

C# Tips

Write Better C#

Jason Roberts

C# Tips

Write Better C#

Jason Roberts

This book is for sale at <http://leanpub.com/cstips>

This version was published on 2018-08-31



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2014 - 2018 Jason Roberts

Tweet This Book!

Please help Jason Roberts by spreading the word about this book on [Twitter](#)!

The suggested tweet for this book is:

I just bought C# Tips by @robertsjason from <http://bit.ly/sharpbook>

The suggested hashtag for this book is [#csharptips](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

[#csharptips](#)

Contents

Introduction	2
About The Author	2
Other Books by Jason Roberts	2
Pluralsight Courses	2
 Part 1: Assorted Tips	 3
Merging IEnumerable Sequences with LINQ	4
Auto-Generating Sequences of Integer Values	5
Improving Struct Equality Performance	6
Overriding Equals() to Improve Performance	7
Creating Generic Methods in Non-Generic Classes	9
Converting Chars to Doubles	11
Non Short Circuiting Conditional Operators	12
More on & and &&	12
The && Operator	12
The & Operator	14
Using & With Boolean Operands	14
Using & With Integral Operands	15
Using C# Keywords for Variable Names	16
Three Part Conditional Numeric Format Strings	17
Customizing the Appearance of Debug Information in Visual Studio	18
The DebuggerDisplay Attribute	18
The DebuggerBrowsable Attribute	19
Partial Types	21
Partial Methods	21
Partial Method Restrictions	22
The Null Coalescing Operator	23
Creating and Using Bit Flag Enums	24

CONTENTS

Defining	24
Manipulating	25
The Continue Statement	26
Preprocessor Directives	27
Defining Custom Symbols	27
Emitting Warnings	28
Emitting Errors	28
Automatically Stepping Through Code	29
Exceptions in Static Constructors	30
Safe Conversion To and From DateTime Strings	32
Parsing Strings into Numbers with NumberStyles	34
Useful LINQ Set Operations	36
Concat	36
Union	36
Intersect	36
Except	36
Working with Zip Files	38
Creating Zip Files	38
Extracting Files	38
Adding Files to Existing Archive	38
Deleting Files from an Existing Archive	39
Comparing Two Lists to Check if they Contain The Same Items	40
Conditional Compilation with the Conditional Attribute	41
Writing Indented Lines of Text	44
Custom Collection Initializers	48
Delaying Creation of Expensive Objects	51
 Part 2: Common Design Patterns	 53
The Decorator Pattern	54
The Factory Pattern	58
The Gateway Pattern	60
The Strategy Pattern	62
The Null Object Pattern	65

Part 3: Useful Tools	68
The NUnit Testing Framework	69
Test Classes and Test Methods	69
Asserting Correct Results	69
Advanced Techniques	70
The xUnit.net Testing Framework	71
Test Classes and Test Methods	71
Asserting Correct Results	71
Advanced Techniques	72
Faking with Moq	74
Dependency Injection with Ninject	76
Using a Dependency Injection Framework	77
BDDfy - A BDD Testing Framework	78
Convention Tests	84
Using the Supplied ClassTypeHasSpecificNamespace Convention	84

Copyright 2014 Jason Roberts. All rights reserved.

No part of this publication may be transmitted or reproduced in any form or by any means without prior written permission from the author.

The information contained herein is provided on an “as is” basis, without warranty. The author and publisher assume no responsibility for omissions or errors, or for losses or damages resulting from the use of the information contained herein.

All trade marks reproduced, referenced, or otherwise used herein which are not the property of, or licensed to, the publisher or author are acknowledged. Trademarked names that may appear are used purely in an editorial fashion with no intention of infringement of the trademark.

Introduction

Welcome!

C# is a wonderful programming language. This book will help you become a better C# programmer by highlighting some great features of C# and .NET.

If you want to learn even more great C# tips, be sure to check out my C# Pluralsight courses: [C# Tips and Traps¹](#) and [C# Tips and Traps 2²](#).

I really hope you enjoy this book and are able to use the great C# features contained inside!

About The Author



Jason Roberts is a Journeyman Software Developer with over 12 years commercial experience. He is a Microsoft C# MVP, a [Pluralsight course author³](#) and holds an honours degree in computing. He is a writer, open source contributor and has worked on numerous apps for both Windows Phone and Windows Store.

You can contact him on Twitter as [@robertsjason⁴](#) and at his blog [DontCodeTired.com⁵](#).

Other Books by Jason Roberts

- [Keeping Software Soft⁶](#) (also available on Kindle).
- [Clean C#⁷](#)

Pluralsight Courses

Watch [Pluralsight courses by Jason Roberts⁸](#).

¹<http://bit.ly/pscstips>

²<http://bit.ly/pscstips2>

³<http://bit.ly/psjason>

⁴<https://twitter.com/robertsjason>

⁵<http://dontcodetired.com>

⁶<http://keepingsoftwaresoft.com>

⁷<http://cleancsharp.com/>

⁸<http://bit.ly/psjason>

Part 1: Assorted Tips

Merging IEnumerable Sequences with LINQ

The `Zip` method allows the joining together of `IEnumerable` sequences by interleaving the elements in the sequences.

`Zip` is an extension method on `IEnumerable`.

For example, to zip together a collection of names with ages:

```
1 var names = new [] { "John", "Sarah", "Amrit" };
2
3 var ages = new[] { 22, 58, 36 };
4
5 var namesAndAges = names.Zip(ages, (name, age) => name + " " + age);
```

This would produce an `IEnumerable<string>` containing three elements:

- “John 22”
- “Sarah 58”
- “Amrit 36”

If one sequence is shorter than the other, the “zipping” will stop when the end of the shorter sequence is reached. So if an extra name “Bob” were added:

```
1 var names = new [] { "John", "Sarah", "Amrit", "Bob" };
2
3 var ages = new[] { 22, 58, 36 };
4
5 var namesAndAges = names.Zip(ages, (name, age) => name + " " + age);
```

The result would be the same as before, “Bob” would not be used as there is no corresponding age for him in ages.

Lambdas can also be used to create objects; the following example shows how to create an `IEnumerable` of two-element Tuples:

```
1 var names = new [] { "John", "Sarah", "Amrit" };
2
3 var ages = new[] { 22, 58, 36 };
4
5 var namesAndAges = names.Zip(ages, (name, age) => Tuple.Create(name, age));
```

This will produce an `IEnumerable<Tuple<String,Int32>>` that contains three Tuples, with each Tuple holding a name and age.

Auto-Generating Sequences of Integer Values

When a sequence of integer values is needed, it can be created manually using some kind of loop, or the `Enumerable.Range` method can be used.

The following code illustrates:

```
1 var oneToTen = Enumerable.Range(1, 10);  
2  
3 int[] twentyToThirty = Enumerable.Range(20, 11).ToArray();  
4  
5 List<int> oneHundredToOneThirty = Enumerable.Range(100, 31).ToList();
```

The results of the `Range` method can also be transformed in some way, for example to get the letters of the alphabet the following could be written:

```
1 var alphabet = Enumerable.Range(0, 26).Select(i => Convert.ToChar('A' + i));
```

This will generate an `IEnumerable<char>` containing the letters A through Z.

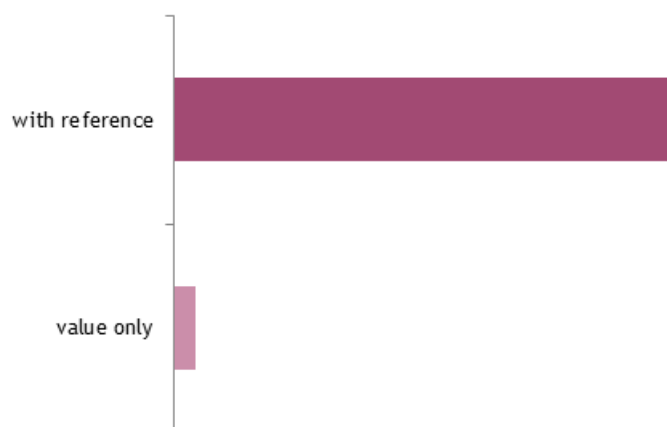
Improving Struct Equality Performance

The performance of struct equality comparisons can be improved by overriding the `Equals` method. This is especially true if the struct contain reference type fields (as opposed to only value types such as `int`).

By default, the equality of structs is automatically determined by doing a byte-by-byte comparison of the two struct objects in memory, but **only** when the struct does not contain any reference types.

When a struct contains reference type fields, reflection is used to compare the fields of the two struct objects. This reflection-based approach results in slower performance.

The following chart shows the relative performance of the default equality of a struct that contains only value types and a struct that additionally contains a reference type.



Performance with reference or value types

This chart is based on performing an equality test 10000000 times and comparing the time in milliseconds. Specific numbers have been omitted so as to concentrate on the relative differences.

These are the structs that were compared:

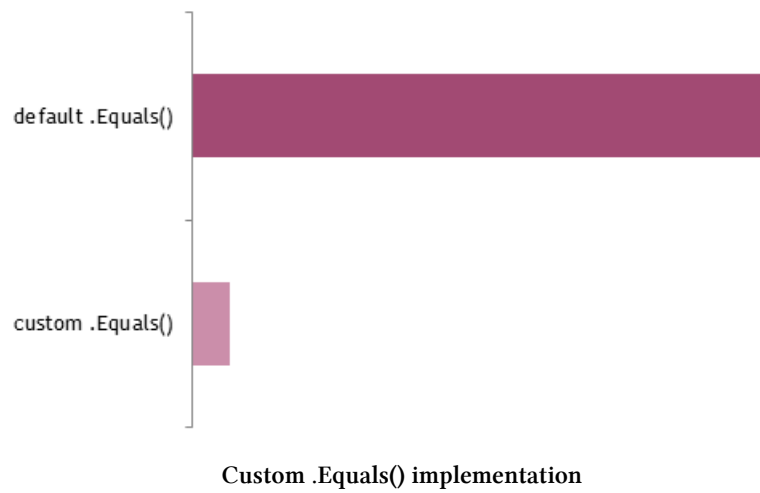
```
1 struct StructWithRefTypesNoOverriddenEquals
2 {
3     public int Age { get; set; }
4     public int Height { get; set; }
5     public string Name { get; set; }
6 }
7
8 struct StructWithNoRefTypesNoOverriddenEquals
9 {
10     public int Age { get; set; }
11     public int Height { get; set; }
12 }
```

Overriding Equals() to Improve Performance

If the `Equals` method is overridden to provide a custom definition of what equality means, this overridden method will be used rather than the default (slower) reflection-based mechanism:

```
1 struct StructWithRefTypesAndOverriddenEquals
2 {
3     public int Age { get; set; }
4     public int Height { get; set; }
5     public string Name { get; set; }
6
7     public override bool Equals(object obj)
8     {
9         if (!(obj is StructWithRefTypesAndOverriddenEquals))
10             return false;
11
12         var other = (StructWithRefTypesAndOverriddenEquals) obj;
13
14         return Age == other.Age &&
15             Height == other.Height &&
16             Name == other.Name;
17     }
18
19     // GetHashCode override and == != operators omitted
20 }
```

Now comparing the performance of `StructWithRefTypesNoOverriddenEquals` against `StructWithRefTypesAndOverriddenEquals` generates the following results:



Implementing an overridden `Equals` means that reflection will not be used, instead the custom `.Equals()` code is executed.



