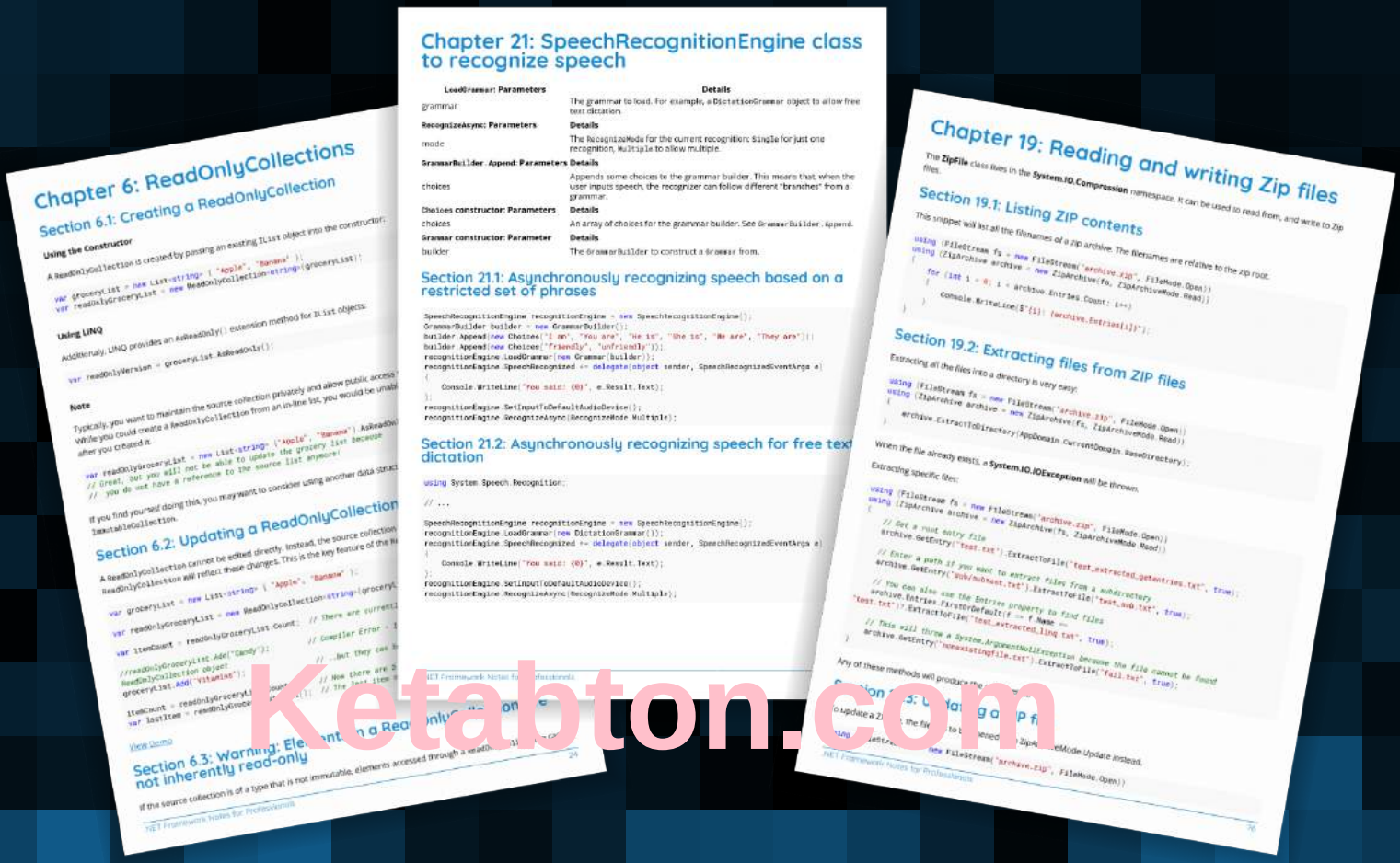
.NET Framework

Notes for Professionals



100+ pages
of professional hints and tricks

Contents

About

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

This *.NET Framework 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 .NET Framework 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 .NET Framework

.NET

Version	Release Date
1.0	2002-02-13
1.1	2003-04-24
2.0	2005-11-07
3.0	2006-11-06
3.5	2007-11-19
3.5 SP1	2008-08-11
4.0	2010-04-12
4.5	2012-08-15
4.5.1	2013-10-17
4.5.2	2014-05-05
4.6	2015-07-20
4.6.1	2015-11-17
4.6.2	2016-08-02
4.7	2017-04-05
4.7.1	2017-10-17

Compact Framework

Version	Release Date
1.0	2000-01-01
2.0	2005-10-01
3.5	2007-11-19
3.7	2009-01-01
3.9	2013-06-01

Micro Framework

Version	Release Date
4.2	2011-10-04
4.3	2012-12-04
4.4	2015-10-20

Section 1.1: Hello World in C#

```
using System;

class Program
{
    // The Main() function is the first function to be executed in a program
    static void Main()
    {
        // Write the string "Hello World to the standard out
        Console.WriteLine("Hello World");
    }
}
```

`Console.WriteLine` has several overloads. In this case, the string "Hello World" is the parameter, and it will output the "Hello World" to the standard out stream during execution. Other overloads may call the `.ToString` of the

argument before writing to the stream. See the .NET Framework Documentation for more information.

Live Demo in Action at .NET Fiddle

Introduction to C#

Section 1.2: Hello World in F#

```
open System

[<EntryPoint>]
let main argv =
    printfn "Hello World"
    0
```

Live Demo in Action at .NET Fiddle

Introduction to F#

Section 1.3: Hello World in Visual Basic .NET

```
Imports System

Module Program
    Public Sub Main()
        Console.WriteLine("Hello World")
    End Sub
End Module
```

Live Demo in Action at .NET Fiddle

Introduction to Visual Basic .NET

Section 1.4: Hello World in C++/CLI

```
using namespace System;

int main(array<String^>^ args)
{
    Console::WriteLine("Hello World");
}
```

Section 1.5: Hello World in IL

```
.class public auto ansi beforefieldinit Program
       extends [mscorlib]System.Object
{
  .method public hidebysig static void  Main() cil managed
  {
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr      "Hello World"
    IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
  }
```

```
  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  }

}
```

Section 1.6: Hello World in PowerShell

```
Write-Host "Hello World"
```

Introduction to PowerShell

Section 1.7: Hello World in Nemerle

```
System.Console.WriteLine("Hello World");
```

Section 1.8: Hello World in Python (IronPython)

```
print "Hello World"
```

```
import clr
from System import Console
Console.WriteLine("Hello World")
```

Section 1.9: Hello World in Oxygene

```
namespace HelloWorld;

interface

type
  App = class
  public
    class method Main(args: array of String);
  end;

implementation

class method App.Main(args: array of String);
begin
  Console.WriteLine('Hello World');
end;

end.
```

Section 1.10: Hello World in Boo

```
print "Hello World"
```

Chapter 2: Strings

Section 2.1: Count characters

If you need to count *characters* then, for the reasons explained in *Remarks* section, you can't simply use Length property because it's the length of the array of `System.Char` which are not characters but code-units (not Unicode code-points nor graphemes). Correct code is then:

```
int length = text.EnumerateCharacters().Count();
```

A small optimization may rewrite `EnumerateCharacters()` extension method specifically for this purpose:

```
public static class StringExtensions
{
    public static int CountCharacters(this string text)
    {
        if (String.IsNullOrEmpty(text))
            return 0;

        int count = 0;
        var enumerator = StringInfo.GetTextElementEnumerator(text);
        while (enumerator.MoveNext())
            ++count;

        return count;
    }
}
```

Section 2.2: Count distinct characters

If you need to count distinct characters then, for the reasons explained in *Remarks* section, you can't simply use `Length` property because it's the length of the array of `System.Char` which are not characters but code-units (not Unicode code-points nor graphemes). If, for example, you simply write `text.Distinct().Count()` you will get incorrect results, correct code:

```
int distinctCharactersCount = text.EnumerateCharacters().Count();
```

One step further is to **count occurrences of each character**, if performance aren't an issue you may simply do it like this (in this example regardless of case):

```
var frequencies = text.EnumerateCharacters()
    .GroupBy(x => x, StringComparer.CurrentCultureIgnoreCase)
    .Select(x => new { Character = x.Key, Count = x.Count() };
```

Section 2.3: Convert string to/from another encoding

.NET strings contain `System.Char` (UTF-16 code-units). If you want to save (or manage) text with another encoding you have to work with an array of `System.Byte`.

Conversions are performed by classes derived from `System.Text.Encoder` and `System.Text.Decoder` which, together, can convert to/from another encoding (from a byte *X* encoded array `byte[]` to an UTF-16 encoded `System.String` and vice-versa).

Because the encoder/decoder usually works very close to each other they're grouped together in a class derived

from `System.Text.Encoding`, derived classes offer conversions to/from popular encodings (UTF-8, UTF-16 and so on).

Examples:

Convert a string to UTF-8

```
byte[] data = Encoding.UTF8.GetBytes("This is my text");
```

Convert UTF-8 data to a string

```
var text = Encoding.UTF8.GetString(data);
```

Change encoding of an existing text file

This code will read content of an UTF-8 encoded text file and save it back encoded as UTF-16. Note that this code is not optimal if file is big because it will read all its content into memory:

```
var content = File.ReadAllText(path, Encoding.UTF8);
File.WriteAllText(content, Encoding.UTF16);
```

Section 2.4: Comparing strings

Despite `String` is a reference type `==` operator compares string values rather than references.

As you may know `string` is just an array of characters. But if you think that strings equality check and comparison is made character by character, you are mistaken. This operation is culture specific (see Remarks below): some character sequences can be treated as equal depending on the culture.

Think twice before short circuiting equality check by comparing `Length` properties of two strings!

Use overloads of `String.Equals` method which accept additional `StringComparison` enumeration value, if you need to change default behavior.

Section 2.5: Count occurrences of a character

Because of the reasons explained in *Remarks* section you can't simply do this (unless you want to count occurrences of a specific code-unit):

```
int count = text.Count(x => x == ch);
```

You need a more complex function:

```
public static int CountOccurrencesOf(this string text, string character)
{
    return text.EnumerateCharacters()
        .Count(x => String.Equals(x, character, StringComparer.CurrentCulture));
}
```

Note that string comparison (in contrast to character comparison which is culture invariant) must always be performed according to rules to a specific culture.

Section 2.6: Split string into fixed length blocks

We cannot break a string into arbitrary points (because a `System.Char` may not be valid alone because it's a combining character or part of a surrogate) then code must take that into account (note that with *length* I mean the

number of *graphemes* not the number of *code-units*):

```
public static IEnumerable<string> Split(this string value, int desiredLength)
{
    var characters = StringInfo.GetTextElementEnumerator(value);
    while (characters.MoveNext())
        yield return String.Concat(Take(characters, desiredLength));
}

private static IEnumerable<string> Take(TextElementEnumerator enumerator, int count)
{
    for (int i = 0; i < count; ++i)
    {
        yield return (string)enumerator.Current;

        if (!enumerator.MoveNext())
            yield break;
    }
}
```

Section 2.7: Object.ToString() virtual method

Everything in .NET is an object, hence every type has `ToString()` method defined in `Object` class which can be overridden. Default implementation of this method just returns the name of the type:

```
public class Foo
{
}

var foo = new Foo();
Console.WriteLine(foo); // outputs Foo
```

`ToString()` is implicitly called when concatinating value with a string:

```
public class Foo
{
    public override string ToString()
    {
        return "I am Foo";
    }
}

var foo = new Foo();
Console.WriteLine("I am bar and "+foo);// outputs I am bar and I am Foo
```

The result of this method is also extensively used by debugging tools. If, for some reason, you do not want to override this method, but want to customize how debugger shows the value of your type, use DebuggerDisplay Attribute (MSDN):

```
// [DebuggerDisplay("Person = FN {FirstName}, LN {LastName}")]
[DebuggerDisplay("Person = FN {"+nameof(Person.FirstName)+"}, LN {"+nameof(Person.LastName)+"}")]
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set;}
    // ...
}
```

Section 2.8: Immutability of strings

Strings are immutable. You just cannot change existing string. Any operation on the string crates a new instance of the string having new value. It means that if you need to replace a single character in a very long string, memory will be allocated for a new value.

```
string veryLongString = ...
// memory is allocated
string newString = veryLongString.Remove(0,1); // removes first character of the string.
```

If you need to perform many operations with string value, use `StringBuilder` class which is designed for efficient strings manipulation:

```
var sb = new StringBuilder(someInitialString);
foreach(var str in manyManyStrings)
{
    sb.Append(str);
}
var finalString = sb.ToString();
```

Chapter 3: DateTime parsing

Section 3.1: ParseExact

```
var dateString = "2015-11-24";

var date = DateTime.ParseExact(dateString, "yyyy-MM-dd", null);
Console.WriteLine(date);
```

> 11/24/2015 12:00:00 AM

Note that passing `CultureInfo.CurrentCulture` as the third parameter is identical to passing `null`. Or, you can pass a specific culture.

Format Strings

Input string can be in any format that matches the format string

```
var date = DateTime.ParseExact("24|201511", "dd|yyyyMM", null);
Console.WriteLine(date);
```

> 11/24/2015 12:00:00 AM

Any characters that are not format specifiers are treated as literals

```
var date = DateTime.ParseExact("2015|11|24", "yyyy|MM|dd", null);
Console.WriteLine(date);
```

> 11/24/2015 12:00:00 AM

Case matters for format specifiers

```
var date = DateTime.ParseExact("2015-01-24 11:11:30", "yyyy-mm-dd hh:MM:ss", null);
Console.WriteLine(date);
```

> 11/24/2015 11:01:30 AM

Note that the month and minute values were parsed into the wrong destinations.

