffee
{
    public Sugar(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Sugar";
        else return "Sugar";
    }
}
public class Topping : AbstractCoffee
{
    public Topping(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Topping";
```

```
        else return "Topping";
    }
}
```

Now we can create our favourite coffee:

```
public class Program
{
    public static void Main(string[] args)
    {
        AbstractCoffee coffee = null; //we cant create instance of abstract class
        coffee = new Topping(coffee); //passing null
        coffee = new Sugar(coffee); //passing topping instance
        coffee = new Milk(coffee);  //passing sugar
        Console.WriteLine("Coffee with " + coffee.ShowCoffee());

    }
}
```

Running the code will produce the following output:

> Coffee with Topping with Sugar with Milk

Chapter 136: Implementing Flyweight Design Pattern

Section 136.1: Implementing map in RPG game

Flyweight is one of structural design patterns. It is used to decrease the amount of used memory by sharing as much data as possible with similiar objects. This document will teach you how to use Flyweight DP properly.

Let me explain the idea of it to you on a simple example. Imagine you're working on a RPG game and you need to load huge file that contains some characters. For example:

- `#` is grass. You can walk on it.
- `$` is starting point
- `@` is rock. You can't walk on it.
- `%` is treasure chest

Sample of a map:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@############@@@@@######@#$@@@

@#############@@@######@###@@@

@#######%######@###########@@@

@############################@

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Since those objects have similiar characteristic, you don't need to create separate object for each map field. I will show you how to use flyweight.

Let's define an interface which our fields will implement:

```
public interface IField
{
    string Name { get; }
    char Mark { get; }
    bool CanWalk { get; }
    FieldType Type { get; }
}
```

Now we can create classes that represent our fields. We also have to identify them somehow (I used an enumeration):

```
public enum FieldType
{
    GRASS,
    ROCK,
    START,
    CHEST
}
public class Grass : IField
{
    public string Name { get { return "Grass"; } }
    public char Mark { get { return '#'; } }
```

```
    public bool CanWalk { get { return true; } }
    public FieldType Type { get { return FieldType.GRASS; } }
}
public class StartingPoint : IField
{
    public string Name { get { return "Starting Point"; } }
    public char Mark { get { return '$'; } }
    public bool CanWalk { get { return true; } }
    public FieldType Type { get { return FieldType.START; } }
}
public class Rock : IField
{
    public string Name { get { return "Rock"; } }
    public char Mark { get { return '@'; } }
    public bool CanWalk { get { return false; } }
    public FieldType Type { get { return FieldType.ROCK; } }
}
public class TreasureChest : IField
{
    public string Name { get { return "Treasure Chest"; } }
    public char Mark { get { return '%'; } }
    public bool CanWalk { get { return true; } } // you can approach it
    public FieldType Type { get { return FieldType.CHEST; } }
}
```

Like I said, we don't need to create separate instance for each field. We have to create a **repository** of fields. The essence of Flyweight DP is that we dynamically create an object only if we need it and it doesn't exist yet in our repo, or return it if it already exists. Let's write simple class that will handle this for us:

```
public class FieldRepository
{
    private List<IField> lstFields = new List<IField>();

    private IField AddField(FieldType type)
    {
        IField f;
        switch(type)
        {
            case FieldType.GRASS: f = new Grass(); break;
            case FieldType.ROCK: f = new Rock(); break;
            case FieldType.START: f = new StartingPoint(); break;
            case FieldType.CHEST:
            default: f = new TreasureChest(); break;
        }
        lstFields.Add(f); //add it to repository
        Console.WriteLine("Created new instance of {0}", f.Name);
        return f;
    }
    public IField GetField(FieldType type)
    {
        IField f = lstFields.Find(x => x.Type == type);
        if (f != null) return f;
        else return AddField(type);
    }
}
```

Great! Now we can test our code:

```
public class Program
{
```

```
    public static void Main(string[] args)
    {
        FieldRepository f = new FieldRepository();
        IField grass = f.GetField(FieldType.GRASS);
        grass = f.GetField(FieldType.ROCK);
        grass = f.GetField(FieldType.GRASS);
    }
}
```

The result in the console should be:

> Created a new instance of Grass
>
> Created a new instance of Rock

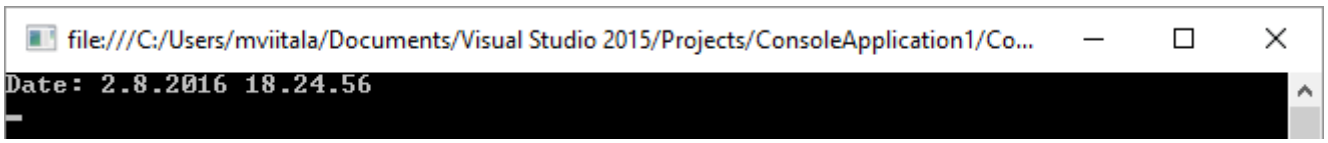
But why grass appears only one time if we wanted to get it twice? That's because first time we call `GetField` grass instance does not exist in our **repository**, so it's created, but next time we need grass it already exist, so we only return it.

Chapter 137: System.Management.Automation

Section 137.1: Invoke simple synchronous pipeline

Get the current date and time.

```
public class Program
{
    static void Main()
    {
        // create empty pipeline
        PowerShell ps = PowerShell.Create();

        // add command
        ps.AddCommand("Get-Date");

        // run command(s)
        Console.WriteLine("Date: {0}", ps.Invoke().First());

        Console.ReadLine();
    }
}
```



Chapter 138: System.DirectoryServices.Protocols.LdapConnection

Section 138.1: Authenticated SSL LDAP connection, SSL cert does not match reverse DNS

Set up some constants for the server and authentication information. Assuming LDAPv3, but it's easy enough to change that.

```
// Authentication, and the name of the server.
private const string LDAPUser = "cn=example:app:mygroup:accts,ou=Applications,dc=example,dc=com";
private readonly char[] password = { 'p', 'a', 's', 's', 'w', 'o', 'r', 'd' };
private const string TargetServer = "ldap.example.com";

// Specific to your company. Might start "cn=manager" instead of "ou=people", for example.
private const string CompanyDN = "ou=people,dc=example,dc=com";
```

Actually create the connection with three parts: an LdapDirectoryIdentifier (the server), and NetworkCredentials.

```
// Configure server and port. LDAP w/ SSL, aka LDAPS, uses port 636.
// If you don't have SSL, don't give it the SSL port.
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer, 636);

// Configure network credentials (userid and password)
var secureString = new SecureString();
foreach (var character in password)
        secureString.AppendChar(character);
NetworkCredential creds = new NetworkCredential(LDAPUser, secureString);

// Actually create the connection
LdapConnection connection = new LdapConnection(identifier, creds)
{
    AuthType = AuthType.Basic,
    SessionOptions =
    {
        ProtocolVersion = 3,
        SecureSocketLayer = true
    }
};

// Override SChannel reverse DNS lookup.
// This gets us past the "The LDAP server is unavailable." exception
// Could be
//    connection.SessionOptions.VerifyServerCertificate += { return true; };
// but some certificate validation is probably good.
connection.SessionOptions.VerifyServerCertificate +=
    (sender, certificate) => certificate.Subject.Contains(string.Format("CN={0},", TargetServer));
```

Use the LDAP server, e.g. search for someone by userid for all objectClass values. The objectClass is present to demonstrates a compound search: The ampersand is the boolean "and" operator for the two query clauses.

```
SearchRequest searchRequest = new SearchRequest(
        CompanyDN,
        string.Format((&(objectClass=*)(uid={0})), uid),
        SearchScope.Subtree,
```

```
        null
);

// Look at your results
foreach (SearchResultEntry entry in searchResponse.Entries) {
    // do something
}
```

Section 138.2: Super Simple anonymous LDAP

Assuming LDAPv3, but it's easy enough to change that. This is anonymous, unencrypted LDAPv3 LdapConnection creation.

```
private const string TargetServer = "ldap.example.com";
```

Actually create the connection with three parts: an LdapDirectoryIdentifier (the server), and NetworkCredentials.

```
// Configure server and credentials
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer);
NetworkCredential creds = new NetworkCredential();
LdapConnection connection = new LdapConnection(identifier, creds)
{
    AuthType=AuthType.Anonymous,
    SessionOptions =
    {
        ProtocolVersion = 3
    }
};
```

To use the connection, something like this would get people with the surname Smith

```
SearchRequest searchRequest = new SearchRequest("dn=example,dn=com", "(sn=Smith)",
SearchScope.Subtree,null);
```

Chapter 139: C# Authentication handler

Section 139.1: Authentication handler

```
public class AuthenticationHandler : DelegatingHandler
    {
        /// <summary>
        /// Holds request's header name which will contains token.
        /// </summary>
        private const string securityToken = "__RequestAuthToken";

        /// <summary>
        /// Default overridden method which performs authentication.
        /// </summary>
        /// <param name="request">Http request message.</param>
        /// <param name="cancellationToken">Cancellation token.</param>
        /// <returns>Returns http response message of type <see cref="HttpResponseMessage"/> class
asynchronously.</returns>
        protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
CancellationToken cancellationToken)
        {
            if (request.Headers.Contains(securityToken))
            {
                bool authorized = Authorize(request);
                if (!authorized)
                {
                    return ApiHttpUtility.FromResult(request, false, HttpStatusCode.Unauthorized,
MessageTypes.Error, Resource.UnAuthenticatedUser);
                }
            }
            else
            {
                return ApiHttpUtility.FromResult(request, false, HttpStatusCode.BadRequest,
MessageTypes.Error, Resource.UnAuthenticatedUser);
            }

            return base.SendAsync(request, cancellationToken);
        }

        /// <summary>
        /// Authorize user by validating token.
        /// </summary>
        /// <param name="requestMessage">Authorization context.</param>
        /// <returns>Returns a value indicating whether current request is authenticated or
not.</returns>
        private bool Authorize(HttpRequestMessage requestMessage)
        {
            try
            {
                HttpRequest request = HttpContext.Current.Request;
                string token = request.Headers[securityToken];
                return SecurityUtility.IsTokenValid(token, request.UserAgent,
HttpContext.Current.Server.MapPath("~/Content/"), requestMessage);
            }
            catch (Exception)
            {
                return false;
            }
        }
```

```
    }
```

Chapter 140: Pointers

Section 140.1: Pointers for array access

This example demonstrates how pointers can be used for C-like access to C# arrays.

```
unsafe
{
    var buffer = new int[1024];
    fixed (int* p = &buffer[0])
    {
        for (var i = 0; i < buffer.Length; i++)
        {
            *(p + i) = i;
        }
    }
}
```

The `unsafe` keyword is required because pointer access will not emit any bounds checks that are normally emitted when accessing C# arrays the regular way.

The `fixed` keyword tells the C# compiler to emit instructions to pin the object in an exception-safe way. Pinning is required to ensure that the garbage collector will not move the array in memory, as that would invalidate any pointers pointing within the array.

Section 140.2: Pointer arithmetic

Addition and subtraction in pointers works differently from integers. When a pointer is incremented or decremented, the address it points to is increased or decreased by the size of the referent type.

For example, the type `int` (alias for `System.Int32`) has a size of 4. If an `int` can be stored in address 0, the subsequent `int` can be stored in address 4, and so on. In code:

```
var ptr = (int*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 4
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8
```

Similarly, the type `long` (alias for `System.Int64`) has a size of 8. If a `long` can be stored in address 0, the subsequent `long`can be stored in address 8, and so on. In code:

```
var ptr = (long*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 16
```

The type `void` is special and `void` pointers are also special and they are used as catch-all pointers when the type isn't known or doesn't matter. Due to their size-agnostic nature, `void` pointers cannot be incremented or decremented:

```
var ptr = (void*)IntPtr.Zero;
```

```
Console.WriteLine(new IntPtr(ptr));
ptr++; // compile-time error
Console.WriteLine(new IntPtr(ptr));
ptr++; // compile-time error
Console.WriteLine(new IntPtr(ptr));
```

Section 140.3: The asterisk is part of the type

In C and C++, the asterisk in the declaration of a pointer variable is *part of the expression* being declared. In C#, the asterisk in the declaration is *part of the type*.

In C, C++ and C#, the following snippet declares an `int` pointer:

```
int* a;
```

In C and C++, the following snippet declares an `int` pointer and an `int` variable. In C#, it declares two `int` pointers:

```
int* a, b;
```

In C and C++, the following snippet declares two `int` pointers. In C#, it is invalid:

```
int *a, *b;
```

Section 140.4: void*

C# inherits from C and C++ the usage of `void*` as a type-agnostic and size-agnostic pointer.

```
void* ptr;
```

Any pointer type can be assigned to `void*` using an implicit conversion:

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
```

The reverse requires an explicit conversion:

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
int* p2 = (int*)ptr;
```

Section 140.5: Member access using ->

C# inherits from C and C++ the usage of the symbol `->` as a means of accessing the members of an instance through a typed pointer.

Consider the following struct:

```
struct Vector2
{
    public int X;
    public int Y;
}
```

This is an example of the usage of `->` to access its members:

```
Vector2 v;
v.X = 5;
v.Y = 10;

Vector2* ptr = &v;
int x = ptr->X;
int y = ptr->Y;
string s = ptr->ToString();

Console.WriteLine(x); // prints 5
Console.WriteLine(y); // prints 10
Console.WriteLine(s); // prints Vector2
```

Section 140.6: Generic pointers

The criteria that a type must satisfy in order to support pointers (see *Remarks*) cannot be expressed in terms of generic constraints. Therefore, any attempt to declare a pointer to a type provided through a generic type parameter will fail.

```
void P<T>(T obj)
    where T : struct
{
    T* ptr = &obj; // compile-time error
}
```

Chapter 141: Pointers & Unsafe Code

Section 141.1: Introduction to unsafe code

C# allows using pointer variables in a function of code block when it is marked by the `unsafe` modifier. The unsafe code or the unmanaged code is a code block that uses a pointer variable.

A pointer is a variable whose value is the address of another variable i.e., the direct address of the memory location. similar to any variable or constant, you must declare a pointer before you can use it to store any variable address.

The general form of a pointer declaration is:

```
type *var-name;
```

Following are valid pointer declarations:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The following example illustrates use of pointers in C#, using the unsafe modifier:

```
using System;
namespace UnsafeCodeApplication
{
   class Program
   {
      static unsafe void Main(string[] args)
      {
         int var = 20;
         int* p = &var;
         Console.WriteLine("Data is: {0} ",  var);
         Console.WriteLine("Address is: {0}",  (int)p);
         Console.ReadKey();
      }
   }
}
```

When the above code wass compiled and executed, it produces the following result:

```
Data is: 20
Address is: 99215364
```

Instead of declaring an entire method as unsafe, you can also declare a part of the code as unsafe:

```
// safe code
unsafe
{
    // you can use pointers here
}
// safe code
```

Section 141.2: Accessing Array Elements Using a Pointer

In C#, an array name and a pointer to a data type same as the array data, are not the same variable type. For example, `int *p` and `int[] p`, are not same type. You can increment the pointer variable `p` because it is not fixed in memory but an array address is fixed in memory, and you can't increment that.

Therefore, if you need to access an array data using a pointer variable, as we traditionally do in C, or C++, you need to fix the pointer using the fixed keyword.

The following example demonstrates this:

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe static void Main()
      {
         int[]  list = {10, 100, 200};
         fixed(int *ptr = list)

         /* let us have array address in pointer */
         for ( int i = 0; i < 3; i++)
         {
            Console.WriteLine("Address of list[{0}]={1}",i,(int)(ptr + i));
            Console.WriteLine("Value of list[{0}]={1}", i, *(ptr + i));
         }

         Console.ReadKey();
      }
   }
}
```

When the above code was compiled and executed, it produces the following result:

```
Address of list[0] = 31627168
Value of list[0] = 10
Address of list[1] = 31627172
Value of list[1] = 100
Address of list[2] = 31627176
Value of list[2] = 200
```

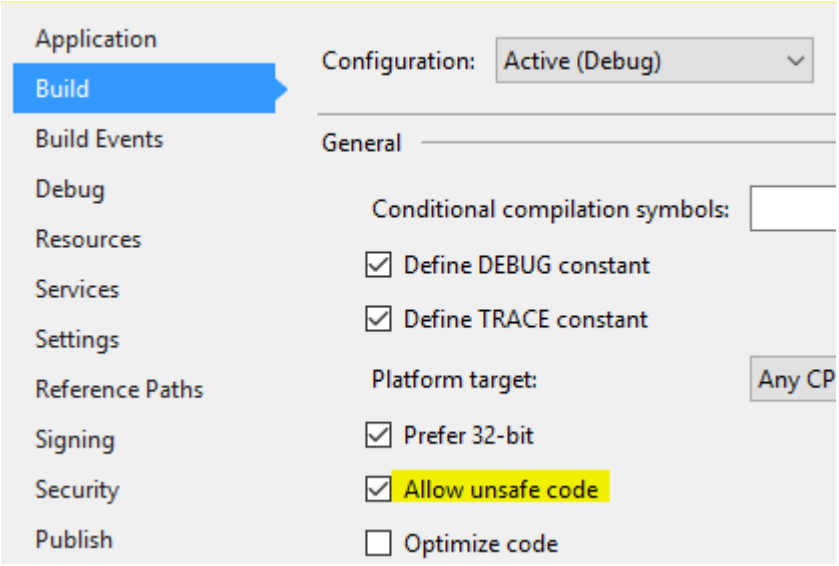
Section 141.3: Compiling Unsafe Code

For compiling unsafe code, you have to specify the `/unsafe` command-line switch with command-line compiler.

For example, to compile a program named prog1.cs containing unsafe code, from command line, give the command:

```
csc /unsafe prog1.cs
```

If you are using Visual Studio IDE then you need to enable use of unsafe code in the project properties.



To do this:

- Open project properties by double clicking the properties node in the Solution Explorer.
- Click on the Build tab.
- Select the option "Allow unsafe code"

Section 141.4: Retrieving the Data Value Using a Pointer

You can retrieve the data stored at the located referenced by the pointer variable, using the ToString() method. The following example demonstrates this:

```
using System;
namespace UnsafeCodeApplication
{
   class Program
   {
      public static void Main()
      {
         unsafe
         {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} " , var);
            Console.WriteLine("Data is: {0} " , p->ToString());
            Console.WriteLine("Address is: {0} " , (int)p);
         }

         Console.ReadKey();
      }
   }
}
```

When the above code was compiled and executed, it produces the following result:

```
Data is: 20
Data is: 20
Address is: 77128984
```

Section 141.5: Passing Pointers as Parameters to Methods

You can pass a pointer variable to a method as parameter. The following example illustrates this:

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe void swap(int* p, int *q)
      {
         int temp = *p;
         *p = *q;
         *q = temp;
      }

      public unsafe static void Main()
      {
         TestPointer p = new TestPointer();
         int var1 = 10;
         int var2 = 20;
         int* x = &var1;
         int* y = &var2;

         Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
         p.swap(x, y);

         Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result:

```
Before Swap: var1: 10, var2: 20
After Swap: var1: 20, var2: 10
```

Chapter 142: How to use C# Structs to create a Union type (Similar to C Unions)

Section 142.1: C-Style Unions in C#

Union types are used in several languages, like C-language, to contain several different types which can "overlap". In other words, they might contain different fields all of which start at the same memory offset, even when they might have different lengths and types. This has the benefit of both saving memory, and doing automatic conversion. Think of an IP address, as an example. Internally, an IP address is represented as an integer, but sometimes we want to access the different Byte component, as in Byte1.Byte2.Byte3.Byte4. This works for any value types, be it primitives like Int32 or long, or for other structs that you define yourself.