As with all things performance related, these differences in performance may or may not be relevant to the application you are writing.

Creating Generic Methods in Non-Generic Classes

Generic classes can be created such as:

```
1 class ThingWriter<T>
2 {
3     public void Write(T thing)
4     {
5         Console.WriteLine(thing);
6     }
7 }
```

This is a generic class that uses the type `T` as a method parameter in the `Write` method. This class could be used in the following way:

```
1 var w = new ThingWriter<int>();
2
3 w.Write(42);
```

Generic methods can be created even if the class itself is not generic.

```
1 class ThingWriter
2 {
3     public void Write<T>(T thing)
4     {
5         Console.WriteLine(thing);
6     }
7 }
```

Notice here that the `ThingWriter` class itself is not generic. To call this generic method:

```
1 var w = new ThingWriter();
2
3 w.Write<int>(42);
```

Or by taking advantage of generic type inference, the compiler can figure out the type by virtue of the fact that the type being passed to the `Write` method is an `int`.

```
1  var w = new ThingWriter();  
2  
3  w.Write(42);
```


Converting Chars to Doubles

Chars with numeric digits in them can be converted to numeric (double) values using the `char.GetNumericValue` method.

The following code:

```
1 double d = char.GetNumericValue('5');  
2  
3 Console.WriteLine(d);
```

Outputs the value: 5

So why does `GetNumericValue` return a double when a `char` can only be a single character and hold a single “number” (“0” to “9”)?

This is because `char` holds Unicode characters so it can hold characters such as the Unicode character for the fraction two-thirds. When this two-thirds Unicode character is used with `GetNumericValue` it returns a value of: 0.6666666666666667.

Non Short Circuiting Conditional Operators

Most of the time when writing C# programs, the short-circuiting conditional operators are used. These operators do not continue evaluation once the outcome has been determined.

For example:

```
1 bool b = false && CheckName(name);  
2  
3 b = false & CheckName(name);
```

Here a logical AND is being performed with the value `false` and the result of the `CheckName` method. Because there is a `false`, the logical AND can never be true.

The first statement using the short-circuiting AND operator (`&&`) will **not** execute the `CheckName` method because as soon as it has determined the outcome (the first value is `false`) the remaining terms are not even evaluated.

The second statement using the non-short-circuiting AND operator (`&`) **will** execute the `CheckName` method, even though the first value is `false`.

If the `CheckName` method needed to always be called then the non-short-circuiting version would need to be used.



Writing code that relies on these kind of side effects is not usually a good idea. It makes code harder to reason about, test, and maintain.

The conditional OR operator also comes in two versions: non-short-circuiting (`|`) and short-circuiting (`||`).

More on & and &&

There can sometimes be confusion over the `&` and `&&` operator in C#.

The && Operator

Works with: `bools`

The `&&` operator performs a logical AND on Boolean values.

For example, the following code:

```
1  bool b1 = true;
2  bool b2 = false;
3
4  bool isBothTrue = b1 && b2;
5
6  Console.WriteLine(isBothTrue);
```

Outputs:

False

If the two bool values come from the result of methods instead and the first method returns false, the && operator will **not** execute the second method. The following code:

```
1  void Main()
2  {
3      bool isBothTrue = Method1() && Method2();
4
5      Console.WriteLine("Result: " + isBothTrue);
6  }
7
8  bool Method1()
9  {
10     Console.WriteLine("Method 1 called - returning false");
11     return false;
12 }
13
14 bool Method2()
15 {
16     Console.WriteLine("Method 2 called - returning true");
17     return true;
18 }
```

Results in this output:

Method 1 called - returning false

Result: False

Notice that there is no “Method 2 called - returning true” line. This is because the && operator “short-circuits”; once it has determined the outcome (in this case the overall AND can never be true because the first Method1 is false) it stops evaluating the rest of the terms. This means Method2 does not need to be executed. If Method1 returned true then Method2 would be executed as the overall result of the AND could still be either true or false.

The & Operator

Works with: bools or integral types (such as int)

What the & operator does depends on the context in which it is used.

“For integral types, & computes the logical bitwise AND of its operands. For bool operands, & computes the logical AND of its operands”.

– [MSDN⁹](#)

Using & With Boolean Operands

If the & operator is used with Boolean operands, it performs a logical AND, much like the && operator. The difference is that & does not “short-circuit”. So even if Method1 returns false, Method2 will still be executed.

In the following code notice the & instead of &&:

```
1 void Main()  
2 {  
3     bool isBothTrue = Method1() & Method2();  
4  
5     Console.WriteLine("Result: " + isBothTrue);  
6 }  
7  
8 bool Method1()  
9 {  
10    Console.WriteLine("Method 1 called - returning false");  
11    return false;  
12 }  
13  
14 bool Method2()  
15 {  
16    Console.WriteLine("Method 2 called - returning true");  
17    return true;  
18 }
```

Results in this output:

```
Method 1 called - returning false  
Method 2 called - returning true  
Result: False
```

Even though Method1 returned false, Method2 was still executed.

⁹<http://msdn.microsoft.com/en-us/library/sbf85k1c.aspx>

Using & With Integral Operands

When the & operator is used with integral operands (for example two int types) it computes a logical bitwise AND of its operands.

The following code:

```
1  void Main()
2  {
3      int a = 0xF0;
4      int b = 0xFF;
5
6      int result = a & b;
7
8      OutputBinary(a);
9      OutputBinary(b);
10
11     Console.WriteLine("Each bit gets AND-ed");
12
13     OutputBinary(result);
14
15     Console.WriteLine("As int: " + result);
16 }
17
18 void OutputBinary(int i)
19 {
20     Console.WriteLine(Convert.ToString(i,2));
21 }
```

Produces this output:

```
11110000
11111111
Each bit gets AND-ed
11110000
As int: 240
```

Here, each bit in a gets AND-ed with its corresponding bit in b to produce the final bitwise AND-ed result of: 11110000 which is 240.

So the & operator can be referred to in the context of bools, whereby it is a logical AND of Boolean values; or in the context of integral types whereby it performs a bitwise AND of the bits of the values. Furthermore, when we refer to & in the context of bools it is referring to a short-circuiting version of the && operator.

Using C# Keywords for Variable Names

It is possible to use C# keywords for variable names, method parameter names, etc.

For example, if a variable called “namespace” was required (which is a C# keyword) the declaration and subsequent use can be prefixed with an @.

```
1 var @namespace = "hello";  
2  
3 @namespace += " world";
```

If the C# code is interoperating with another language, for example if a Visual Basic .NET project has been referenced from the C# project, there could be a class defined in VB.NET called “abstract”. This class could be used in C#:

```
1 var a = new @abstract();  
2  
3 a.@foreach = "hello";  
4  
5 a.@foreach += " world";
```

Here the VB.NET class also has a string property called “foreach” which is also a C# keyword. Notice the use of the @ in this case also.

Three Part Conditional Numeric Format Strings

When formatting numbers, a format string can be used that allows a different format to be used depending on whether the number is positive, negative, or zero.

To do this, the 3 individual formats are separated by a semicolon.

The first format will be used if the value is positive, the second format if negative, and the third if zero.

For example the format string `(+)#.##;(-)#.##;(sorry nothing at all)` will output:

`(+)99.99` if the value was 99.99

`(-)23.55` if the value was -23.55

`(sorry nothing at all)` if the value was zero

A two part conditional format string can also be used, where the first part will be used if the number is positive or zero; the second part if the number is negative.

Customizing the Appearance of Debug Information in Visual Studio

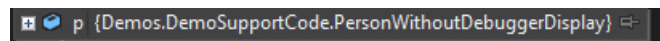
Sometimes the display and formatting of information in the Visual Studio debugger is not descriptive enough. It is possible to customize debug information in a number of ways.

The DebuggerDisplay Attribute

The DebuggerDisplay attribute allows the customization of how an object is portrayed in the debug window.

Imagine a Person object `p` with an `AgeInYears` and a `Name` property.

By default, in the debugger it would look like this:



Default display in the debugger window

Notice by default at the “root” level for the object there is just the type name displayed.

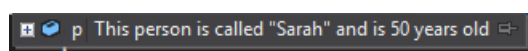
If `ToString()` is overridden then its output would be displayed here instead.

If `ToString()` is overridden but a different output is required at the root debug level, the `DebuggerDisplay` attribute can be applied at the class level:

```
1 [DebuggerDisplay("This person is called {Name} and is {AgeInYears} years old")]
2 class PersonWithDebuggerDisplay
3 {
4     [DebuggerDisplay("{AgeInYears} years old")]
5     public int AgeInYears { get; set; }
6
7     public string Name { get; set; }
8 }
```

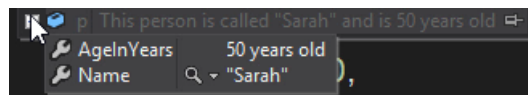
Notice that in the string parameter of the `DebuggerDisplay` attribute the properties, fields, and methods within the class can be referenced and output to the debugger by wrapping them in `{ }`.

The output in the debugger now looks as follows:



Customized display in the debugger window

Note that in the sample code above, the `DebuggerDisplay` attribute is also applied to the `AgeInYears` property which produces the following when the object is expanded in the debugger:



Customized AgeInYears property output

The DebuggerDisplay attribute can be applied to:

- Structs
- Delegates
- Classes
- Enums
- Fields
- Properties
- Assemblies

The DebuggerBrowsable Attribute

The DebuggerBrowsable attribute allows the hiding of members when viewed in the debugger.