Single-character format strings must be one of the standard formats

```
var date = DateTime.ParseExact("11/24/2015", "d", new CultureInfo("en-US"));
var date = DateTime.ParseExact("2015-11-24T10:15:45", "s", null);
var date = DateTime.ParseExact("2015-11-24 10:15:45Z", "u", null);
```

Exceptions

ArgumentNullException

```
var date = DateTime.ParseExact(null, "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", null, null);
```

FormatException

```
var date = DateTime.ParseExact("", "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", "", null);
var date = DateTime.ParseExact("2015-0C-24", "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", "yyyy-QQ-dd", null);

// Single-character format strings must be one of the standard formats
var date = DateTime.ParseExact("2015-11-24", "q", null);

// Format strings must match the input exactly* (see next section)
var date = DateTime.ParseExact("2015-11-24", "d", null); // Expects 11/24/2015 or 24/11/2015 for
most cultures
```

Handling multiple possible formats

```
var date = DateTime.ParseExact("2015-11-24T10:15:45",
  new [] { "s", "t", "u", "yyyy-MM-dd" }, // Will succeed as long as input matches one of these
  CultureInfo.CurrentCulture, DateTimeStyles.None);
```

Handling culture differences

```
var dateString = "10/11/2015";
var date = DateTime.ParseExact(dateString, "d", new CultureInfo("en-US"));
Console.WriteLine("Day: {0}; Month: {1}", date.Day, date.Month);
```

> Day: 11; Month: 10

```
date = DateTime.ParseExact(dateString, "d", new CultureInfo("en-GB"));
Console.WriteLine("Day: {0}; Month: {1}", date.Day, date.Month);
```

> Day: 10; Month: 11

Section 3.2: TryParse

This method accepts a string as input, attempts to parse it into a `DateTime`, and returns a Boolean result indicating success or failure. If the call succeeds, the variable passed as the `out` parameter is populated with the parsed result.

If the parse fails, the variable passed as the `out` parameter is set to the default value, `DateTime.MinValue`.

TryParse(string, out DateTime)

```
DateTime parsedValue;

if (DateTime.TryParse("monkey", out parsedValue))
{
   Console.WriteLine("Apparently, 'monkey' is a date/time value. Who knew?");
}
```

This method attempts to parse the input string based on the system regional settings and known formats such as

ISO 8601 and other common formats.

```
DateTime.TryParse("11/24/2015 14:28:42", out parsedValue); // true
DateTime.TryParse("2015-11-24 14:28:42", out parsedValue); // true
DateTime.TryParse("2015-11-24T14:28:42", out parsedValue); // true
DateTime.TryParse("Sat, 24 Nov 2015 14:28:42", out parsedValue); // true
```

Since this method does not accept culture info, it uses the system locale. This can lead to unexpected results.

```
// System set to en-US culture
bool result = DateTime.TryParse("24/11/2015", out parsedValue);
Console.WriteLine(result);
```

> False

```
// System set to en-GB culture
bool result = DateTime.TryParse("11/24/2015", out parsedValue);
Console.WriteLine(result);
```

> False

```
// System set to en-GB culture
bool result = DateTime.TryParse("10/11/2015", out parsedValue);
Console.WriteLine(result);
```

> True

Note that if you are in the US, you might be surprised that the parsed result is November 10, not October 11.

TryParse(string, IFormatProvider, DateTimeStyles, out DateTime)

```
if (DateTime.TryParse(" monkey ", new CultureInfo("en-GB"),
    DateTimeStyles.AllowLeadingWhite | DateTimeStyles.AllowTrailingWhite, out parsedValue)
{
    Console.WriteLine("Apparently, ' monkey ' is a date/time value. Who knew?");
}
```

Unlike its sibling method, this overload allows a specific culture and style(s) to be specified. Passing `null` for the `IFormatProvider` parameter uses the system culture.

Exceptions

Note that it is possible for this method to throw an exception under certain conditions. These relate to the parameters introduced for this overload: `IFormatProvider` and `DateTimeStyles`.

- `NotSupportedException`: `IFormatProvider` specifies a neutral culture
- `ArgumentException`: `DateTimeStyles` is not a valid option, or contains incompatible flags such as `AssumeLocal` and `AssumeUniversal`.

Section 3.3: TryParseExact

This method behaves as a combination of `TryParse` and `ParseExact`: It allows custom format(s) to be specified, and returns a Boolean result indicating success or failure rather than throwing an exception if the parse fails.

TryParseExact(string, string, IFormatProvider, DateTimeStyles, out DateTime)

This overload attempts to parse the input string against a specific format. The input string must match that format in order to be parsed.

```
DateTime.TryParseExact("11242015", "MMddyyyy", null, DateTimeStyles.None, out parsedValue); // true
```

TryParseExact(string, string[], IFormatProvider, DateTimeStyles, out DateTime)

This overload attempts to parse the input string against an array of formats. The input string must match at least one format in order to be parsed.

```
DateTime.TryParseExact("11242015", new [] { "yyyy-MM-dd", "MMddyyyy" }, null, DateTimeStyles.None,
out parsedValue); // true
```

Chapter 4: Dictionaries

Section 4.1: Initializing a Dictionary with a Collection Initializer

```
// Translates to `dict.Add(1, "First")` etc.
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};

// Translates to `dict[1] = "First"` etc.
// Works in C# 6.0.
var dict = new Dictionary<int, string>()
{
    [1] = "First",
    [2] = "Second",
    [3] = "Third"
};
```

Section 4.2: Adding to a Dictionary

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict.Add(1, "First");
dict.Add(2, "Second");

// To safely add items (check to ensure item does not already exist - would throw)
if(!dict.ContainsKey(3))
{
   dict.Add(3, "Third");
}
```

Alternatively they can be added/set via the an indexer. (An indexer internally looks like a property, having a get and set, but takes a parameter of any type which is specified between the brackets) :

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict[1] = "First";
dict[2] = "Second";
dict[3] = "Third";
```

Unlike the `Add` method which throws an exception, if a key is already contained in the dictionary, the indexer just replaces the existing value.

For thread-safe dictionary use `ConcurrentDictionary<TKey, TValue>`:

```
var dict = new ConcurrentDictionary<int, string>();
dict.AddOrUpdate(1, "First", (oldKey, oldValue) => "First");
```

Section 4.3: Getting a value from a dictionary

Given this setup code:

```
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
```

```
    { 2, "Second" },
    { 3, "Third" }
};
```

You may want to read the value for the entry with key 1. If key doesn't exist getting a value will throw `KeyNotFoundException`, so you may want to first check for that with `ContainsKey`:

```
if (dict.ContainsKey(1))
    Console.WriteLine(dict[1]);
```

This has one disadvantage: you will search through your dictionary twice (once to check for existence and one to read the value). For a large dictionary this can impact performance. Fortunately both operations can be performed together:

```
string value;
if (dict.TryGetValue(1, out value))
    Console.WriteLine(value);
```

Section 4.4: Make a Dictionary<string, T> with Case-Insensivitve keys

```
var MyDict = new Dictionary<string,T>(StringComparison.InvariantCultureIgnoreCase)
```

Section 4.5: IEnumerable to Dictionary (≥ .NET 3.5)

Create a Dictionary<TKey, TValue> from an IEnumerable<T>:

```
using System;
using System.Collections.Generic;
using System.Linq;


public class Fruits
{
    public int Id { get; set; }
    public string Name { get; set; }
}


var fruits = new[]
{
    new Fruits { Id = 8 , Name = "Apple" },
    new Fruits { Id = 3 , Name = "Banana" },
    new Fruits { Id = 7 , Name = "Mango" },
};


// Dictionary<int, string>                  key      value
var dictionary = fruits.ToDictionary(x => x.Id, x => x.Name);
```

Section 4.6: Enumerating a Dictionary

You can enumerate through a Dictionary in one of 3 ways:

Using KeyValue pairs

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(KeyValuePair<int, string> kvp in dict)
{
   Console.WriteLine("Key : " + kvp.Key.ToString() + ", Value : " + kvp.Value);
}
```

Using Keys

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(int key in dict.Keys)
{
    Console.WriteLine("Key : " + key.ToString() + ", Value : " + dict[key]);
}
```

Using Values

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(string s in dict.Values)
{
    Console.WriteLine("Value : " + s);
}
```

Section 4.7: ConcurrentDictionary<TKey,TValue> (from .NET 4.0)

> Represents a thread-safe collection of key/value pairs that can be accessed by multiple threads concurrently.

Creating an instance

Creating an instance works pretty much the same way as with `Dictionary<TKey, TValue>`, e.g.:

```
var dict = new ConcurrentDictionary<int, string>();
```

Adding or Updating

You might be surprised, that there is no `Add` method, but instead there is `AddOrUpdate` with 2 overloads:

(1) `AddOrUpdate(TKey key, TValue, Func<TKey, TValue, TValue> addValue)` - *Adds a key/value pair if the key does not already exist, or updates a key/value pair by using the specified function if the key already exists.*

(2) `AddOrUpdate(TKey key, Func<TKey, TValue> addValue, Func<TKey, TValue, TValue> updateValueFactory)` - *Uses the specified functions to add a key/value pair to the if the key does not already exist, or to update a key/value pair if the key already exists.*

Adding or updating a value, no matter what was the value if it was already present for given key (1):

```
string addedValue = dict.AddOrUpdate(1, "First", (updateKey, valueOld) => "First");
```

Adding or updating a value, but now altering the value in update, based on the previous value (1):

```
string addedValue2 = dict.AddOrUpdate(1, "First", (updateKey, valueOld) => $"{valueOld} Updated");
```

Using the overload (2) we can also add new value using a factory:

```
string addedValue3 = dict.AddOrUpdate(1, (key) => key == 1 ? "First" : "Not First", (updateKey,
valueOld) => $"{valueOld} Updated");
```

Getting value

Getting a value is the same as with the `Dictionary<TKey,TValue>`:

```
string value = null;
bool success = dict.TryGetValue(1, out value);
```

Getting or Adding a value

There are two mehod overloads, that will **get or add** a value in a thread-safe manner.

Get value with key 2, or add value "Second" if the key is not present:

```
string theValue = dict.GetOrAdd(2, "Second");
```

Using a factory for adding a value, if value is not present:

```
string theValue2 = dict.GetOrAdd(2, (key) => key == 2 ? "Second" : "Not Second." );
```

Section 4.8: Dictionary to List

Creating a list of KeyValuePair:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<KeyValuePair<int, int>> list = new List<KeyValuePair<int, int>>();
list.AddRange(dictionary);
```

Creating a list of keys:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<int> list = new List<int>();
list.AddRange(dictionary.Keys);
```

Creating a list of values:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<int> list = new List<int>();
list.AddRange(dictionary.Values);
```

Section 4.9: Removing from a Dictionary

Given this setup code:

```
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};
```

Use the `Remove` method to remove a key and its associated value.

```
bool wasRemoved = dict.Remove(2);
```

Executing this code removes the key 2 and it's value from the dictionary. `Remove` returns a boolean value indicating whether the specified key was found and removed from the dictionary. If the key does not exist in the dictionary, nothing is removed from the dictionary, and false is returned (no exception is thrown).

It's **incorrect** to try and remove a key by setting the value for the key to `null`.

```
dict[2] = null; // WRONG WAY TO REMOVE!
```

This will not remove the key. It will just replace the previous value with a value of `null`.

To remove all keys and values from a dictionary, use the `Clear` method.

```
dict.Clear();
```

After executing `Clear` the dictionary's `Count` will be 0, but the internal capacity remains unchanged.

Section 4.10: ContainsKey(TKey)

To check if a `Dictionary` has an specifique key, you can call the method `ContainsKey(TKey)` and provide the key of `TKey` type. The method returns a `bool` value when the key exists on the dictionary. For sample:

```
var dictionary = new Dictionary<string, Customer>()
{
   {"F1", new Customer() { FirstName = "Felipe", ... } },
   {"C2", new Customer() { FirstName = "Carl", ... } },
   {"J7", new Customer() { FirstName = "John", ... } },
   {"M5", new Customer() { FirstName = "Mary", ... } },
};
```

And check if a `C2` exists on the Dictionary:

```
if (dictionary.ContainsKey("C2"))
{
   // exists
}
```

The ContainsKey method is available on the generic version `Dictionary<TKey, TValue>`.

Section 4.11: ConcurrentDictionary augmented with Lazy'1 reduces duplicated computation

Problem

ConcurrentDictionary shines when it comes to instantly returning of existing keys from cache, mostly lock free, and contending on a granular level. But what if the object creation is really expensive, outweighing the cost of context switching, and some cache misses occur?

If the same key is requested from multiple threads, one of the objects resulting from colliding operations will be eventually added to the collection, and the others will be thrown away, wasting the CPU resource to create the object and memory resource to store the object temporarily. Other resources could be wasted as well. This is really bad.

Solution

We can combine `ConcurrentDictionary<TKey, TValue>` with `Lazy<TValue>`. The idea is that ConcurrentDictionary GetOrAdd method can only return the value which was actually added to the collection. The loosing Lazy objects could be wasted in this case too, but that's not much problem, as the Lazy object itself is relatively unexpensive. The Value property of the losing Lazy is never requested, because we are smart to only request the Value property of the one actually added to the collection - the one returned from the GetOrAdd method:

```
public static class ConcurrentDictionaryExtensions
{
    public static TValue GetOrCreateLazy<TKey, TValue>(
        this ConcurrentDictionary<TKey, Lazy<TValue>> d,
        TKey key,
        Func<TKey, TValue> factory)
    {
        return
            d.GetOrAdd(
                key,
                key1 =>
                    new Lazy<TValue>(() => factory(key1),
                    LazyThreadSafetyMode.ExecutionAndPublication)).Value;
    }
}
```

Caching of XmlSerializer objects can be particularly expensive, and there is a lot of contention at the application startup too. And there is more to this: if those are custom serializers, there will be a memory leak too for the rest of the process lifecycle. The only benefit of the ConcurrentDictionary in this case is that for the rest of the process lifecycle there will be no locks, but application startup and memory usage would be inacceptable. This is a job for our ConcurrentDictionary, augmented with Lazy:

```
private ConcurrentDictionary<Type, Lazy<XmlSerializer>> _serializers =
    new ConcurrentDictionary<Type, Lazy<XmlSerializer>>();

public XmlSerializer GetSerialier(Type t)
{
    return _serializers.GetOrCreateLazy(t, BuildSerializer);
}

private XmlSerializer BuildSerializer(Type t)
{
    throw new NotImplementedException("and this is a homework");
}
```

Chapter 5: Collections

Section 5.1: Using collection initializers

Some collection types can be initialized at the declaration time. For example, the following statement creates and initializes the `numbers` with some integers:

```
List<int> numbers = new List<int>(){10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1};
```

Internally, the C# compiler actually converts this initialization to a series of calls to the Add method. Consequently, you can use this syntax only for collections that actually support the `Add` method.

> The `Stack<T>` and `Queue<T>` classes do not support it.

For complex collections such as the `Dictionary<TKey, TValue>` class, that take key/value pairs, you can specify each key/value pair as an anonymous type in the initializer list.

```
Dictionary<int, string> employee = new Dictionary<int, string>()
     {{44, "John"}, {45, "Bob"}, {47, "James"}, {48, "Franklin"}};
```

The first item in each pair is the key, and the second is the value.