We can achieve the same effect in C# by using Explicit Layout Structs.

```
using System;
using System.Runtime.InteropServices;

// The struct needs to be annotated as "Explicit Layout"
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // The "FieldOffset" means that this Integer starts, an offset in bytes.
    // sizeof(int) 4, sizeof(byte) = 1
    [FieldOffset(0)] public int Address;
    [FieldOffset(0)] public byte Byte1;
    [FieldOffset(1)] public byte Byte2;
    [FieldOffset(2)] public byte Byte3;
    [FieldOffset(3)] public byte Byte4;

    public IpAddress(int address) : this()
    {
        // When we init the Int, the Bytes will change too.
        Address = address;
    }

    // Now we can use the explicit layout to access the
    // bytes separately, without doing any conversion.
    public override string ToString() => $"{Byte1}.{Byte2}.{Byte3}.{Byte4}";
}
```

Having defined out Struct in this way, we can use it as we would use a Union in C. For example, let's create an IP address as a Random Integer and then modify the first token in the address to '100', by changing it from 'A.B.C.D' to '100.B.C.D':

```
var ip = new IpAddress(new Random().Next());
Console.WriteLine($"{ip} = {ip.Address}");
ip.Byte1 = 100;
Console.WriteLine($"{ip} = {ip.Address}");
```

Output:

```
75.49.5.32 = 537211211
100.49.5.32 = 537211236
```

View Demo

Section 142.2: Union Types in C# can also contain Struct fields

Apart from primitives, the Explicit Layout structs (Unions) in C#, can also contain other Structs. As long as a field is a Value type and not a Reference, it can be contained in a Union:

```
using System;
using System.Runtime.InteropServices;

// The struct needs to be annotated as "Explicit Layout"
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // Same definition of IpAddress, from the example above
}

// Now let's see if we can fit a whole URL into a long

// Let's define a short enum to hold protocols
enum Protocol : short { Http, Https, Ftp, Sftp, Tcp }

// The Service struct will hold the Address, the Port and the Protocol
[StructLayout(LayoutKind.Explicit)]
struct Service
{
    [FieldOffset(0)] public IpAddress Address;
    [FieldOffset(4)] public ushort Port;
    [FieldOffset(6)] public Protocol AppProtocol;
    [FieldOffset(0)] public long Payload;

    public Service(IpAddress address, ushort port, Protocol protocol)
    {
        Payload = 0;
        Address = address;
        Port  = port;
        AppProtocol = protocol;
    }

    public Service(long payload)
    {
        Address = new IpAddress(0);
        Port = 80;
        AppProtocol = Protocol.Http;
        Payload = payload;
    }

    public Service Copy() => new Service(Payload);

    public override string ToString() => $"{AppProtocol}//{Address}:{Port}/";
}
```

We can now verify that the whole Service Union fits into the size of a long (8 bytes).

```
var ip = new IpAddress(new Random().Next());
Console.WriteLine($"Size: {Marshal.SizeOf(ip)} bytes. Value: {ip.Address} = {ip}.");

var s1 = new Service(ip, 8080, Protocol.Https);
var s2 = new Service(s1.Payload);
s2.Address.Byte1 = 100;
s2.AppProtocol = Protocol.Ftp;
```

```
Console.WriteLine($"Size: {Marshal.SizeOf(s1)} bytes. Value: {s1.Address} = {s1}.");
Console.WriteLine($"Size: {Marshal.SizeOf(s2)} bytes. Value: {s2.Address} = {s2}.");
```

View Demo

Chapter 143: Reactive Extensions (Rx)

Section 143.1: Observing TextChanged event on a TextBox

An observable is created from the TextChanged event of the TextBox. Also any input is only selected if it's different from the last input and if there was no input within 0.5 seconds. The output in this example is sent to the console.

```
Observable
     .FromEventPattern(textBoxInput, "TextChanged")
     .Select(s => ((TextBox) s.Sender).Text)
     .Throttle(TimeSpan.FromSeconds(0.5))
     .DistinctUntilChanged()
     .Subscribe(text => Console.WriteLine(text));
```

Section 143.2: Streaming Data from Database with Observable

Assume having a method returning `IEnumerable<T>`, f.e.

```
private IEnumerable<T> GetData()
{
    try
    {
        // return results from database
    }
    catch(Exception exception)
    {
        throw;
    }
}
```

Creates an Observable and starts a method asynchronously. `SelectMany` flattens the collection and the subscription is fired every 200 elements through `Buffer`.

```
int bufferSize = 200;

Observable
    .Start(() => GetData())
    .SelectMany(s => s)
    .Buffer(bufferSize)
    .ObserveOn(SynchronizationContext.Current)
    .Subscribe(items =>
    {
        Console.WriteLine("Loaded {0} elements", items.Count);

        // do something on the UI like incrementing a ProgressBar
    },
    () => Console.WriteLine("Completed loading"));
```

Chapter 144: AssemblyInfo.cs Examples

Section 144.1: Global and local AssemblyInfo

Having a global allows for better DRYness, you need only put values that are different into AssemblyInfo.cs for projects that have variance. This use assumes your product has more than one visual studio project.

GlobalAssemblyInfo.cs

```
using System.Reflection;
using System.Runtime.InteropServices;
//using Stackoverflow domain as a made up example

// It is common, and mostly good, to use one GlobalAssemblyInfo.cs that is added
// as a link to many projects of the same product, details below
// Change these attribute values in local assembly info to modify the information.
[assembly: AssemblyProduct("Stackoverflow Q&A")]
[assembly: AssemblyCompany("Stackoverflow")]
[assembly: AssemblyCopyright("Copyright © Stackoverflow 2016")]

// The following GUID is for the ID of the typelib if this project is exposed to COM
[assembly: Guid("4e4f2d33-aaab-48ea-a63d-1f0a8e3c935f")]
[assembly: ComVisible(false)] //not going to expose ;)

// Version information for an assembly consists of the following four values:
// roughly translated from I reckon it is for SO, note that they most likely
// dynamically generate this file
//      Major Version  - Year 6 being 2016
//      Minor Version  - The month
//      Day Number     - Day of month
//      Revision       - Build number
// You can specify all the values or you can default the Build and Revision Numbers
// by using the '*' as shown below: [assembly: AssemblyVersion("year.month.day.*")]
[assembly: AssemblyVersion("2016.7.00.00")]
[assembly: AssemblyFileVersion("2016.7.27.3839")]
```

AssemblyInfo.cs - one for each project

```
//then the following might be put into a separate Assembly file per project, e.g.
[assembly: AssemblyTitle("Stackoveflow.Redis")]
```

You can add the GlobalAssemblyInfo.cs to the local project using the following procedure:

1. Select Add/Existing Item... in the context menu of the project
2. Select GlobalAssemblyInfo.cs
3. Expand the Add-Button by clicking on that little down-arrow on the right hand
4. Select "Add As Link" in the buttons drop down list

Section 144.2: [AssemblyVersion]

This attribute applies a version to the assembly.

```
[assembly: AssemblyVersion("1.0.*")]
```

The * character is used to auto-increment a portion of the version automatically every time you compile (often used for the "build" number)

Section 144.3: Automated versioning

Your code in source control has version numbers either by default (SVN ids or Git SHA1 hashes) or explicitly (Git tags). Rather than manually updating versions in AssemblyInfo.cs you can use a build time process to write the version from your source control system into your AssemblyInfo.cs files and thus onto your assemblies.

The GitVersionTask or SemVer.Git.Fody NuGet packages are examples of the above. To use GitVersionTask, for instance, after installing the package in your project remove the `Assembly*Version` attributes from your AssemblyInfo.cs files. This puts GitVersionTask in charge of versioning your assemblies.

Note that Semantic Versioning is increasingly the *de facto* standard so these methods recommend using source control tags that follow SemVer.

Section 144.4: Common fields

It's good practice to complete your AssemblyInfo's default fields. The information may be picked up by installers and will then appear when using Programs and Features (Windows 10) to uninstall or change a program.

The minimum should be:

- AssemblyTitle - usually the namespace, *i.e.* MyCompany.MySolution.MyProject
- AssemblyCompany - the legal entities full name
- AssemblyProduct - marketing may have a view here
- AssemblyCopyright - keep it up to date as it looks scruffy otherwise

'AssemblyTitle' becomes the 'File description' when examining the DLL's Properties Details tab.

Section 144.5: [AssemblyTitle]

This attribute is used to give a name to this particular assembly.

```
[assembly: AssemblyTitle("MyProduct")]
```

Section 144.6: [AssemblyProduct]

This attribute is used to describe the product that this particular assembly is for. Multiple assemblies can be components of the same product, in which case they can all share the same value for this attribute.

```
[assembly: AssemblyProduct("MyProduct")]
```

Section 144.7: [InternalsVisibleTo]

If you want to make `internal` classes or functions of an assembly accessable from another assembly you declare this by `InternalsVisibleTo` and the assembly name that is allowed to access.

In this example code in the assembly `MyAssembly.UnitTests` is allowed to call `internal` elements from `MyAssembly`.

```
[assembly: InternalsVisibleTo("MyAssembly.UnitTests")]
```

This is especially useful for unit testing to prevent unnecessary `public` declarations.

Section 144.8: [AssemblyConfiguration]

AssemblyConfiguration: The AssemblyConfiguration attribute must have the configuration that was used to build the assembly. Use conditional compilation to properly include different assembly configurations. Use the block similar to the example below. Add as many different configurations as you commonly use.

```
#if (DEBUG)

[assembly: AssemblyConfiguration("Debug")]

#else

[assembly: AssemblyConfiguration("Release")]

#endif
```

Section 144.9: [AssemblyKeyFile]

Whenever we want our assembly to install in GAC then it is must to have a strong name. For strong naming assembly we have to create a public key. To generate the `.snk` file.

To create a strong name key file

> 1. Developers command prompt for VS2015 (with administrator Access)
> 2. At the command prompt, type cd C:\Directory_Name and press ENTER.
> 3. At the command prompt, type sn -k KeyFileName.snk, and then press ENTER.

once the keyFileName.snk is created at specified directory then give refernce in your project . give `AssemblyKeyFileAttribute` attribute the path to `snk` file to generate the key when we build our class library.

> Properties -> AssemblyInfo.cs

```
[assembly: AssemblyKeyFile(@"c:\Directory_Name\KeyFileName.snk")]
```

Thi will create a strong name assembly after build. After creating your strong name assembly you can then install it in GAC

Happy Coding :)

Section 144.10: Reading Assembly Attributes

Using .NET's rich reflection APIs, you can gain access to an assembly's metadata. For example, you can get `this` assembly's title attribute with the following code

```
using System.Linq;
using System.Reflection;

...

Assembly assembly = typeof(this).Assembly;
var titleAttribute = assembly.GetCustomAttributes<AssemblyTitleAttribute>().FirstOrDefault();
```

```
Console.WriteLine($"This assembly title is {titleAttribute?.Title}");
```

Chapter 145: Creating a Console Application using a Plain-Text Editor and the C# Compiler (csc.exe)

Section 145.1: Creating a Console application using a Plain-Text Editor and the C# Compiler

In order to use a plain-text editor to create a Console application that is written in C#, you'll need the C# Compiler. The C# Compiler (csc.exe), can be found at the following location:
`%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe`

N.B. Depending upon which version of the .NET Framework that is installed on your system, you may need to change the path above, accordingly.

Saving the Code
The purpose of this topic is not to teach you *how* to write a Console application, but to teach you how to *compile* one [to produce a single executable file], with nothing other than the C# Compiler and any Plain-Text Editor (such as Notepad).

1. Open the Run dialog, by using the keyboard shortcut `Windows Key` + `R`
2. Type `notepad`, then hit `Enter`
3. Paste the example code below, into Notepad
4. Save the file as `ConsoleApp.cs`, by going to **File** → **Save As...**, then entering `ConsoleApp.cs` in the 'File Name' text field, then selecting `All Files` as the file-type.
5. Click `Save`

Compiling the Source Code
1. Open the Run dialog, using `Windows Key` + `R`
2. Enter:

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe /t:exe
/out:"C:\Users\yourUserName\Documents\ConsoleApp.exe"
"C:\Users\yourUserName\Documents\ConsoleApp.cs"
```

Now, go back to where you originally saved your `ConsoleApp.cs` file. You should now see an executable file (`ConsoleApp.exe`). Double-click `ConsoleApp.exe` to open it.

That's it! Your console application has been compiled. An executable file has been created and you now have a working Console app.

```
using System;

namespace ConsoleApp
{
    class Program
    {
        private static string input = String.Empty;

        static void Main(string[] args)
        {
            goto DisplayGreeting;
```

```
            DisplayGreeting:
            {
                Console.WriteLine("Hello! What is your name?");

                input = Console.ReadLine();

                if (input.Length >= 1)
                {
                    Console.WriteLine(
                        "Hello, " +
                        input +
                        ", enter 'Exit' at any time to exit this app.");

                    goto AwaitFurtherInstruction;
                }
                else
                {
                    goto DisplayGreeting;
                }
            }

            AwaitFurtherInstruction:
            {
                input = Console.ReadLine();

                if(input.ToLower() == "exit")
                {
                    input = String.Empty;

                    Environment.Exit(0);
                }
                else
                {
                    goto AwaitFurtherInstruction;
                }
            }
        }
    }
}
```

Chapter 146: CLSCompliantAttribute

Constructor	Parameter
CLSCompliantAttribute(Boolean)	Initializes an instance of the CLSCompliantAttribute class with a Boolean value indicating whether the indicated program element is CLS-compliant.

Section 146.1: Access Modifier to which CLS rules apply

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Cat
    {
        internal UInt16 _age = 0;
        private UInt16 _daysTillVacination = 0;

        //Warning CS3003  Type of 'Cat.DaysTillVacination' is not CLS-compliant
        protected UInt16 DaysTillVacination
        {
            get { return _daysTillVacination; }
        }

        //Warning    CS3003    Type of 'Cat.Age' is not CLS-compliant
        public UInt16 Age
        { get { return _age; } }

        //valid behaviour by CLS-compliant rules
        public int IncreaseAge()
        {
            int increasedAge = (int)_age + 1;

            return increasedAge;
        }

    }
}
```

The rules for CLS compliance apply only to a public/protected components.

Section 146.2: Violation of CLS rule: Unsigned types / sbyte

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        internal UInt16 _yearOfCreation = 0;

        //Warning CS3008  Identifier '_numberOfDoors' is not CLS-compliant
        //Warning CS3003  Type of 'Car._numberOfDoors' is not CLS-compliant
        public UInt32 _numberOfDoors = 0;
```

```
        //Warning    CS3003    Type of 'Car.YearOfCreation' is not CLS-compliant
        public UInt16 YearOfCreation
        {
            get { return _yearOfCreation; }
        }


        //Warning CS3002  Return type of 'Car.CalculateDistance()' is not CLS-compliant
        public UInt64 CalculateDistance()
        {
            return 0;
        }


        //Warning CS3002  Return type of 'Car.TestDummyUnsignedPointerMethod()' is not CLS-compliant
        public UIntPtr TestDummyUnsignedPointerMethod()
        {
            int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            UIntPtr ptr = (UIntPtr)arr[0];


            return ptr;
        }

        //Warning CS3003  Type of 'Car.age' is not CLS-compliant
        public sbyte age = 120;


    }
}
```

Section 146.3: Violation of CLS rule: Same naming

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        //Warning    CS3005    Identifier 'Car.CALCULATEAge()' differing only in case is not CLS-
compliant
        public int CalculateAge()
        {
            return 0;
        }

        public int CALCULATEAge()
        {
            return 0;
        }

    }
}
```

Visual Basic is not case sensitive

Section 146.4: Violation of CLS rule: Identifier _

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        //Warning CS3008  Identifier '_age' is not CLS-complian
        public int _age = 0;
    }

}
```

You can not start variable with _

Section 146.5: Violation of CLS rule: Inherit from non CLSComplaint class

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    [CLSCompliant(false)]
    public class Animal
    {
        public int age = 0;
    }

    //Warning    CS3009    'Dog': base type 'Animal' is not CLS-compliant
    public class Dog : Animal
    {
    }

}
```

Chapter 147: ObservableCollection<T>

Section 147.1: Initialize ObservableCollection<T>

`ObservableCollection` is a collection of type `T` like `List<T>` which means that it holds objects of type `T`.

From documentation we read that :

> `ObservableCollection`represents a dynamic data collection that provides notifications when items get added, removed, or when the entire list is refreshed.

The key difference from other collections is that `ObservableCollection` implements the interfaces `INotifyCollectionChanged` and `INotifyPropertyChanged` and immediately raise notification event when a new object is added or removed and when collection is cleared.

This is especially useful for conneting the UI and backend of an application without having to write extra code because when an object is added to or removed from an observable collection, the UI is automatically updated.

The first step in order to use it is to include

```
using System.Collections.ObjectModel
```

You can either create an empty instance of a collection for example of type `string`

```
ObservableCollection<string> collection = new ObservableCollection<string>();
```

or an instance that is filled with data

```
ObservableCollection<string> collection = new ObservableCollection<string>()
{
 "First_String", "Second_String"
};
```

Remember as in all IList collection, index starts from 0 (IList.Item Property).

Chapter 148: Hash Functions

Section 148.1: PBKDF2 for Password Hashing

PBKDF2 ("Password-Based Key Derivation Function 2") is one of the recommended hash-functions for password-hashing. It is part of rfc-2898.

.NET's `Rfc2898DeriveBytes`-Class is based upon HMACSHA1.

```
    using System.Security.Cryptography;

    ...

    public const int SALT_SIZE = 24; // size in bytes
    public const int HASH_SIZE = 24; // size in bytes
    public const int ITERATIONS = 100000; // number of pbkdf2 iterations

    public static byte[] CreateHash(string input)
    {
        // Generate a salt
        RNGCryptoServiceProvider provider = new RNGCryptoServiceProvider();
        byte[] salt = new byte[SALT_SIZE];
        provider.GetBytes(salt);

        // Generate the hash
        Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(input, salt, ITERATIONS);
        return pbkdf2.GetBytes(HASH_SIZE);
    }
```

PBKDF2 requires a salt and the number of iterations.

Iterations:

A high number of iterations will slow the algorithm down, which makes password cracking a lot harder. A high number of iterations is therefor recommended. PBKDF2 is order of magnitudes slower than MD5 for example.

Salt:

A salt will prevent the lookup of hash values in rainbow tables. It has to be stored alongside the password hash. One salt per password (not one global salt) is recommended.