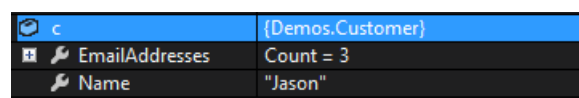
Take a Customer class:

```

1  public class Customer
2  {
3      public Customer()
4      {
5          EmailAddresses = new List<string>();
6      }
7
8      public string Name { get; set; }
9
10     public List<string> EmailAddresses { get; set; }
11 }

```

If an instance of this class is viewed in the debugger, by default it appears like this:

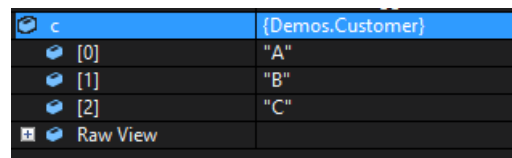


Default view of customer class

If the DebuggerBrowsable attribute is added:

```
1 public class Customer
2 {
3     public Customer()
4     {
5         EmailAddresses = new List<string>();
6     }
7
8     [DebuggerBrowsable(DebuggerBrowsableState.Never)]
9     public string Name { get; set; }
10
11    [DebuggerBrowsable(DebuggerBrowsableState.RootHidden)]
12    public List<string> EmailAddresses { get; set; }
13 }
```

Now the debug display looks like this:



c	{Demos.Customer}
[0]	"A"
[1]	"B"
[2]	"C"
Raw View	

DebuggerBrowsable view of customer class

Note that now the Name property is no longer visible.

The collection of EmailAddresses is now expanded automatically, its “root” in the debugger has been hidden and each item within the list now appears at the root level of the debugger.



While customizing debug information is not something that needs to be used on every class, it may be helpful when a lot of debug work is being performed and we want to make life easier and enable at-a-glance understanding of the state of objects.

Partial Types

The definition for a class, struct, or interface can be split across multiple source code files.

This is typically useful when part of a class is generated, but some additional non-generated code needs to be added to the class. If it were added to the generated .cs file, when the code generation tool runs again it will be overwritten and the custom code will be lost.

For example, a tool might generate a file called “DataMapper.generated.cs”. This contains a class that is marked as `partial`:

```
1 partial class IMapper
```

Another file can be created, for example “DataMapper.cs” that can contain user-defined code:

```
1 partial class IMapper
```

When compiled, it is still a single `IMapper` class that contains the combination of members defined in both the individual source code files.

Partial Methods

Partial methods allow a method to be defined in one of the .cs source files but one without any implementation.

```
1 partial void PreRead();
```

An implementation for this method can be provided in another source code file, but it *does not have to be*.

Take the generated class:

```
1 // IMapper.generated.cs
2 partial class IMapper
3 {
4     public void ReadData()
5     {
6         PreRead();
7     }
8
9     partial void PreRead();
10 }
```

This class defines a partial method `PreRead()` that is called from the non-partial method `ReadData()`. This allows for the *optional* implementation of the `PreRead` method in another partial file:

```
1  // IMapper.cs
2  partial class IMapper
3  {
4      public void SomeOtherMethod() {}
5
6      partial void PreRead()
7      {
8          // implementation code
9      }
10 }
```

Note the implementation of `PreLoad()`.

If “`IMapper.cs`” did not implement `PreLoad()`, there will be no compile error, the compiler will simply take care of removing any calls to the unimplemented method.

Partial Method Restrictions

Partial methods have a number of restrictions:

- The method name and parameter signatures of the definition and implementation must be the same
- The method can only return `void`
- Partial methods are private, they cannot have access modifiers applied
- They also cannot have `virtual`, `abstract`, `override`, `new`, `sealed`, or `extern` modifiers applied to them

The Null Coalescing Operator

The null coalescing operator `??` can reduce the amount of code required to check for a null value and provide a default.

Take this `if` statement:

```
1  if (color == null)
2  {
3      result = "white";
4  }
5  else
6  {
7      result = color;
8  }
```

Here the code is providing a default result of “white” if `color` is null.

This `if` statement can be reduced to a single line of code using the null coalescing operator.

```
1  result = color ?? "white";
```

The `??` operator can also be used with nullable value types:

```
1  int? count = null;
2
3  int result = count ?? 0;
```

It can also be chained to together:

```
1  int? localDefaultCount = null;
2  int globalDefaultCount = 42;
3
4  int? count = null;
5
6  int result = count ?? localDefaultCount ?? globalDefaultCount;
```

Here if `count` is null, then the `localDefaultCount` will be used, unless it is null, in which case `globalDefaultCount` will be used.

Creating and Using Bit Flag Enums

Enumeration types can be declared in such a way as to create a set of bitwise-combinable flags. This enables a single variable to be created that can represent different combinations of the enum named constants.

For example, an enum called `BorderVisibility` allows the representation of which sides of a shape are currently visible.

To create a variable that represents both the top and right sides of the shape being set to visible:

```
1 var topAndRight = BorderVisibility.Top | BorderVisibility.Right;
```

Here the `Top` and `Right` values have been combined using the bitwise OR operator (`|`).

Defining

To define a bit flags enum, each named constant within the enum needs to be given a specific value. The values that are assigned are in powers of 2, e.g. 2, 4, 8, etc.

A named constant also needs to be specified with a value of 0 if it needs to represent “nothing”. In the case of `BorderVisibility` this could represent no borders being visible.

So in this example, the enumeration type can be declared as follows:

```
1 enum BorderVisibility
2 {
3     None = 0,
4     Top = 1,
5     Right = 2,
6     Bottom = 4,
7     Left = 8
8 }
```

Using this definition, if the `topAndRight` variable was examined in the debugger, or converted `.ToString()` the value of “3” would be seen.

To improve this, the `[Flags]` attribute can be applied:

```
1  [Flags]
2  enum BorderVisibility
3  {
4      None = 0,
5      Top = 1,
6      Right = 2,
7      Bottom = 4,
8      Left = 8
9  }
```

Now the debugger will show “Top | Right” and `.ToString()` will now result in “Top, Right”.

Manipulating

A variable containing a combination of flags can be checked for the existence of a given value in a number of ways.

First, the bitwise AND operator can be applied to the variable and the enum named constant, then the result checked. If the result is non-zero then the flag has been set:

```
1  bool isTopVisible = (topAndRight & BorderVisibility.Top) != 0; // True
```

Another option is to use the `Enum.HasFlag()` method.

```
1  bool isTopVisible = topAndRight.HasFlag(BorderVisibility.Top); // True
```

It is possible to combine sets of flags:

```
1  var topAndRight = BorderVisibility.Top | BorderVisibility.Right;
2
3  var bottomAndLeft = BorderVisibility.Bottom | BorderVisibility.Left;
4
5  var allSidesVisible = topAndRight | bottomAndLeft;
```

Individual flags can be toggled using the exclusive OR assignment operator (“^=”), to turn off top:

```
1  allSidesVisible ^= BorderVisibility.Top;
```

And to toggle it back on again:

```
1  allSidesVisible ^= BorderVisibility.Top;
```

The Continue Statement

The continue statement can be used inside a for, foreach, while, or do statement.

When used, it skips the rest of the code following it, jumps back to the start of the loop, and starts the next iteration.

```
1  for (int i = 0; i < 10; i++)
2  {
3      if (i < 5)
4      {
5          continue;
6          // the Debug.WriteLine below won't be executed
7      }
8
9      Debug.WriteLine(i);
10 }
```

The output of this is:

```
5
6
7
8
9
```


Preprocessor Directives

Preprocessor directives allows different lines of source code to be compiled depending on a conditional compilation symbol being defined or not.

For example, by default in Visual Studio, when the Debug build configuration is being used then the Debug symbol will be automatically defined.

Similarly, in Release configuration, the “Release” symbol is defined.

Based on these pre-configured symbols, a string variable can be given a different value depending on whether the project is built using debug or release.

```
1         string buildUsed = null;
2 # if DEBUG
3         buildUsed = "building in debug";
4 # elif RELEASE
5         buildUsed = "building in release";
6 # else
7         buildUsed = "build something else";
8 # endif
```

Here the conditional preprocessor directives `#if`, `#elif` (else if), `#else` and `#endif` are being used to change the value of `buildUsed`.

The value of `buildUsed` will be different in the compiled assembly, it is not logic that executes at runtime, rather it is chosen at compile time.

Defining Custom Symbols

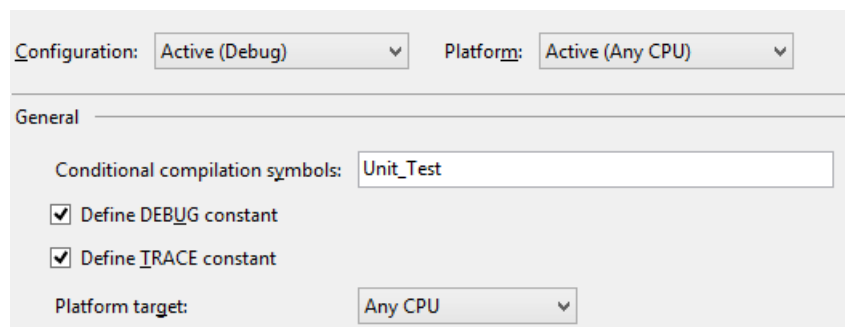
It is also possible to define custom symbols.

For example:

```
1 # if Unit_Test
2         // code that only gets compiled if unit testing
3 #endif
```

This code inside this if will only run if the symbol “Unit_Test” is defined.

One way to define this symbol is in the project’s properties in Visual Studio.



Visual Studio project properties

Emitting Warnings

Compilation warnings can be created by using the `#warning` directive, for example:

```
1 # warning This is a sample warning
```

Emitting Errors

Errors can also be emitted (that will break the build):

```
1 # error This is a sample error
```

The `#if` and `#error` directives can be combined. For example, if both release and debug symbols are defined at the same time then there is some ambiguity and the build should probably error.

```
1 # if DEBUG && RELEASE
2     # error Both Debug and release are defined
3 # endif
```

Automatically Stepping Through Code

The `[DebuggerStepThrough]` attribute allows a method to be marked so that when debugging in Visual Studio, the method will not be entered into even if using F11 (Step Into). Instead the method will still be executed, but the next line highlighted in Visual Studio will be the one after the method call.

To apply the attribute to a method:

```
1  [DebuggerStepThrough]
2  public void AMethod()
3  {
4  }
```

It is also valid to apply this attribute to an entire class or struct.

Exceptions in Static Constructors

Types can have static constructors defined. A static constructor is executed the first time an instance of the type is created, or if it is a static type, before the first static member is referenced.