Section 5.2: Stack

There is a collection in .Net used to manage values in a `Stack` that uses the LIFO (last-in first-out) concept. The basics of stacks is the method `Push(T item)` which is used to add elements in the stack and `Pop()` which is used to get the last element added and remove it from the stack. The generic version can be used like the following code for a queue of strings.

First, add the namespace:

```
using System.Collections.Generic;
```

and use it:

```
Stack<string> stack = new Stack<string>();
stack.Push("John");
stack.Push("Paul");
stack.Push("George");
stack.Push("Ringo");

string value;
value = stack.Pop(); // return Ringo
value = stack.Pop(); // return George
value = stack.Pop(); // return Paul
value = stack.Pop(); // return John
```

There is a non generic version of the type, which works with objects.

The namespace is:

```
using System.Collections;
```

And a code sample of non generic stack:

```
Stack stack = new Stack();
stack.Push("Hello World"); // string
stack.Push(5); // int
stack.Push(1d); // double
stack.Push(true); // bool
stack.Push(new Product()); // Product object

object value;
value = stack.Pop(); // return Product (Product type)
value = stack.Pop(); // return true (bool)
value = stack.Pop(); // return 1d (double)
value = stack.Pop(); // return 5 (int)
value = stack.Pop(); // return Hello World (string)
```

There is also a method called Peek() which returns the last element added but without removing it from the `Stack`.

```
Stack<int> stack = new Stack<int>();
stack.Push(10);
stack.Push(20);

var lastValueAdded = stack.Peek(); // 20
```

It is possible to iterate on the elements on the stack and it will respect the order of the stack (LIFO).

```
Stack<int> stack = new Stack<int>();
stack.Push(10);
stack.Push(20);
stack.Push(30);
stack.Push(40);
stack.Push(50);

foreach (int element in stack)
{
   Console.WriteLine(element);
}
```

The output (without removing):

```
50
40
30
20
10
```

Section 5.3: Creating an initialized List with Custom Types

```
public class Model
{
    public string Name { get; set; }
    public bool? Selected { get; set; }
}
```

Here we have a Class with no constructor with two properties: `Name` and a nullable boolean property `Selected`. If we wanted to initialize a `List<Model>`, there are a few different ways to execute this.

```
var SelectedEmployees = new List<Model>
 {
      new Model() {Name = "Item1", Selected = true},
      new Model() {Name = "Item2", Selected = false},
      new Model() {Name = "Item3", Selected = false},
      new Model() {Name = "Item4"}
 };
```

Here, we are creating several `new` instances of our `Model` class, and initializing them with data. What if we added a constructor?

```
public class Model
{

    public Model(string name, bool? selected = false)
    {
        Name = name;
        selected = Selected;
    }
    public string Name { get; set; }
    public bool? Selected { get; set; }
}
```

This allows us to initialize our List a *little* differently.

```
var SelectedEmployees = new List<Model>
{
    new Model("Mark", true),
    new Model("Alexis"),
    new Model("")
};
```

What about a Class where one of the properties is a class itself?

```
public class Model
{
    public string Name { get; set; }
    public bool? Selected { get; set; }
}

public class ExtendedModel : Model
{
    public ExtendedModel()
    {
        BaseModel = new Model();
    }

    public Model BaseModel { get; set; }
    public DateTime BirthDate { get; set; }
}
```

Notice we reverted the constructor on the `Model` class to simplify the example a little bit.

```
var SelectedWithBirthDate = new List<ExtendedModel>
{
    new ExtendedModel()
    {
        BaseModel = new Model { Name = "Mark", Selected = true},
        BirthDate = new DateTime(2015, 11, 23)
```

```
    },
                    new ExtendedModel()
    {
        BaseModel = new Model { Name = "Random"},
        BirthDate = new DateTime(2015, 11, 23)
    }
};
```

Note that we can interchange our `List<ExtendedModel>` with `Collection<ExtendedModel>`, `ExtendedModel[]`, `object[]`, or even simply `[]`.

Section 5.4: Queue

There is a collection in .Net used to manage values in a `Queue` that uses the FIFO (first-in first-out) concept. The basics of queues is the method `Enqueue(T item)` which is used to add elements in the queue and `Dequeue()` which is used to get the first element and remove it from the queue. The generic version can be used like the following code for a queue of strings.

First, add the namespace:

```
using System.Collections.Generic;
```

and use it:

```
Queue<string> queue = new Queue<string>();
queue.Enqueue("John");
queue.Enqueue("Paul");
queue.Enqueue("George");
queue.Enqueue("Ringo");

string dequeueValue;
dequeueValue = queue.Dequeue(); // return John
dequeueValue = queue.Dequeue(); // return Paul
dequeueValue = queue.Dequeue(); // return George
dequeueValue = queue.Dequeue(); // return Ringo
```

There is a non generic version of the type, which works with objects.

The namespace is:

```
using System.Collections;
```

Adn a code sample fo non generic queue:

```
Queue queue = new Queue();
queue.Enqueue("Hello World"); // string
queue.Enqueue(5); // int
queue.Enqueue(1d); // double
queue.Enqueue(true); // bool
queue.Enqueue(new Product()); // Product object

object dequeueValue;
dequeueValue = queue.Dequeue(); // return Hello World (string)
dequeueValue = queue.Dequeue(); // return 5 (int)
dequeueValue = queue.Dequeue(); // return 1d (double)
dequeueValue = queue.Dequeue(); // return true (bool)
dequeueValue = queue.Dequeue(); // return Product (Product type)
```

There is also a method called Peek() which returns the object at the beginning of the queue without removing it the elements.

```
Queue<int> queue = new Queue<int>();
queue.Enqueue(10);
queue.Enqueue(20);
queue.Enqueue(30);
queue.Enqueue(40);
queue.Enqueue(50);

foreach (int element in queue)
{
    Console.WriteLine(i);
}
```

The output (without removing):

```
10
20
30
40
50
```

Chapter 6: ReadOnlyCollections

Section 6.1: Creating a ReadOnlyCollection

Using the Constructor

A `ReadOnlyCollection` is created by passing an existing `IList` object into the constructor:

```
var groceryList = new List<string> { "Apple", "Banana" };
var readOnlyGroceryList = new ReadOnlyCollection<string>(groceryList);
```

Using LINQ

Additionaly, LINQ provides an `AsReadOnly()` extension method for `IList` objects:

```
var readOnlyVersion = groceryList.AsReadOnly();
```

Note

Typically, you want to maintain the source collection privately and allow public access to the `ReadOnlyCollection`. While you could create a `ReadOnlyCollection` from an in-line list, you would be unable to modify the collection after you created it.

```
var readOnlyGroceryList = new List<string> {"Apple", "Banana"}.AsReadOnly();
// Great, but you will not be able to update the grocery list because
//  you do not have a reference to the source list anymore!
```

If you find yourself doing this, you may want to consider using another data structure, such as an `ImmutableCollection`.

Section 6.2: Updating a ReadOnlyCollection

A `ReadOnlyCollection` cannot be edited directly. Instead, the source collection is updated and the `ReadOnlyCollection` will reflect these changes. This is the key feature of the `ReadOnlyCollection`.

```
var groceryList = new List<string> { "Apple", "Banana" };

var readOnlyGroceryList = new ReadOnlyCollection<string>(groceryList);

var itemCount = readOnlyGroceryList.Count;  // There are currently 2 items

//readOnlyGroceryList.Add("Candy");         // Compiler Error - Items cannot be added to a
ReadOnlyCollection object
groceryList.Add("Vitamins");                 // ..but they can be added to the original collection

itemCount = readOnlyGroceryList.Count;       // Now there are 3 items
var lastItem = readOnlyGroceryList.Last();  // The last item on the read only list is now "Vitamins"
```

View Demo

Section 6.3: Warning: Elements in a ReadOnlyCollection are not inherently read-only

If the source collection is of a type that is not immutable, elements accessed through a `ReadOnlyCollection` can be

modified.

```
public class Item
{
    public string Name { get; set; }
    public decimal Price { get; set; }
}

public static void FillOrder()
{
    // An order is generated
    var order = new List<Item>
    {
        new Item { Name = "Apple", Price = 0.50m },
        new Item { Name = "Banana", Price = 0.75m },
        new Item { Name = "Vitamins", Price = 5.50m }
    };

    // The current sub total is $6.75
    var subTotal = order.Sum(item => item.Price);

    // Let the customer preview their order
    var customerPreview = new ReadOnlyCollection<Item>(order);

    // The customer can't add or remove items, but they can change
    //   the price of an item, even though it is a ReadOnlyCollection
    customerPreview.Last().Price = 0.25m;

    // The sub total is now only $1.50!
    subTotal = order.Sum(item => item.Price);
}
```

View Demo

Chapter 7: Stack and Heap

Section 7.1: Value types in use

Value types simply contain a ***value***.

All value types are derived from the System.ValueType class, and this includes most of the built in types.

When creating a new value type, the an area of memory called ***the stack*** is used.
The stack will grow accordingly, by the size the declared type. So for example, an int will always be allocated 32 bits of memory on the stack. When the value type is no longer in scope, the space on the stack will be deallocated.

The code below demonstrates a value type being assigned to a new variable. A struct is being used as a convenient way to create a custom value type (the System.ValueType class cannot be otherwise extended).

The important thing to understand is that when assigning a value type, the value itself ***copied*** to the new variable, meaning we have two distinct instances of the object, that cannot affect each other.

```
struct PersonAsValueType
{
    public string Name;
}

class Program
{
    static void Main()
    {
        PersonAsValueType personA;

        personA.Name = "Bob";

        var personB = personA;

        personA.Name = "Linda";

        Console.WriteLine(                // Outputs 'False' - because
            object.ReferenceEquals(       // personA and personB are referencing
                personA,                  // different areas of memory
                personB));

        Console.WriteLine(personA.Name);  // Outputs 'Linda'
        Console.WriteLine(personB.Name);  // Outputs 'Bob'
    }
}
```

Section 7.2: Reference types in use

Reference types are comprised of both a ***reference*** to a memory area, and a ***value*** stored within that area.
This is analogous to pointers in C/C++.

All reference types are stored on what is known as ***the heap***.
The heap is simply a managed area of memory where objects are stored. When a new object is instantiated, a part of the heap will be allocated for use by that object, and a reference to that location of the heap will be returned. The heap is managed and maintained by the *garbage collector*, and does not allow for manual intervention.

In addition to the memory space required for the instance itself, additional space is required to store the reference

itself, along with additional temporary information required by the .NET CLR.

The code below demonstrates a reference type being assigned to a new variable. In this instance, we are using a class, all classes are reference types (even if static).

When a reference type is assigned to another variable, it is the ***reference*** to the object that is copied over, **not** the value itself. This is an important distinction between value types and reference types.

The implications of this are that we now have *two* references to the same object.
Any changes to the values within that object will be reflected by both variables.

```
class PersonAsReferenceType
{
    public string Name;
}

class Program
{
    static void Main()
    {
        PersonAsReferenceType personA;

        personA = new PersonAsReferenceType { Name = "Bob" };

        var personB = personA;

        personA.Name = "Linda";

        Console.WriteLine(               // Outputs 'True' - because
            object.ReferenceEquals(      // personA and personB are referencing
                personA,                 // the *same* memory location
                personB));

        Console.WriteLine(personA.Name); // Outputs 'Linda'
        Console.WriteLine(personB.Name); // Outputs 'Linda'
    }
```

Chapter 8: LINQ

LINQ (Language Integrated Query) is an expression that retrieves data from a data source. LINQ simplifies this situation by offering a consistent model for working with data across various kinds of data sources and formats. In a LINQ query, you are always working with objects. 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 provider is available. LINQ can be used in C# and VB.

Section 8.1: SelectMany (flat map)

`Enumerable.Select` returns an output element for every input element. Whereas `Enumerable.SelectMany` produces a variable number of output elements for each input element. This means that the output sequence may contain more or fewer elements than were in the input sequence.

`Lambda expressions` passed to `Enumerable.Select` must return a single item. Lambda expressions passed to `Enumerable.SelectMany` must produce a child sequence. This child sequence may contain a varying number of elements for each element in the input sequence.

Example

```
class Invoice
{
    public int Id { get; set; }
}

class Customer
{
    public Invoice[] Invoices {get;set;}
}

var customers = new[] {
    new Customer {
        Invoices = new[] {
            new Invoice {Id=1},
            new Invoice {Id=2},
        }
    },
    new Customer {
        Invoices = new[] {
            new Invoice {Id=3},
            new Invoice {Id=4},
        }
    },
    new Customer {
        Invoices = new[] {
            new Invoice {Id=5},
            new Invoice {Id=6},
        }
    }
};

var allInvoicesFromAllCustomers = customers.SelectMany(c => c.Invoices);

Console.WriteLine(
    string.Join(",", allInvoicesFromAllCustomers.Select(i => i.Id).ToArray()));
```

Output:

> 1,2,3,4,5,6

View Demo

`Enumerable.SelectMany` can also be achieved with a syntax-based query using two consecutive `from` clauses:

```
var allInvoicesFromAllCustomers
    = from customer in customers
      from invoice in customer.Invoices
      select invoice;
```

Section 8.2: Where (filter)

This method returns an IEnumerable with all the elements that meets the lambda expression

Example

```
var personNames = new[]
{
    "Foo", "Bar", "Fizz", "Buzz"
};

var namesStartingWithF = personNames.Where(p => p.StartsWith("F"));
Console.WriteLine(string.Join(",", namesStartingWithF));
```

Output:

> Foo,Fizz

View Demo

Section 8.3: Any

Returns `true` if the collection has any elements that meets the condition in the lambda expression:

```
var numbers = new[] {1,2,3,4,5};

var isNotEmpty = numbers.Any();
Console.WriteLine(isNotEmpty); //True

var anyNumberIsOne = numbers.Any(n => n == 1);
Console.WriteLine(anyNumberIsOne); //True

var anyNumberIsSix = numbers.Any(n => n == 6);
Console.WriteLine(anyNumberIsSix); //False

var anyNumberIsOdd = numbers.Any(n => (n & 1) == 1);
Console.WriteLine(anyNumberIsOdd); //True

var anyNumberIsNegative = numbers.Any(n => n < 0);
Console.WriteLine(anyNumberIsNegative); //False
```