Section 148.2: Complete Password Hashing Solution using Pbkdf2

```
using System;
using System.Linq;
using System.Security.Cryptography;

namespace YourCryptoNamespace
{
    /// <summary>
    /// Salted password hashing with PBKDF2-SHA1.
  /// Compatibility: .NET 3.0 and later.
  /// </summary>
  /// <remarks>See http://crackstation.net/hashing-security.htm for much more on password
hashing.</remarks>
  public static class PasswordHashProvider
```

```
  {
    /// <summary>
    /// The salt byte size, 64 length ensures safety but could be increased / decreased
    /// </summary>
    private const int SaltByteSize = 64;
    /// <summary>
    /// The hash byte size,
    /// </summary>
    private const int HashByteSize = 64;
    /// <summary>
    ///  High iteration count is less likely to be cracked
    /// </summary>
    private const int Pbkdf2Iterations = 10000;

    /// <summary>
    /// Creates a salted PBKDF2 hash of the password.
    /// </summary>
    /// <remarks>
    /// The salt and the hash have to be persisted side by side for the password. They could be
persisted as bytes or as a string using the convenience methods in the next class to convert from
byte[] to string and later back again when executing password validation.
    /// </remarks>
    /// <param name="password">The password to hash.</param>
    /// <returns>The hash of the password.</returns>
    public static PasswordHashContainer CreateHash(string password)
    {
      // Generate a random salt
      using (var csprng = new RNGCryptoServiceProvider())
      {
        // create a unique salt for every password hash to prevent rainbow and dictionary based
attacks
        var salt = new byte[SaltByteSize];
        csprng.GetBytes(salt);

        // Hash the password and encode the parameters
        var hash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);

        return new PasswordHashContainer(hash, salt);
      }
    }
    /// <summary>
    /// Recreates a password hash based on the incoming password string and the stored salt
    /// </summary>
    /// <param name="password">The password to check.</param>
    /// <param name="salt">The salt existing.</param>
    /// <returns>the generated hash based on the password and salt</returns>
    public static byte[] CreateHash(string password, byte[] salt)
    {
      // Extract the parameters from the hash
      return Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
    }

    /// <summary>
    /// Validates a password given a hash of the correct one.
    /// </summary>
    /// <param name="password">The password to check.</param>
    /// <param name="salt">The existing stored salt.</param>
    /// <param name="correctHash">The hash of the existing password.</param>
    /// <returns><c>true</c> if the password is correct. <c>false</c> otherwise. </returns>
    public static bool ValidatePassword(string password, byte[] salt, byte[] correctHash)
    {
      // Extract the parameters from the hash
```

```
      byte[] testHash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
      return CompareHashes(correctHash, testHash);
    }
    /// <summary>
    /// Compares two byte arrays (hashes)
    /// </summary>
    /// <param name="array1">The array1.</param>
    /// <param name="array2">The array2.</param>
    /// <returns><c>true</c> if they are the same, otherwise <c>false</c></returns>
    public static bool CompareHashes(byte[] array1, byte[] array2)
    {
      if (array1.Length != array2.Length) return false;
      return !array1.Where((t, i) => t != array2[i]).Any();
    }

    /// <summary>
    /// Computes the PBKDF2-SHA1 hash of a password.
    /// </summary>
    /// <param name="password">The password to hash.</param>
    /// <param name="salt">The salt.</param>
    /// <param name="iterations">The PBKDF2 iteration count.</param>
    /// <param name="outputBytes">The length of the hash to generate, in bytes.</param>
    /// <returns>A hash of the password.</returns>
    private static byte[] Pbkdf2(string password, byte[] salt, int iterations, int outputBytes)
    {
      using (var pbkdf2 = new Rfc2898DeriveBytes(password, salt))
      {
        pbkdf2.IterationCount = iterations;
        return pbkdf2.GetBytes(outputBytes);
      }
    }
  }

  /// <summary>
  /// Container for password hash and salt and iterations.
  /// </summary>
  public sealed class PasswordHashContainer
  {
    /// <summary>
    /// Gets the hashed password.
    /// </summary>
    public byte[] HashedPassword { get; private set; }
    /// <summary>
    /// Gets the salt.
    /// </summary>
    public byte[] Salt { get; private set; }

    /// <summary>
    /// Initializes a new instance of the <see cref="PasswordHashContainer" /> class.
    /// </summary>
    /// <param name="hashedPassword">The hashed password.</param>
    /// <param name="salt">The salt.</param>
    public PasswordHashContainer(byte[] hashedPassword, byte[] salt)
    {
      this.HashedPassword = hashedPassword;
      this.Salt = salt;
    }
  }

  /// <summary>
  /// Convenience methods for converting between hex strings and byte array.
  /// </summary>
```

```
  public static class ByteConverter
  {
    /// <summary>
    /// Converts the hex representation string to an array of bytes
    /// </summary>
    /// <param name="hexedString">The hexed string.</param>
    /// <returns></returns>
    public static byte[] GetHexBytes(string hexedString)
    {
      var bytes = new byte[hexedString.Length / 2];
      for (var i = 0; i < bytes.Length; i++)
      {
        var strPos = i * 2;
        var chars = hexedString.Substring(strPos, 2);
        bytes[i] = Convert.ToByte(chars, 16);
      }
      return bytes;
    }
    /// <summary>
    /// Gets a hex string representation of the byte array passed in.
    /// </summary>
    /// <param name="bytes">The bytes.</param>
    public static string GetHexString(byte[] bytes)
    {
      return BitConverter.ToString(bytes).Replace("-", "").ToUpper();
    }
  }
}

/*
 * Password Hashing With PBKDF2 (http://crackstation.net/hashing-security.htm).
 * Copyright (c) 2013, Taylor Hornby
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 * this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright notice,
 * this list of conditions and the following disclaimer in the documentation
 * and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */
```

Please see this excellent resource Crackstation - Salted Password Hashing - Doing it Right for more information. Part of this solution (the hashing function) was based on the code from that site.

Section 148.3: MD5

Hash functions map binary strings of an arbitrary length to small binary strings of a fixed length.

The MD5 algorithm is a widely used hash function producing a 128-bit hash value (16 Bytes, 32 Hexdecimal characters).

The ComputeHash method of the System.Security.Cryptography.MD5 class returns the hash as an array of 16 bytes.

Example:

```
using System;
using System.Security.Cryptography;
using System.Text;

internal class Program
{
    private static void Main()
    {
        var source = "Hello World!";

        // Creates an instance of the default implementation of the MD5 hash algorithm.
        using (var md5Hash = MD5.Create())
        {
            // Byte array representation of source string
            var sourceBytes = Encoding.UTF8.GetBytes(source);

            // Generate hash value(Byte Array) for input data
            var hashBytes = md5Hash.ComputeHash(sourceBytes);

            // Convert hash byte array to string
            var hash = BitConverter.ToString(hashBytes).Replace("-", string.Empty);

            // Output the MD5 hash
            Console.WriteLine("The MD5 hash of " + source + " is: " + hash);
        }
    }
}
```

> **Output:** The MD5 hash of Hello World! is: ED076287532E86365E841E92BFC50D8C

Security Issues:

Like most hash functions, MD5 is neither encryption nor encoding. It can be reversed by brute-force attack and suffers from extensive vulnerabilities against collision and preimage attacks.

Section 148.4: SHA1

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA1 sha1Hash = SHA1.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha1Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-",String.Empty);

                Console.WriteLine("The SHA1 hash of " + source + " is: " + hash);
            }
        }
    }
 }
```

Output:

The SHA1 hash of Hello Word! is: 2EF7BDE608CE5404E97D5F042F95F89F1C232871

Section 148.5: SHA256

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA256 sha256Hash = SHA256.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha256Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA256 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

Output:

The SHA256 hash of Hello World! is:
7F83B1657FF1FC53B92DC18148A1D65DFC2D4B1FA3D677284ADDD200126D9069

Section 148.6: SHA384

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA384 sha384Hash = SHA384.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha384Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA384 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

Output:

The SHA384 hash of Hello World! is:
BFD76C0EBBD006FEE583410547C1887B0292BE76D582D96C242D2A792723E3FD6FD061F9D5CFD13B8F961358E6A
DBA4A

Section 148.7: SHA512

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA512 sha512Hash = SHA512.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha512Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA512 hash of " + source + " is: " + hash);
            }
        }
    }
}
```

Output: The SHA512 hash of Hello World! is:
861844D6704E8573FEC34D967E20BCFEF3D424CF48BE04E6DC08F2BD58C729743371015EAD891CC3CF1C9D34B49
264B510751B1FF9E537937BC46B5D6FF4ECC8

Chapter 149: Generating Random Numbers in C#

Parameters	Details
Seed	A value for generating random numbers. If not set, the default value is determined by the current system time.
minValue	Generated numbers won't be smaller than this value. If not set, the default value is 0.
maxValue	Generated numbers will be smaller than this value. If not set, the default value is `Int32.MaxValue`.
return value	Returns a number with random value.

Section 149.1: Generate a random int

This example generates random values between 0 and 2147483647.

```
Random rnd = new Random();
int randomNumber = rnd.Next();
```

Section 149.2: Generate a random int in a given range

Generate a random number between `minValue` and `maxValue - 1`.

```
Random rnd = new Random();
var randomBetween10And20 = rnd.Next(10, 20);
```

Section 149.3: Generating the same sequence of random numbers over and over again

When creating `Random` instances with the same seed, the same numbers will be generated.

```
int seed = 5;
for (int i = 0; i < 2; i++)
{
   Console.WriteLine("Random instance " + i);
   Random rnd = new Random(seed);
   for (int j = 0; j < 5; j++)
   {
      Console.Write(rnd.Next());
      Console.Write(" ");
   }

   Console.WriteLine();
}
```

Output:

```
Random instance 0
726643700 610783965 564707973 1342984399 995276750
Random instance 1
726643700 610783965 564707973 1342984399 995276750
```

Section 149.4: Create multiple random class with different seeds simultaneously

Two Random class created at the same time will have the same seed value.

Using `System.Guid.NewGuid().GetHashCode()` can get a different seed even in the same time.

```
Random rnd1 = new Random();
Random rnd2 = new Random();
Console.WriteLine("First 5 random number in rnd1");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd1.Next());

Console.WriteLine("First 5 random number in rnd2");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd2.Next());

rnd1 = new Random(Guid.NewGuid().GetHashCode());
rnd2 = new Random(Guid.NewGuid().GetHashCode());
Console.WriteLine("First 5 random number in rnd1 using Guid");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd1.Next());
Console.WriteLine("First 5 random number in rnd2 using Guid");
for (int i = 0; i < 5; i++)
    Console.WriteLine(rnd2.Next());
```

Another way to achieve different seeds is to use another `Random` instance to retrieve the seed values.

```
Random rndSeeds = new Random();
Random rnd1 = new Random(rndSeeds.Next());
Random rnd2 = new Random(rndSeeds.Next());
```

This also makes it possible to control the result of all the `Random` instances by setting only the seed value for the `rndSeeds`. All the other instances will be deterministically derived from that single seed value.

Section 149.5: Generate a Random double

Generate a random number between 0 and 1.0. (not including 1.0)

```
Random rnd = new Random();
var randomDouble = rnd.NextDouble();
```

Section 149.6: Generate a random character

Generate a random letter between `a` and `z` by using the `Next()` overload for a given range of numbers, then converting the resulting `int` to a `char`

```
Random rnd = new Random();
char randomChar = (char)rnd.Next('a','z');
//'a' and 'z' are interpreted as ints for parameters for Next()
```

Section 149.7: Generate a number that is a percentage of a max value

A common need for random numbers it to generate a number that is `X%` of some max value. this can be done by treating the result of `NextDouble()` as a percentage:

```
var rnd = new Random();
var maxValue = 5000;
var percentage = rnd.NextDouble();
var result = maxValue * percentage;
//suppose NextDouble() returns .65, result will hold 65% of 5000: 3250.
```

Chapter 150: Cryptography (System.Security.Cryptography)

Section 150.1: Modern Examples of Symmetric Authenticated Encryption of a string

Cryptography is something very hard and after spending a lot of time reading different examples and seeing how easy it is to introduce some form of vulnerability I found an answer originally written by @jbtule that I think is very good. Enjoy reading:

"The general best practice for symmetric encryption is to use Authenticated Encryption with Associated Data (AEAD), however this isn't a part of the standard .net crypto libraries. So the first example uses AES256 and then HMAC256, a two step Encrypt then MAC, which requires more overhead and more keys.

The second example uses the simpler practice of AES256-GCM using the open source Bouncy Castle (via nuget).

Both examples have a main function that takes secret message string, key(s) and an optional non-secret payload and return and authenticated encrypted string optionally prepended with the non-secret data. Ideally you would use these with 256bit key(s) randomly generated see `NewKey()`.

Both examples also have a helper methods that use a string password to generate the keys. These helper methods are provided as a convenience to match up with other examples, however they are *far less secure* because the strength of the password is going to be *far weaker than a 256 bit key*.

Update: Added `byte[]` overloads, and only the Gist has the full formatting with 4 spaces indent and api docs due to StackOverflow answer limits."

.NET Built-in Encrypt(AES)-Then-MAC(HMAC) [Gist]

```
/*
 * This work (Modern Encryption of a String C#, by James Tuley),
 * identified by James Tuley, is free of known copyright restrictions.
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
 */

using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;

namespace Encryption
{
  public static class AESThenHMAC
  {
    private static readonly RandomNumberGenerator Random = RandomNumberGenerator.Create();

    //Preconfigured Encryption Parameters
    public static readonly int BlockBitSize = 128;
    public static readonly int KeyBitSize = 256;

    //Preconfigured Password Key Derivation Parameters
    public static readonly int SaltBitSize = 64;
    public static readonly int Iterations = 10000;
    public static readonly int MinPasswordLength = 12;
```

```
    /// <summary>
    /// Helper that generates a random key on each call.
    /// </summary>
    /// <returns></returns>
    public static byte[] NewKey()
    {
      var key = new byte[KeyBitSize / 8];
      Random.GetBytes(key);
      return key;
    }

    /// <summary>
    /// Simple Encryption (AES) then Authentication (HMAC) for a UTF8 Message.
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="cryptKey">The crypt key.</param>
    /// <param name="authKey">The auth key.</param>
    /// <param name="nonSecretPayload">(Optional) Non-Secret Payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Secret Message Required!;secretMessage</exception>
    /// <remarks>
    /// Adds overhead of (Optional-Payload + BlockSize(16) + Message-Padded-To-Blocksize +  HMac-
Tag(32)) * 1.33 Base64
    /// </remarks>
    public static string SimpleEncrypt(string secretMessage, byte[] cryptKey, byte[] authKey,
                        byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncrypt(plainText, cryptKey, authKey, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }

    /// <summary>
    /// Simple Authentication (HMAC) then Decryption (AES) for a secrets UTF8 Message.
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="cryptKey">The crypt key.</param>
    /// <param name="authKey">The auth key.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    public static string SimpleDecrypt(string encryptedMessage, byte[] cryptKey, byte[] authKey,
                        int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecrypt(cipherText, cryptKey, authKey, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    /// <summary>
    /// Simple Encryption (AES) then Authentication (HMAC) of a UTF8 message
```

```
    /// using Keys derived from a Password (PBKDF2).
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayload">The non secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">password</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// Adds additional non secret payload for key generation parameters.
    /// </remarks>
    public static string SimpleEncryptWithPassword(string secretMessage, string password,
                            byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }

    /// <summary>
    /// Simple Authentication (HMAC) and then Descryption (AES) of a UTF8 Message
    /// using keys derived from a password (PBKDF2).
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// </remarks>
    public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                            int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] cryptKey, byte[] authKey,
byte[] nonSecretPayload = null)
    {
      //User Error Checks
      if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"cryptKey");

      if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
"authKey");
```

```
      if (secretMessage == null || secretMessage.Length < 1)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      //non-secret payload optional
      nonSecretPayload = nonSecretPayload ?? new byte[] { };

      byte[] cipherText;
      byte[] iv;

      using (var aes = new AesManaged
      {
        KeySize = KeyBitSize,
        BlockSize = BlockBitSize,
        Mode = CipherMode.CBC,
        Padding = PaddingMode.PKCS7
      })
      {

        //Use random IV
        aes.GenerateIV();
        iv = aes.IV;

        using (var encrypter = aes.CreateEncryptor(cryptKey, iv))
        using (var cipherStream = new MemoryStream())
        {
          using (var cryptoStream = new CryptoStream(cipherStream, encrypter,
CryptoStreamMode.Write))
          using (var binaryWriter = new BinaryWriter(cryptoStream))
          {
            //Encrypt Data
            binaryWriter.Write(secretMessage);
          }

          cipherText = cipherStream.ToArray();
        }

      }

      //Assemble encrypted message and add authentication
      using (var hmac = new HMACSHA256(authKey))
      using (var encryptedStream = new MemoryStream())
      {
        using (var binaryWriter = new BinaryWriter(encryptedStream))
        {
          //Prepend non-secret payload if any
          binaryWriter.Write(nonSecretPayload);
          //Prepend IV
          binaryWriter.Write(iv);
          //Write Ciphertext
          binaryWriter.Write(cipherText);
          binaryWriter.Flush();

          //Authenticate all data
          var tag = hmac.ComputeHash(encryptedStream.ToArray());
          //Postpend tag
          binaryWriter.Write(tag);
        }
        return encryptedStream.ToArray();
      }

    }
```

```
    public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] cryptKey, byte[] authKey,
int nonSecretPayloadLength = 0)
    {

      //Basic Usage Error Checks
      if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("CryptKey needs to be {0} bit!", KeyBitSize),
"cryptKey");

      if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("AuthKey needs to be {0} bit!", KeyBitSize),
"authKey");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      using (var hmac = new HMACSHA256(authKey))
      {
        var sentTag = new byte[hmac.HashSize / 8];
        //Calculate Tag
        var calcTag = hmac.ComputeHash(encryptedMessage, 0, encryptedMessage.Length -
sentTag.Length);
        var ivLength = (BlockBitSize / 8);

        //if message length is to small just return null
        if (encryptedMessage.Length < sentTag.Length + nonSecretPayloadLength + ivLength)
          return null;

        //Grab Sent Tag
        Array.Copy(encryptedMessage, encryptedMessage.Length - sentTag.Length, sentTag, 0,
sentTag.Length);

        //Compare Tag with constant time comparison
        var compare = 0;
        for (var i = 0; i < sentTag.Length; i++)
          compare |= sentTag[i] ^ calcTag[i];

        //if message doesn't authenticate return null
        if (compare != 0)
          return null;

        using (var aes = new AesManaged
        {
          KeySize = KeyBitSize,
          BlockSize = BlockBitSize,
          Mode = CipherMode.CBC,
          Padding = PaddingMode.PKCS7
        })
        {

          //Grab IV from message
          var iv = new byte[ivLength];
          Array.Copy(encryptedMessage, nonSecretPayloadLength, iv, 0, iv.Length);

          using (var decrypter = aes.CreateDecryptor(cryptKey, iv))
          using (var plainTextStream = new MemoryStream())
          {
            using (var decrypterStream = new CryptoStream(plainTextStream, decrypter,
CryptoStreamMode.Write))
            using (var binaryWriter = new BinaryWriter(decrypterStream))
            {
              //Decrypt Cipher Text from Message
```

```
            binaryWriter.Write(
              encryptedMessage,
              nonSecretPayloadLength + iv.Length,
              encryptedMessage.Length - nonSecretPayloadLength - iv.Length - sentTag.Length
            );
          }
          //Return Plain Text
          return plainTextStream.ToArray();
        }
      }
    }
  }

  public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password, byte[]
nonSecretPayload = null)
  {
    nonSecretPayload = nonSecretPayload ?? new byte[] {};

    //User Error Checks
    if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
      throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

    if (secretMessage == null || secretMessage.Length ==0)
      throw new ArgumentException("Secret Message Required!", "secretMessage");

    var payload = new byte[((SaltBitSize / 8) * 2) + nonSecretPayload.Length];

    Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
    int payloadIndex = nonSecretPayload.Length;

    byte[] cryptKey;
    byte[] authKey;
    //Use Random Salt to prevent pre-generated weak password attacks.
    using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
    {
      var salt = generator.Salt;

      //Generate Keys
      cryptKey = generator.GetBytes(KeyBitSize / 8);

      //Create Non Secret Payload
      Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
      payloadIndex += salt.Length;
    }

    //Deriving separate key, might be less efficient than using HKDF,
    //but now compatible with RNEncryptor which had a very similar wireformat and requires less
code than HKDF.
    using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
    {
      var salt = generator.Salt;

      //Generate Keys
      authKey = generator.GetBytes(KeyBitSize / 8);

      //Create Rest of Non Secret Payload
      Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
    }

    return SimpleEncrypt(secretMessage, cryptKey, authKey, payload);
  }
```

```
    public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password, int
nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cryptSalt = new byte[SaltBitSize / 8];
      var authSalt = new byte[SaltBitSize / 8];

      //Grab Salt from Non-Secret Payload
      Array.Copy(encryptedMessage, nonSecretPayloadLength, cryptSalt, 0, cryptSalt.Length);
      Array.Copy(encryptedMessage, nonSecretPayloadLength + cryptSalt.Length, authSalt, 0,
authSalt.Length);

      byte[] cryptKey;
      byte[] authKey;

      //Generate crypt key
      using (var generator = new Rfc2898DeriveBytes(password, cryptSalt, Iterations))
      {
        cryptKey = generator.GetBytes(KeyBitSize / 8);
      }
      //Generate auth key
      using (var generator = new Rfc2898DeriveBytes(password, authSalt, Iterations))
      {
        authKey = generator.GetBytes(KeyBitSize / 8);
      }

      return SimpleDecrypt(encryptedMessage, cryptKey, authKey, cryptSalt.Length + authSalt.Length
+ nonSecretPayloadLength);
    }
  }
}
```