The runtime will only ever execute static constructors once (perhaps not at all if no reference to the static type is made). Because of this, if the static constructor throws an exception, it will not get another chance to execute.

Possibly more importantly, the type will not be able to be used again for the life of the application.

Take the following class:

```
1 public class StaticCtorException
2 {
3     public static readonly string SomeString = "Initialized";
4
5     static StaticCtorException()
6     {
7         throw new ApplicationException("Fake Exception");
8     }
9 }
```

Notice the unhandled exception being thrown in the static constructor.

If this class is used:

```
1 try
2 {
3     // Try and get the value of SomeString.
4     // This is the first time the static SomeString field is accessed,
5     // so the static constructor will be called.
6     var s = StaticCtorException.SomeString;
7 }
8 catch
9 {
10    // The exception thrown in the static constructor will be caught
11    // here, so the program won't crash (yet)
12    Debug.WriteLine("Static constructor ex caught");
13 }
14
15 // Because the static constructor threw an exception,
16 // the StaticCtorException class can no longer be used.
17
18 // Either of the following lines will crash the program:
```

```
19
20 Debug.WriteLine(StaticCtorException.SomeString);
21
22 // or:
23
24 var o = new StaticCtorException();
```

When either of the last two lines of code are executed, a `System.TypeInitializationException` will be thrown with a message of: “The type initializer for ‘StaticCtorException’ threw an exception.”.

The type can no longer be used in the application without getting an exception, because of the uncaught exception in the static constructor.

Safe Conversion To and From DateTime Strings

When converting a `DateTime` to a string representation and back again (“round-tripping”) it is possible for the re-parsed date to be different from the original `DateTime` value.

Take the following code:

```
1 var startingDate = new DateTime(2000, 12, 1, 13, 30, 0, DateTimeKind.Local);
2
3 var nonRoundTripString = startingDate.ToString();
4 var reparsedNonRoundTrip = DateTime.Parse(nonRoundTripString);
5
6 Debug.WriteLine(startingDate);
7 Debug.WriteLine(nonRoundTripString);
8 Debug.WriteLine(reparsedNonRoundTrip);
9 Debug.WriteLine(reparsedNonRoundTrip.Kind);
10 Debug.WriteLine(reparsedNonRoundTrip.ToLocalTime());
```

The output from this is:

```
1/12/2000 1:30:00 PM
1/12/2000 1:30:00 PM
1/12/2000 1:30:00 PM
Unspecified
1/12/2000 9:30:00 PM
```

Here the re-parsed `DateTime` seems identical to the original until the `DateTimeKind` is examined. The re-parsed `DateTime` has lost the original `DateTimeKind.Local`. When the re-parsed value is converted to the local time (the last line of output), the value is different.

This is because when the original `DateTime` was converted to a string, crucial information has been lost.

If a `DateTime` needs to be converted to a string and then back to an identical value, the “round-trip format specifier” can be used.

The .NET “o” standard format specifier will ensure the string is re-parsed correctly. Take the following code:

```
1 var startingDate = new DateTime(2000, 12, 1, 13, 30, 0, DateTimeKind.Local);
2
3 var roundTripString = startingDate.ToString("o");
4 var reparsedRoundTrip = DateTime.Parse(roundTripString,
5     null, DateTimeStyles.RoundtripKind);
6
7 Debug.WriteLine(startingDate);
8 Debug.WriteLine(roundTripString);
9 Debug.WriteLine(reparsedRoundTrip);
10 Debug.WriteLine(reparsedRoundTrip.Kind);
11 Debug.WriteLine(reparsedRoundTrip.ToLocalTime());
```

The output from this is:

```
1/12/2000 1:30:00 PM
2000-12-01T13:30:00.0000000+08:00
1/12/2000 1:30:00 PM
Local
1/12/2000 1:30:00 PM
```

Notice this time that the local time zone is retained in the re-parsed date, and when it is converted `.ToLocalTime()` the value remains the same as the original `DateTime`.

Parsing Strings into Numbers with NumberStyles

When parsing strings into numbers it is possible to specify the style of number that is expected in the string.

For example, parsing the string "(50)" into an `int` will result in a `FormatException` being thrown:

```
1 int error = int.Parse("(50)");
```

An exception is thrown because by default the `Parse` method defaults to a number style that only allows leading/trailing space and a leading sign indicator such as "-50".

When parsing, the expected `NumberStyles` can be explicitly set.

If the string being parsed indicates a negative value by enclosing it in parentheses, the following syntax can be used:

```
1 int negative50 = int.Parse("(50)", NumberStyles.AllowParentheses);  
2 int positive50 = int.Parse("50", NumberStyles.AllowParentheses);
```

Here the variable `negative50` will be -50 and `positive50` will be 50. Notice in the positive example the lack of parentheses does not produce an exception because they are allowed, **not** required.

If `NumberStyles.None` is specified, the string cannot even have a leading sign indicator. The following produces an exception:

```
1 int errorSign = int.Parse("-24000", NumberStyles.None);
```

The available `NumberStyles` are:

- `None`
- `AllowLeadingWhite`
- `AllowTrailingWhite`
- `AllowLeadingSign`
- `AllowTrailingSign`
- `AllowParentheses`
- `AllowDecimalPoint`
- `AllowThousands`
- `AllowExponent`
- `AllowCurrencySymbol`
- `AllowHexSpecifier`

- Integer (AllowLeadingSign | AllowTrailingWhite | AllowLeadingWhite)
- HexNumber (AllowHexSpecifier | AllowTrailingWhite | AllowLeadingWhite)
- Number (Integer | AllowThousands | AllowDecimalPoint | AllowTrailingSign)
- Float (Integer | AllowExponent | AllowDecimalPoint)
- Currency (Number | AllowCurrencySymbol | AllowParentheses)
- Any (Currency | AllowExponent)

Notice that the bottom styles are actually composites of other values.

It is also possible to use a custom composite:

```
1 double custom = Double.Parse("(24,000)",  
2     NumberStyles.AllowParentheses |  
3     NumberStyles.AllowThousands);
```

Here, the resulting value is -24000.0.

Useful LINQ Set Operations

There are a number of useful LINQ set-based operations that are often overlooked.

Concat

To join two `IEnumerable` sequences together:

```
1 IEnumerable<int> oneToFive = new [] { 1, 2, 3, 4, 5 };
2 IEnumerable<int> sixToTen = new [] { 6, 7, 8, 9, 10 };
3
4 IEnumerable<int> oneToTen = oneToFive.Concat(sixToTen);
```

`oneToTen` now contains the values 1 through 10;

Union

To combine all the unique values:

```
1 IEnumerable<int> oneToFive = new [] { 1, 2, 3, 4, 5 };
2 IEnumerable<int> twoToSix = new [] { 2, 3, 4, 5, 6 };
3
4 IEnumerable<int> oneToSix = oneToFive.Union(twoToSix);
```

`oneToSix` now contains 1 through 6. The values 2, 3, 4, 5 are not duplicated in the result.

Intersect

To get values that only appear in both sequences:

```
1 IEnumerable<int> oneToFive = new [] { 1, 2, 3, 4, 5 };
2 IEnumerable<int> twoToSix = new [] { 2, 3, 4, 5, 6 };
3
4 IEnumerable<int> twoToFive = oneToFive.Intersect(twoToSix);
```

`twoToFive` contains only 2 through 5. 1 and 6 do not appear as they do not appear in both sequences.

Except

To get only those values inside the first sequence that do **not** appear in the second sequence:

```
1 IEnumerable<int> oneToFive = new[] { 1, 2, 3, 4, 5 };
2 IEnumerable<int> sixToTen = new[] { 6, 7, 8, 9, 10 };
3
4 IEnumerable<int> resultOneToFive = oneToFive.Except(sixToTen);
```

resultOneToFive contains all the values from the first sequence as none of them appear in the second sequence sixToTen.

As another example:

```
1 var colors = new[] { "Red", "Green", "Blue" };
2
3 var colorsResult = colors.Except(new[] { "Red", "Green", "Orange" });
```

colorsResult contains only “Blue” because “Blue” does not appear in the second sequence "Red", "Green", "Orange".

Working with Zip Files

The `System.IO.Compression` namespace and related assemblies allow the creation and manipulation of compressed Zip files.

Creating Zip Files

To create a Zip file containing all the files under a given directory:

```
1 ZipFile.CreateFromDirectory(@"C:\Files", @"C:\Files.zip");
```

This new Zip file will contain the files in the Files directory but will not create the root Files directory in the archive file itself.

If the root directory is required in the new Zip file then an overload of `CreateFromDirectory` can be used (that also allows the setting of the compression level).

```
1 ZipFile.CreateFromDirectory(@"C:\Files",  
2     @"C:\Files.zip",  
3     CompressionLevel.Fastest,  
4     includeBaseDirectory: true);
```

Extracting Files

The contents of a Zip archive can be extracted to a specified target directory:

```
1 ZipFile.ExtractToDirectory(@"C:\Files.zip", @"C:\Temp\ExtractedFiles");
```

Adding Files to Existing Archive

If an archive already exists, additional files can be added to it. First the existing Zip file is opened with the `ZipFile.Open` method and the mode is set to `Update`.

Once the reference to the existing Zip has been obtained, the `CreateEntryFromFile` method adds a new file to it. In the code below the file on disk is called “extrafile.txt” but when it is added to the Zip file it will be added with a different name: “differentNameInArchive.txt”, though this does not have to be different.

```
1 using (ZipArchive zip = ZipFile.Open(@"C:\Files.zip", ZipArchiveMode.Update))
2 {
3     zip.CreateEntryFromFile(@"C:\Temp\extrafile.txt",
4         "differentNameInArchive.txt");
5 }
```

Deleting Files from an Existing Archive

Files can be deleted from an existing Zip file by first getting a reference to the file in the Zip and calling the Delete method:

```
1 using (ZipArchive zip = ZipFile.Open(@"C:\Files.zip", ZipArchiveMode.Update))
2 {
3     var fileToDelete = zip.GetEntry("differentNameInArchive.txt");
4
5     fileToDelete.Delete();
6 }
```

Comparing Two Lists to Check if they Contain The Same Items

The `SequenceEqual` method in the `System.Linq` namespace allows two `IEnumerable` sequences to be compared and returns a boolean result of `true` if they both contain the same items in the same order.