Section 8.4: GroupJoin

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

class Project
{
    public int DeveloperId { get; set; }
    public string Name { get; set; }
}

var developers = new[] {
    new Developer {
        Id = 1,
        Name = "Foobuzz"
    },
    new Developer {
        Id = 2,
        Name = "Barfizz"
    }
};

var projects = new[] {
    new Project {
        DeveloperId = 1,
        Name = "Hello World 3D"
    },
    new Project {
        DeveloperId = 1,
        Name = "Super Fizzbuzz Maker"
    },
    new Project {
        DeveloperId = 2,
        Name = "Citizen Kane - The action game"
    },
    new Project {
        DeveloperId = 2,
        Name = "Pro Pong 2016"
    }
};

var grouped = developers.GroupJoin(
    inner: projects,
    outerKeySelector: dev => dev.Id,
    innerKeySelector: proj => proj.DeveloperId,
    resultSelector:
        (dev, projs) => new {
            DeveloperName = dev.Name,
            ProjectNames = projs.Select(p => p.Name).ToArray()});

foreach(var item in grouped)
{
    Console.WriteLine(
        "{0}'s projects: {1}",
        item.DeveloperName,
        string.Join(", ", item.ProjectNames));
}
```

```
//Foobuzz's projects: Hello World 3D, Super Fizzbuzz Maker
//Barfizz's projects: Citizen Kane - The action game, Pro Pong 2016
```

Section 8.5: Except

```
var numbers = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var evenNumbersBetweenSixAndFourteen = new[] { 6, 8, 10, 12 };

var result = numbers.Except(evenNumbersBetweenSixAndFourteen);

Console.WriteLine(string.Join(",", result));

//1, 2, 3, 4, 5, 7, 9
```

Section 8.6: Zip

.NET Version ≥ 4.0

```
var tens = new[] {10,20,30,40,50};
var units = new[] {1,2,3,4,5};

var sums = tens.Zip(units, (first, second) => first + second);

Console.WriteLine(string.Join(",", sums));

//11,22,33,44,55
```

Section 8.7: Aggregate (fold)

Generating a new object in each step:

```
var elements = new[] {1,2,3,4,5};

var commaSeparatedElements = elements.Aggregate(
    seed: "",
    func: (aggregate, element) => $"{aggregate}{element},");

Console.WriteLine(commaSeparatedElements);  //1,2,3,4,5,
```

Using the same object in all steps:

```
var commaSeparatedElements2 = elements.Aggregate(
    seed: new StringBuilder(),
    func: (seed, element) => seed.Append($"{element},"));

Console.WriteLine(commaSeparatedElements2.ToString());  //1,2,3,4,5,
```

Using a result selector:

```
var commaSeparatedElements3 = elements.Aggregate(
    seed: new StringBuilder(),
    func: (seed, element) => seed.Append($"{element},"),
    resultSelector: (seed) => seed.ToString());
Console.WriteLine(commaSeparatedElements3);  //1,2,3,4,5,
```

If a seed is omitted, the first element becomes the seed:

```
var seedAndElements = elements.Select(n=>n.ToString());
var commaSeparatedElements4 = seedAndElements.Aggregate(
    func: (aggregate, element) => $"{aggregate}{element},");

Console.WriteLine(commaSeparatedElements4);  //12,3,4,5,
```

Section 8.8: ToLookup

```
var persons = new[] {
    new { Name="Fizz", Job="Developer"},
    new { Name="Buzz", Job="Developer"},
    new { Name="Foo", Job="Astronaut"},
    new { Name="Bar", Job="Astronaut"},
};

var groupedByJob = persons.ToLookup(p => p.Job);

foreach(var theGroup in groupedByJob)
{
    Console.WriteLine(
        "{0} are {1}s",
        string.Join(",", theGroup.Select(g => g.Name).ToArray()),
        theGroup.Key);
}

//Fizz,Buzz are Developers
//Foo,Bar are Astronauts
```

Section 8.9: Intersect

```
var numbers1to10 = new[] {1,2,3,4,5,6,7,8,9,10};
var numbers5to15 = new[] {5,6,7,8,9,10,11,12,13,14,15};

var numbers5to10 = numbers1to10.Intersect(numbers5to15);

Console.WriteLine(string.Join(",", numbers5to10));

//5,6,7,8,9,10
```

Section 8.10: Concat

```
var numbers1to5 = new[] {1, 2, 3, 4, 5};
var numbers4to8 = new[] {4, 5, 6, 7, 8};

var numbers1to8 = numbers1to5.Concat(numbers4to8);

Console.WriteLine(string.Join(",", numbers1to8));

//1,2,3,4,5,4,5,6,7,8
```

Note that duplicates are kept in the result. If this is undesirable, use `Union` instead.

Section 8.11: All

```
var numbers = new[] {1,2,3,4,5};

var allNumbersAreOdd = numbers.All(n => (n & 1) == 1);
```

```
Console.WriteLine(allNumbersAreOdd); //False

var allNumbersArePositive = numbers.All(n => n > 0);
Console.WriteLine(allNumbersArePositive); //True
```

Note that the `All` method functions by checking for the first element to evaluate as `false` according to the predicate. Therefore, the method will return `true` for *any* predicate in the case that the set is empty:

```
var numbers = new int[0];
var allNumbersArePositive = numbers.All(n => n > 0);
Console.WriteLine(allNumbersArePositive); //True
```

Section 8.12: Sum

```
var numbers = new[] {1,2,3,4};

var sumOfAllNumbers = numbers.Sum();
Console.WriteLine(sumOfAllNumbers); //10

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var totalPopulation = cities.Sum(c => c.Population);
Console.WriteLine(totalPopulation); //7500
```

Section 8.13: SequenceEqual

```
var numbers = new[] {1,2,3,4,5};
var sameNumbers = new[] {1,2,3,4,5};
var sameNumbersInDifferentOrder = new[] {5,1,4,2,3};

var equalIfSameOrder = numbers.SequenceEqual(sameNumbers);
Console.WriteLine(equalIfSameOrder); //True

var equalIfDifferentOrder = numbers.SequenceEqual(sameNumbersInDifferentOrder);
Console.WriteLine(equalIfDifferentOrder); //False
```

Section 8.14: Min

```
var numbers = new[] {1,2,3,4};

var minNumber = numbers.Min();
Console.WriteLine(minNumber); //1

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var minPopulation = cities.Min(c => c.Population);
Console.WriteLine(minPopulation); //1000
```

Section 8.15: Distinct

```
var numbers = new[] {1, 1, 2, 2, 3, 3, 4, 4, 5, 5};
var distinctNumbers = numbers.Distinct();

Console.WriteLine(string.Join(",", distinctNumbers));

//1,2,3,4,5
```

Section 8.16: Count

```
IEnumerable<int> numbers = new[] {1,2,3,4,5,6,7,8,9,10};

var numbersCount = numbers.Count();
Console.WriteLine(numbersCount); //10

var evenNumbersCount = numbers.Count(n => (n & 1) == 0);
Console.WriteLine(evenNumbersCount); //5
```

Section 8.17: Cast

`Cast` is different from the other methods of `Enumerable` in that it is an extension method for `IEnumerable`, not for `IEnumerable<T>`. Thus it can be used to convert instances of the former into instances of the later.

This does not compile since `ArrayList` does not implement `IEnumerable<T>`:

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.First());
```

This works as expected:

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.Cast<int>().First()); //1
```

`Cast` does **not** perform conversion casts. The following compiles but throws `InvalidCastException` at runtime:

```
var numbers = new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Cast<decimal>().ToArray();
```

The proper way to perform a converting cast to a collection is as follows:

```
var numbers= new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Select(n => (decimal)n).ToArray();
```

Section 8.18: Range

The two parameters to `Range` are the *first* number and the *count* of elements to produce (not the last number).

```
// prints 1,2,3,4,5,6,7,8,9,10
Console.WriteLine(string.Join(",", Enumerable.Range(1, 10)));

// prints 10,11,12,13,14
Console.WriteLine(string.Join(",", Enumerable.Range(10, 5)));
```

Section 8.19: ThenBy

`ThenBy` can only be used after a `OrderBy` clause allowing to order using multiple criteria

```
var persons = new[]
{
    new {Id = 1, Name = "Foo", Order = 1},
    new {Id = 1, Name = "FooTwo", Order = 2},
    new {Id = 2, Name = "Bar", Order = 2},
    new {Id = 2, Name = "BarTwo", Order = 1},
    new {Id = 3, Name = "Fizz", Order = 2},
    new {Id = 3, Name = "FizzTwo", Order = 1},
};

var personsSortedByName = persons.OrderBy(p => p.Id).ThenBy(p => p.Order);

Console.WriteLine(string.Join(",", personsSortedByName.Select(p => p.Name)));
//This will display :
//Foo,FooTwo,BarTwo,Bar,FizzTwo,Fizz
```

Section 8.20: Repeat

`Enumerable.Repeat` generates a sequence of a repeated value. In this example it generates "Hello" 4 times.

```
var repeats = Enumerable.Repeat("Hello", 4);

foreach (var item in repeats)
{
    Console.WriteLine(item);
}

/* output:
    Hello
    Hello
    Hello
    Hello
*/
```

Section 8.21: Empty

To create an empty IEnumerable of int:

```
IEnumerable<int> emptyList = Enumerable.Empty<int>();
```

This empty IEnumerable is cached for each Type T, so that:

```
Enumerable.Empty<decimal>() == Enumerable.Empty<decimal>(); // This is True
Enumerable.Empty<int>() == Enumerable.Empty<decimal>();     // This is False
```

Section 8.22: Select (map)

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
```

```
};

var names = persons.Select(p => p.Name);
Console.WriteLine(string.Join(",", names.ToArray()));

//Foo,Bar,Fizz,Buzz
```

This type of function is usually called `map` in functional programming languages.

Section 8.23: OrderBy

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var personsSortedByName = persons.OrderBy(p => p.Name);

Console.WriteLine(string.Join(",", personsSortedByName.Select(p => p.Id).ToArray()));

//2,4,3,1
```

Section 8.24: OrderByDescending

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var personsSortedByNameDescending = persons.OrderByDescending(p => p.Name);

Console.WriteLine(string.Join(",", personsSortedByNameDescending.Select(p => p.Id).ToArray()));

//1,3,4,2
```

Section 8.25: Contains

```
var numbers = new[] {1,2,3,4,5};
Console.WriteLine(numbers.Contains(3)); //True
Console.WriteLine(numbers.Contains(34)); //False
```

Section 8.26: First (find)

```
var numbers = new[] {1,2,3,4,5};

var firstNumber = numbers.First();
Console.WriteLine(firstNumber); //1

var firstEvenNumber = numbers.First(n => (n & 1) == 0);
Console.WriteLine(firstEvenNumber); //2
```

The following throws `InvalidOperationException` with message "Sequence contains no matching element":

```
var firstNegativeNumber = numbers.First(n => n < 0);
```

Section 8.27: Single

```
var oneNumber = new[] {5};
var theOnlyNumber = oneNumber.Single();
Console.WriteLine(theOnlyNumber);  //5

var numbers = new[] {1,2,3,4,5};

var theOnlyNumberSmallerThanTwo = numbers.Single(n => n < 2);
Console.WriteLine(theOnlyNumberSmallerThanTwo);  //1
```

The following throws `InvalidOperationException` since there is more than one element in the sequence:

```
var theOnlyNumberInNumbers = numbers.Single();
var theOnlyNegativeNumber = numbers.Single(n => n < 0);
```

Section 8.28: Last

```
var numbers = new[] {1,2,3,4,5};

var lastNumber = numbers.Last();
Console.WriteLine(lastNumber); //5

var lastEvenNumber = numbers.Last(n => (n & 1) == 0);
Console.WriteLine(lastEvenNumber); //4
```

The following throws `InvalidOperationException`:

```
var lastNegativeNumber = numbers.Last(n => n < 0);
```

Section 8.29: LastOrDefault

```
var numbers = new[] {1,2,3,4,5};

var lastNumber = numbers.LastOrDefault();
Console.WriteLine(lastNumber); //5

var lastEvenNumber = numbers.LastOrDefault(n => (n & 1) == 0);
Console.WriteLine(lastEvenNumber); //4

var lastNegativeNumber = numbers.LastOrDefault(n => n < 0);
Console.WriteLine(lastNegativeNumber); //0

var words = new[] { "one", "two", "three", "four", "five" };

var lastWord = words.LastOrDefault();
Console.WriteLine(lastWord); // five

var lastLongWord = words.LastOrDefault(w => w.Length > 4);
Console.WriteLine(lastLongWord); // three

var lastMissingWord = words.LastOrDefault(w => w.Length > 5);
```

```
Console.WriteLine(lastMissingWord); // null
```

Section 8.30: SingleOrDefault

```
var oneNumber = new[] {5};
var theOnlyNumber = oneNumber.SingleOrDefault();
Console.WriteLine(theOnlyNumber);  //5

var numbers = new[] {1,2,3,4,5};

var theOnlyNumberSmallerThanTwo = numbers.SingleOrDefault(n => n < 2);
Console.WriteLine(theOnlyNumberSmallerThanTwo);  //1

var theOnlyNegativeNumber = numbers.SingleOrDefault(n => n < 0);
Console.WriteLine(theOnlyNegativeNumber);  //0
```

The following throws `InvalidOperationException`:

```
var theOnlyNumberInNumbers = numbers.SingleOrDefault();
```

Section 8.31: FirstOrDefault

```
var numbers = new[] {1,2,3,4,5};

var firstNumber = numbers.FirstOrDefault();
Console.WriteLine(firstNumber); //1

var firstEvenNumber = numbers.FirstOrDefault(n => (n & 1) == 0);
Console.WriteLine(firstEvenNumber); //2

var firstNegativeNumber = numbers.FirstOrDefault(n => n < 0);
Console.WriteLine(firstNegativeNumber); //0

var words = new[] { "one", "two", "three", "four", "five" };

var firstWord = words.FirstOrDefault();
Console.WriteLine(firstWord); // one

var firstLongWord = words.FirstOrDefault(w => w.Length > 3);
Console.WriteLine(firstLongWord); // three

var firstMissingWord = words.FirstOrDefault(w => w.Length > 5);
Console.WriteLine(firstMissingWord); // null
```

Section 8.32: Skip

Skip will enumerate the first N items without returning them. Once item number N+1 is reached, Skip starts returning every enumerated item:

```
var numbers = new[] {1,2,3,4,5};

var allNumbersExceptFirstTwo = numbers.Skip(2);
Console.WriteLine(string.Join(",", allNumbersExceptFirstTwo.ToArray()));

//3,4,5
```

Section 8.33: Take

This method takes the first `n` elements from an enumerable.

```
var numbers = new[] {1,2,3,4,5};

var threeFirstNumbers = numbers.Take(3);
Console.WriteLine(string.Join(",", threeFirstNumbers.ToArray()));

//1,2,3
```

Section 8.34: Reverse

```
var numbers = new[] {1,2,3,4,5};
var reversed = numbers.Reverse();

Console.WriteLine(string.Join(",", reversed.ToArray()));

//5,4,3,2,1
```

Section 8.35: OfType

```
var mixed = new object[] {1,"Foo",2,"Bar",3,"Fizz",4,"Buzz"};
var numbers = mixed.OfType<int>();

Console.WriteLine(string.Join(",", numbers.ToArray()));

//1,2,3,4
```

Section 8.36: Max

```
var numbers = new[] {1,2,3,4};

var maxNumber = numbers.Max();
Console.WriteLine(maxNumber); //4

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var maxPopulation = cities.Max(c => c.Population);
Console.WriteLine(maxPopulation); //4000
```

Section 8.37: Average

```
var numbers = new[] {1,2,3,4};

var averageNumber = numbers.Average();
Console.WriteLine(averageNumber);
// 2,5
```

This method calculates the average of enumerable of numbers.

```
var cities = new[] {
```

```
    new {Population = 1000},
    new {Population = 2000},
    new {Population = 4000}
};

var averagePopulation = cities.Average(c => c.Population);
Console.WriteLine(averagePopulation);
// 2333,33
```

This method calculates the average of enumerable using delegated function.