Bouncy Castle AES-GCM [Gist]

```
/*
 * This work (Modern Encryption of a String C#, by James Tuley),
 * identified by James Tuley, is free of known copyright restrictions.
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
 */

using System;
using System.IO;
using System.Text;
using Org.BouncyCastle.Crypto;
using Org.BouncyCastle.Crypto.Engines;
using Org.BouncyCastle.Crypto.Generators;
using Org.BouncyCastle.Crypto.Modes;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Security;
namespace Encryption
{

  public static class AESGCM
```

```
  {
    private static readonly SecureRandom Random = new SecureRandom();

    //Preconfigured Encryption Parameters
    public static readonly int NonceBitSize = 128;
    public static readonly int MacBitSize = 128;
    public static readonly int KeyBitSize = 256;

    //Preconfigured Password Key Derivation Parameters
    public static readonly int SaltBitSize = 128;
    public static readonly int Iterations = 10000;
    public static readonly int MinPasswordLength = 12;


    /// <summary>
    /// Helper that generates a random new key on each call.
    /// </summary>
    /// <returns></returns>
    public static byte[] NewKey()
    {
      var key = new byte[KeyBitSize / 8];
      Random.NextBytes(key);
      return key;
    }

    /// <summary>
    /// Simple Encryption And Authentication (AES-GCM) of a UTF8 string.
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="key">The key.</param>
    /// <param name="nonSecretPayload">Optional non-secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Secret Message Required!;secretMessage</exception>
    /// <remarks>
    /// Adds overhead of (Optional-Payload + BlockSize(16) + Message +  HMac-Tag(16)) * 1.33 Base64
    /// </remarks>
    public static string SimpleEncrypt(string secretMessage, byte[] key, byte[] nonSecretPayload =
null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncrypt(plainText, key, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }


    /// <summary>
    /// Simple Decryption & Authentication (AES-GCM) of a UTF8 Message
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="key">The key.</param>
    /// <param name="nonSecretPayloadLength">Length of the optional non-secret payload.</param>
    /// <returns>Decrypted Message</returns>
    public static string SimpleDecrypt(string encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrEmpty(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");
```

```
      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecrypt(cipherText, key, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    /// <summary>
    /// Simple Encryption And Authentication (AES-GCM) of a UTF8 String
    /// using key derived from a password (PBKDF2).
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayload">The non secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// Adds additional non secret payload for key generation parameters.
    /// </remarks>
    public static string SimpleEncryptWithPassword(string secretMessage, string password,
                            byte[] nonSecretPayload = null)
    {
      if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var plainText = Encoding.UTF8.GetBytes(secretMessage);
      var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
      return Convert.ToBase64String(cipherText);
    }


    /// <summary>
    /// Simple Decryption and Authentication (AES-GCM) of a UTF8 message
    /// using a key derived from a password (PBKDF2)
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="password">The password.</param>
    /// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
    /// <returns>
    /// Decrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
    /// <remarks>
    /// Significantly less secure than using random binary keys.
    /// </remarks>
    public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                            int nonSecretPayloadLength = 0)
    {
      if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var cipherText = Convert.FromBase64String(encryptedMessage);
      var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
      return plainText == null ? null : Encoding.UTF8.GetString(plainText);
    }

    public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] key, byte[] nonSecretPayload =
null)
    {
      //User Error Checks
      if (key == null || key.Length != KeyBitSize / 8)
```

```
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize), "key");

      if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      //Non-secret Payload Optional
      nonSecretPayload = nonSecretPayload ?? new byte[] { };

      //Using random nonce large enough not to repeat
      var nonce = new byte[NonceBitSize / 8];
      Random.NextBytes(nonce, 0, nonce.Length);

      var cipher = new GcmBlockCipher(new AesFastEngine());
      var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
      cipher.Init(true, parameters);

      //Generate Cipher Text With Auth Tag
      var cipherText = new byte[cipher.GetOutputSize(secretMessage.Length)];
      var len = cipher.ProcessBytes(secretMessage, 0, secretMessage.Length, cipherText, 0);
      cipher.DoFinal(cipherText, len);

      //Assemble Message
      using (var combinedStream = new MemoryStream())
      {
        using (var binaryWriter = new BinaryWriter(combinedStream))
        {
          //Prepend Authenticated Payload
          binaryWriter.Write(nonSecretPayload);
          //Prepend Nonce
          binaryWriter.Write(nonce);
          //Write Cipher Text
          binaryWriter.Write(cipherText);
        }
        return combinedStream.ToArray();
      }
    }

    public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (key == null || key.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize), "key");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      using (var cipherStream = new MemoryStream(encryptedMessage))
      using (var cipherReader = new BinaryReader(cipherStream))
      {
        //Grab Payload
        var nonSecretPayload = cipherReader.ReadBytes(nonSecretPayloadLength);

        //Grab Nonce
        var nonce = cipherReader.ReadBytes(NonceBitSize / 8);

        var cipher = new GcmBlockCipher(new AesFastEngine());
        var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
        cipher.Init(false, parameters);
```

```
        //Decrypt Cipher Text
        var cipherText = cipherReader.ReadBytes(encryptedMessage.Length - nonSecretPayloadLength -
nonce.Length);
        var plainText = new byte[cipher.GetOutputSize(cipherText.Length)];

        try
        {
          var len = cipher.ProcessBytes(cipherText, 0, cipherText.Length, plainText, 0);
          cipher.DoFinal(plainText, len);

        }
        catch (InvalidCipherTextException)
        {
          //Return null if it doesn't authenticate
          return null;
        }

        return plainText;
      }

    }

    public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password, byte[]
nonSecretPayload = null)
    {
      nonSecretPayload = nonSecretPayload ?? new byte[] {};

      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

      var generator = new Pkcs5S2ParametersGenerator();

      //Use Random Salt to minimize pre-generated weak password attacks.
      var salt = new byte[SaltBitSize / 8];
      Random.NextBytes(salt);

      generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

      //Generate Key
      var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

      //Create Full Non Secret Payload
      var payload = new byte[salt.Length + nonSecretPayload.Length];
      Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
      Array.Copy(salt,0, payload,nonSecretPayload.Length, salt.Length);

      return SimpleEncrypt(secretMessage, key.GetKey(), payload);
    }

    public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password, int
nonSecretPayloadLength = 0)
    {
      //User Error Checks
      if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
```

```
        throw new ArgumentException(String.Format("Must have a password of at least {0}
characters!", MinPasswordLength), "password");

      if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

      var generator = new Pkcs5S2ParametersGenerator();

      //Grab Salt from Payload
      var salt = new byte[SaltBitSize / 8];
      Array.Copy(encryptedMessage, nonSecretPayloadLength, salt, 0, salt.Length);

      generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

      //Generate Key
      var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

      return SimpleDecrypt(encryptedMessage, key.GetKey(), salt.Length + nonSecretPayloadLength);
    }
  }
}
```

Section 150.2: Introduction to Symmetric and Asymmetric Encryption

You can improve the security for data transit or storing by implementing encrypting techniques. Basically there are two approaches when using *System.Security.Cryptography*: **symmetric** and **asymmetric.**

Symmetric Encryption

This method uses a private key in order to perform the data transformation.

Pros:

- Symmetric algorithms consume less resources and are faster than asymmetric ones.
- The amount of data you can encrypt is unlimited.

Cons:

- Encryption and decryption use the same key. Someone will be able to decrypt your data if the key is compromised.
- You could end up with many different secret keys to manage if you choose to use a different secret key for different data.

Under System.Security.Cryptography you have different classes that perform symmetric encryption, they are known as block ciphers:

- AesManaged (AES algorithm).
- AesCryptoServiceProvider (AES algorithm FIPS 140-2 complaint).
- DESCryptoServiceProvider (DES algorithm).
- RC2CryptoServiceProvider (Rivest Cipher 2 algorithm).
- RijndaelManaged (AES algorithm). *Note*: RijndaelManaged is **not** FIPS-197 complaint.
- TripleDES (TripleDES algorithm).

Asymmetric Encryption

This method uses a combination of public and private keys in order to perform the data transformation.

Pros:

- It uses larger keys than symmetric algorithms, thus they are less susceptible to being cracked by using brute force.
- It is easier to guarantee who is able to encrypt and decrypt the data because it relies on two keys (public and private).

Cons:

- There is a limit on the amount of data that you can encrypt. The limit is different for each algorithm and is typically proportional with the key size of the algorithm. For example, an RSACryptoServiceProvider object with a key length of 1,024 bits can only encrypt a message that is smaller than 128 bytes.
- Asymmetric algorithms are very slow in comparison to symmetric algorithms.

Under System.Security.Cryptography you have access to different classes that perform asymmetric encryption:

- DSACryptoServiceProvider (Digital Signature Algorithm algorithm)
- RSACryptoServiceProvider (RSA Algorithm algorithm)

Section 150.3: Simple Symmetric File Encryption

The following code sample demonstrates a quick and easy means of encrypting and decrypting files using the AES symmetric encryption algorithm.

The code randomly generates the Salt and Initialization Vectors each time a file is encrypted, meaning that encrypting the same file with the same password will always lead to different output. The salt and IV are written to the output file so that only the password is required to decrypt it.

```
public static void ProcessFile(string inputPath, string password, bool encryptMode, string
outputPath)
{
    using (var cypher = new AesManaged())
    using (var fsIn = new FileStream(inputPath, FileMode.Open))
    using (var fsOut = new FileStream(outputPath, FileMode.Create))
    {
        const int saltLength = 256;
        var salt = new byte[saltLength];
        var iv = new byte[cypher.BlockSize / 8];

        if (encryptMode)
        {
            // Generate random salt and IV, then write them to file
            using (var rng = new RNGCryptoServiceProvider())
            {
                rng.GetBytes(salt);
                rng.GetBytes(iv);
            }
            fsOut.Write(salt, 0, salt.Length);
            fsOut.Write(iv, 0, iv.Length);
        }
        else
        {
            // Read the salt and IV from the file
            fsIn.Read(salt, 0, saltLength);
```

```
            fsIn.Read(iv, 0, iv.Length);
        }

        // Generate a secure password, based on the password and salt provided
        var pdb = new Rfc2898DeriveBytes(password, salt);
        var key = pdb.GetBytes(cypher.KeySize / 8);

        // Encrypt or decrypt the file
        using (var cryptoTransform = encryptMode
            ? cypher.CreateEncryptor(key, iv)
            : cypher.CreateDecryptor(key, iv))
        using (var cs = new CryptoStream(fsOut, cryptoTransform, CryptoStreamMode.Write))
        {
            fsIn.CopyTo(cs);
        }
    }
}
```

Section 150.4: Cryptographically Secure Random Data

There are times when the framework's Random() class may not be considered random enough, given that it is based on a psuedo-random number generator. The framework's Crypto classes do, however, provide something more robust in the form of RNGCryptoServiceProvider.

The following code samples demonstrate how to generate Cryptographically Secure byte arrays, strings and numbers.

Random Byte Array

```
public static byte[] GenerateRandomData(int length)
{
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    return rnd;
}
```

Random Integer (with even distribution)

```
public static int GenerateRandomInt(int minVal=0, int maxVal=100)
{
    var rnd = new byte[4];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    var i = Math.Abs(BitConverter.ToInt32(rnd, 0));
    return Convert.ToInt32(i % (maxVal - minVal + 1) + minVal);
}
```

Random String

```
public static string GenerateRandomString(int length, string allowableChars=null)
{
    if (string.IsNullOrEmpty(allowableChars))
        allowableChars = @"ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    // Generate random data
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
```

```
    // Generate the output string
    var allowable = allowableChars.ToCharArray();
    var l = allowable.Length;
    var chars = new char[length];
    for (var i = 0; i < length; i++)
        chars[i] = allowable[rnd[i] % l];

    return new string(chars);
}
```

Section 150.5: Password Hashing

Passwords should never be stored as plain text! They should be hashed with a randomly generated salt (to defend against rainbow table attacks) using a slow password hashing algorithm. A high number of iterations (> 10k) can be used to slow down brute force attacks. A delay of ~100ms is acceptable to a user logging in, but makes breaking a long password difficult. When choosing a number of iterations you should use the maximum tolerable value for your application and increase it as computer performance improves. You will also need to consider stopping repeated requests which could be used as a DoS attack.

When hashing for the first time a salt can be generated for you, the resulting hash and salt can then be stored to a file.

```
private void firstHash(string userName, string userPassword, int numberOfItterations)
{
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, 8, numberOfItterations);
 //Hash the password with a 8 byte salt
    byte[] hashedPassword = PBKDF2.GetBytes(20);    //Returns a 20 byte hash
    byte[] salt = PBKDF2.Salt;
    writeHashToFile(userName, hashedPassword, salt, numberOfItterations); //Store the hashed
password with the salt and number of itterations to check against future password entries
}
```

Checking an existing users password, read their hash and salt from a file and compare to the hash of the entered password

```
private bool checkPassword(string userName, string userPassword, int numberOfItterations)
{
    byte[] usersHash = getUserHashFromFile(userName);
    byte[] userSalt = getUserSaltFromFile(userName);
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, userSalt,
numberOfItterations);    //Hash the password with the users salt
    byte[] hashedPassword = PBKDF2.GetBytes(20);    //Returns a 20 byte hash
    bool passwordsMach = comparePasswords(usersHash, hashedPassword);    //Compares byte arrays
    return passwordsMach;
}
```

Section 150.6: Fast Asymmetric File Encryption

Asymmetric encryption is often regarded as preferable to Symmetric encryption for transferring messages to other parties. This is mainly because it negates many of the risks related to the exchange of a shared key and ensures that whilst anyone with the public key can encrypt a message for the intended recipient, only that recipient can decrypt it. Unfortunately the major down-side of asymmetric encryption algorithms is that they are significantly slower than their symmetric cousins. As such the asymmetric encryption of files, especially large ones, can often be a very computationally intensive process.

In order to provide both security AND performance, a hybrid approach can be taken. This entails the

cryptographically random generation of a key and initialization vector for *Symmetric* encryption. These values are then encrypted using an *Asymmetric* algorithm and written to the output file, before being used to encrypt the source data *Symmetrically* and appending it to the output.

This approach provides a high degree of both performance and security, in that the data is encrypted using a symmetric algorithm (fast) and the key and iv, both randomly generated (secure) are encrypted by an asymmetric algorithm (secure). It also has the added advantage that the same payload encrypted on different occasions will have very different cyphertext, because the symmetric keys are randomly generated each time.

The following class demonstrates asymmetric encryption of strings and byte arrays, as well as hybrid file encryption.

```
public static class AsymmetricProvider
{
    #region Key Generation
    public class KeyPair
    {
        public string PublicKey { get; set; }
        public string PrivateKey { get; set; }
    }

    public static KeyPair GenerateNewKeyPair(int keySize = 4096)
    {
        // KeySize is measured in bits. 1024 is the default, 2048 is better, 4096 is more robust but
takes a fair bit longer to generate.
        using (var rsa = new RSACryptoServiceProvider(keySize))
        {
            return new KeyPair {PublicKey = rsa.ToXmlString(false), PrivateKey =
rsa.ToXmlString(true)};
        }
    }

    #endregion

    #region Asymmetric Data Encryption and Decryption

    public static byte[] EncryptData(byte[] data, string publicKey)
    {
        using (var asymmetricProvider = new RSACryptoServiceProvider())
        {
            asymmetricProvider.FromXmlString(publicKey);
            return asymmetricProvider.Encrypt(data, true);
        }
    }

    public static byte[] DecryptData(byte[] data, string publicKey)
    {
        using (var asymmetricProvider = new RSACryptoServiceProvider())
        {
            asymmetricProvider.FromXmlString(publicKey);
            if (asymmetricProvider.PublicOnly)
                throw new Exception("The key provided is a public key and does not contain the
private key elements required for decryption");
            return asymmetricProvider.Decrypt(data, true);
        }
    }

    public static string EncryptString(string value, string publicKey)
    {
        return Convert.ToBase64String(EncryptData(Encoding.UTF8.GetBytes(value), publicKey));
```

```
    }

    public static string DecryptString(string value, string privateKey)
    {
        return Encoding.UTF8.GetString(EncryptData(Convert.FromBase64String(value), privateKey));
    }

    #endregion

    #region Hybrid File Encryption and Decription

    public static void EncryptFile(string inputFilePath, string outputFilePath, string publicKey)
    {
        using (var symmetricCypher = new AesManaged())
        {
            // Generate random key and IV for symmetric encryption
            var key = new byte[symmetricCypher.KeySize / 8];
            var iv = new byte[symmetricCypher.BlockSize / 8];
            using (var rng = new RNGCryptoServiceProvider())
            {
                rng.GetBytes(key);
                rng.GetBytes(iv);
            }

            // Encrypt the symmetric key and IV
            var buf = new byte[key.Length + iv.Length];
            Array.Copy(key, buf, key.Length);
            Array.Copy(iv, 0, buf, key.Length, iv.Length);
            buf = EncryptData(buf, publicKey);

            var bufLen = BitConverter.GetBytes(buf.Length);

            // Symmetrically encrypt the data and write it to the file, along with the encrypted key
and iv
            using (var cypherKey = symmetricCypher.CreateEncryptor(key, iv))
            using (var fsIn = new FileStream(inputFilePath, FileMode.Open))
            using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
            using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
            {
                fsOut.Write(bufLen,0, bufLen.Length);
                fsOut.Write(buf, 0, buf.Length);
                fsIn.CopyTo(cs);
            }
        }
    }

    public static void DecryptFile(string inputFilePath, string outputFilePath, string privateKey)
    {
        using (var symmetricCypher = new AesManaged())
        using (var fsIn = new FileStream(inputFilePath, FileMode.Open))
        {
            // Determine the length of the encrypted key and IV
            var buf = new byte[sizeof(int)];
            fsIn.Read(buf, 0, buf.Length);
            var bufLen = BitConverter.ToInt32(buf, 0);

            // Read the encrypted key and IV data from the file and decrypt using the asymmetric
algorithm
            buf = new byte[bufLen];
            fsIn.Read(buf, 0, buf.Length);
            buf = DecryptData(buf, privateKey);
```

```
            var key = new byte[symmetricCypher.KeySize / 8];
            var iv = new byte[symmetricCypher.BlockSize / 8];
            Array.Copy(buf, key, key.Length);
            Array.Copy(buf, key.Length, iv, 0, iv.Length);

            // Decript the file data using the symmetric algorithm
            using (var cypherKey = symmetricCypher.CreateDecryptor(key, iv))
            using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
            using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
            {
                fsIn.CopyTo(cs);
            }
        }
    }

    #endregion

    #region Key Storage

    public static void WritePublicKey(string publicKeyFilePath, string publicKey)
    {
        File.WriteAllText(publicKeyFilePath, publicKey);
    }
    public static string ReadPublicKey(string publicKeyFilePath)
    {
        return File.ReadAllText(publicKeyFilePath);
    }

    private const string SymmetricSalt = "Stack_Overflow!"; // Change me!

    public static string ReadPrivateKey(string privateKeyFilePath, string password)
    {
        var salt = Encoding.UTF8.GetBytes(SymmetricSalt);
        var cypherText = File.ReadAllBytes(privateKeyFilePath);

        using (var cypher = new AesManaged())
        {
            var pdb = new Rfc2898DeriveBytes(password, salt);
            var key = pdb.GetBytes(cypher.KeySize / 8);
            var iv = pdb.GetBytes(cypher.BlockSize / 8);

            using (var decryptor = cypher.CreateDecryptor(key, iv))
            using (var msDecrypt = new MemoryStream(cypherText))
            using (var csDecrypt = new CryptoStream(msDecrypt, decryptor, CryptoStreamMode.Read))
            using (var srDecrypt = new StreamReader(csDecrypt))
            {
                return srDecrypt.ReadToEnd();
            }
        }
    }

    public static void WritePrivateKey(string privateKeyFilePath, string privateKey, string
password)
    {
        var salt = Encoding.UTF8.GetBytes(SymmetricSalt);
        using (var cypher = new AesManaged())
        {
            var pdb = new Rfc2898DeriveBytes(password, salt);
            var key = pdb.GetBytes(cypher.KeySize / 8);
            var iv = pdb.GetBytes(cypher.BlockSize / 8);

            using (var encryptor = cypher.CreateEncryptor(key, iv))
```

```
            using (var fsEncrypt = new FileStream(privateKeyFilePath, FileMode.Create))
            using (var csEncrypt = new CryptoStream(fsEncrypt, encryptor, CryptoStreamMode.Write))
            using (var swEncrypt = new StreamWriter(csEncrypt))
            {
                swEncrypt.Write(privateKey);
            }
        }
    }

    #endregion
}
```

Example of use:

```
private static void HybridCryptoTest(string privateKeyPath, string privateKeyPassword, string
inputPath)
{
    // Setup the test
    var publicKeyPath = Path.ChangeExtension(privateKeyPath, ".public");
    var outputPath = Path.Combine(Path.ChangeExtension(inputPath, ".enc"));
    var testPath = Path.Combine(Path.ChangeExtension(inputPath, ".test"));

    if (!File.Exists(privateKeyPath))
    {
        var keys = AsymmetricProvider.GenerateNewKeyPair(2048);
        AsymmetricProvider.WritePublicKey(publicKeyPath, keys.PublicKey);
        AsymmetricProvider.WritePrivateKey(privateKeyPath, keys.PrivateKey, privateKeyPassword);
    }

    // Encrypt the file
    var publicKey = AsymmetricProvider.ReadPublicKey(publicKeyPath);
    AsymmetricProvider.EncryptFile(inputPath, outputPath, publicKey);

    // Decrypt it again to compare against the source file
    var privateKey = AsymmetricProvider.ReadPrivateKey(privateKeyPath, privateKeyPassword);
    AsymmetricProvider.DecryptFile(outputPath, testPath, privateKey);

    // Check that the two files match
    var source = File.ReadAllBytes(inputPath);
    var dest = File.ReadAllBytes(testPath);

    if (source.Length != dest.Length)
        throw new Exception("Length does not match");

    if (source.Where((t, i) => t != dest[i]).Any())
        throw new Exception("Data mismatch");
}
```

Chapter 151: ASP.NET Identity

Tutorials concerning asp.net Identity such as user management, role management, creating tokens and more.