The following code demonstrates that simply comparing the equality of two `List<string>` instances will return `false` because the equality test is testing that the two references point to the same object, which they do not.

```
1 IEnumerable<string> names1 = new List<string>{"Sarah", "Amrit", "Gentry"};
2 IEnumerable<string> names2 = new List<string> { "Sarah", "Amrit", "Gentry" };
3
4 bool equal = names1 == names2; // false
```

Compare this with using `SequenceEqual` which returns `true`:

```
1 bool containSameItems = names1.SequenceEqual(names2); // true
```

If the two sequences contain the same items but in a different order `SequenceEqual` returns `false`:

```
1 IEnumerable<string> names3 = new List<string> {"Gentry", "Sarah", "Amrit"};
2 IEnumerable<string> names4 = new List<string> {"Sarah", "Amrit", "Gentry"};
3
4 containSameItems = names3.SequenceEqual(names4); // false
```

Conditional Compilation with the Conditional Attribute

The `[Conditional]` attribute can be applied to methods to tell the compiler to only include them when a given conditional compilation symbol is defined (for example “DEBUG”).

Examine the following code:

```
1  using System;
2
3  namespace ClassLibrary1.CondtionalAttr
4  {
5      class SomeClass
6      {
7          public static void SomeMethod()
8          {
9              Console.WriteLine("In SomeMethod");
10         }
11     }
12 }
```

```
1  using System;
2
3  namespace ClassLibrary1.CondtionalAttr
4  {
5      class SomeProgram
6      {
7          public static void Run()
8          {
9              Console.WriteLine("Running SomeProgram");
10
11              SomeClass.SomeMethod();
12         }
13     }
14 }
```

```
1 using System;
2
3 namespace ClassLibrary1.CondtionalAttr
4 {
5     class SomeOtherProgram
6     {
7         public static void Run()
8         {
9             Console.WriteLine("Running SomeOtherProgram");
10
11             SomeClass.SomeMethod();
12         }
13     }
14 }
```

If the following code is run:

```
1 SomeProgram.Run();
2 SomeOtherProgram.Run();
```

The following will be output:

```
Running SomeProgram
In SomeMethod
Running SomeOtherProgram
In SomeMethod
```

To only compile `SomeClass.SomeMethod()` when in a DEBUG build, the `[Conditional]` attribute can be added:

```
1 class SomeClass
2 {
3     [Conditional("DEBUG")]
4     public static void SomeMethod()
5     {
6         Console.WriteLine("In SomeMethod");
7     }
8 }
```

Now, assuming the build is in DEBUG mode, the output remains the same. If the build is changed to RELEASE, then the following output is received:

```
Running SomeProgram
Running SomeOtherProgram
```


Notice that `SomeMethod` never gets executed now.

The compilation decision is made at the point of the **caller**. So if `DEBUG` is undefined in `SomeOtherProgram` and even in `SomeClass` itself, the following output is obtained:

```
Running SomeProgram
In SomeMethod
Running SomeOtherProgram
```

Notice that because `DEBUG` is **not** active `SomeClass.SomeMethod()` is not called from `SomeOtherProgram`, i.e. because `DEBUG` is not defined at the point if the caller (`SomeOtherProgram`).

There are a number of restrictions when applying the `[Conditional]` attribute:

- The decorated method must be in a `class` or `struct`
- The decorated method must return `void`
- The decorated method must not be marked as `override`
- The decorated method must not be implementing an interface method


```

35         Name = "Executive in charge of team bonding",
36         Minions =
37         {
38             new Role
39             {
40                 Name = "Paintball organizer"
41             }
42         }
43     }
44 },
45 new Role
46 {
47     Name = "VP of Software development",
48     Minions =
49     {
50         new Role
51         {
52             Name = "Executive in charge of getting best tools"
53         },
54         new Role
55         {
56             Name = "Executive in charge of quiet"
57         },
58         new Role
59         {
60             Name = "Executive in charge of training"
61         }
62     }
63 },
64 },
65 };

```

Where Role is defined as:

```

1  class Role
2  {
3      public Role()
4      {
5          Minions = new List<Role>();
6      }
7
8      public string Name { get; set; }
9
10     public List<Role> Minions { get; set; }
11 }

```

The following code defines an action that is used recursively to loop through all roles and sub-roles. The `IndentedTextWriter` is indented using `writer.Indent++` and un-indented `writer.Indent--` for each layer in the organization:

```

1 Action<Role, IndentedTextWriter> writeRole = null;
2
3 writeRole = (role, writer) =>
4     {
5         writer.WriteLine(role.Name);
6
7         foreach (var minion in role.Minions)
8         {
9             writer.Indent++;
10
11             writeRole(minion, writer);
12
13             writer.Indent--;
14         }
15     };
16
17 var stringWriter = new StringWriter();
18 var indentedWriter = new IndentedTextWriter(stringWriter);
19
20 writeRole(organisation, indentedWriter);
21
22 Console.Write(stringWriter);

```

This produces the following output:

```

1 The Big Boss
2     VP of Lunchtimes
3         Executive in charge of burgers
4             Burger griller
5             Bun toaster
6     VP of Fun Times
7         Executive in charge of team bonding
8             Paintball organizer
9     VP of Software development
10         Executive in charge of getting best tools
11         Executive in charge of quiet
12         Executive in charge of training

```

Optionally the indentation string can be specified:

```
1  var indentedWriter = new IndentedTextWriter(stringWriter, "-");
```

Which results in:

```
1  The Big Boss
2  -VP of Lunchtimes
3  --Executive in charge of burgers
4  ---Burger griller
5  ---Bun toaster
6  -VP of Fun Times
7  --Executive in charge of team bonding
8  ---Paintball organizer
9  -VP of Software development
10 --Executive in charge of getting best tools
11 --Executive in charge of quiet
12 --Executive in charge of training
```

Custom Collection Initializers

Collection initializers allow a collection class to be initialized using a shorthand version, as opposed to using multiple `.Add()` statements.

For example to initialize a list of `Ingredient`:

```
1 var cake = new List<Ingredient>
2     {
3         new Ingredient("Flour"),
4         new Ingredient("Sugar"),
5         new Ingredient("Butter")
6     };
```

If a custom class implements `IEnumerable` then this functionality can be enabled by creating an `Add` method.

The following class doesn't have an `.Add()` method:

```
1 public class Recipe : IEnumerable<Ingredient>
2 {
3     private readonly List<Ingredient> _foods = new List<Ingredient>();
4
5     public IEnumerator<Ingredient> GetEnumerator()
6     {
7         return _foods.GetEnumerator();
8     }
9
10    IEnumerator IEnumerable.GetEnumerator()
11    {
12        return GetEnumerator();
13    }
14 }
```

So the following code will not compile:

```
1 var cake = new Recipe
2     {
3         new Ingredient("Flour"),
4         new Ingredient("Sugar"),
5         new Ingredient("Butter")
6     };
```

To fix this an `.Add()` method can be added:

```
1 public class Recipe : IEnumerable<Ingredient>
2 {
3     private readonly List<Ingredient> _foods = new List<Ingredient>();
4
5     public IEnumerator<Ingredient> GetEnumerator()
6     {
7         return _foods.GetEnumerator();
8     }
9
10    IEnumerator IEnumerable.GetEnumerator()
11    {
12        return GetEnumerator();
13    }
14
15    public void Add(Ingredient ingredient)
16    {
17        _foods.Add(ingredient);
18    }
19
20    public void Add(string ingredientName)
21    {
22        _foods.Add(new Ingredient(ingredientName));
23    }
24 }
```

Notice here there are two overloads of `Add`: one takes an `Ingredient` and the other for convenience allows a string to be passed.

This means the following code will compile:

```
1 var cake = new Recipe
2     {
3         new Ingredient("Flour"),
4         new Ingredient("Sugar"),
5         new Ingredient("Butter"),
6         "Milk"
7     };
```


Delaying Creation of Expensive Objects

If there are a number of objects being created that have a significant creation cost and not all instances may actually end up being used, it can make sense to delay the actual creation until the object instance is actually needed. This is called lazy initialization.

The `Lazy<T>` class is one way to achieve this in C#.

Take the following class that simulates a one second resource-intensive creation cost in the constructors:

```
1 public class FileSystemReportGenerator
2 {
3     public FileSystemReportGenerator()
4     {
5         RootPath = @"C:\";
6
7         // Simulated one second "resource intensive" setup cost
8         Thread.Sleep(1000);
9     }
10
11    public FileSystemReportGenerator(string rootPath)
12    {
13        RootPath = rootPath;
14
15        // Simulated one second "resource intensive" setup cost
16        Thread.Sleep(1000);
17    }
18
19    public string RootPath { get; set; }
20
21    public void RunReport()
22    {
23        // etc.
24    }
25 }
```

Every time one of these objects is created there will be a one second delay. If multiple objects are created in a list (for example 10) then there will be a ten second delay to create all of these. If depending on the execution only *some* of these objects are actually used (e.g. `RunReport()`) then this creation time is wasted.

Using the `Lazy<T>` class it is possible to declare these 10 reports, but they will only actually be instantiated when they are first used.

To create a lazy `FileSystemReportGenerator`:

```
1 var lazyRepGen = new Lazy<FileSystemReportGenerator>();
```

At this point the `FileSystemReportGenerator` constructor has *not* been called so no one second delay has been executed yet.

To access the actual object, the `.Value` property is used:

```
1 lazyRepGen.Value.RunReport();
```

The first use of the `.Value` property will cause the underlying object to be created and the constructor called, and now the one second delay will occur.

In the preceding examples, the default constructor is called on initialization. Alternatively, a specific constructor can be called:

```
1 var lazyRepGen = new Lazy<FileSystemReportGenerator>(
2     () => new FileSystemReportGenerator(@"C:\temp"));
```

Again this constructor has not been executed at this point, it will be executed when the `.Value` property is accessed for the first time.



There are some considerations regarding the caching of exceptions when using `Lazy<T>`, please see the MSDN documentation for more details.

Part 2: Common Design Patterns

This section illustrates how to implement some common design patterns using C#.

The Decorator Pattern

The Decorator pattern allows the addition of behaviour to existing classes without changing the original class. The pattern also allows the “wrapping” of multiple decorators together to compose additional behaviour on top of an existing class.

The decorator pattern typically starts with an abstraction of some behaviour. In the example below this takes the form of an interface called `Tweeter` that allows a message to be sent:

```
1 public interface ITweeter
2 {
3     void SendTweet(string message);
4 }
```

This interface is implemented in a class called `Tweeter` that can be considered the basic class that will be decorated later:

```
1 public class Tweeter : ITweeter
2 {
3     public void SendTweet(string message)
4     {
5         Console.WriteLine("TWEETING: '{0}'", message);
6     }
7 }
```

If `SendTweet` is called with a message of “DontCodeTired.com” the following is output:

TWEETING: ‘DontCodeTired.com’

The `Tweeter` can be decorated.