Section 8.38: GroupBy

```
var persons = new[] {
    new { Name="Fizz", Job="Developer"},
    new { Name="Buzz", Job="Developer"},
    new { Name="Foo", Job="Astronaut"},
    new { Name="Bar", Job="Astronaut"},
};

var groupedByJob = persons.GroupBy(p => p.Job);

foreach(var theGroup in groupedByJob)
{
    Console.WriteLine(
        "{0} are {1}s",
        string.Join(",", theGroup.Select(g => g.Name).ToArray()),
        theGroup.Key);
}

//Fizz,Buzz are Developers
//Foo,Bar are Astronauts
```

Group invoices by country, generating a new object with the number of record, total paid, and average paid

```
var a = db.Invoices.GroupBy(i => i.Country)
          .Select(g => new { Country = g.Key,
                             Count = g.Count(),
                             Total = g.Sum(i => i.Paid),
                             Average = g.Average(i => i.Paid) });
```

If we want only the totals, no group

```
var a = db.Invoices.GroupBy(i => 1)
          .Select(g => new { Count = g.Count(),
                             Total = g.Sum(i => i.Paid),
                             Average = g.Average(i => i.Paid) });
```

If we need several counts

```
var a = db.Invoices.GroupBy(g => 1)
          .Select(g => new { High = g.Count(i => i.Paid >= 1000),
                             Low = g.Count(i => i.Paid < 1000),
                             Sum = g.Sum(i => i.Paid) });
```

Section 8.39: ToDictionary

Returns a new dictionary from the source `IEnumerable` using the provided keySelector function to determine keys.

Will throw an `ArgumentException` if keySelector is not injective(returns a unique value for each member of the source collection.) There are overloads which allow one to specify the value to be stored as well as the key.

```
var persons = new[] {
    new { Name="Fizz", Id=1},
    new { Name="Buzz", Id=2},
    new { Name="Foo", Id=3},
    new { Name="Bar", Id=4},
};
```

Specifying just a key selector function will create a `Dictionary<TKey,TVal>` with `TKey` the return Type of the key selector, `TVal` the original object Type, and the original object as the stored value.

```
var personsById = persons.ToDictionary(p => p.Id);
// personsById is a Dictionary<int,object>

Console.WriteLine(personsById[1].Name); //Fizz
Console.WriteLine(personsById[2].Name); //Buzz
```

Specifying a value selector function as well will create a `Dictionary<TKey,TVal>` with `TKey` still the return type of the key selector, but `TVal` now the return type of the value selector function, and the returned value as the stored value.

```
var namesById = persons.ToDictionary(p => p.Id, p => p.Name);
//namesById is a Dictionary<int,string>

Console.WriteLine(namesById[3]); //Foo
Console.WriteLine(namesById[4]); //Bar
```

As stated above, the keys returned by the key selector must be unique. The following will throw an exception.

```
var persons = new[] {
    new { Name="Fizz", Id=1},
    new { Name="Buzz", Id=2},
    new { Name="Foo", Id=3},
    new { Name="Bar", Id=4},
    new { Name="Oops", Id=4}
};

var willThrowException = persons.ToDictionary(p => p.Id)
```

If a unique key can not be given for the source collection, consider using ToLookup instead. On the surface, ToLookup behaves similarly to ToDictionary, however, in the resulting Lookup each key is paired with a collection of values with matching keys.

Section 8.40: Union

```
var numbers1to5 = new[] {1,2,3,4,5};
var numbers4to8 = new[] {4,5,6,7,8};

var numbers1to8 = numbers1to5.Union(numbers4to8);

Console.WriteLine(string.Join(",", numbers1to8));

//1,2,3,4,5,6,7,8
```

Note that duplicates are removed from the result. If this is undesirable, use `Concat` instead.

Section 8.41: ToArray

```
var numbers = new[] {1,2,3,4,5,6,7,8,9,10};
var someNumbers = numbers.Where(n => n < 6);

Console.WriteLine(someNumbers.GetType().Name);
//WhereArrayIterator`1

var someNumbersArray = someNumbers.ToArray();

Console.WriteLine(someNumbersArray.GetType().Name);
//Int32[]
```

Section 8.42: ToList

```
var numbers = new[] {1,2,3,4,5,6,7,8,9,10};
var someNumbers = numbers.Where(n => n < 6);

Console.WriteLine(someNumbers.GetType().Name);
//WhereArrayIterator`1

var someNumbersList = someNumbers.ToList();

Console.WriteLine(
    someNumbersList.GetType().Name + " - " +
    someNumbersList.GetType().GetGenericArguments()[0].Name);
//List`1 - Int32
```

Section 8.43: ElementAt

```
var names = new[] {"Foo","Bar","Fizz","Buzz"};

var thirdName = names.ElementAt(2);
Console.WriteLine(thirdName); //Fizz

//The following throws ArgumentOutOfRangeException

var minusOnethName = names.ElementAt(-1);
var fifthName = names.ElementAt(4);
```

Section 8.44: ElementAtOrDefault

```
var names = new[] {"Foo","Bar","Fizz","Buzz"};

var thirdName = names.ElementAtOrDefault(2);
Console.WriteLine(thirdName); //Fizz

var minusOnethName = names.ElementAtOrDefault(-1);
Console.WriteLine(minusOnethName); //null

var fifthName = names.ElementAtOrDefault(4);
Console.WriteLine(fifthName); //null
```

Section 8.45: SkipWhile

```
var numbers = new[] {2,4,6,8,1,3,5,7};
```

```
var oddNumbers = numbers.SkipWhile(n => (n & 1) == 0);

Console.WriteLine(string.Join(",", oddNumbers.ToArray()));

//1,3,5,7
```

Section 8.46: TakeWhile

```
var numbers = new[] {2,4,6,1,3,5,7,8};

var evenNumbers = numbers.TakeWhile(n => (n & 1) == 0);

Console.WriteLine(string.Join(",", evenNumbers.ToArray()));

//2,4,6
```

Section 8.47: DefaultIfEmpty

```
var numbers = new[] {2,4,6,8,1,3,5,7};

var numbersOrDefault = numbers.DefaultIfEmpty();
Console.WriteLine(numbers.SequenceEqual(numbersOrDefault)); //True

var noNumbers = new int[0];

var noNumbersOrDefault = noNumbers.DefaultIfEmpty();
Console.WriteLine(noNumbersOrDefault.Count()); //1
Console.WriteLine(noNumbersOrDefault.Single()); //0

var noNumbersOrExplicitDefault = noNumbers.DefaultIfEmpty(34);
Console.WriteLine(noNumbersOrExplicitDefault.Count()); //1
Console.WriteLine(noNumbersOrExplicitDefault.Single()); //34
```

Section 8.48: Join

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

class Project
{
    public int DeveloperId { get; set; }
    public string Name { get; set; }
}

var developers = new[] {
    new Developer {
        Id = 1,
        Name = "Foobuzz"
    },
    new Developer {
        Id = 2,
        Name = "Barfizz"
    }
};

var projects = new[] {
```

```
    new Project {
        DeveloperId = 1,
        Name = "Hello World 3D"
    },
    new Project {
        DeveloperId = 1,
        Name = "Super Fizzbuzz Maker"
    },
    new Project {
        DeveloperId = 2,
        Name = "Citizen Kane - The action game"
    },
    new Project {
        DeveloperId = 2,
        Name = "Pro Pong 2016"
    }
};

var denormalized = developers.Join(
    inner: projects,
    outerKeySelector: dev => dev.Id,
    innerKeySelector: proj => proj.DeveloperId,
    resultSelector:
        (dev, proj) => new {
            ProjectName = proj.Name,
            DeveloperName = dev.Name});

foreach(var item in denormalized)
{
    Console.WriteLine("{0} by {1}", item.ProjectName, item.DeveloperName);
}

//Hello World 3D by Foobuzz
//Super Fizzbuzz Maker by Foobuzz
//Citizen Kane - The action game by Barfizz
//Pro Pong 2016 by Barfizz
```

Section 8.49: Left Outer Join

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void Main(string[] args)
{
    var magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    var terry = new Person { FirstName = "Terry", LastName = "Adams" };

    var barley = new Pet { Name = "Barley", Owner = terry };

    var people = new[] { magnus, terry };
    var pets = new[] { barley };
```

```
    var query =
        from person in people
        join pet in pets on person equals pet.Owner into gj
        from subpet in gj.DefaultIfEmpty()
        select new
        {
            person.FirstName,
            PetName = subpet?.Name ?? "-" // Use - if he has no pet
        };

    foreach (var p in query)
        Console.WriteLine($"{p.FirstName}: {p.PetName}");
}
```

Chapter 9: ForEach

Section 9.1: Extension method for IEnumerable

`ForEach()` is defined on the `List<T>` class, but not on `IQueryable<T>` or `IEnumerable<T>`. You have two choices in those cases:

ToList first

The enumeration (or query) will be evaluated, copying the results into a new list or calling the database. The method is then called on each item.

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ToList().ForEach(c => c.SendEmail());
```

This method has obvious memory usage overhead, as an intermediate list is created.

Extension method

Write an extension method:

```
public static void ForEach<T>(this IEnumerable<T> enumeration, Action<T> action)
{
    foreach(T item in enumeration)
    {
        action(item);
    }
}
```

Use:

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ForEach(c => c.SendEmail());
```

Caution: The Framework's LINQ methods have been designed with the intention of being *pure*, which means they do not produce side effects. The `ForEach` method's only purpose is to produce side effects, and deviates from the other methods in this aspect. You may consider just using a plain `foreach` loop instead.

Section 9.2: Calling a method on an object in a list

```
public class Customer {
   public void SendEmail()
   {
       // Sending email code here
   }
}

List<Customer> customers = new List<Customer>();

customers.Add(new Customer());
customers.Add(new Customer());

customers.ForEach(c => c.SendEmail());
```

Chapter 10: Reflection

Section 10.1: What is an Assembly?

Assemblies are the building block of any Common Language Runtime (CLR) application. Every type you define, together with its methods, properties and their bytecode, is compiled and packaged inside an Assembly.

```
using System.Reflection;
```

```
Assembly assembly = this.GetType().Assembly;
```

Assemblies are self-documenting: they do not only contain types, methods and their IL code, but also the Metadata necessary to inspect and consume them, both at compile and runtime:

```
Assembly assembly = Assembly.GetExecutingAssembly();

foreach (var type in assembly.GetTypes())
{
    Console.WriteLine(type.FullName);
}
```

Assemblies have names which describes their full, unique identity:

```
Console.WriteLine(typeof(int).Assembly.FullName);
// Will print: "mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
```

If this name includes a `PublicKeyToken`, it is called a *strong name*. Strong-naming an assembly is the process of creating a signature by using the private key that corresponds to the public key distributed with the assembly. This signature is added to the Assembly manifest, which contains the names and hashes of all the files that make up the assembly, and its `PublicKeyToken` becomes part of the name. Assemblies that have the same strong name should be identical; strong names are used in versioning and to prevent assembly conflicts.

Section 10.2: Compare two objects with reflection

```
public class Equatable
{
    public string field1;

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

        var type = obj.GetType();
        if (GetType() != type)
            return false;

        var fields = type.GetFields(BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public);
        foreach (var field in fields)
            if (field.GetValue(this) != field.GetValue(obj))
                return false;

        return true;
    }
```

```
    public override int GetHashCode()
    {
        var accumulator = 0;
        var fields = GetType().GetFields(BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public);
        foreach (var field in fields)
            accumulator = unchecked ((accumulator * 937) ^ field.GetValue(this).GetHashCode());

        return accumulator;
    }
}
```

Note: this example do a field based comparasion (ignore static fields and properties) for simplicity

Section 10.3: Creating Object and setting properties using reflection

Lets say we have a class `Classy` that has property Propertua

```
public class Classy
{
    public string Propertua {get; set;}
}
```

to set `Propertua` using reflection:

```
var typeOfClassy = typeof (Classy);
var classy = new Classy();
var prop = typeOfClassy.GetProperty("Propertua");
prop.SetValue(classy, "Value");
```

Section 10.4: How to create an object of T using Reflection

Using the default constructor

```
T variable = Activator.CreateInstance(typeof(T));
```

Using parameterized constructor

```
T variable = Activator.CreateInstance(typeof(T), arg1, arg2);
```

Section 10.5: Getting an attribute of an enum with reflection (and caching it)

Attributes can be useful for denoting metadata on enums. Getting the value of this can be slow, so it is important to cache results.

```
    private static Dictionary<object, object> attributeCache = new Dictionary<object, object>();

    public static T GetAttribute<T, V>(this V value)
        where T : Attribute
        where V : struct
    {
        object temp;

        // Try to get the value from the static cache.
```

```
        if (attributeCache.TryGetValue(value, out temp))
        {
            return (T) temp;
        }
        else
        {
            // Get the type of the struct passed in.
            Type type = value.GetType();
            FieldInfo fieldInfo = type.GetField(value.ToString());

            // Get the custom attributes of the type desired found on the struct.
            T[] attribs = (T[])fieldInfo.GetCustomAttributes(typeof(T), false);

            // Return the first if there was a match.
            var result = attribs.Length > 0 ? attribs[0] : null;

            // Cache the result so future checks won't need reflection.
            attributeCache.Add(value, result);

            return result;
        }
    }
```

Chapter 11: Expression Trees

Section 11.1: building a predicate of form field == value

To build up an expression like `_ => _.Field == "VALUE"` at runtime.