Section 151.1: How to implement password reset token in asp.net identity using user manager

1. Create a new folder called MyClasses and create and add the following class

```
public class GmailEmailService:SmtpClient
{
    // Gmail user-name
    public string UserName { get; set; }

    public GmailEmailService() :
        base(ConfigurationManager.AppSettings["GmailHost"],
Int32.Parse(ConfigurationManager.AppSettings["GmailPort"]))
    {
        //Get values from web.config file:
        this.UserName = ConfigurationManager.AppSettings["GmailUserName"];
        this.EnableSsl = Boolean.Parse(ConfigurationManager.AppSettings["GmailSsl"]);
        this.UseDefaultCredentials = false;
        this.Credentials = new System.Net.NetworkCredential(this.UserName,
ConfigurationManager.AppSettings["GmailPassword"]);
    }
}
```

2. Configure your Identity Class

```
public async Task SendAsync(IdentityMessage message)
{
    MailMessage email = new MailMessage(new MailAddress("youremailadress@domain.com", "(any
subject here)"),
    new MailAddress(message.Destination));
    email.Subject = message.Subject;
    email.Body = message.Body;

    email.IsBodyHtml = true;

    GmailEmailService mailClient = new GmailEmailService();
    await mailClient.SendMailAsync(email);
}
```

3. Add your credentials to the web.config. I did not use gmail in this portion because the use of gmail is blocked in my workplace and it still works perfectly.

   lt;add key="GmailUserName" value="youremail@yourdomain.com"/> <add key="GmailPassword" value="yourPassword"/> <add key="GmailHost" value="yourServer"/> <add key="GmailPort" value="yourPort"/> <add key="GmailSsl" value="chooseTrueOrFalse"/> <!--Smptp Server (confirmations emails)-->

4. Make necessary changes to your Account Controller. Add the following highlighted code.

```
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.DataProtection;
using System;
using System.Linq;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;
using MYPROJECT.Models;

namespace MYPROJECT.Controllers
{
    [Authorize]
    public class AccountController : Controller
    {
        private ApplicationSignInManager _signInManager;
        private ApplicationUserManager _userManager;
        ApplicationDbContext _context;
        DpapiDataProtectionProvider provider = new DpapiDataProtectionProvider("MYPROJECT");
        EmailService EmailService = new EmailService();


        public AccountController()
            : this(new UserManager<ApplicationUser>(new UserStore<ApplicationUser>(new ApplicationDbContext())))
        {
            _context = new ApplicationDbContext();
        }

        public AccountController(UserManager<ApplicationUser> userManager, ApplicationSignInManager signInManager)
        {
            UserManager = userManager;
            SignInManager = signInManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider<ApplicationUser>(
                    provider.Create("EmailConfirmation"));
        }

        private AccountController(UserManager<ApplicationUser> userManager)
        {
            UserManager = userManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider<ApplicationUser>(
                    provider.Create("EmailConfirmation"));
        }

        public UserManager<ApplicationUser> UserManager { get; private set; }

        public ApplicationSignInManager SignInManager
        {
            get
            {
                return _signInManager ?? HttpContext.GetOwinContext().Get<ApplicationSignInManager>();
            }
            private set
            {
                _signInManager = value;
            }
        }
```

```
//
// GET: /Account/ForgotPassword
[AllowAnonymous]
public ActionResult ForgotPassword()
{
    return View();
}

//
// POST: /Account/ForgotPassword
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> ForgotPassword(ForgotPasswordViewModel model)
{
    if (ModelState.IsValid)
    {

        var user = await UserManager.FindByNameAsync(model.UserName);

        if (user == null || !(await UserManager.IsEmailConfirmedAsync(user.Id)))
        {
            // Don't reveal that the user does not exist or is not confirmed
            return View("ForgotPasswordConfirmation");
        }

        // For more information on how to enable account confirmation and password reset please visit http://go.microsoft.com/fwlink/?LinkID=320771
        // Send an email with this link
        string code = await UserManager.GeneratePasswordResetTokenAsync(user.Id);
        code = HttpUtility.UrlEncode(code);
        var callbackUrl = Url.Action("ResetPassword", "Account", new { userId = user.Id, code = HttpUtility.UrlDecode(code) }, protocol: Request.Url.Scheme);
        await EmailService.SendAsync(new IdentityMessage
        {
            Body = "Please reset your password by clicking <a href=\"" + callbackUrl + "\">here</a>",
            Destination = user.Email,
            Subject = "Reset Password"
        });
        //await UserManager.SendEmailAsync(user.Id, "Reset Password", "Please reset your password by clicking <a href=\"" + callbackUrl + "\">here</a>");
        return RedirectToAction("ForgotPasswordConfirmation", "Account");
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Compile then run. Cheers!

Chapter 152: Unsafe Code in .NET

Section 152.1: Using unsafe with arrays

When accessing arrays with pointers, there are no bounds check and therefore no `IndexOutOfRangeException` will be thrown. This makes the code faster.

Assigning values to an array with a pointer:

```
class Program
{
    static void Main(string[] args)
    {
        unsafe
        {
            int[] array = new int[1000];
            fixed (int* ptr = array)
            {
                for (int i = 0; i < array.Length; i++)
                {
                    *(ptr+i) = i; //assigning the value with the pointer
                }
            }
        }
    }
}
```

While the safe and normal counterpart would be:

```
class Program
{
    static void Main(string[] args)
    {
        int[] array = new int[1000];

        for (int i = 0; i < array.Length; i++)
        {
            array[i] = i;
        }
    }
}
```

The unsafe part will generally be faster and the difference in performance can vary depending on the complexity of the elements in the array as well as the logic applied to each one. Even though it may be faster, it should be used with care since it is harder to maintain and easier to break.

Section 152.2: Using unsafe with strings

```
var s = "Hello";       // The string referenced by variable 's' is normally immutable, but
                       // since it is memory, we could change it if we can access it in an
                       // unsafe way.

unsafe                 // allows writing to memory; methods on System.String don't allow this
{
  fixed (char* c = s) // get pointer to string originally stored in read only memory
    for (int i = 0; i < s.Length; i++)
      c[i] = 'a';      // change data in memory allocated for original string "Hello"
```

```
}
Console.WriteLine(s); // The variable 's' still refers to the same System.String
                      // value in memory, but the contents at that location were
                      // changed by the unsafe write above.
                      // Displays: "aaaaa"
```

Section 152.3: Unsafe Array Index

```
void Main()
{
    unsafe
    {
        int[] a = {1, 2, 3};
        fixed(int* b = a)
        {
            Console.WriteLine(b[4]);
        }
    }
}
```

Running this code creates an array of length 3, but then tries to get the 5th item (index 4). On my machine, this printed `1910457872`, but the behavior is not defined.

Without the `unsafe` block, you cannot use pointers, and therefore cannot access values past the end of an array without causing an exception to be thrown.

Chapter 153: C# Script

Section 153.1: Simple code evaluation

You can evaluate any valid C# code:

```
int value = await CSharpScript.EvaluateAsync<int>("15 * 89 + 95");
var span = await CSharpScript.EvaluateAsync<TimeSpan>("new DateTime(2016,1,1) - DateTime.Now");
```

If type is not specified, the result is `object`:

```
object value = await CSharpScript.EvaluateAsync("15 * 89 + 95");
```

Chapter 154: Runtime Compile

Section 154.1: RoslynScript

`Microsoft.CodeAnalysis.CSharp.Scripting.CSharpScript` is a new C# script engine.

```
var code = "(1 + 2).ToString()";
var run = await CSharpScript.RunAsync(code, ScriptOptions.Default);
var result = (string)run.ReturnValue;
Console.WriteLine(result); //output 3
```

You can compile and run any statements, variables, methods, classes or any code segments.

Section 154.2: CSharpCodeProvider

`Microsoft.CSharp.CSharpCodeProvider` can be used to compile C# classes.

```
var code = @"
    public class Abc {
       public string Get() { return ""abc""; }
    }
";

var options = new CompilerParameters();
options.GenerateExecutable = false;
options.GenerateInMemory = false;

var provider = new CSharpCodeProvider();
var compile = provider.CompileAssemblyFromSource(options, code);

var type = compile.CompiledAssembly.GetType("Abc");
var abc = Activator.CreateInstance(type);

var method = type.GetMethod("Get");
var result = method.Invoke(abc, null);

Console.WriteLine(result); //output: abc
```

Chapter 155: Interoperability

Section 155.1: Import function from unmanaged C++ DLL

Here is an example of how to import a function that is defined in an unmanaged C++ DLL. In the C++ source code for "myDLL.dll", the function `add` is defined:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
{
    return a + b;
}
```

Then it can be included into a C# program as follows:

```
class Program
{
    // This line will import the C++ method.
    // The name specified in the DllImport attribute must be the DLL name.
    // The names of parameters are unimportant, but the types must be correct.
    [DllImport("myDLL.dll")]
    private static extern int add(int left, int right);

    static void Main(string[] args)
    {
        //The extern method can be called just as any other C# method.
        Console.WriteLine(add(1, 2));
    }
}
```

See Calling conventions and C++ name mangling for explanations about why `extern "C"` and `__stdcall` are necessary.

Finding the dynamic library

When the extern method is first invoked the C# program will search for and load the appropriate DLL. For more information about where is searched to find the DLL, and how you can influence the search locations see this stackoverflow question.

Section 155.2: Calling conventions

There're several conventions of calling functions, specifying who (caller or callee) pops arguments from the stack, how arguments are passed and in what order. C++ uses `Cdecl` calling convention by default, but C# expects `StdCall`, which is usually used by Windows API. You need to change one or the other:

- Change calling convention to `StdCall` in C++:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
```

```
[DllImport("myDLL.dll")]
```

- Or, change calling convention to `Cdecl` in C#:

```
extern "C" __declspec(dllexport) int /*__cdecl*/ add(int a, int b)
```

```
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl)]
```

If you want to use a function with `Cdecl` calling convention and a mangled name, your code will look like this:

```
__declspec(dllexport) int add(int a, int b)
```

```
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl,
           EntryPoint = "?add@@YAHHH@Z")]
```

- **thiscall**(**__thiscall**) is mainly used in functions that are members of a class.
- When a function uses **thiscall**(**__thiscall**) , a pointer to the class is passed down as the first parameter.

Section 155.3: C++ name mangling

C++ compilers encode additional information in the names of exported functions, such as argument types, to make overloads with different arguments possible. This process is called name mangling. This causes problems with importing functions in C# (and interop with other languages in general), as the name of `int add(int a, int b)` function is no longer `add`, it can be `?add@@YAHHH@Z`, `_add@8` or anything else, depending on the compiler and the calling convention.

There're several ways of solving the problem of name mangling:

- Exporting functions using `extern "C"` to switch to C external linkage which uses C name mangling:

  ```
  extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
  ```

  ```
  [DllImport("myDLL.dll")]
  ```

  Function name will still be mangled (`_add@8`), but `StdCall`+`extern "C"` name mangling is recognized by C# compiler.

- Specifying exported function names in `myDLL.def` module definition file:

  ```
  EXPORTS
    add
  ```

  ```
  int __stdcall add(int a, int b)
  ```

  ```
  [DllImport("myDLL.dll")]
  ```

  The function name will be pure `add` in this case.

- Importing mangled name. You'll need some DLL viewer to see the mangled name, then you can specify it explicitly:

  ```
  __declspec(dllexport) int __stdcall add(int a, int b)
  ```

  ```
  [DllImport("myDLL.dll", EntryPoint = "?add@@YGHHH@Z")]
  ```

Section 155.4: Dynamic loading and unloading of unmanaged DLLs

When using the `DllImport` attribute you have to know the correct dll and method name at *compile time*. If you want to be more flexible and decide at *runtime* which dll and methods to load, you can use the Windows API methods `LoadLibrary()`, `GetProcAddress()` and `FreeLibrary()`. This can be helpful if the library to use depends on runtime

conditions.

The pointer returned by `GetProcAddress()` can be casted into a delegate using `Marshal.GetDelegateForFunctionPointer()`.

The following code sample demonstrates this with the `myDLL.dll` from the previous examples:

```
class Program
{
    // import necessary API as shown in other examples
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr LoadLibrary(string lib);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern void FreeLibrary(IntPtr module);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr GetProcAddress(IntPtr module, string proc);

    // declare a delegate with the required signature
    private delegate int AddDelegate(int a, int b);

    private static void Main()
    {
        // load the dll
        IntPtr module = LoadLibrary("myDLL.dll");
        if (module == IntPtr.Zero) // error handling
        {
            Console.WriteLine($"Could not load library: {Marshal.GetLastWin32Error()}");
            return;
        }

        // get a "pointer" to the method
        IntPtr method = GetProcAddress(module, "add");
        if (method == IntPtr.Zero) // error handling
        {
            Console.WriteLine($"Could not load method: {Marshal.GetLastWin32Error()}");
            FreeLibrary(module);  // unload library
            return;
        }

        // convert "pointer" to delegate
        AddDelegate add = (AddDelegate)Marshal.GetDelegateForFunctionPointer(method,
typeof(AddDelegate));

        // use function
        int result = add(750, 300);

        // unload library
        FreeLibrary(module);
    }
}
```

Section 155.5: Reading structures with Marshal

Marshal class contains a function named **PtrToStructure**, this function gives us the ability of reading structures by an unmanaged pointer.

PtrToStructure function got many overloads, but they all have the same intention.

Generic **PtrToStructure**:

```
public static T PtrToStructure<T>(IntPtr ptr);
```

T - structure type.

ptr - A pointer to an unmanaged block of memory.

Example:

```
NATIVE_STRUCT result = Marshal.PtrToStructure<NATIVE_STRUCT>(ptr);
```

- If you dealing with managed objects while reading native structures, don't forget to pin your object :)

```
T Read<T>(byte[] buffer)
   {
       T result = default(T);

       var gch = GCHandle.Alloc(buffer, GCHandleType.Pinned);

       try
       {
           result = Marshal.PtrToStructure<T>(gch.AddrOfPinnedObject());
       }
       finally
       {
           gch.Free();
       }

       return result;
   }
```

Section 155.6: Dealing with Win32 Errors

When using interop methods, you can use **GetLastError** API to get additional information on you API calls.

DllImport Attribute SetLastError Attribute

SetLastError=true

Indicates that the callee will call SetLastError (Win32 API function).

SetLastError=false

Indicates that the callee **will not** call SetLastError (Win32 API function), therefore you will not get an error information.

- When SetLastError isn't set, it is set to false (Default value).
- You can obtain the error code using Marshal.GetLastWin32Error Method:

Example:

```
[DllImport("kernel32.dll", SetLastError=true)]
public static extern IntPtr OpenMutex(uint access, bool handle, string lpName);
```

If you trying to open mutex which does not exist, GetLastError will return **ERROR_FILE_NOT_FOUND**.

```
var lastErrorCode = Marshal.GetLastWin32Error();
```

```
if (lastErrorCode == (uint)ERROR_FILE_NOT_FOUND)
{
    //Deal with error
}
```

System Error Codes can be found here:

https://msdn.microsoft.com/en-us/library/windows/desktop/ms681382(v=vs.85).aspx

GetLastError API

There is a native **GetLastError** API which you can use as well :

```
[DllImport("coredll.dll", SetLastError=true)]
static extern Int32 GetLastError();
```

- When calling Win32 API from managed code, you must always use the **Marshal.GetLastWin32Error**.

Here's why:

Between your Win32 call which sets the error (calls SetLastError), the CLR can call other Win32 calls which could call **SetLastError** as well, this behavior can override your error value. In this scenario, if you call **GetLastError** you can obtain an invalid error.

Setting **SetLastError = true**, makes sure that the CLR retrieves the error code before it executes other Win32 calls.

Section 155.7: Pinned Object

GC (Garbage Collector) is responsible for cleaning our garbage.

While **GC** cleans our garbage, he removes the unused objects from the managed heap which cause heap fragmentation. When **GC** is done with the removal, it performs a heap compression (defragmintation) which involves moving objects on the heap.

Since **GC** isn't deterministic, when passing managed object reference/pointer to native code, **GC** can kick in at any time, if it occurs just after Inerop call, there is a very good possibility that object (which reference passed to native) will be moved on the managed heap - as a result, we get an invalid reference on managed side.

In this scenario, you should **pin** the object before passing it to native code.

Pinned Object

Pinned object is an object that is not allowed to move by GC.

Gc Pinned Handle

You can create a pin object using **Gc.Alloc** method

```
GCHandle handle = GCHandle.Alloc(yourObject, GCHandleType.Pinned);
```

- Obtaining a pinned **GCHandle** to managed object marks a specific object as one that cannot be moved by **GC**, until freeing the handle

Example:

```
[DllImport("kernel32.dll", SetLastError = true)]
```

```
public static extern void EnterCriticalSection(IntPtr ptr);

[DllImport("kernel32.dll", SetLastError = true)]
public static extern void LeaveCriticalSection(IntPtr ptr);

public void EnterCriticalSection(CRITICAL_SECTION section)
{
    try
    {
        GCHandle handle = GCHandle.Alloc(section, GCHandleType.Pinned);
        EnterCriticalSection(handle.AddrOfPinnedObject());
        //Do Some Critical Work
        LeaveCriticalSection(handle.AddrOfPinnedObject());
    }
    finaly
    {
        handle.Free()
    }
}
```

Precautions

- When pinning (especially large ones) object try to release the pinned **GcHandle** as fast as possible, since it interrupt heap defragmentation.
- If you forget to free **GcHandle** nothing will. Do it in a safe code section (such as finaly)

Section 155.8: Simple code to expose class for com

```
using System;
using System.Runtime.InteropServices;

namespace ComLibrary
{
    [ComVisible(true)]
    public interface IMainType
    {
        int GetInt();

        void StartTime();

        int StopTime();
    }

    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.None)]
    public class MainType : IMainType
    {
        private Stopwatch stopWatch;

        public int GetInt()
        {
            return 0;
        }

        public void StartTime()
        {
            stopWatch= new Stopwatch();
            stopWatch.Start();
        }
```

```
        public int StopTime()
        {
            return (int)stopWatch.ElapsedMilliseconds;
        }
    }
}
```

Chapter 156: .NET Compiler Platform (Roslyn)

Section 156.1: Semantic model

A **Semantic Model** offers a deeper level of interpretation and insight of code compare to a syntax tree. Where syntax trees can tell the names of variables, semantic models also give the type and all references. Syntax trees notice method calls, but semantic models give references to the precise location the method is declared (after overload resolution has been applied.)

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var sln = await workspace.OpenSolutionAsync(solutionFilePath);
var project = sln.Projects.First();
var compilation = await project.GetCompilationAsync();

foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();

    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>();

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di.Identifier);
        // => "root"

        var variableSymbol = compilation
            .GetSemanticModel(syntaxTree)
            .GetDeclaredSymbol(di) as ILocalSymbol;

        Console.WriteLine(variableSymbol.Type);
        // => "Microsoft.CodeAnalysis.SyntaxNode"

        var references = await SymbolFinder.FindReferencesAsync(variableSymbol, sln);
        foreach (var reference in references)
        {
            foreach (var loc in reference.Locations)
            {
                Console.WriteLine(loc.Location.SourceSpan);
                // => "[1375..1379)"
            }
        }
    }
}
```

This outputs a list of local variables using a syntax tree. Then it consults the semantic model to get the full type name and find all references of every variable.