A decorator implements the same interface as the class it is decorating. It also contains a reference to the decorated (“wrapped”) instance.

As an example, suppose that a new requirement exists to be able to add the hashtag **#DontCodeTired** to some messages. One way to do this would be to subclass `Tweeter` or implement `ITweeter` a second time.

Another approach is to create a `DontCodeTiredDecorator` that automatically appends the hashtag to all messages:

```
1 public class DontCodeTiredDecorator : ITweeter
2 {
3     private readonly ITweeter _decoratedTweeter;
4
5     public DontCodeTiredDecorator(ITweeter tweeter)
6     {
7         _decoratedTweeter = tweeter;
8     }
9
10    public void SendTweet(string message)
11    {
12        _decoratedTweeter.SendTweet(message + " #DontCodeTired");
13    }
14 }
```

This decorator delegates to the “wrapped” instance of `ITweeter` (as supplied via the constructor) but adds the extra hashtag string.

To use this decorated version:

```
1 ITweeter t = new Tweeter();
2
3 ITweeter decorated = new DontCodeTiredDecorator(t);
4
5 decorated.SendTweet("DontCodeTired.com");
```

This produces the following output (note the hashtag):

TWEETING: ‘DontCodeTired.com #DontCodeTired’

The key thing here is that extra runtime behaviour has been added **without** having to modify the original `Tweeter` class.

Suppose another new requirement surfaces that requires that certain messages need to be repeated (spammed) ten times. At this point it would be possible to create another implementation of `ITweeter` or subclass `Tweeter`, but instead another decorator can be defined:

```
1 public class SpamDecorator : ITweeter
2 {
3     private readonly ITweeter _decoratedTweeter;
4
5     public SpamDecorator(ITweeter tweeter)
6     {
7         _decoratedTweeter = tweeter;
8     }
9
10    public void SendTweet(string message)
```

```
11     {
12         const int numberOfSpamMessages = 10;
13
14         for (var i = 0; i < numberOfSpamMessages; i++)
15         {
16             _decoratedTweeter.SendTweet(message);
17         }
18     }
19 }
```

This decorator simply delegates to the “wrapped” `ITweeter` and calls it 10 times.

To use this decorated version:

```
1 ITweeter t = new Tweeter();
2
3 ITweeter decorated = new SpamDecorator(t);
4
5 decorated.SendTweet("DontCodeTired.com");
```

This produces the following output:

```
TWEETING: 'DontCodeTired.com'
TWEETING: 'DontCodeTired.com'
TWEETING: 'DontCodeTired.com'
TWEETING: 'DontCodeTired.com'
TWEETING: 'DontCodeTired.com'
TWEETING: 'DontCodeTired.com'
TWEETING: 'DontCodeTired.com'
TWEETING: 'DontCodeTired.com'
TWEETING: 'DontCodeTired.com'
TWEETING: 'DontCodeTired.com'
```

At this point it is possible to send messages with a hashtag or send spam. To the calling code, everything is still an `ITweeter`, it does not care if it happens to be decorated or not.

Decorators can be composed with other decorators. For example, if some messages now need the hashtag to be added **and** sent 10 times, the existing two decorators can be combined:

```
1 ITweeter t = new SpamDecorator(  
2     new DontCodeTiredDecorator(  
3         new Tweeter());  
4  
5 t.SendTweet("DontCodeTired.com");
```

This produces the following output (note the hashtag and the 10 repeats):

```
TWEETING: 'DontCodeTired.com #DontCodeTired'  
TWEETING: 'DontCodeTired.com #DontCodeTired'  
TWEETING: 'DontCodeTired.com #DontCodeTired'  
TWEETING: 'DontCodeTired.com #DontCodeTired'  
TWEETING: 'DontCodeTired.com #DontCodeTired'  
TWEETING: 'DontCodeTired.com #DontCodeTired'  
TWEETING: 'DontCodeTired.com #DontCodeTired'  
TWEETING: 'DontCodeTired.com #DontCodeTired'  
TWEETING: 'DontCodeTired.com #DontCodeTired'  
TWEETING: 'DontCodeTired.com #DontCodeTired'
```



“Decorator works best when you have just a single I/O channel...[it] is a great way to allow to dynamically compose the way we apply processing to it.” [Rahien, A. [Design patterns in the test of time: Decorator](http://ayende.com/blog/159553/design-patterns-in-the-test-of-time-decorator)¹⁰].

¹⁰<http://ayende.com/blog/159553/design-patterns-in-the-test-of-time-decorator>

The Factory Pattern

A Factory creates objects of particular kind as opposed to just new-ing up objects manually in code.

The type of the returned object can depend on some data supplied by the client or could also be based on the current context or state of the application.

As an example, suppose there is an `IGreeter` that is responsible for outputting a message dependant upon either a time based value or a gender based value.

The interface:

```
1 public interface IGreeter
2 {
3     void Greet();
4 }
```

The factory:

```
1 public static class GreeterFactory
2 {
3     public static IGreeter FromTimeOfDay(DateTime time)
4     {
5         var isEvening = time.Hour > 18 || time.Hour < 5;
6
7         if (isEvening)
8             return new EveningGreeter();
9
10        return new DaytimeGreeter();
11    }
12
13    public static IGreeter FromGender(bool isMale)
14    {
15        if (isMale)
16            return new MaleGreeter();
17
18        return new FemaleGreeter();
19    }
20 }
```

Now client code can ask for an `IGreeter` to be created:


```
1 GreeterFactory.FromGender(true).Greet();
2 GreeterFactory.FromGender(false).Greet();
3
4 var midnight = new DateTime(2000, 1, 1, 0, 0, 0);
5 var midday = new DateTime(2000, 1, 1, 12, 0, 0);
6
7 GreeterFactory.FromTimeOfDay(midnight).Greet();
8 GreeterFactory.FromTimeOfDay(midday).Greet();
```

Client code does not care how the factory goes about creating an object, it just knows that it should get back an `IGreeter` of the correct kind.

The factory method does not have to have any parameters passed to it, instead it may just examine the existing state of the application to make the decision.

The Gateway Pattern

The Gateway pattern “*encapsulates access to an external system or resource*” [Fowler, M¹¹](#). A gateway class provides a simplified way of interacting with something external to the application/system that perhaps also has a peculiar or difficult to use API.

For example, suppose an application needs to use some third-party external system to validate an address. The external system provides an API: `int val(string adLn1, string adLn2, string cntry)` that returns a 1 for valid address and a 0 for an invalid address.

A gateway class can be created to facilitate interaction with this external system:

```
1  public interface IAddressValidationGateway
2  {
3      bool Validate(string addressLine1,
4                   string addressLine2,
5                   string country);
6  }
7
8  public class AddressValidationGateway : IAddressValidationGateway
9  {
10     public bool Validate(string addressLine1,
11                        string addressLine2,
12                        string country)
13     {
14         const string licenceKey = "xyz";
15
16         var v = new AddVal(licenceKey);
17
18         var result = v.val(addressLine1, addressLine2, country);
19
20         const int validAddress = 1;
21
22         return result == validAddress;
23     }
24 }
```

The application being developed can now use the gateway:

¹¹<http://martinfowler.com/eaCatalog/gateway.html>

```
1 IAddressValidationGateway a = new AddressValidationGateway();
2
3 var isValidAddress = a.Validate("Unit 2", "First Street", "Australia");
4
5 if (isValidAddress)
6 {
7     // etc.
8 }
```

The Gateway pattern helps to insulate the system being developed from its external dependencies. By also defining an interface for the gateway, it makes it possible to provide fake versions for testing purposes or swap out one third-party address validation service for another without the change rippling throughout the system.

The gateway also helps to deal with anything unique to the external service, in the example above this can be seen in the provision of the `licenceKey` to the third-party service.

The Strategy Pattern

The Strategy pattern encapsulates a set of related algorithms into individual classes and makes them interchangeable with each other, for example by defining an interface.

This means that client code can choose which algorithm to use without needing to know the internal details of how each one works.

The example code below shows what this might look like without using the strategy pattern:

```
1  public enum ForecastSimulationMethod
2  {
3      ConsiderLongTermTrendData,
4      ConsiderShortTermTrendData
5  }
6
7  public class Forecast
8  {
9      public int Temperature { get; set; }
10     public int ChanceOfRain { get; set; }
11 }
12
13 public class WeatherForecasterNoStrategyPattern
14 {
15     public Forecast PredictForecast(DateTime targetDate,
16                                     ForecastSimulationMethod simulationMethod)
17     {
18         switch (simulationMethod)
19         {
20             case ForecastSimulationMethod.ConsiderLongTermTrendData:
21                 // actual forecast code omitted for demo purposes
22                 return new Forecast {Temperature = 33, ChanceOfRain = 0};
23             case ForecastSimulationMethod.ConsiderShortTermTrendData:
24                 // actual forecast code omitted for demo purposes
25                 return new Forecast { Temperature = 100, ChanceOfRain = 42 };
26             break;
27             default:
28                 throw new ArgumentOutOfRangeException("simulationMethod");
29         }
30     }
31 }
```

A client would create a forecast for today using short term trends with the following code:

```
1 var f = new WeatherForecasterNoStrategyPattern();
2
3 var forecastForToday = f.PredictForecast(DateTime.Now,
4     ForecastSimulationMethod.ConsiderShortTermTrendData);
```

When new forecast algorithms/methods are required in the future, a new enum member needs adding to `ForecastSimulationMethod` and also a new case in the switch statement also needs adding. This means changing existing code (and potentially breaking it) rather than just creating a new algorithm.

To use the Strategy pattern, an interface could be defined as:

```
1 public interface IForecastSimulation
2 {
3     Forecast RunSimulation(DateTime targetDate);
4 }
```

This interface represents an algorithm that produces a forecast for a specified date.