Given a predicate `_ => _.Field` and a string value `"VALUE"`, create an expression that tests whether or not the predicate is true.

The expression is suitable for:

- `IQueryable<T>`, `IEnumerable<T>` to test the predicate.
- entity framework or `Linq` to `SQL` to create a `Where` clause that tests the predicate.

This method will build an appropriate `Equal` expression that tests whether or not `Field` equals `"VALUE"`.

```
public static Expression<Func<T, bool>> BuildEqualPredicate<T>(
    Expression<Func<T, string>> memberAccessor,
    string term)
{
    var toString = Expression.Convert(Expression.Constant(term), typeof(string));
    Expression expression = Expression.Equal(memberAccessor.Body, toString);
    var predicate = Expression.Lambda<Func<T, bool>>(
        expression,
        memberAccessor.Parameters);
    return predicate;
}
```

The predicate can be used by including the predicate in a `Where` extension method.

```
var predicate = PredicateExtensions.BuildEqualPredicate<Entity>(
    _ => _.Field,
    "VALUE");
var results = context.Entity.Where(predicate).ToList();
```

Section 11.2: Simple Expression Tree Generated by the C# Compiler

Consider the following C# code

```
Expression<Func<int, int>> expression = a => a + 1;
```

Because the C# compiler sees that the lambda expression is assigned to an Expression type rather than a delegate type it generates an expression tree roughly equivalent to this code

```
ParameterExpression parameterA = Expression.Parameter(typeof(int), "a");
var expression = (Expression<Func<int, int>>)Expression.Lambda(
                                                Expression.Add(
                                                    parameterA,
                                                    Expression.Constant(1)),
                                                parameterA);
```

The root of the tree is the lambda expression which contains a body and a list of parameters. The lambda has 1 parameter called "a". The body is a single expression of CLR type BinaryExpression and NodeType of Add. This expression represents addition. It has two subexpressions denoted as Left and Right. Left is the

ParameterExpression for the parameter "a" and Right is a ConstantExpression with the value 1.

The simplest usage of this expression is printing it:

```
Console.WriteLine(expression); //prints a => (a + 1)
```

Which prints the equivalent C# code.

The expression tree can be compiled into a C# delegate and executed by the CLR

```
Func<int, int> lambda = expression.Compile();
Console.WriteLine(lambda(2)); //prints 3
```

Usually expressions are translated to other languages like SQL, but can be also used to invoke private, protected and internal members of public or non-public types as alternative to Reflection.

Section 11.3: Expression for retrieving a static field

Having example type like this:

```
public TestClass
{
    public static string StaticPublicField = "StaticPublicFieldValue";
}
```

We can retrieve value of StaticPublicField:

```
var fieldExpr = Expression.Field(null, typeof(TestClass), "StaticPublicField");
var labmda = Expression.Lambda<Func<string>>(fieldExpr);
```

It can be then i.e. compiled into a delegate for retrieving field value.

```
Func<string> retriever = lambda.Compile();
var fieldValue = retriever();
```

//fieldValue result is StaticPublicFieldValue

Section 11.4: InvocationExpression Class

InvocationExpression class allows invocation of other lambda expressions that are parts of the same Expression tree.

You create them with static `Expression.Invoke` method.

Problem We want to get on the items which have "car" in their description. We need to check it for null before searching for a string inside but we don't want it to be called excessively, as the computation could be expensive.

```
using System;
using System.Linq;
using System.Linq.Expressions;

public class Program
{
    public static void Main()
    {
        var elements = new[] {
```

```
            new Element { Description = "car" },
            new Element { Description = "cargo" },
            new Element { Description = "wheel" },
            new Element { Description = null },
            new Element { Description = "Madagascar" },
        };

        var elementIsInterestingExpression = CreateSearchPredicate(
            searchTerm: "car",
            whereToSearch: (Element e) => e.Description);

        Console.WriteLine(elementIsInterestingExpression.ToString());

        var elementIsInteresting = elementIsInterestingExpression.Compile();
        var interestingElements = elements.Where(elementIsInteresting);
        foreach (var e in interestingElements)
        {
            Console.WriteLine(e.Description);
        }

        var countExpensiveComputations = 0;
        Action incCount = () => countExpensiveComputations++;
        elements
            .Where(
                CreateSearchPredicate(
                    "car",
                    (Element e) => ExpensivelyComputed(
                        e, incCount
                    )
                ).Compile()
            )
            .Count();

        Console.WriteLine("Property extractor is called {0} times.", countExpensiveComputations);
    }

    private class Element
    {
        public string Description { get; set; }
    }

    private static string ExpensivelyComputed(Element source, Action count)
    {
        count();
        return source.Description;
    }

    private static Expression<Func<T, bool>> CreateSearchPredicate<T>(
            string searchTerm,
            Expression<Func<T, string>> whereToSearch)
    {
        var extracted = Expression.Parameter(typeof(string), "extracted");

        Expression<Func<string, bool>> coalesceNullCheckWithSearch =
            Expression.Lambda<Func<string, bool>>(
                Expression.AndAlso(
                    Expression.Not(
                        Expression.Call(typeof(string), "IsNullOrEmpty", null, extracted)
                    ),
                    Expression.Call(extracted, "Contains", null, Expression.Constant(searchTerm))
                ),
                extracted);
```

```
        var elementParameter = Expression.Parameter(typeof(T), "element");

        return Expression.Lambda<Func<T, bool>>(
            Expression.Invoke(
                coalesceNullCheckWithSearch,
                Expression.Invoke(whereToSearch, elementParameter)
            ),
            elementParameter
        );
    }
}
```

Output

```
element => Invoke(extracted => (Not(IsNullOrEmpty(extracted)) AndAlso extracted.Contains("car")),
Invoke(e => e.Description, element))
car
cargo
Madagascar
Predicate is called 5 times.
```

First thing to note is how the actual propery access, wrapped in an Invoke:

```
Invoke(e => e.Description, element)
```

, and this is the only part that touches `e.Description`, and in place of it, `extracted` parameter of type `string` is passed to the next one:

```
(Not(IsNullOrEmpty(extracted)) AndAlso extracted.Contains("car"))
```

Another important thing to note here is `AndAlso`. It computes only the left part, if the first part returns 'false'. It's a common mistake to use the bitwise operator 'And' instead of it, which always computes both parts, and would fail with a NullReferenceException in this example.

Chapter 12: Custom Types

Section 12.1: Struct Definition

Structs inherit from System.ValueType, are value types, and live on the stack. When value types are passed as a parameter, they are passed by value.

```
Struct MyStruct
{
    public int x;
    public int y;
}
```

Passed by value means that the value of the parameter is *copied* for the method, and any changes made to the parameter in the method are not reflected outside of the method. For instance, consider the following code, which calls a method named `AddNumbers`, passing in the variables `a` and `b`, which are of type `int`, which is a Value type.

```
int a = 5;
int b = 6;

AddNumbers(a,b);

public AddNumbers(int x, int y)
{
    int z = x + y; // z becomes 11
    x = x + 5; // now we changed x to be 10
    z = x + y; // now z becomes 16
}
```

Even though we added 5 to `x` inside the method, the value of `a` remains unchanged, because it's a Value type, and that means `x` was a *copy* of `a`'s value, but not actually `a`.

Remember, Value types live on the stack, and are passed by value.

Section 12.2: Class Definition

Classes inherit from System.Object, are reference types, and live on the heap. When reference types are passed as a parameter, they are passed by reference.

```
public Class MyClass
{
    public int a;
    public int b;
}
```

Passed by reference means that a *reference* to the parameter is passed to the method, and any changes to the parameter will be reflected outside of the method when it returns, because the reference is *to the exact same object in memory*. Let's use the same example as before, but we'll "wrap" the `int`s in a class first.

```
MyClass instanceOfMyClass = new MyClass();
instanceOfMyClass.a = 5;
instanceOfMyClass.b = 6;

AddNumbers(instanceOfMyClass);

public AddNumbers(MyClass sample)
{
```

```
    int z = sample.a + sample.b; // z becomes 11
    sample.a = sample.a + 5; // now we changed a to be 10
    z = sample.a + sample.b; // now z becomes 16
}
```

This time, when we changed `sample.a` to `10`, the value of `instanceOfMyClass.a` *also* changes, because it was *passed by reference*. Passed by reference means that a *reference* (also sometimes called a *pointer*) to the object was passed into the method, instead of a copy of the object itself.

Remember, Reference types live on the heap, and are passed by reference.

Chapter 13: Code Contracts

Section 13.1: Contracts for Interfaces

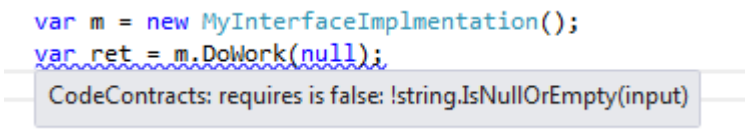
Using Code Contracts it is possible to apply a contract to an interface. This is done by declaring an abstract class that implments the interfaces. The interface should be tagged with the `ContractClassAttribute` and the contract definition (the abstract class) should be tagged with the `ContractClassForAttribute`

C# Example...

```
[ContractClass(typeof(MyInterfaceContract))]
public interface IMyInterface
{
    string DoWork(string input);
}
//Never inherit from this contract defintion class
[ContractClassFor(typeof(IMyInterface))]
internal abstract class MyInterfaceContract : IMyInterface
{
    private MyInterfaceContract() { }

    public string DoWork(string input)
    {
        Contract.Requires(!string.IsNullOrEmpty(input));
        Contract.Ensures(!string.IsNullOrEmpty(Contract.Result<string>()));
        throw new NotSupportedException();
    }
}
public class MyInterfaceImplmentation : IMyInterface
{
    public string DoWork(string input)
    {
        return input;
    }
}
```

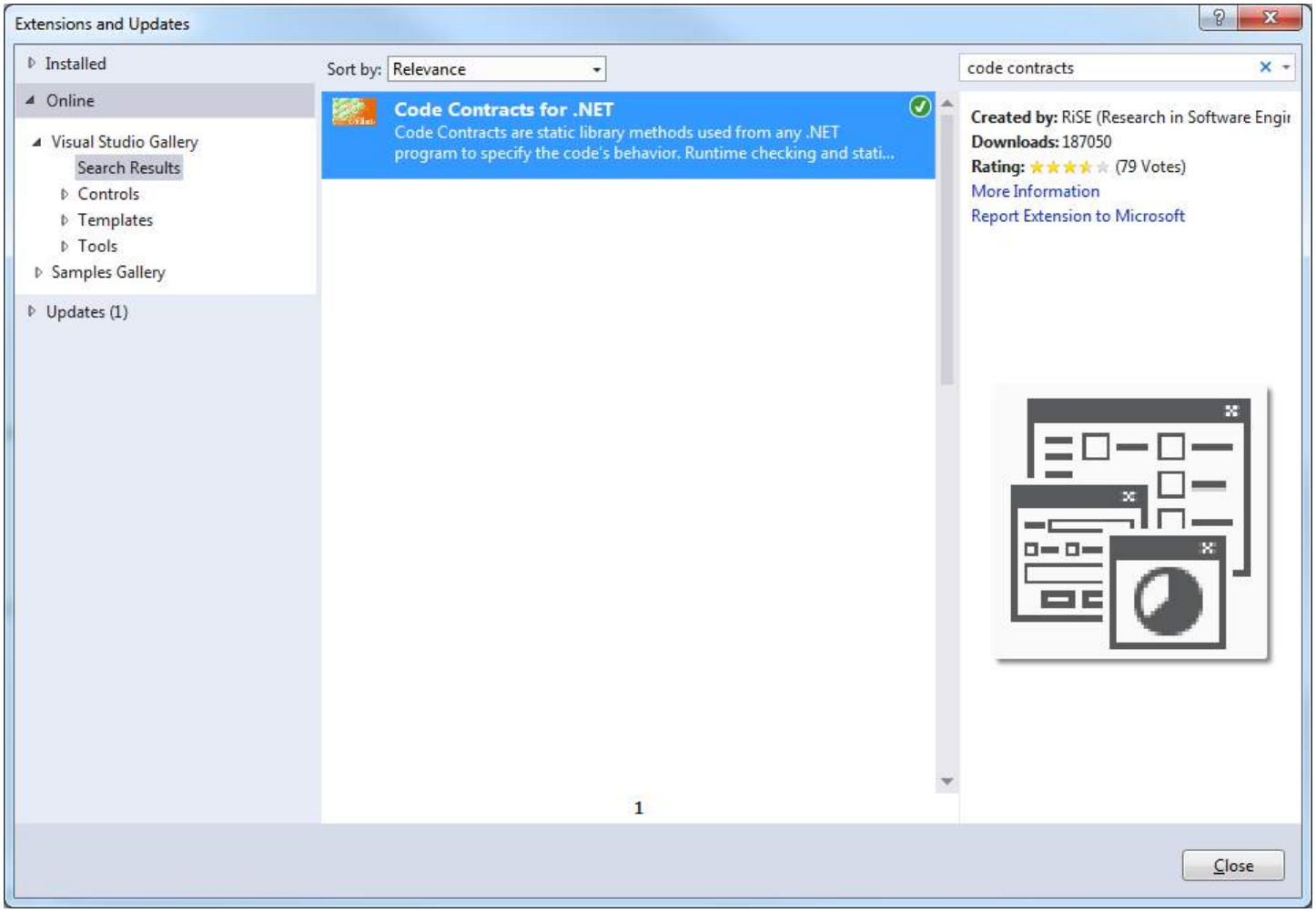
Static Analysis Result...