Section 156.2: Syntax tree

A **Syntax Tree** is an immutable data structure representing the program as a tree of names, commands and marks (as previously configured in the editor.)

For example, assume a `Microsoft.CodeAnalysis.Compilation` instance named `compilation` has been configured.

There are multiple ways to list the names of every variable declared in the loaded code. To do so naively, take all pieces of syntax in every document (the `DescendantNodes` method) and use Linq to select nodes that describe variable declaration:

```
foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();
    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>()
        .Select(vd => vd.Identifier);

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di);
    }
}
```

Every type of C# construct with a corresponding type will exist in the syntax tree. To quickly find specific types, use the `Syntax Visualizer` window from Visual Studio. This will interpret the current opened document as a Roslyn syntax tree.

Section 156.3: Create workspace from MSBuild project

First obtain the `Microsoft.CodeAnalysis.CSharp.Workspaces` nuget before continuing.

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var project = await workspace.OpenProjectAsync(projectFilePath);
var compilation = await project.GetCompilationAsync();

foreach (var diagnostic in compilation.GetDiagnostics()
    .Where(d => d.Severity == Microsoft.CodeAnalysis.DiagnosticSeverity.Error))
{
    Console.WriteLine(diagnostic);
}
```

To load existing code to the workspace, compile and report errors. Afterwards the code will be located in memory. From here, both the syntactic and semantic side will be available to work with.

Chapter 157: ILGenerator

Section 157.1: Creates a DynamicAssembly that contains a UnixTimestamp helper method

This example shows the usage of the ILGenerator by generating code that makes use of already existing and new created members as well as basic Exception handling. The following code emits a DynamicAssembly that contains an equivalent to this c# code:

```
public static class UnixTimeHelper
{
    private readonly static DateTime EpochTime = new DateTime(1970, 1, 1);

    public static int UnixTimestamp(DateTime input)
    {
        int totalSeconds;
        try
        {
            totalSeconds = checked((int)input.Subtract(UnixTimeHelper.EpochTime).TotalSeconds);
        }
        catch (OverflowException overflowException)
        {
            throw new InvalidOperationException("It's to late for an Int32 timestamp.",
overflowException);
        }
        return totalSeconds;
    }
}
```

```
//Get the required methods
var dateTimeCtor = typeof (DateTime)
    .GetConstructor(new[] {typeof (int), typeof (int), typeof (int)});
var dateTimeSubstract = typeof (DateTime)
    .GetMethod(nameof(DateTime.Subtract), new[] {typeof (DateTime)});
var timeSpanSecondsGetter = typeof (TimeSpan)
    .GetProperty(nameof(TimeSpan.TotalSeconds)).GetGetMethod();
var invalidOperationCtor = typeof (InvalidOperationException)
    .GetConstructor(new[] {typeof (string), typeof (Exception)});

if (dateTimeCtor == null || dateTimeSubstract == null ||
    timeSpanSecondsGetter == null || invalidOperationCtor == null)
{
    throw new Exception("Could not find a required Method, can not create Assembly.");
}

//Setup the required members
var an = new AssemblyName("UnixTimeAsm");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(an, AssemblyBuilderAccess.RunAndSave);
var dynMod = dynAsm.DefineDynamicModule(an.Name, an.Name + ".dll");

var dynType = dynMod.DefineType("UnixTimeHelper",
    TypeAttributes.Abstract | TypeAttributes.Sealed | TypeAttributes.Public);

var epochTimeField = dynType.DefineField("EpochStartTime", typeof (DateTime),
    FieldAttributes.Private | FieldAttributes.Static | FieldAttributes.InitOnly);

var cctor =
    dynType.DefineConstructor(
```

```
        MethodAttributes.Private | MethodAttributes.HideBySig | MethodAttributes.SpecialName |
        MethodAttributes.RTSpecialName | MethodAttributes.Static, CallingConventions.Standard,
        Type.EmptyTypes);

var cctorGen = cctor.GetILGenerator();
cctorGen.Emit(OpCodes.Ldc_I4, 1970); //Load the DateTime constructor arguments onto the stack
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Newobj, dateTimeCtor); //Call the constructor
cctorGen.Emit(OpCodes.Stsfld, epochTimeField); //Store the object in the static field
cctorGen.Emit(OpCodes.Ret);

var unixTimestampMethod = dynType.DefineMethod("UnixTimestamp",
    MethodAttributes.Public | MethodAttributes.HideBySig | MethodAttributes.Static,
    CallingConventions.Standard, typeof (int), new[] {typeof (DateTime)});

unixTimestampMethod.DefineParameter(1, ParameterAttributes.None, "input");

var methodGen = unixTimestampMethod.GetILGenerator();
methodGen.DeclareLocal(typeof (TimeSpan));
methodGen.DeclareLocal(typeof (int));
methodGen.DeclareLocal(typeof (OverflowException));

methodGen.BeginExceptionBlock(); //Begin the try block
methodGen.Emit(OpCodes.Ldarga_S, (byte) 0); //To call a method on a struct we need to load the
address of it
methodGen.Emit(OpCodes.Ldsfld, epochTimeField);
    //Load the object of the static field we created as argument for the following call
methodGen.Emit(OpCodes.Call, dateTimeSubstract); //Call the substract method on the input DateTime
methodGen.Emit(OpCodes.Stloc_0); //Store the resulting TimeSpan in a local
methodGen.Emit(OpCodes.Ldloca_S, (byte) 0); //Load the locals address to call a method on it
methodGen.Emit(OpCodes.Call, timeSpanSecondsGetter); //Call the TotalSeconds Get method on the
TimeSpan
methodGen.Emit(OpCodes.Conv_Ovf_I4); //Convert the result to Int32; throws an exception on overflow
methodGen.Emit(OpCodes.Stloc_1); //store the result for returning later
//The leave instruction to jump behind the catch block will be automatically emitted
methodGen.BeginCatchBlock(typeof (OverflowException)); //Begin the catch block
//When we are here, an OverflowException was thrown, that is now on the stack
methodGen.Emit(OpCodes.Stloc_2); //Store the exception in a local.
methodGen.Emit(OpCodes.Ldstr, "It's to late for an Int32 timestamp.");
    //Load our error message onto the stack
methodGen.Emit(OpCodes.Ldloc_2); //Load the exception again
methodGen.Emit(OpCodes.Newobj, invalidOperationCtor);
    //Create an InvalidOperationException with our message and inner Exception
methodGen.Emit(OpCodes.Throw); //Throw the created exception
methodGen.EndExceptionBlock(); //End the catch block
//When we are here, everything is fine
methodGen.Emit(OpCodes.Ldloc_1); //Load the result value
methodGen.Emit(OpCodes.Ret); //Return it

dynType.CreateType();

dynAsm.Save(an.Name + ".dll");
```

Section 157.2: Create method override

This example shows how to override `ToString` method in generated class

```
// create an Assembly and new type
var name = new AssemblyName("MethodOverriding");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(name, AssemblyBuilderAccess.RunAndSave);
```

```
var dynModule = dynAsm.DefineDynamicModule(name.Name, $"{name.Name}.dll");
var typeBuilder = dynModule.DefineType("MyClass", TypeAttributes.Public | TypeAttributes.Class);

// define a new method
var toStr = typeBuilder.DefineMethod(
    "ToString", // name
    MethodAttributes.Public | MethodAttributes.Virtual, // modifiers
    typeof(string), // return type
    Type.EmptyTypes); // argument types
var ilGen = toStr.GetILGenerator();
ilGen.Emit(OpCodes.Ldstr, "Hello, world!");
ilGen.Emit(OpCodes.Ret);

// set this method as override of object.ToString
typeBuilder.DefineMethodOverride(toStr, typeof(object).GetMethod("ToString"));
var type = typeBuilder.CreateType();

// now test it:
var instance = Activator.CreateInstance(type);
Console.WriteLine(instance.ToString());
```

Chapter 158: T4 Code Generation

Section 158.1: Runtime Code Generation

```
<#@ template language="C#" #> //Language of your project
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
```

Chapter 159: Creating Own MessageBox in Windows Form Application

First we need to know what a MessageBox is...

The MessageBox control displays a message with specified text, and can be customised by specifying a custom image, title and button sets (These button sets allow the user to choose more than a basic yes/no answer).

By creating our own MessageBox we can re-use that MessageBox Control in any new applications just by using the generated dll, or copying the file containing the class.

Section 159.1: How to use own created MessageBox control in another Windows Form application

To find your existing .cs files, right click on the project in your instance of Visual Studio, and click Open Folder in File Explorer.

1. Visual Studio --> Your current project (Windows Form) --> Solution Explorer --> Project Name --> Right Click --> Add --> Existing Item --> Then locate your existing .cs file.
2. Now there's one last thing to do in order to use the control. Add a using statement to your code, so that your assembly knows about its dependencies.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
.
.
.
using CustomMsgBox; //Here's the using statement for our dependency.
```

3. To display the messagebox, simply use the following...

   CustomMsgBox.Show("Your Message for Message Box...","MSG","OK");

Section 159.2: Creating Own MessageBox Control

To create our own MessageBox control simply follow the guide below...

1. Open up your instance of Visual Studio (VS 2008/2010/2012/2015/2017)
2. Go to the toolbar at the top and click File -> New Project --> Windows Forms Application --> Give the project a name and then click ok.
3. Once loaded, drag and drop a button control from the Toolbox (found on the left) onto the form (as shown below).

4. Double click the button and the Integrated Development Environment will automatically generate the click event handler for you.
5. Edit the code for the form so that it looks like the following (You can right-click the form and click Edit Code):

```
namespace MsgBoxExample {
    public partial class MsgBoxExampleForm : Form {
        //Constructor, called when the class is initialised.
        public MsgBoxExampleForm() {
            InitializeComponent();
        }

        //Called whenever the button is clicked.
        private void btnShowMessageBox_Click(object sender, EventArgs e) {
           CustomMsgBox.Show($"I'm a {nameof(CustomMsgBox)}!", "MSG", "OK");
        }
    }
}
```

6. Solution Explorer -> Right Click on your project --> Add --> Windows Form and set the name as "CustomMsgBox.cs"
7. Drag in a button & label control from the Toolbox to the form (It'll look something like the form below after doing it):

8. Now write out the code below into the newly created form:

```
private DialogResult result = DialogResult.No;
public static DialogResult Show(string text, string caption, string btnOkText) {
    var msgBox = new CustomMsgBox();
    msgBox.lblText.Text = text; //The text for the label...
    msgBox.Text = caption; //Title of form
    msgBox.btnOk.Text = btnOkText; //Text on the button
    //This method is blocking, and will only return once the user
    //clicks ok or closes the form.
    msgBox.ShowDialog();
    return result;
}

private void btnOk_Click(object sender, EventArgs e) {
    result = DialogResult.Yes;
    MsgBox.Close();
}
```

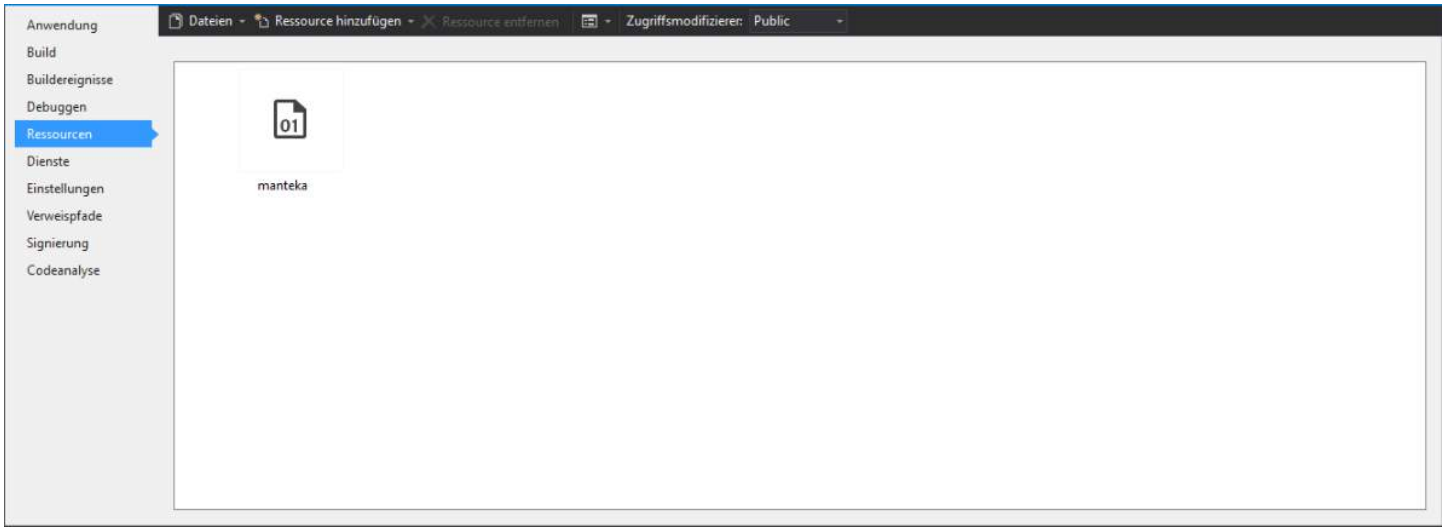
9. Now run the program by just pressing F5 Key. Congratulations, you've made a reusable control.

Chapter 160: Including Font Resources

Parameter	Details
fontbytes	byte array from the binary .ttf

Section 160.1: Instantiate 'Fontfamily' from Resources

```
public FontFamily Maneteke = GetResourceFontFamily(Properties.Resources.manteka);
```



Section 160.2: Integration method

```
public static FontFamily GetResourceFontFamily(byte[] fontbytes)
{
    PrivateFontCollection pfc = new PrivateFontCollection();
    IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
    Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
    pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
    Marshal.FreeCoTaskMem(fontMemPointer);
    return pfc.Families[0];
}
```

Section 160.3: Usage with a 'Button'

```
    public static class Res
    {
        /// <summary>
        /// URL: https://www.behance.net/gallery/2846011/Manteka
        /// </summary>
        public static FontFamily Maneteke = GetResourceFontFamily(Properties.Resources.manteka);

        public static FontFamily GetResourceFontFamily(byte[] fontbytes)
        {
            PrivateFontCollection pfc = new PrivateFontCollection();
            IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
            Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
            pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
            Marshal.FreeCoTaskMem(fontMemPointer);
            return pfc.Families[0];
        }
    }
```

```
public class FlatButton : Button
{
    public FlatButton() : base()
    {
        Font = new Font(Res.Maneteke, Font.Size);
    }

    protected override void OnFontChanged(EventArgs e)
    {
        base.OnFontChanged(e);
        this.Font = new Font(Res.Maneteke, this.Font.Size);
    }
}
```

Chapter 161: Import Google Contacts

Section 161.1: Requirements

To Import Google(Gmail) contacts in ASP.NET MVC application, first download "Google API setup" This will grant the following references:

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
```

Add these to the relevant application.

Section 161.2: Source code in the controller

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using System.Web;
using System.Web.Mvc;

namespace GoogleContactImport.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Import()
        {
            string clientId = "";  // here you need to add your google client id
            string redirectUrl = "http://localhost:1713/Home/AddGoogleContacts"; // here your
redirect action method NOTE: you need to configure same url in google console
            Response.Redirect("https://accounts.google.com/o/oauth2/auth?redirect_uri=" +
redirectUrl + "&amp&response_type=code&amp&client_id=" + clientId +
"&amp&scope=https://www.google.com/m8/feeds/&amp;approval_prompt=force&amp;access_type=offline");
            return View();
        }

        public ActionResult AddGoogleContacts()
        {
            string code = Request.QueryString["code"];
            if (!string.IsNullOrEmpty(code))
            {
                var contacts = GetAccessToken().ToArray();
                if (contacts.Length > 0)
                {
```

```
                // You will get all contacts here
                    return View("Index",contacts);
                }
                else
                {
                    return RedirectToAction("Index","Home");
                }
            }
            else
            {
                return RedirectToAction("Index", "Home");
            }
        }
        public List<GmailContacts> GetAccessToken()
        {
            string code = Request.QueryString["code"];
            string google_client_id = ""; //your google client Id
            string google_client_sceret = ""; // your google secret key
            string google_redirect_url = "http://localhost:1713/MyContact/AddGoogleContacts";


            HttpWebRequest webRequest =
(HttpWebRequest)WebRequest.Create("https://accounts.google.com/o/oauth2/token");
            webRequest.Method = "POST";
            string parameters = "code=" + code + "&client_id=" + google_client_id +
"&client_secret=" + google_client_sceret + "&redirect_uri=" + google_redirect_url +
"&grant_type=authorization_code";
            byte[] byteArray = Encoding.UTF8.GetBytes(parameters);
            webRequest.ContentType = "application/x-www-form-urlencoded";
            webRequest.ContentLength = byteArray.Length;
            Stream postStream = webRequest.GetRequestStream();
            // Add the post data to the web request
            postStream.Write(byteArray, 0, byteArray.Length);
            postStream.Close();
            WebResponse response = webRequest.GetResponse();
            postStream = response.GetResponseStream();
            StreamReader reader = new StreamReader(postStream);
            string responseFromServer = reader.ReadToEnd();
            GooglePlusAccessToken serStatus =
JsonConvert.DeserializeObject<GooglePlusAccessToken>(responseFromServer);
            /*End*/
            return GetContacts(serStatus);
        }

        public List<GmailContacts> GetContacts(GooglePlusAccessToken serStatus)
        {
            string google_client_id = "";  //client id
            string google_client_sceret = ""; //secret key
            /*Get Google Contacts From Access Token and Refresh Token*/
            // string refreshToken = serStatus.refresh_token;
            string accessToken = serStatus.access_token;
            string scopes = "https://www.google.com/m8/feeds/contacts/default/full/";
            OAuth2Parameters oAuthparameters = new OAuth2Parameters()
            {
                ClientId = google_client_id,
                ClientSecret = google_client_sceret,
                RedirectUri = "http://localhost:1713/Home/AddGoogleContacts",
                Scope = scopes,
                AccessToken = accessToken,
                //  RefreshToken = refreshToken
            };
```

```
            RequestSettings settings = new RequestSettings("App Name", oAuthparameters);
            ContactsRequest cr = new ContactsRequest(settings);
            ContactsQuery query = new ContactsQuery(ContactsQuery.CreateContactsUri("default"));
            query.NumberToRetrieve = 5000;
            Feed<Contact> ContactList = cr.GetContacts();

            List<GmailContacts> olist = new List<GmailContacts>();
            foreach (Contact contact in ContactList.Entries)
            {
                foreach (EMail email in contact.Emails)
                {
                    GmailContacts gc = new GmailContacts();
                    gc.EmailID = email.Address;
                    var a = contact.Name.FullName;
                    olist.Add(gc);
                }
            }
            return olist;
        }

        public class GmailContacts
        {
            public string EmailID
            {
                get { return _EmailID; }
                set { _EmailID = value; }
            }
            private string _EmailID;
        }

        public class GooglePlusAccessToken
        {

            public GooglePlusAccessToken()
            { }


            public string access_token
            {
                get { return _access_token; }
                set { _access_token = value; }
            }
            private string _access_token;

            public string token_type
            {
                get { return _token_type; }
                set { _token_type = value; }
            }
            private string _token_type;

            public string expires_in
            {
                get { return _expires_in; }
                set { _expires_in = value; }
            }
            private string _expires_in;

        }
    }
}
```

Section 161.3: Source code in the view

The only action method you need to add is to add an action link present below

```
<a href='@Url.Action("Import", "Home")'>Import Google Contacts</a>
```

Chapter 162: Garbage Collector in .Net

Section 162.1: Weak References

In .NET, the GC allocates objects when there are no references left to them. Therefore, while an object can still be reached from code (there is a strong reference to it), the GC will not allocate this object. This can become a problem if there are a lot of large objects.