Next, this interface can be implemented in multiple classes, each class representing a specific way of forecasting weather:

```
1 public class ConsiderLongTermTrendDataSimulation : IForecastSimulation
2 {
3     public Forecast RunSimulation(DateTime targetDate)
4     {
5         // actual forecast code omitted for demo purposes
6         return new Forecast { Temperature = 33, ChanceOfRain = 0 };
7     }
8 }
9
10 public class ConsiderShortTermTrendDataSimulation : IForecastSimulation
11 {
12     public Forecast RunSimulation(DateTime targetDate)
13     {
14         // actual forecast code omitted for demo purposes
15         return new Forecast { Temperature = 100, ChanceOfRain = 42 };
16     }
17 }
```

Now the original class can be refactored and the switch removed:

```
1 public class WeatherForecasterUsingStrategyPattern
2 {
3     public Forecast PredictForecast(DateTime targetDate,
4                                     IForecastSimulation simulationMethod)
5     {
6         return simulationMethod.RunSimulation(targetDate);
7     }
8 }
```

Now the client, rather than an enum value, supplies an instance of an `IForecastSimulation` to choose what algorithm to use:

```
1 var f = new WeatherForecasterUsingStrategyPattern();
2
3 var forecastForToday = f.PredictForecast(DateTime.Now,
4     new ConsiderShortTermTrendDataSimulation());
```

The Null Object Pattern

Often code can has a large number of `if` statements to check whether an item exists before calling some behaviour on it.

The following code shows a `Car` that checks to see if a turbo is fitted to increase the acceleration of the car:

```
1  public class Car
2  {
3      private readonly ITurbo _turbo;
4
5      public Car(ITurbo turbo)
6      {
7          _turbo = turbo;
8      }
9
10     public void IncreaseSpeed()
11     {
12         // Check if car has turbo before trying to use it
13
14         if (_turbo != null)
15         {
16             _turbo.Boost();
17         }
18     }
19 }
```

Notice that in the `IncreaseSpeed` method, before the turbo can be used, a check must first be made to see if the car even has a turbo fitted. So to use this `Car` class either an instance of `ITurbo` can be supplied or a `null` if the car does not have a turbo fitted:

```
1  var carWithTurbo = new Car(new CeramicTurbo());
2
3  carWithTurbo.IncreaseSpeed();
4
5  var carWithNoTurbo = new Car(null);
6
7  carWithNoTurbo.IncreaseSpeed();
```

Here `CeramicTurbo` is defined as:

```
1 public interface ITurbo
2 {
3     void Boost();
4 }
5
6 public class CeramicTurbo : ITurbo
7 {
8     public void Boost()
9     {
10         Debug.WriteLine("Engage turbo boost!");
11     }
12 }
```

The Null Object Pattern removes the need for these kind of checks throughout the codebase.

To represent the absence (null) turbo, the following can be defined:

```
1 public class NullTurbo : ITurbo
2 {
3     public void Boost()
4     {
5         Debug.WriteLine("Do nothing :(");
6     }
7 }
```

Notice here the Boost method in the null object won't do anything if it's called, it will act in a neutral way and not affect the car.

The car can be refactored to remove the null check when increasing speed:

```
1 public class CarUsingNullObjectPattern
2 {
3     private readonly ITurbo _turbo;
4
5     public CarUsingNullObjectPattern(ITurbo turbo)
6     {
7         _turbo = turbo;
8     }
9
10    public void IncreaseSpeed()
11    {
12        _turbo.Boost();
13    }
14 }
```

Notice the IncreaseSpeed method here is a lot more succinct.

To define a car with no turbo:


```
1  var carWithNoTurbo = new CarUsingNullObjectPattern(new NullTurbo());  
2  
3  carWithNoTurbo.IncreaseSpeed();
```

Part 3: Useful Tools

This section introduces some useful tools for the C# developer.

The NUnit Testing Framework

NUnit¹² is a testing framework for .NET that is available for install via NuGet.

NUnit based tests can be executed in Visual Studio using the built in test runner (after installing the [test adapter](#)¹³) or with a third-party tool such as Resharper.

NUnit based tests can also be run from the command line and inside build servers such as TeamCity.

Test Classes and Test Methods

Individual tests sit inside a test class.

To indicate to NUnit that a class contains tests, it is marked with the `TestFixture` attribute.

```
1 [TestFixture]
2 public class CalculatorTests
3 {
4 }
```

Now that NUnit knows to look inside this class, test methods can be written. A test method contains a single logical test and is marked by used the `Test` attribute.

```
1 [TestFixture]
2 public class CalculatorTests
3 {
4     [Test]
5     public void ShouldDivideNumbers()
6     {
7     }
8 }
```

Now whichever NUnit test runner is being used can execute this `ShouldDivideNumbers` test.

Asserting Correct Results

Once the system under test (sut) has been executed, the resulting state needs to be checked for correctness. To signal to the test runner that something is correct or broken, the `Assert` class is used.

The following example checks (asserts) that the result of dividing two numbers is correct.

¹²<http://www.nunit.org/>

¹³<http://visualstudiogallery.msdn.microsoft.com/6ab922d0-21c0-4f06-ab5f-4ecd1fe7175d>

```
1  [TestFixture]
2  public class CalculatorTests
3  {
4      [Test]
5      public void ShouldDivideNumbers()
6      {
7          var sut = new Calculator();
8
9          var actualResult = sut.Divide(10, 2);
10
11         Assert.AreEqual(5, actualResult);
12     }
13 }
```

Here the `Assert's AreEqual` method is being used. The first value “5” is the result that we expect, the second parameter `actualResult` is the resulting value from the system we are testing (the `Calculator`).

There are a number of `Assert` methods such as:

- `Assert.AreEqual`
- `Assert.False`
- `Assert.True`
- `Assert.Null`
- `Assert.NotNull`
- etc

These types of asserts are known as the classic asserts.

A newer way of asserting (from v.2.4) is called the constraint-based model. With this approach the above assert could be written as:

```
1  Assert.That(actualResult, Is.EqualTo(5));
```

For a full list of assert methods, see the [documentation](#)¹⁴.

Advanced Techniques

There are a number of advanced techniques (and associated attributes) that can be used to perform tasks such as setup and cleanup code that runs before and after each test runs.

There are also attributes such as `Category` that allow the categorising of tests into arbitrary groups. Most test runners can then run just the tests that belong to a specific category.

¹⁴<http://www.nunit.org/index.php?p=assertions&r=2.6.3>

The xUnit.net Testing Framework

[xUnit.net](#)¹⁵ is a testing framework for .NET that is available for install via NuGet.

xUnit.net based tests can be executed in Visual Studio using the built in test runner (after installing the [test adapter](#)¹⁶) or with a third-party tool such as Resharper.

Test Classes and Test Methods

Individual tests sit inside a test class.

Unlike NUnit, a class that contains tests does not need to be marked with any attribute, xUnit will discover tests in any class.

```
1 public class CalculatorTests
2 {
3 }
```

An xUnit.net test method contains a single logical test and is marked by using the `Fact` attribute.

```
1 public class CalculatorTests
2 {
3     [Fact]
4     public void ShouldDivideNumbers()
5     {
6
7     }
8 }
```

Asserting Correct Results

As with NUnit, to signal to the test runner that something is correct or broken, the `Assert` class is used.

The following example checks (asserts) that the result of dividing two numbers is correct.

¹⁵<https://github.com/xunit/xunit>

¹⁶<http://visualstudiogallery.msdn.microsoft.com/463c5987-f82b-46c8-a97e-b1cde42b9099>

```
1 public class CalculatorTests
2 {
3     [Fact]
4     public void ShouldDivideNumbers()
5     {
6         var sut = new Calculator();
7
8         var actualResult = sut.Divide(10, 2);
9
10        Assert.Equal(5, actualResult);
11    }
12 }
```

Here the `Assert.Equal` method is being used. The first value “5” is the result that we expect, the second parameter `actualResult` is the resulting value from the system we are testing (the `Calculator`).

There are a number of `Assert` methods such as:

- `Assert.NotEqual`
- `Assert.False`
- `Assert.True`
- `Assert.Null`
- `Assert.NotNull`
- etc

Advanced Techniques

There are a number of advanced techniques that can be used to perform tasks such as setup and cleanup code that runs before and after each test runs.

There is also the `xUnit.Extensions` NuGet package that among other things, provides for data-driven tests.

The following code tests the `Calculator` twice, the first time dividing 10 by 2, the second time dividing 20 by 1.

```
1 [Theory]
2 [InlineData(10, 2, 5)]
3 [InlineData(20, 1, 20)]
4 public void ShouldDivideNumbersWithDataDrivenTest(int number, int by, int expectedResult)
5 {
6     var sut = new Calculator();
7
8     var actualResult = sut.Divide(number, by);
```

```
10
11     Assert.Equal(expectedResult, actualResult);
12 }
```

Faking with Moq

When testing something in isolation, any external dependencies need to be provided as fakes. These fake versions can be created manually which is unwieldy, time-consuming, and creates a maintenance overhead.

The alternative is to use a library to generate fake versions of these dependencies.

Take the following class:

```
1 public class OrderViewModel
2 {
3     private readonly ITaxCalculator _calculator;
4
5     public OrderViewModel(ITaxCalculator calculator)
6     {
7         _calculator = calculator;
8     }
9
10    public decimal Total { get; set; }
11
12    public void CalculateTotals()
13    {
14        var itemCost = 100; // hardcoded for demo purposes
15        Total = itemCost + _calculator.CalculateItemTax(itemCost);
16    }
17 }
```

This class has an external dependency on an `ITaxCalculator`.

[Moq¹⁷](#) (or another mocking/faking framework) can create a fake version of an `ITaxCalculator` and allow this fake to be configured to respond in a specified way.

The interface is defined as follows:

```
1 public interface ITaxCalculator
2 {
3     decimal CalculateItemTax(decimal itemValue);
4 }
```

Once a fake version is created, the fake `CalculateItemTax` method can be configured to return a value, for example 10. Because the unit test now knows what the fake external dependency will return, the relevant `assert(s)` can be written in the test:

¹⁷<https://github.com/Moq/moq4>


```
1  [Fact]
2  public void ShouldCalculateTotalWithTax()
3  {
4      // create a Moq version
5      var fakeCalculator = new Moq.Mock<ITaxCalculator>();
6
7      // setup the fake how we want
8      fakeCalculator.Setup(x => x.CalculateItemTax(It.IsAny<decimal>()))
9          .Returns(10);
10
11
12     var sut = new OrderViewModel(fakeCalculator.Object);
13     sut.CalculateTotals();
14
15     var expectedResult = 110; // 100 + 10 tax
16     Assert.Equal(expectedResult, sut.Total);
17 }
```

Notice the `It.IsAny<decimal>()`, this is instructing Moq to return the value 10 regardless of what value is passed to the `CalculateItemTax` method.

Moq can also work with things such as faked property values and also perform interaction tests to check the the system under test (sut) calls specific methods on faked dependants.