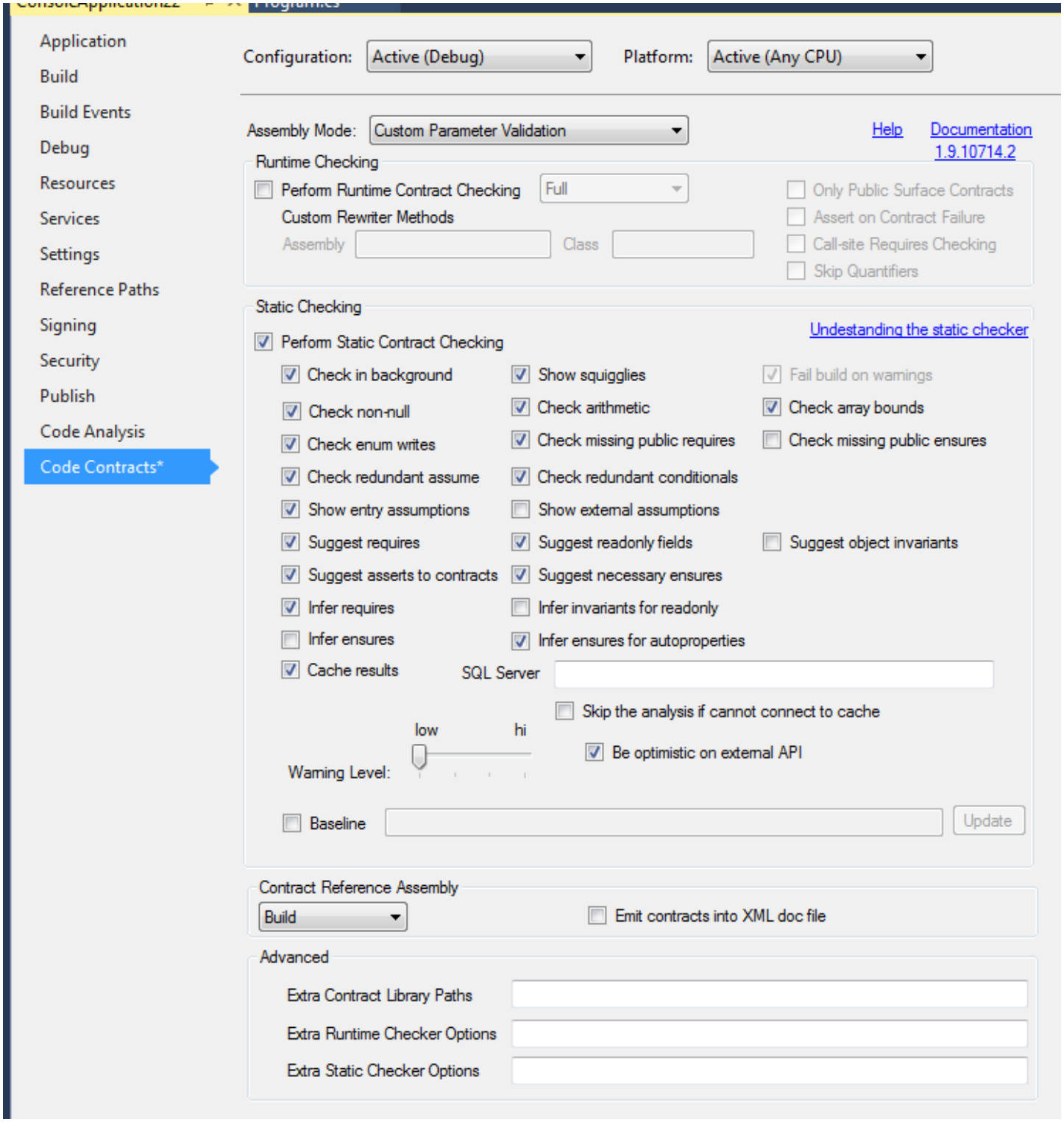
Section 13.2: Installing and Enabling Code Contracts

While `System.Diagnostics.Contracts` is included within the .Net Framework. To use Code Contracts you must install the Visual Studio extensions.

Under `Extensions and Updates` search for `Code Contracts` then install the `Code Contracts Tools`



After the tools are installed you must enable `Code Contracts` within your Project solution. At the minimum you probably want to enable the `Static Checking` (check after build). If you are implementing a library that will be used by other solutions you may want to consider also enabling `Runtime Checking`.



Section 13.3: Preconditions

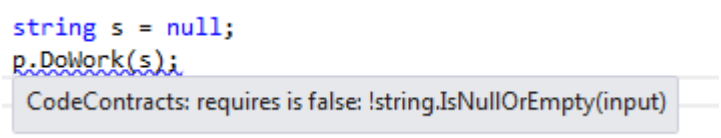
Preconditions allows methods to provide minimum required values for input parameters

Example...

```
void DoWork(string input)
{
    Contract.Requires(!string.IsNullOrEmpty(input));

    //do work
}
```

Static Analysis Result...



Section 13.4: Postconditions

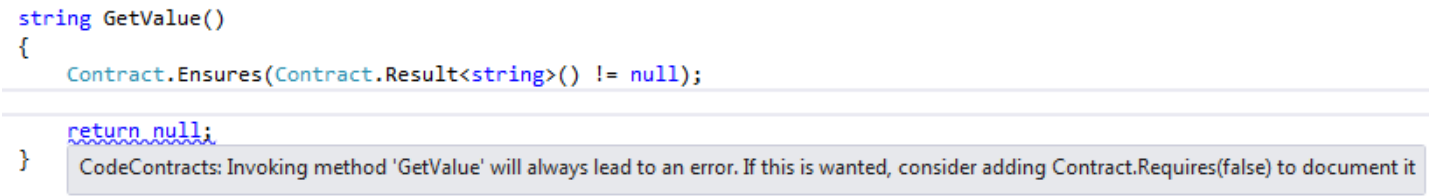
Postconditions ensure that the returned results from a method will match the provided definition. This provides the caller with a definition of the expected result. Postconditions may allowed for simplied implmentations as some possible outcomes can be provided by the static analyizer.

Example...

```
string GetValue()
{
    Contract.Ensures(Contract.Result<string>() != null);

    return null;
}
```

Static Analyis Result...



Chapter 14: Settings

Section 14.1: AppSettings from ConfigurationSettings in .NET 1.x

Deprecated usage

The ConfigurationSettings class was the original way to retrieve settings for an assembly in .NET 1.0 and 1.1. It has been superseded by the ConfigurationManager class and the WebConfigurationManager class.

If you have two keys with the same name in the `appSettings` section of the configuration file, the last one is used.

app.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="keyName" value="anything, as a string"/>
    <add key="keyNames" value="123"/>
    <add key="keyNames" value="234"/>
  </appSettings>
</configuration>
```

Program.cs

```
using System;
using System.Configuration;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            string keyValue = ConfigurationSettings.AppSettings["keyName"];
            Debug.Assert("anything, as a string".Equals(keyValue));

            string twoKeys = ConfigurationSettings.AppSettings["keyNames"];
            Debug.Assert("234".Equals(twoKeys));

            Console.ReadKey();
        }
    }
}
```

Section 14.2: Reading AppSettings from ConfigurationManager in .NET 2.0 and later

The ConfigurationManager class supports the `AppSettings` property, which allows you to continue reading settings from the `appSettings` section of a configuration file the same way as .NET 1.x supported.

app.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
```

```
  <appSettings>
    <add key="keyName" value="anything, as a string"/>
    <add key="keyNames" value="123"/>
    <add key="keyNames" value="234"/>
  </appSettings>
</configuration>
```

Program.cs

```
using System;
using System.Configuration;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            string keyValue = ConfigurationManager.AppSettings["keyName"];
            Debug.Assert("anything, as a string".Equals(keyValue));

            var twoKeys = ConfigurationManager.AppSettings["keyNames"];
            Debug.Assert("234".Equals(twoKeys));

            Console.ReadKey();
        }
    }
}
```

Section 14.3: Introduction to strongly-typed application and user settings support from Visual Studio

Visual Studio helps manage user and application settings. Using this approach has these benefits over using the `appSettings` section of the configuration file.

1. Settings can be made strongly typed. Any type which can be serialized can be used for a settings value.
2. Application settings can be easily separated from user settings. Application settings are stored in a single configuration file: `web.config` for Web sites and Web applications, and app.config, renamed as *assembly*.exe.config, where *assembly* is the name of the executable. User settings (not used by Web projects) are stored in a `user.config` file in the user's Application Data folder (which varies with the operating system version).
3. Application settings from class libraries can be combined into a single configuration file without risk of name collisions, since each class library can have its own custom settings section.

In most project types, the Project Properties Designer has a Settings tab which is the starting point for creating custom application and user settings. Initially, the Settings tab will be blank, with a single link to create a default settings file. Clicking the link results in these changes:

1. If a configuration file (`app.config` or `web.config`) does not exist for the project, one will be created.
2. The Settings tab will be replaced with a grid control which enables you to create, edit, and delete individual settings entries.
3. In Solution Explorer, a `Settings.settings` item is added under the Properties special folder. Opening this

item will open the Settings tab.

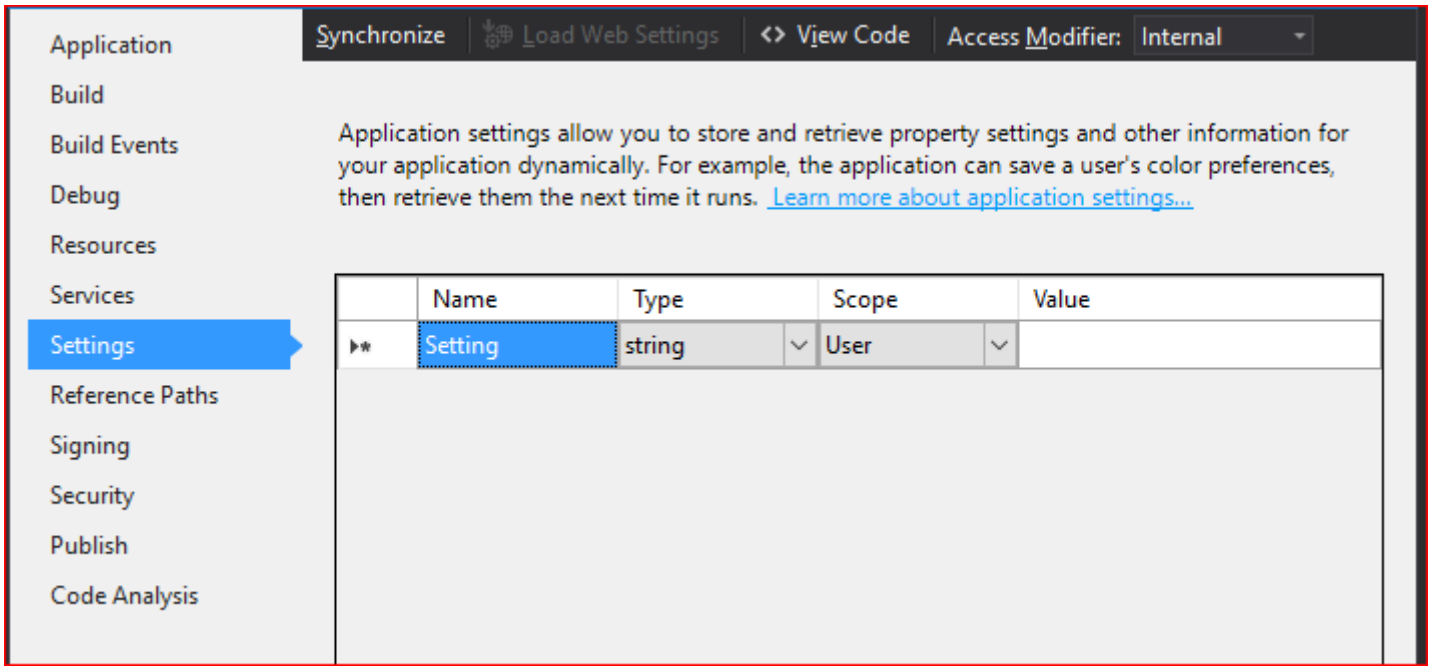
4. A new file with a new partial class is added under the `Properties` folder in the project folder. This new file is named `Settings.Designer.__` (.cs, .vb, etc.), and the class is named `Settings`. The class is code-generated, so it should not be edited, but the class is a partial class, so you can extend the class by putting additional members in a separate file. Furthermore, the class is implemented using the Singleton Pattern, exposing the singleton instance with the property named `Default`.

As you add each new entry to the Settings tab, Visual Studio does these two things:

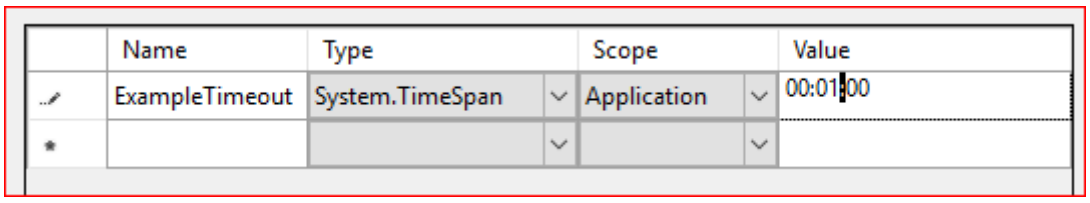
1. Saves the setting in the configuration file, in a custom configuration section designed to be managed by the Settings class.
2. Creates a new member in the Settings class to read, write, and present the setting in the specific type selected from the Settings tab.

Section 14.4: Reading strongly-typed settings from custom section of configuration file

Starting from a new Settings class and custom configuration section:



Add an application setting named ExampleTimeout, using the time System.Timespan, and set the value to 1 minute:



Save the Project Properties, which saves the Settings tab entries, as well as re-generates the custom Settings class and updates the project configuration file.