A weak reference is a reference, that allows the GC to collect the object while still allowing to access the object. A weak reference is valid only during the indeterminate amount of time until the object is collected when no strong references exist. When you use a weak reference, the application can still obtain a strong reference to the object, which prevents it from being collected. So weak references can be useful for holding on to large objects that are expensive to initialize, but should be available for garbage collection if they are not actively in use.

Simple usage:

```
WeakReference reference = new WeakReference(new object(), false);

GC.Collect();

object target = reference.Target;
if (target != null)
  DoSomething(target);
```

So weak references could be used to maintain, for example, a cache of objects. However, it is important to remember that there is always the risk that the garbage collector will get to the object before a strong reference is reestablished.

Weak references are also handy for avoiding memory leaks. A typical use case is with events.

Suppose we have some handler to an event on a source:

```
Source.Event += new EventHandler(Handler)
```

This code registers an event handler and creates a strong reference from the event source to the listening object. If the source object has a longer lifetime than the listener, and the listener doesn't need the event anymore when there are no other references to it, using normal .NET events causes a memory leak: the source object holds listener objects in memory that should be garbage collected.

In this case, it may be a good idea is to use the Weak Event Pattern.

Something like:

```
public static class WeakEventManager
    {
    public static void SetHandler<S, TArgs>(
    Action<EventHandler<TArgs>> add,
    Action<EventHandler<TArgs>> remove,
    S subscriber,
    Action<S, TArgs> action)
    where TArgs : EventArgs
    where S : class
        {
            var subscrWeakRef = new WeakReference(subscriber);
            EventHandler<TArgs> handler = null;
```

```
            handler = (s, e) =>
            {
                var subscrStrongRef = subscrWeakRef.Target as S;
                if (subscrStrongRef != null)
                {
                    action(subscrStrongRef, e);
                }
                else
                {
                    remove(handler);
                    handler = null;
                }
            };

            add(handler);
        }
    }
```

and used like this:

```
 EventSource s = new EventSource();
 Subscriber subscriber = new Subscriber();
 WeakEventManager.SetHandler<Subscriber, SomeEventArgs>(a => s.Event += a, r => s.Event -= r,
subscriber, (s,e) => { s.HandleEvent(e); });
```

In this case of course we have some restrictions - the event must be a

```
public event EventHandler<SomeEventArgs> Event;
```

As MSDN suggests:

- Use long weak references only when necessary as the state of the object is unpredictable after finalization.
- Avoid using weak references to small objects because the pointer itself may be as large or larger.
- Avoid using weak references as an automatic solution to memory management problems. Instead, develop an effective caching policy for handling your application's objects.

Section 162.2: Large Object Heap compaction

By default the Large Object Heap is not compacted unlike the classic Object Heap which can lead to memory fragmentation and further, can lead to `OutOfMemoryExceptions`

Starting with .NET 4.5.1 there is an option to explicitly compact the Large Object Heap (along with a garbage collection):

```
GCSettings.LargeObjectHeapCompactionMode = GCLargeObjectHeapCompactionMode.CompactOnce;
GC.Collect();
```

Just as any explicit garbage collection request (it's called request because the CLR is not forced to conduct it) use with care and by default avoid it if you can since it can de-calibrate `GC`s statistics, decreasing its performance.

Chapter 163: Microsoft.Exchange.WebServices

Section 163.1: Retrieve Specified User's Out of Office Settings

First let's create an `ExchangeManager` object, where the constructor will connect to the services for us. It also has a `GetOofSettings` method, which will return the `OofSettings` object for the specified email address :

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;

namespace SetOutOfOffice
{
    class ExchangeManager
    {
        private ExchangeService Service;

        public ExchangeManager()
        {
            var password = WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
            Connect("exchangeadmin", password);
        }
        private void Connect(string username, string password)
        {
            var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
            service.Credentials = new WebCredentials(username, password);
            service.AutodiscoverUrl("autodiscoveremail@domain.com" ,
RedirectionUrlValidationCallback);

            Service = service;
        }
        private static bool RedirectionUrlValidationCallback(string redirectionUrl)
        {
            return redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
        }
        public OofSettings GetOofSettings(string email)
        {
            return Service.GetUserOofSettings(email);
        }
    }
}
```

We can now call this elsewhere like this:

```
var em = new ExchangeManager();
var oofSettings = em.GetOofSettings("testemail@domain.com");
```

Section 163.2: Update Specific User's Out of Office Settings

Using the class below, we can connect to Exchange and then set a specific user's out of office settings with `UpdateUserOof`:

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;
```

```
class ExchangeManager
{
    private ExchangeService Service;

    public ExchangeManager()
    {
        var password = WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
        Connect("exchangeadmin", password);
    }
    private void Connect(string username, string password)
    {
        var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
        service.Credentials = new WebCredentials(username, password);
        service.AutodiscoverUrl("autodiscoveremail@domain.com" , RedirectionUrlValidationCallback);

        Service = service;
    }
    private static bool RedirectionUrlValidationCallback(string redirectionUrl)
    {
        return redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
    }
    /// <summary>
    /// Updates the given user's Oof settings with the given details
    /// </summary>
    public void UpdateUserOof(int oofstate, DateTime starttime, DateTime endtime, int
externalaudience, string internalmsg, string externalmsg, string emailaddress)
    {
        var newSettings = new OofSettings
        {
            State = (OofState)oofstate,
            Duration = new TimeWindow(starttime, endtime),
            ExternalAudience = (OofExternalAudience)externalaudience,
            InternalReply = internalmsg,
            ExternalReply = externalmsg
        };

        Service.SetUserOofSettings(emailaddress, newSettings);
    }
}
```

Update the user settings with the following:

```
var oofState = 1;
var startDate = new DateTime(01,08,2016);
var endDate = new DateTime(15,08,2016);
var externalAudience = 1;
var internalMessage = "I am not in the office!";
var externalMessage = "I am not in the office <strong>and neither are you!</strong>"
var theUser = "theuser@domain.com";

var em = new ExchangeManager();
em.UpdateUserOof(oofstate, startDate, endDate, externalAudience, internalMessage, externalMessage,
theUser);
```

Note that you can format the messages using standard `html` tags.

Chapter 164: Windows Communication Foundation

Windows Communication Foundation (WCF) is a framework for building service-oriented applications. Using WCF, you can send data as asynchronous messages from one service endpoint to another. A service endpoint can be part of a continuously available service hosted by IIS, or it can be a service hosted in an application. The messages can be as simple as a single character or word sent as XML, or as complex as a stream of binary data.

Section 164.1: Getting started sample

The service describes the operations it performs in a service contract that it exposes publicly as metadata.

```
// Define a service contract.
[ServiceContract(Namespace="http://StackOverflow.ServiceModel.Samples")]
public interface ICalculator
{
    [OperationContract]
    double Add(double n1, double n2);
}
```

The service implementation calculates and returns the appropriate result, as shown in the following example code.

```
// Service class that implements the service contract.
public class CalculatorService : ICalculator
{
    public double Add(double n1, double n2)
    {
        return n1 + n2;
    }
}
```

The service exposes an endpoint for communicating with the service, defined using a configuration file (Web.config), as shown in the following sample configuration.

```
<services>
    <service
       name="StackOverflow.ServiceModel.Samples.CalculatorService"
       behaviorConfiguration="CalculatorServiceBehavior">
        <!-- ICalculator is exposed at the base address provided by
        host: http://localhost/servicemodelsamples/service.svc.  -->
       <endpoint address=""
             binding="wsHttpBinding"
             contract="StackOverflow.ServiceModel.Samples.ICalculator" />
       ...
    </service>
</services>
```

The framework does not expose metadata by default. As such, the service turns on the ServiceMetadataBehavior and exposes a metadata exchange (MEX) endpoint at http://localhost/servicemodelsamples/service.svc/mex. The following configuration demonstrates this.

```
<system.serviceModel>
  <services>
    <service
        name="StackOverflow.ServiceModel.Samples.CalculatorService"
```

```
        behaviorConfiguration="CalculatorServiceBehavior">
      ...
      <!-- the mex endpoint is explosed at
       http://localhost/servicemodelsamples/service.svc/mex -->
      <endpoint address="mex"
                binding="mexHttpBinding"
                contract="IMetadataExchange" />
    </service>
  </services>

  <!--For debugging purposes set the includeExceptionDetailInFaults
   attribute to true-->
  <behaviors>
    <serviceBehaviors>
      <behavior name="CalculatorServiceBehavior">
        <serviceMetadata httpGetEnabled="True"/>
        <serviceDebug includeExceptionDetailInFaults="False" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

The client communicates using a given contract type by using a client class that is generated by the ServiceModel Metadata Utility Tool (Svcutil.exe).

Run the following command from the SDK command prompt in the client directory to generate the typed proxy:

```
svcutil.exe /n:"http://StackOverflow.ServiceModel.Samples,StackOverflow.ServiceModel.Samples"
http://localhost/servicemodelsamples/service.svc/mex /out:generatedClient.cs
```

Like the service, the client uses a configuration file (App.config) to specify the endpoint with which it wants to communicate. The client endpoint configuration consists of an absolute address for the service endpoint, the binding, and the contract, as shown in the following example.

```
<client>
     <endpoint
        address="http://localhost/servicemodelsamples/service.svc"
        binding="wsHttpBinding"
        contract="StackOverflow.ServiceModel.Samples.ICalculator" />
</client>
```

The client implementation instantiates the client and uses the typed interface to begin communicating with the service, as shown in the following example code.

```
// Create a client.
CalculatorClient client = new CalculatorClient();

// Call the Add service operation.
double value1 = 100.00D;
double value2 = 15.99D;
double result = client.Add(value1, value2);
Console.WriteLine("Add({0},{1}) = {2}", value1, value2, result);