Dependency Injection with Ninject

Dependency injection is a technique where dependencies of an object are created and given to it, rather than the object creating them itself. In this way, the dependency (or dependencies) are “injected” into the object.

Given the following class (from the Moq code sample), the constructor requires that a dependency of type `ITaxCalculator` is required in the constructor:

```
1 public class OrderViewModel
2 {
3     private readonly ITaxCalculator _calculator;
4
5     public OrderViewModel(ITaxCalculator calculator)
6     {
7         _calculator = calculator;
8     }
9
10    public decimal Total { get; set; }
11
12    public void CalculateTotals()
13    {
14        var itemCost = 100; // hardcoded for demo purposes
15        Total = itemCost + _calculator.CalculateItemTax(itemCost);
16    }
17 }
```

Assuming that `ITaxCalculator` has been implemented in a class called `SimpleTaxCalculator`:

```
1 public class SimpleTaxCalculator : ITaxCalculator
2 {
3     public decimal CalculateItemTax(decimal itemValue)
4     {
5         return itemValue * 0.1m;
6     }
7 }
```

A new `OrderViewModel` instance can be created as follows:

```
1 var vm = new OrderViewModel(new SimpleTaxCalculator());
```

If the `SimpleTaxCalculator` also had dependencies then they would also have to be manually new-ed. If those dependencies also had dependencies then it becomes increasingly messy to just manually create an `OrderViewModel`.

Using a Dependency Injection Framework

Ninject¹⁸ is a dependency injection framework for .NET

It is used to create objects and at the same time automatically supply the object with all its dependencies.

The following code shows how to create an instance of Ninject's `StandardKernel` that creates objects.

Once the “kernel” is created, it needs to know what concrete types to provide when a constructor requires an interface. The `OrderViewModel` requires an `ITaxCalculator` so Ninject needs to know what concrete type to use when it sees an `ITaxCalculator`.

Finally, the kernel is used to create the `OrderViewModel`, rather than creating it manually using the `new` keyword.

```
1 var dependencyInjector = new StandardKernel();
2
3 dependencyInjector.Bind<ITaxCalculator>().To<SimpleTaxCalculator>();
4
5 var vm = dependencyInjector.Get<OrderViewModel>();
```

This is only scratching the surface of dependency injection frameworks and Ninject specifically. Visit the [documentation](http://www.ninject.org/index.html)¹⁹ for more details.

¹⁸<http://www.ninject.org/index.html>

¹⁹<http://www.ninject.org/index.html>

BDDfy - A BDD Testing Framework

BDDfy is part of the open source TestStack suite of tools.

It allows the creation of tests that produce business-readable output.

There are two styles, one where methods are created using certain naming conventions and then found via reflection, and another more fluent style.

Take the following class that represents an ice cream maker machine:

```
1  public class IceCreamMaker
2  {
3      private readonly List<Ingredient> _ingredients = new List<Ingredient>();
4      public int Temp { get; set; }
5      public Consistency Consistency { get; set; }
6
7      public void AddIngredient(Ingredient ingredient)
8      {
9          _ingredients.Add(ingredient);
10     }
11
12     public void Make(ProgramType normal)
13     {
14         Chill();
15         Mix();
16         Freeze();
17     }
18
19     private void Freeze()
20     {
21         Temp = -10;
22     }
23
24     private void Mix()
25     {
26         Consistency = Consistency.Smooth;
27     }
28
29     private void Chill()
30     {
31         Temp = 5;
32     }
33 }
```

BDDfy can be used to test this class and create a business readable report from the output.

BDDfy works with existing test frameworks, so if using xUnit.net a BDDfy test would look something like the following:

```
1  [Fact]
2  public void ShouldMakeTastyChocolate()
3  {
4      Configurator.BatchProcessors.HtmlMetroReport.Enable();
5
6      this.Given(x => IHaveAddedCream())
7          .And(x => IHaveAddedChocolate())
8          .When(x => IRunTheNormalProgram())
9          .Then(x => IShouldHaveYummyChocolateIceCream())
10         .BDDfy();
11 }
```

This first line here is telling BDDfy to use a metro inspired HTML report format rather than the standard styled report.

Notice here also that BDDfy using the “Given... When... Then...” format to describe the pre-conditions, the acting, and the expected results respectively.

Each of the methods represent stages in the test:

```
1  private void IHaveAddedCream()
2  {
3      _sut = new IceCreamMaker();
4
5      _sut.AddIngredient(Ingredient.Cream);
6  }
7
8  private void IHaveAddedChocolate()
9  {
10     _sut.AddIngredient(Ingredient.Chocolate);
11 }
12
13 private void IRunTheNormalProgram()
14 {
15     _sut.Make(ProgramType.Normal);
16 }
17
18
19 private void IShouldHaveYummyChocolateIceCream()
20 {
21     Assert.Equal(-10, _sut.Temp);
22     Assert.Equal(Consistency.Smooth, _sut.Consistency);
23 }
```

When the test is run, in the output directory (e.g. bin/debug) there will be a BDDfy.html file that looks like the following:



BDDfy metro themed HTML report

There are a host of other features that BDDfy supports, visit the [documentation](#)²⁰, [GitHub site](#)²¹, or my [Pluralsight course](#)²² for more information.

The full listing for this demo code is:

²⁰<http://docs.teststack.net/bddfy/index.html>

²¹<https://github.com/TestStack>

²²<http://bit.ly/psteststack>

```
1  using System.Collections.Generic;
2  using TestStack.BDDfy;
3  using TestStack.BDDfy.Configuration;
4  using Xunit;
5
6  namespace BDDfyDemo
7  {
8      public class IceCreamMaker
9      {
10         private readonly List<Ingredient> _ingredients = new List<Ingredient>();
11         public int Temp { get; set; }
12         public Consistency Consistency { get; set; }
13
14         public void AddIngredient(Ingredient ingredient)
15         {
16             _ingredients.Add(ingredient);
17         }
18
19         public void Make(ProgramType normal)
20         {
21             Chill();
22             Mix();
23             Freeze();
24         }
25
26         private void Freeze()
27         {
28             Temp = -10;
29         }
30
31         private void Mix()
32         {
33             Consistency = Consistency.Smooth;
34         }
35
36         private void Chill()
37         {
38             Temp = 5;
39         }
40     }
41
42     [Story(AsA = "Connoisseur of ice cream",
43           IWant = "To be able to make my own ice cream",
44           SoThat = "I can satisfy my cravings")]
45     public class IceCreamMakerTests
46     {
```

```
47         private IceCreamMaker _sut;
48
49         [Fact]
50         public void ShouldMakeTastyChocolate()
51         {
52             Configurator.BatchProcessors.HtmlMetroReport.Enable();
53
54             this.Given(x => IHaveAddedCream())
55                 .And(x => IHaveAddedChocolate())
56                 .When(x => IRunTheNormalProgram())
57                 .Then(x => IShouldHaveYummyChocolateIceCream())
58                 .BDDfy();
59         }
60
61         private void IHaveAddedCream()
62         {
63             _sut = new IceCreamMaker();
64
65             _sut.AddIngredient(Ingredient.Cream);
66         }
67
68         private void IHaveAddedChocolate()
69         {
70             _sut.AddIngredient(Ingredient.Chocolate);
71         }
72
73         private void IRunTheNormalProgram()
74         {
75             _sut.Make(ProgramType.Normal);
76         }
77
78
79         private void IShouldHaveYummyChocolateIceCream()
80         {
81             Assert.Equal(-10, _sut.Temp);
82             Assert.Equal(Consistency.Smooth, _sut.Consistency);
83         }
84     }
85
86     public enum Consistency
87     {
88         Smooth,
89         Liquid,
90         Bitty
91     }
92
```



```
93      public enum ProgramType
94      {
95          Normal
96      }
97
98      public enum Ingredient
99      {
100          Cream,
101          Chocolate
102      }
103 }
```

Convention Tests

Convention Tests (part of [TestStack²³](http://docs.teststack.net/)) allow the creation of automated tests to check if a codebase conforms to given conventions.

Some examples of what Convention Test can test for include:

- All classes have a default constructor
- Methods are declared as `virtual`
- Classes with certain names are only defined in a certain namespaces

In addition to the supplied conventions, it is also possible to create new custom conventions as required.

Using the Supplied `ClassTypeHasSpecificNamespace` Convention

One of the supplied conventions allows the checking that certain types are declared in a specified namespace.

At a high level there are three steps:

1. Select the types to check from the production code assembly
2. Create an instance of the convention to be used
3. Execute the convention

Take the following classes that have been defined in a production assembly:

```
1 namespace CTDemo
2 {
3     public class ComplexConverter
4     {
5     }
6 }
7
8 namespace CTDemo
9 {
10    public class SimpleConverter
11    {
12    }
13 }
```

²³<http://docs.teststack.net/>

Both these classes are defined in the CTDemo namespace.

A test could be written to ensure all “converter” classes are only defined in a namespace called “CTDemo.Converters”.

The following code shows an example of this test:

```
1  [Fact]
2  public void AllConvertorsShouldBeInCorrectNamespace()
3  {
4      // get a list of all the types to check against the convention
5      var typesToCheck = Types.InAssemblyOf<SimpleConverter>();
6
7      // create a convention to check code against
8      // for all the types defined in the assembly containing SimpleConverter
9      // where the type's name ends with "Convention"
10     // ensure they are all defined in the CTDemo.Converters" namespace
11     // and help to output a helpful error message
12     // with "Converter related classes"
13     var convention = new ClassTypeHasSpecificNamespace(
14         x => x.Name.EndsWith("Converter"),
15         "CTDemo.Converters",
16         "Converter related classes");
17
18     // execute check
19     Convention.Is(convention, typesToCheck);
20 }
```

When this test is run it will fail with the following message:

```
'Converter related classess must be under the 'CTDemo.Converters' namespace' for
'Types in CTDemo'
CTDemo.ComplexConverter
CTDemo.SimpleConverter
```

This is because ComplexConverter and SimpleConverter have not been defined in the CTDemo.Converters namespace.

If the namespaces are changed:

```
1 namespace CTDemo.Converters
2 {
3     public class ComplexConverter
4     {
5     }
6 }
7
8 namespace CTDemo.Converters
9 {
10    public class SimpleConverter
11    {
12    }
13 }
```

The test will now pass.

For a list of the supplied conventions, check out the [GitHub site](https://github.com/TestStack/ConventionTests/tree/master/TestStack.ConventionTests/Conventions)²⁴.

²⁴<https://github.com/TestStack/ConventionTests/tree/master/TestStack.ConventionTests/Conventions>