//Closing the client releases all communication resources.
client.Close();
```

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

Đan	Chapter 48
4444	Chapter 156
Aaron Anodide	Chapter 72
Aaron Hudon	Chapters 20, 22, 27, 48, 89, 96, 114 and 120
aashishkoirala	Chapter 7
Abbas Galiyakotwala	Chapter 139
Abdul Rehman Sayed	Chapters 13, 22, 31 and 76
AbdulRahman Ansari	Chapter 19
abishekshivan	Chapter 1
Abob	Chapter 73
abto	Chapter 94
Adam	Chapters 96, 111 and 124
Adam Clifford	Chapter 66
Adam Houldsworth	Chapters 3 and 60
Adam Sills	Chapter 39
Ade Stringer	Chapters 1, 52, 66 and 105
Adi Lester	Chapters 39, 43, 48, 52, 66, 68, 89, 98, 148 and 149
Adil Mammadov	Chapters 3, 48, 66, 71 and 72
Aditya Korti	Chapter 52
Adriano Repetti	Chapters 39, 52 and 72
AFT	Chapter 21
AGB	Chapters 59 and 72
Ahmet Can Aydemir	Chapter 149
AJ.	Chapter 52
Akshay Anand	Chapters 3 and 81
Alan McBee	Chapters 3 and 10
Aleks Andreev	Chapters 83 and 157
Aleksey L.	Chapter 66
alex	Chapter 127
Alex Wiese	Chapter 60
Alexander Mandt	Chapters 4, 56, 95 and 149
Alexander Pacha	Chapter 52
Alexey Koptyaev	Chapter 66
Alexey Groshev	Chapters 20 and 94
Aliaksei Futryn	Chapter 130
Alisson	Chapter 28
Alok Singh	Chapter 1
Alpha	Chapter 8
Aman Sharma	Chapter 63
Ameya Deshpande	Chapters 4 and 144
Amir Pourmand امیر پورمند	Chapter 31
Anas Tasadduq	Chapter 20
anaximander	Chapters 48, 66 and 72
Andrei	Chapters 46 and 93
Andrei Epure	Chapter 122
Andrei Rînea	Chapters 28, 39, 43, 52 and 162

Andrew Diamond	Chapters 39, 71, 76 and 95
Andrew Piliser	Chapter 152
Andrew Stollak	Chapter 138
andre_ss6	Chapters 58, 59 and 60
Andrius	Chapter 12
Angela	Chapter 52
Ankit Rana	Chapter 7
Ankit Vijay	Chapter 66
Anna	Chapter 52
Ant P	Chapter 114
Aphelion	Chapters 26 and 48
Aravind Suresh	Chapter 1
Ares	Chapter 72
Arjan Einbu	Chapters 10 and 15
Arthur Rizzo	Chapter 72
Aseem Gautam	Chapters 12, 63 and 76
ASh	Chapter 26
ATechieThought	Chapter 48
Athafoud	Chapter 76
Athari	Chapters 3, 16, 26, 48, 52, 57, 71, 72, 74, 76, 89, 153 and 155
Austin T French	Chapter 112
Avi Turner	Chapters 3 and 71
Avia	Chapters 30, 39, 41 and 52
avs099	Chapter 15
Axarydax	Chapter 63
A_Arnold	Chapters 1, 52, 72 and 107
Bad	Chapter 57
Balen Danny	Chapter 155
BanksySan	Chapters 39 and 97
Bassie	Chapter 163
Ben Aaronson	Chapters 48, 55, 59, 72, 96 and 120
Ben Fogel	Chapter 3
Ben Jenkinson	Chapter 45
Benjamin Hodgson	Chapters 38, 39, 59, 94 and 96
Benjol	Chapters 12, 39, 48, 58, 59, 60, 84 and 89
binki	Chapters 48 and 62
Bjørn	Chapter 52
Blachshma	Chapters 1, 11, 14, 22 and 97
Blorgbeard	Chapter 87
Blubberguy22	Chapter 3
bmadtiger	Chapter 66
bob0the0mighty	Chapter 75
BOBS	Chapter 66
Bobson	Chapter 59
Boggin	Chapters 122 and 144
Boris Callens	Chapter 60
Botond Balázs	Chapters 16, 24, 28, 40, 54, 57, 86, 108, 122 and 141
Botz3000	Chapters 47, 52 and 94
Bovaz	Chapters 23, 116 and 155
bpoiss	Chapter 72
Bradley Grainger	Chapter 60
Bradley Uffner	Chapters 60, 72 and 96
Brandon	Chapter 117

Braydie	Chapter 38
Brendan L	Chapter 15
brijber	Chapter 52
BrunoLM	Chapters 12 and 114
Bryan Crosby	Chapters 10 and 62
bumbeishvili	Chapter 66
BunkerMentality	Chapter 52
BurnsBA	Chapter 52
C4u	Chapters 18 and 19
Callum Watkins	Chapters 13, 48, 55, 72 and 86
Carlos Muñoz	Chapter 39
Carmine	Chapter 129
Carsten	Chapter 59
cbale	Chapter 66
Chad	Chapter 9
Chad Levy	Chapters 1 and 72
Chad McGrath	Chapters 48 and 66
CharithJ	Chapter 11
Charlie H	Chapter 72
Chawin	Chapter 23
Chief Wiggum	Chapter 11
Chris	Chapter 52
Chris H.	Chapter 66
Chris Jester	Chapter 3
Chris Marisic	Chapter 60
Chris Oldwood	Chapter 22
Christopher Currens	Chapter 119
Christopher Ronning	Chapter 1
ChrisWue	Chapters 60 and 62
Chronocide	Chapter 48
Chuu	Chapter 52
Cigano Morrison Mendez	Chapter 63
Cihan Yakar	Chapters 49, 52 and 119
codeape	Chapter 115
CodeCaster	Chapter 26
codekaizen	Chapter 152
CodeWarrior	Chapter 96
Confiqure	Chapter 1
coyote	Chapters 20 and 66
Craig Brett	Chapters 31 and 114
Crowcoder	Chapter 70
cubrr	Chapters 16 and 48
CyberFox	Chapters 26 and 41
cyberj0g	Chapter 11
Dale Chen	Chapter 71
Dan Hulme	Chapter 39
Daniel	Chapter 52
Daniel Argüelles	Chapter 66
Daniel J.G.	Chapter 52
Daniel Stradowski	Chapter 72
Danny Chen	Chapters 9 and 49
Danny Varod	Chapter 152
Darren Davies	Chapter 52

Daryl	Chapters 35 and 52
dasblinkenlight	Chapter 90
das_keyboard	Chapter 20
Dave Zych	Chapter 15
David	Chapters 35, 52, 66, 72, 94 and 125
David Pine	Chapters 7, 48, 52, 66, 71, 72, 84 and 120
David Stockinger	Chapter 60
dav_i	Chapter 60
DAXaholic	Chapters 1 and 52
dazerdude	Chapters 8 and 60
da_sann	Chapters 63 and 162
dbmuller	Chapter 97
dcastro	Chapter 89
deadManN	Chapter 52
Deepak gupta	Chapters 48 and 72
delete me	Chapter 145
deloreyk	Chapters 11 and 58
demonplus	Chapter 7
Denis Elkhov	Chapters 7 and 67
Dialecticus	Chapter 28
Didgeridoo	Chapter 71
die maus	Chapter 66
Dileep	Chapter 90
Diligent Key Presser	Chapters 22, 71 and 119
Disk Crasher	Chapter 43
DividedByZero	Chapter 1
DJCubed	Chapters 28 and 97
dlatikay	Chapter 22
DLeh	Chapters 3, 8, 10, 15, 26, 39, 48, 100, 142 and 149
Dmitry Bychenko	Chapters 22, 48, 52, 60 and 66
Dmitry Egorov	Chapters 12 and 66
DmitryG	Chapter 38
docesam	Chapter 31
Dodzi Dzakuma	Chapter 144
Donald Webb	Chapter 114
Doruk	Chapters 13 and 95
dotctor	Chapters 3, 66, 111 and 120
dove	Chapters 22 and 144
DPenner1	Chapter 52
Draken	Chapter 116
Drew Kennedy	Chapter 52
DrewJordan	Chapter 52
Droritos	Chapter 121
Dunno	Chapter 91
Durgpal Singh	Chapters 1 and 124
DVJex	Chapter 63
Dweeberly	Chapter 13
DWright	Chapters 1, 48 and 134
Eamon Charles	Chapters 7 and 162
ebattulga	Chapter 12
Ed Gibbs	Chapter 16
EFrank	Chapter 23
Ehsan Sajjad	Chapters 1, 7, 10, 15, 36, 39, 43, 48, 52, 58, 59, 62, 63, 66, 72 and 120

EJoshuaS	Chapters 4, 52 and 114
ekolis	Chapter 48
el2iot2	Chapter 48
Elad Lachmi	Chapters 52 and 94
Eldar Dordzhiev	Chapter 112
elibyy	Chapter 24
Elton	Chapter 48
enrico.bacis	Chapters 48 and 72
eouw0o83hf	Chapter 7
ephtee	Chapters 54, 110 and 116
Erick	Chapter 66
Erik Schierboom	Chapters 60, 66 and 72
EvenPrime	Chapters 52, 66 and 148
fabriciorissetto	Chapters 15 and 66
Facebamm	Chapter 160
fahadash	Chapter 48
faso	Chapters 66 and 72
Federico Allocati	Chapter 84
Feelbad Soussi Wolfgun DZ	Chapter 97
Felipe Oriani	Chapter 19
fernacolo	Chapter 52
Fernando Matsumoto	Chapters 3, 20, 34, 48 and 52
Finickyflame	Chapter 66
Florian Koch	Chapter 1
Florin M	Chapter 66
flq	Chapter 76
forsvarir	Chapter 52
FortyTwo	Chapter 65
Freelex	Chapter 28
fubo	Chapters 47 and 66
F_V	Chapters 32, 47 and 52
G	Chapter 1
Gajendra	Chapters 120 and 155
ganchito55	Chapters 94, 105 and 148
Gavin Greenwalt	Chapters 52 and 111
gbellmann	Chapter 66
gdyrrahitis	Chapter 114
gdziadkiewicz	Chapter 48
George Duckett	Chapters 8, 76, 90, 119 and 120
GeralexGR	Chapter 147
Gilad Green	Chapter 66
Gilad Naaman	Chapters 52 and 76
GingerHead	Chapter 72
glaubergft	Chapter 150
Gordon Bell	Chapters 1 and 99
goric	Chapter 34
granmirupa	Chapter 20
GregC	Chapter 59
Guilherme de Jesus Santos	Chapter 19
guntbert	Chapter 114
H. Pauwelyn	Chapters 4, 12, 15, 22, 40, 72 and 75
haim770	Chapter 84

hankide	Chapter 72
Hari Prasad	Chapter 66
harriyott	Chapter 58
Haseeb Asif	Chapter 32
hatcyl	Chapters 6 and 40
Hayden	Chapter 4
hellyale	Chapters 41 and 66
Henrik H	Chapters 3 and 71
HimBromBeere	Chapters 66 and 120
iaminvinicble	Chapter 44
Ian	Chapter 39
Icy Defiance	Chapter 71
Igor	Chapter 148
iliketocode	Chapters 1, 71 and 72
intox	Chapter 97
Ioannis Karadimas	Chapter 66
Isac	Chapter 152
Ivan Yurchenko	Chapter 72
J. Steen	Chapter 72
J3soon	Chapter 149
ja72	Chapters 22, 58, 59 and 126
Jack	Chapter 76
Jacob Linney	Chapter 26
Jacobr365	Chapter 116
Jake	Chapter 52
Jake Farley	Chapter 3
James	Chapter 56
James Ellis	Chapters 63 and 66
James Hughes	Chapters 19, 25, 52 and 76
Jamie Rees	Chapters 63 and 72
Jamie Twells	Chapter 112
Jan Bońkowski	Chapters 134, 135 and 136
Jan Peldřimovský	Chapter 63
jao	Chapter 66
Jaroslav Kadlec	Chapter 152
Jarrod Dixon	Chapter 62
Jasmin Solanki	Chapters 3 and 47
jaycer	Chapter 87
Jaydip Jadhav	Chapters 11 and 20
Jcoffman	Chapter 52
Jean	Chapter 71
Jephron	Chapter 3
Jeppe Stig Nielsen	Chapters 3, 6, 16, 20, 48, 52, 72, 96 and 140
Jeremy Kato	Chapters 10, 17, 27, 41 and 47
Jeromy Irvine	Chapter 26
Jesse Williams	Chapters 3, 61 and 71
jHilscher	Chapters 107 and 148
Jim	Chapters 59 and 72
João Lourenço	Chapters 52 and 120
Jodrell	Chapter 66
Joe	Chapter 72
Joe Amenta	Chapter 120
Joel Martinez	Chapter 144

John	Chapters 40 and 86
John Meyer	Chapter 76
John Peters	Chapter 48
John Slegers	Chapters 48 and 52
Johnbot	Chapter 72
JohnLBevan	Chapters 3, 61, 86 and 112
Johny Skovdal	Chapter 63
Jojodmo	Chapter 72
Jon Bates	Chapter 66
Jon Erickson	Chapter 48
Jon Ericson	Chapter 58
Jon G	Chapter 66
Jon Schneider	Chapters 7, 11, 13, 14, 17, 20, 22, 27, 28, 38, 47, 52, 66, 70 and 99
Jon Skeet	Chapters 52 and 60
Jonas S	Chapters 3, 10, 26, 45, 48 and 72
Jonesopolis	Chapters 7 and 60
JoshuaBehrens	Chapter 52
JT.	Chapter 60
juergen d	Chapter 11
Juha Palomäki	Chapter 27
Julien Roncaglia	Chapters 39 and 120
just.another.programmer	Chapter 90
just.ru	Chapter 120
kamilk	Chapter 39
Kane	Chapter 9
karaken12	Chapter 66
Karthik	Chapter 105
Ken Keenan	Chapter 60
ken2k	Chapters 1 and 72
Kevin DiTraglia	Chapter 48
Kevin Green	Chapter 65
Kevin Montrose	Chapters 1 and 114
Kilazur	Chapters 10 and 35
Kimmax	Chapters 3 and 71
Kit	Chapter 61
Kjartan	Chapter 52
Kobi	Chapters 8, 40 and 60
Konamiman	Chapters 11, 39, 52, 58, 63 and 72
Konstantin Vdovkin	Chapter 38
Koopakiller	Chapter 66
Krikor Ailanjian	Chapter 72
krimog	Chapter 77
Kritner	Chapters 1 and 71
Kroltan	Chapter 46
Krzyserious	Chapter 1
Krzysztof Branicki	Chapter 43
Kyle Trauberman	Chapter 60
Kyrylo M	Chapter 113
Lasse Vågsæther Karlsen	Chapters 60 and 72
LegionMammal978	Chapter 60
leondepdelaw	Chapter 112
levininja	Chapter 52
Lijo	Chapters 29 and 36

lloyd	Chapter 158
LMK	Chapter 48
Lohitha Palagiri	Chapter 66
Lokesh_Ram	Chapter 104
lokusking	Chapters 3 and 52
Lorentz Vedeler	Chapter 60
LosManos	Chapter 95
lothlarias	Chapters 43 and 48
Lovy	Chapter 63
ltiveron	Chapter 39
Lukáš Lánský	Chapters 48 and 156
Lukas Kolletzki	Chapter 95
Luke Ryan	Chapters 30 and 80
M Monis Ahmed Khan	Chapter 1
MadddinTribleD	Chapter 76
maf	Chapter 71
Mafii	Chapters 52, 66 and 72
Makyen	Chapter 72
Mamta D	Chapters 52 and 112
Mango Wong	Chapter 52
Maniero	Chapter 11
ManselD	Chapter 159
Marc Gravell	Chapter 71
Marc Wittmann	Chapter 155
MarcE	Chapters 41 and 48
MarcinJuraszek	Chapters 52, 71 and 72
Marco	Chapter 47
Marco Scabbiolo	Chapter 52
Marek Musielak	Chapter 12
Mark Shevchenko	Chapters 12, 43, 72, 99, 115 and 134
Martin	Chapter 52
Martin Klinke	Chapter 52
Martin Zikmund	Chapters 7, 22, 48, 52, 60 and 66
Matas Vaitkevicius	Chapters 3, 12, 19, 48, 52, 58, 59, 62, 63, 66, 72, 76, 84, 90 and 93
Mateen Ulhaq	Chapters 1, 52 and 66
Mathieu Guindon	Chapter 52
Matt	Chapters 1, 1, 66, 72, 72, 76 and 96
Matt Rowland	Chapter 71
Matt Thomas	Chapter 71
matteeyah	Chapters 76, 89 and 90
Matthew Whited	Chapter 52
Matthijs Wessels	Chapter 112
Max	Chapters 8 and 52
Maxim	Chapter 125
Maxime	Chapters 22, 51, 62 and 66
Maximilian Ast	Chapters 38, 52 and 105
mbrdev	Chapters 72 and 88
mburleigh	Chapter 66
MCronin	Chapter 42
MDTech.us_MAN	Chapter 48
Medeni Baykal	Chapter 52
mehrandvd	Chapters 49 and 153
meJustAndrew	Chapters 3, 6, 48, 59, 89 and 90

Mellow	Chapters 72 and 112
MGB	Chapter 12
Michael B	Chapter 1
Michael Benford	Chapter 48
Michael Brandon Morris	Chapter 54
Michael Mairegger	Chapters 22, 27, 31 and 52
Michael Richardson	Chapters 27, 39, 48, 52, 72, 114 and 120
Michael Sorens	Chapter 87
Michele Ceo	Chapter 31
Michiel van Oosterhout	Chapter 64
mike z	Chapter 72
MikeS159	Chapter 150
Mikko Viitala	Chapters 3, 38, 51, 66, 97, 119 and 137
Millan Sanchez	Chapter 1
Milton Hernandez	Chapter 142
Mitch Talmadge	Chapter 72
Moe Farag	Chapter 31
Mohsin khan	Chapter 102
mok	Chapter 66
Mokhtar Ashour	Chapter 117
Mostafiz	Chapter 19
MotKohn	Chapters 9 and 71
Mourndark	Chapter 63
Mr.Mindor	Chapter 72
MrDKOz	Chapter 57
MSE	Chapters 37, 41 and 152
MSL	Chapter 47
Muhammad Albarmawi	Chapter 23
MuiBienCarlota	Chapters 72 and 147
Mujassir Nasir	Chapter 59
Mulder	Chapter 54
Murtuza Vohra	Chapter 117
mybirthname	Chapter 146
Nate Barbettini	Chapters 3, 15, 35, 66, 72 and 114
Nathan Tuggy	Chapter 90
Naveen Gogineni	Chapter 100
Neha Jain	Chapter 66
Neo Vijay	Chapter 82
Nicholas Sizer	Chapters 8 and 52
Nicholaus Lawson	Chapters 52 and 98
Nick	Chapter 1
Nick Larsen	Chapter 48
nickguletskii	Chapter 114
Nico	Chapters 3, 48, 66 and 71
nietras	Chapter 71
nik	Chapter 113
nikchi	Chapter 12
Nikhil Vartak	Chapter 86
Nikita	Chapters 23 and 27
NikolayKondratyev	Chapters 22, 71, 90, 94, 95, 96 and 129
Noctis	Chapter 66
nollidge	Chapter 72
NotEnoughData	Chapters 2, 52, 84 and 89

NotMyself	Chapter 1
NtFreX	Chapter 164
numaroth	Chapter 52
Nuri Tasdemir	Chapters 48, 52, 61, 63, 71, 72, 81, 109 and 119
Ogglas	Chapter 150
Ogoun	Chapter 20
Ojen	Chapter 1
Oliver Mellet	Chapter 72
Olivier De Meulder	Chapter 30
Oluwafemi	Chapters 13, 19, 28, 106 and 119
Oly	Chapters 3, 39, 59 and 84
Orlando William	Chapter 72
Pan	Chapter 54
Parth Patel	Chapters 48, 72, 87 and 134
pascalhein	Chapters 3 and 52
Patrick Hofman	Chapters 32 and 38
Paul Weiland	Chapter 11
paulius_l	Chapter 89
Pavel Durov	Chapter 155
Pavel Mayorov	Chapters 66 and 71
Pavel Pája Halbich	Chapters 52 and 76
Pavel Sapehin	Chapters 39 and 95
Pavel Voronin	Chapters 3, 59, 72, 76 and 114
Pavel Yermalovich	Chapters 19 and 52
Paweł Hemperek	Chapter 66
Paweł Mach	Chapter 52
Pedro	Chapter 66
PedroSouki	Chapters 85 and 134
Pete	Chapter 150
Pete Uh	Chapter 58
petelids	Chapters 3 and 7
Peter Gordon	Chapter 16
Peter Hommel	Chapter 39
Peter K	Chapter 60
Peter L.	Chapter 52
Petr Hudeček	Chapters 16 and 91
petrzjunior	Chapter 23
Philip C	Chapters 9 and 12
pid	Chapter 60
Pierre Theate	Chapter 134
pinkfloydx33	Chapters 9, 38, 39, 48, 66 and 114
PMF	Chapters 41, 52, 55 and 76
Prateek	Chapter 1
Proxima	Chapter 74
PSGuy	Chapter 89
Pushpendra	Chapter 109
Pyritie	Chapter 71
qJake	Chapters 66, 71, 72 and 93
Rémi	Chapter 66
Racil Hilan	Chapters 2, 3, 9, 11, 28, 66 and 72
Rahul Nikate	Chapters 7, 12, 33, 45 and 108
raidensan	Chapter 76
Raidri	Chapter 12

Raj Rao	Chapter 1
Rajput	Chapters 1 and 96
Rakitić	Chapters 1 and 52
Ralf Bönning	Chapter 144
Raphael Pantaleão	Chapter 65
ravindra	Chapter 128
RaYell	Chapter 23
Razan	Chapter 52
RBT	Chapter 52
rdans	Chapters 66 and 72
recursive	Chapter 22
René Vogt	Chapters 63 and 155
Renan Gemignani	Chapter 52
Reza Aghaei	Chapter 89
RhysO	Chapter 3
Ricardo Amores	Chapters 10 and 120
Richa Garg	Chapters 22, 59, 72 and 96
Richard	Chapters 12, 12, 72 and 119
richard	Chapter 122
Ringil	Chapter 52
Rion Williams	Chapters 1, 7, 10, 12, 15, 48, 52, 58, 60, 66, 72, 120 and 128
Rob	Chapters 26, 52, 72 and 119
Robban	Chapter 72
Robert Columbia	Chapters 3, 9, 17, 20, 39, 48, 52, 54, 58, 72 and 76
RobSiklos	Chapter 52
rocky	Chapter 120
RoelF	Chapter 11
Rokey Ge	Chapter 69
Roman Marusyk	Chapter 131
RomCoo	Chapters 1, 52 and 155
Roy Dictus	Chapters 54 and 132
RubberDuck	Chapter 13
Rubens Farias	Chapter 69
Ryan Abbott	Chapter 60
Ryan Hilbert	Chapter 72
ryanyuyu	Chapters 12, 52 and 72
S. Rangeley	Chapter 66
S.Akbari	Chapter 66
S.Dav	Chapter 47
S.L. Barth	Chapters 79 and 84
Sachin Chavan	Chapter 63
Salvador Rubio Martinez	Chapter 66
Sam	Chapters 3, 52 and 59
Sam Axe	Chapter 72
samuelesque	Chapter 71
Sanjay Radadiya	Chapters 52 and 66
Satish Yadav	Chapter 66
scher	Chapter 6
Scott	Chapter 52
Scott Koland	Chapters 60 and 76
Sean	Chapter 60
sebingel	Chapter 113
SeeuD1	Chapter 1

Serg Rogovtsev	Chapter 114
Sergio Domínguez	Chapter 66
sferencik	Chapter 14
shawty	Chapter 98
Shelby115	Chapter 77
Shoe	Chapter 72
Shuffler	Chapter 63
Shyju	Chapter 20
Sibeesh Venu	Chapters 48, 50 and 74
Sivanantham Padikkasu	Chapter 66
Sjoerd222888	Chapter 59
Skipper	Chapter 52
Sklivvz	Chapters 39 and 43
Skullomania	Chapter 151
SlaterCodes	Chapter 54
slawekwin	Chapter 3
slinzerthegod	Chapter 3
smead	Chapter 7
snickro	Chapter 43
Snipzwolf	Chapter 83
Snympi	Chapter 31
Sobieck	Chapter 52
sohnryang	Chapter 52
solidcell	Chapter 72
somebody	Chapter 52
Sompom	Chapter 120
Sondre	Chapters 19, 48, 52, 66, 101, 111 and 134
Sophie Jackson	Chapter 10
Squirrel	Chapters 43, 72 and 98
stackptr	Chapter 72
Stavm	Chapter 71
Stefan Steiger	Chapter 11
stefankmitph	Chapter 143
Stefano d'Antonio	Chapters 22, 71 and 114
Steven	Chapters 41, 48 and 89
Stilgar	Chapters 71 and 72
StriplingWarrior	Chapter 37
Stu	Chapter 96
Sujay Sarma	Chapter 52
Sunny R Gupta	Chapter 60
Supraj v	Chapter 161
Suren Srapyan	Chapters 58 and 162
Suyash Kumar Singh	Chapter 52
Sworgkh	Chapter 72
SynerCoder	Chapter 114
Tagc	Chapters 29, 31 and 84
Tamir Vered	Chapters 26, 66 and 72
Tanner Swett	Chapter 22
Taras	Chapter 45
TarkaDaal	Chapters 22, 24, 26, 27, 47 and 52
tehDorf	Chapter 8
teo van kot	Chapters 56 and 66
Testing123	Chapters 3, 28 and 50

th1rdey3	Chapters 10, 52, 66, 129 and 130
Thaoden	Chapter 40
theB	Chapter 93
thehennyy	Chapter 157
theinarasu	Chapter 19
Thennarasan	Chapters 62 and 110
Theodoros Chatzigiannakis	Chapters 22, 31, 52, 59, 66, 74, 123, 140 and 144
The_Outsider	Chapters 11, 13, 14, 20, 24, 25, 42, 50, 51 and 128
Thomas Weller	Chapter 20
Thorsten Dittmar	Chapter 52
Thriggle	Chapters 63 and 84
Thulani Chivandikwa	Chapters 19, 76 and 96
tiedied61	Chapter 11
Tim Ebenezer	Chapters 24, 52 and 106
Tim Iles	Chapter 66
Tim Medora	Chapter 39
Timon Post	Chapters 69, 103 and 133
Timothy Rascher	Chapter 48
Timothy Shields	Chapter 8
TKharaishvili	Chapters 48 and 95
Tofix	Chapter 1
Tolga Evcimen	Chapters 28, 97 and 124
Tom Bowers	Chapter 1
Tom Droste	Chapters 60 and 72
Tommy	Chapter 154
tonirush	Chapters 16, 20, 52 and 81
topolm	Chapter 52
Tormod Haugene	Chapter 20
Tot Zam	Chapter 52
Travis J	Chapters 1, 66 and 72
Trent	Chapter 72
Tseng	Chapter 114
tsjnsn	Chapter 56
Tushar patel	Chapters 1, 15, 48, 72 and 134
TyCobb	Chapter 3
Uali	Chapter 67
un	Chapters 11, 47 and 78
undefined	Chapter 141
Underscore	Chapter 113
Uriil	Chapters 71 and 72
user2321864	Chapters 71, 114 and 128
user3185569	Chapters 1, 48, 52, 59, 66 and 71
user5389107	Chapters 48 and 72
usr	Chapter 39
uTeisT	Chapters 71 and 72
Vadim Martynov	Chapter 5
Vaibhav_Welcomes_You	Chapter 159
Valentin	Chapters 52, 66 and 123
varocarbas	Chapter 66
vbnet3d	Chapter 89
vcsjones	Chapters 52 and 72
Ven	Chapters 1, 48 and 72

vesi	Chapter 76
Victor Tomaili	Chapters 1 and 48
VictorB	Chapters 59 and 66
viggity	Chapters 1, 6, 48 and 72
VirusParadox	Chapter 58
Vitaliy Fedorchenko	Chapters 38 and 66
VitorCioletti	Chapter 82
Vlad	Chapters 88 and 120
void	Chapters 24, 52 and 66
volvis	Chapter 93
wablab	Chapter 63
Wai Ha Lee	Chapters 32, 71 and 72
Wasabi Fan	Chapters 8, 9, 15, 20, 22, 52, 71, 97 and 120
wertzui	Chapter 27
Wiktor Dębski	Chapter 44
Will	Chapter 116
Will Ray	Chapters 7, 27 and 51
WizardOfMenlo	Chapter 11
Woodchipper	Chapter 52
Wouter	Chapter 39
WQYeo	Chapter 20
Wyck	Chapter 118
Xander Luciano	Chapter 98
Xandrmoro	Chapter 52
Xiaoy312	Chapters 3, 42 and 66
Yaakov Ellis	Chapter 66
Yanai	Chapter 22
Yashar Aliabasi	Chapter 53
Yotam Salmon	Chapter 20
yumaikas	Chapter 92
Yves Schelpe	Chapter 54
Zalomon	Chapter 52
ZenLulz	Chapter 108
Ziad Akiki	Chapter 1
Zoba	Chapter 112
Zohar Peled	Chapters 52 and 86

You may also like