Use the setting from code (C#):

Program.cs

```
using System;
using System.Diagnostics;
using ConsoleApplication1.Properties;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            TimeSpan exampleTimeout = Settings.Default.ExampleTimeout;
            Debug.Assert(TimeSpan.FromMinutes(1).Equals(exampleTimeout));

            Console.ReadKey();
        }
    }
}
```

Under the covers

Look in the project configuration file to see how the application setting entry has been created:

app.config (Visual Studio updates this automatically)

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <sectionGroup name="applicationSettings" type="System.Configuration.ApplicationSettingsGroup,
System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" >
      <section name="ConsoleApplication1.Properties.Settings"
type="System.Configuration.ClientSettingsSection, System, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" requirePermission="false" />
    </sectionGroup>
  </configSections>
  <appSettings />
  <applicationSettings>
    <ConsoleApplication1.Properties.Settings>
      <setting name="ExampleTimeout" serializeAs="String">
        <value>00:01:00</value>
      </setting>
    </ConsoleApplication1.Properties.Settings>
  </applicationSettings>
</configuration>
```

Notice that the `appSettings` section is not used. The `applicationSettings` section contains a custom namespace-qualified section that has a `setting` element for each entry. The type of the value is not stored in the configuration file; it is only known by the `Settings` class.

Look in the `Settings` class to see how it uses the `ConfigurationManager` class to read this custom section.

Settings.designer.cs (for C# projects)

```
...
    [global::System.Configuration.ApplicationScopedSettingAttribute()]
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.Configuration.DefaultSettingValueAttribute("00:01:00")]
    public global::System.TimeSpan ExampleTimeout {
        get {
            return ((global::System.TimeSpan)(this["ExampleTimeout"]));
```

```
        }
    }
...
```

Notice that a `DefaultSettingValueAttribute` was created to stored the value entered in the Settings tab of the Project Properties Designer. If the entry is missing from the configuration file, this default value is used instead.

Chapter 15: Regular Expressions (System.Text.RegularExpressions)

Section 15.1: Check if pattern matches input

```
public bool Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. W.rld!";

    // true
    return Regex.IsMatch(input, pattern);
}
```

Section 15.2: Remove non alphanumeric characters from string

```
public string Remove()
{
    string input = "Hello./!";

    return Regex.Replace(input, "[^a-zA-Z0-9]", "");
}
```

Section 15.3: Passing Options

```
public bool Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. W.rld!";

    // true
    return Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase | RegexOptions.Singleline);
}
```

Section 15.4: Match into groups

```
public string Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. (?<Subject>W.rld)!";

    Match match = Regex.Match(input, pattern);

    // World
    return match.Groups["Subject"].Value;
}
```

Section 15.5: Find all matches

Using

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

Code

```
static void Main(string[] args)
{
    string input = "Carrot Banana Apple Cherry Clementine Grape";
    // Find words that start with uppercase 'C'
    string pattern = @"\bC\w*\b";

    MatchCollection matches = Regex.Matches(input, pattern);
    foreach (Match m in matches)
        Console.WriteLine(m.Value);
}
```

Output

```
Carrot
Cherry
Clementine
```

Section 15.6: Simple match and replace

```
public string Check()
{
    string input = "Hello World!";
    string pattern = @"W.rld";

    // Hello Stack Overflow!
    return Regex.Replace(input, pattern, "Stack Overflow");
}
```

Chapter 16: File Input/Output

Parameter	Details
string path	Path of the file to check. (relative or fully qualified)

Section 16.1: C# File.Exists()

```
using System;
using System.IO;

public class Program
{
    public static void Main()
    {
        string filePath = "somePath";

        if(File.Exists(filePath))
        {
            Console.WriteLine("Exists");
        }
        else
        {
            Console.WriteLine("Does not exist");
        }
    }
}
```

Can also be used in a ternary operator.

```
Console.WriteLine(File.Exists(pathToFile) ? "Exists" : "Does not exist");
```

Section 16.2: VB WriteAllText

```
Imports System.IO

Dim filename As String = "c:\path\to\file.txt"
File.WriteAllText(filename, "Text to write" & vbCrLf)
```

Section 16.3: VB StreamWriter

```
Dim filename As String = "c:\path\to\file.txt"
If System.IO.File.Exists(filename) Then
    Dim writer As New System.IO.StreamWriter(filename)
    writer.Write("Text to write" & vbCrLf) 'Add a newline
    writer.close()
End If
```

Section 16.4: C# StreamWriter

```
using System.Text;
using System.IO;

string filename = "c:\path\to\file.txt";
//'using' structure allows for proper disposal of stream.
using (StreamWriter writer = new StreamWriter(filename"))
```

```
{
    writer.WriteLine("Text to Write\n");
}
```

Section 16.5: C# WriteAllText()

```
using System.IO;
using System.Text;

string filename = "c:\path\to\file.txt";
File.writeAllText(filename, "Text to write\n");
```

Chapter 17: System.IO

Section 17.1: Reading a text file using StreamReader

```
string fullOrRelativePath = "testfile.txt";

string fileData;

using (var reader = new StreamReader(fullOrRelativePath))
{
    fileData = reader.ReadToEnd();
}
```

Note that this `StreamReader` constructor overload does some auto encoding detection, which may or may not conform to the actual encoding used in the file.

Please note that there are some convenience methods that read all text from file available on the `System.IO.File` class, namely `File.ReadAllText(path)` and `File.ReadAllLines(path)`.

Section 17.2: Serial Ports using System.IO.SerialPorts

Iterating over connected serial ports

```
using System.IO.Ports;
string[] ports = SerialPort.GetPortNames();
for (int i = 0; i < ports.Length; i++)
{
    Console.WriteLine(ports[i]);
}
```

Instantiating a System.IO.SerialPort object

```
using System.IO.Ports;
SerialPort port = new SerialPort();
SerialPort port = new SerialPort("COM 1"); ;
SerialPort port = new SerialPort("COM 1", 9600);
```

NOTE: Those are just three of the seven overloads of the constructor for the SerialPort type.

Reading/Writing data over the SerialPort

The simplest way is to use the `SerialPort.Read` and `SerialPort.Write` methods. However you can also retrieve a `System.IO.Stream` object which you can use to stream data over the SerialPort. To do this, use `SerialPort.BaseStream`.

Reading

```
int length = port.BytesToRead;
//Note that you can swap out a byte-array for a char-array if you prefer.
byte[] buffer = new byte[length];
port.Read(buffer, 0, length);
```

You can also read all data available:

```
string curData = port.ReadExisting();
```

Or simply read to the first newline encountered in the incoming data:

```
string line = port.ReadLine();
```

Writing

The easiest way to write data over the SerialPort is:

```
port.Write("here is some text to be sent over the serial port.");
```

However you can also send data over like this when needed:

```
//Note that you can swap out the byte-array with a char-array if you so choose.
byte[] data = new byte[1] { 255 };
port.Write(data, 0, data.Length);
```

Section 17.3: Reading/Writing Data Using System.IO.File

First, let's see three different ways of extracting data from a file.

```
string fileText = File.ReadAllText(file);
string[] fileLines = File.ReadAllLines(file);
byte[] fileBytes = File.ReadAllBytes(file);
```

- On the first line, we read all the data in the file as a string.
- On the second line, we read the data in the file into a string-array. Each line in the file becomes an element in the array.
- On the third we read the bytes from the file.

Next, let's see three different methods of **appending** data to a file. If the file you specify doesn't exist, each method will automatically create the file before attempting to append the data to it.

```
File.AppendAllText(file, "Here is some data that is\nappended to the file.");
File.AppendAllLines(file, new string[2] { "Here is some data that is", "appended to the file." });
using (StreamWriter stream = File.AppendText(file))
{
    stream.WriteLine("Here is some data that is");
    stream.Write("appended to the file.");
}
```

- On the first line we simply add a string to the end of the specified file.
- On the second line we add each element of the array onto a new line in the file.
- Finally on the third line we use `File.AppendText` to open up a streamwriter which will append whatever data is written to it.

And lastly, let's see three different methods of **writing** data to a file. The difference between *appending* and *writing* being that writing **over-writes** the data in the file while appending **adds** to the data in the file. If the file you specify doesn't exist, each method will automatically create the file before attempting to write the data to it.

```
File.WriteAllText(file, "here is some data\nin this file.");
File.WriteAllLines(file, new string[2] { "here is some data", "in this file" });
File.WriteAllBytes(file, new byte[2] { 0, 255 });
```

- The first line writes a string to the file.
- The second line writes each string in the array on it's own line in the file.

- And the third line allows you to write a byte array to the file.

Chapter 18: System.IO.File class

Parameter	Details
`source`	The file that is to be moved to another location.
`destination`	The directory in which you would like to move `source` to (this variable should also contain the name (and file extension) of the file.

Section 18.1: Delete a file

To delete a file (if you have required permissions) is as simple as:

```
File.Delete(path);
```

However many things may go wrong:

- You do not have required permissions (`UnauthorizedAccessException` is thrown).
- File may be in use by someone else (`IOException` is thrown).
- File cannot be deleted because of low level error or media is read-only (`IOException` is thrown).
- File does not exist anymore (`IOException` is thrown).

Note that last point (file does not exist) is usually *circumvented* with a code snippet like this:

```
if (File.Exists(path))
    File.Delete(path);
```

However it's not an atomic operation and file may be delete by someone else between the call to `File.Exists()` and before `File.Delete()`. Right approach to handle I/O operation requires exception handling (assuming an alternative course of actions may be taken when operation fails):

```
if (File.Exists(path))
{
    try
    {
        File.Delete(path);
    }
    catch (IOException exception)
    {
        if (!File.Exists(path))
            return; // Someone else deleted this file

        // Something went wrong...
    }
    catch (UnauthorizedAccessException exception)
    {
        // I do not have required permissions
    }
}
```

Note that this I/O errors sometimes are transitory (file in use, for example) and if a network connection is involved then it may automatically recover without any action from our side. It's then common to *retry* an I/O operation few times with a small delay between each attempt:

```
public static void Delete(string path)
{
    if (!File.Exists(path))
```

```
        return;

    for (int i=1; ; ++i)
    {
        try
        {
            File.Delete(path);
            return;
        }
        catch (IOException e)
        {
            if (!File.Exists(path))
                return;

            if (i == NumberOfAttempts)
                throw;

            Thread.Sleep(DelayBetweenEachAttempt);
        }

        // You may handle UnauthorizedAccessException but this issue
        // will probably won't be fixed in few seconds...
    }
}

private const int NumberOfAttempts = 3;
private const int DelayBetweenEachAttempt = 1000; // ms
```

Note: in Windows environment file will not be really deleted when you call this function, if someone else open the file using `FileShare.Delete` then file can be deleted but it will effectively happen only when owner will close the file.

Section 18.2: Strip unwanted lines from a text file

To change a text file is not easy because its content must be moved around. For *small* files easiest method is to read its content in memory and then write back modified text.

In this example we read all lines from a file and drop all blank lines then we write back to original path:

```
File.WriteAllLines(path,
    File.ReadAllLines(path).Where(x => !String.IsNullOrWhiteSpace(x)));
```

If file is too big to load it in memory and output path is different from input path:

```
File.WriteAllLines(outputPath,
    File.ReadLines(inputPath).Where(x => !String.IsNullOrWhiteSpace(x)));
```

Section 18.3: Convert text file encoding

Text is saved encoded (see also Strings topic) then sometimes you may need to change its encoding, this example assumes (for simplicity) that file is not too big and it can be entirely read in memory:

```
public static void ConvertEncoding(string path, Encoding from, Encoding to)
{
    File.WriteAllText(path, File.ReadAllText(path, from), to);
}
```

When performing conversions do not forget that file may contain BOM (Byte Order Mark), to better understand how it's managed refer to Encoding.UTF8.GetString doesn't take into account the Preamble/BOM.

Section 18.4: Enumerate files older than a specified amount

This snippet is an helper function to enumerate all files older than a specified age, it's useful - for example - when you have to delete old log files or old cached data.

```
static IEnumerable<string> EnumerateAllFilesOlderThan(
                                TimeSpan maximumAge,
                                string path,
                                string searchPattern = "*.*",
                                SearchOption options = SearchOption.TopDirectoryOnly)
{
    DateTime oldestWriteTime = DateTime.Now - maximumAge;

    return Directory.EnumerateFiles(path, searchPattern, options)
        .Where(x => Directory.GetLastWriteTime(x) < oldestWriteTime);
}
```

Used like this:

```
var oldFiles = EnumerateAllFilesOlderThan(TimeSpan.FromDays(7), @"c:\log", "*.log");
```

Few things to note:

- Search is performed using `Directory.EnumerateFiles()` instead of `Directory.GetFiles()`. Enumeration is *alive* then you won't need to wait until all file system entries have been fetched.
- We're checking for last write time but you may use creation time or last access time (for example to delete *unused* cached files, note that access time may be disabled).
- Granularity isn't uniform for all those properties (write time, access time, creation time), check MSDN for details about this.

Section 18.5: Move a File from one location to another

File.Move
In order to move a file from one location to another, one simple line of code can achieve this:

```
File.Move(@"C:\TemporaryFile.txt", @"C:\TemporaryFiles\TemporaryFile.txt");
```

However, there are many things that could go wrong with this simple operation. For instance, what if the user running your program does not have a Drive that is labelled 'C'? What if they did - but they decided to rename it to 'B', or 'M'?

What if the Source file (the file in which you would like to move) has been moved without your knowing - or what if it simply doesn't exist.

This can be circumvented by first checking to see whether the source file does exist:

```
string source = @"C:\TemporaryFile.txt", destination = @"C:\TemporaryFiles\TemporaryFile.txt";
if(File.Exists("C:\TemporaryFile.txt"))
{
    File.Move(source, destination);
}
```

This will ensure that at that very moment, the file does exist, and can be moved to another location. There may be

times where a simple call to `File.Exists` won't be enough. If it isn't, check again, convey to the user that the operation failed - or handle the exception.

A `FileNotFoundException` is not the only exception you are likely to encounter.

See below for possible exceptions:

Exception Type	Description
`IOException`	The file already exists or the source file could not be found.
`ArgumentNullException`	The value of the Source and/or Destination parameters is null.
`ArgumentException`	The value of the Source and/or Destination parameters are empty, or contain invalid characters.
`UnauthorizedAccessException`	You do not have the required permissions in order to perform this action.
`PathTooLongException`	The Source, Destination or specified path(s) exceed the maximum length. On Windows, a Path's length must be less than 248 characters, while File names must be less than 260 characters.
`DirectoryNotFoundException`	The specified directory could not be found.
`NotSupportedException`	The Source or Destination paths or file names are in an invalid format.

Chapter 19: Reading and writing Zip files

The **ZipFile** class lives in the **System.IO.Compression** namespace. It can be used to read from, and write to Zip files.

Section 19.1: Listing ZIP contents

This snippet will list all the filenames of a zip archive. The filenames are relative to the zip root.

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    for (int i = 0; i < archive.Entries.Count; i++)
    {
        Console.WriteLine($"{i}: {archive.Entries[i]}");
    }
}
```

Section 19.2: Extracting files from ZIP files

Extracting all the files into a directory is very easy:

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    archive.ExtractToDirectory(AppDomain.CurrentDomain.BaseDirectory);
}
```

When the file already exists, a **System.IO.IOException** will be thrown.

Extracting specific files:

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    // Get a root entry file
    archive.GetEntry("test.txt").ExtractToFile("test_extracted_getentries.txt", true);

    // Enter a path if you want to extract files from a subdirectory
    archive.GetEntry("sub/subtest.txt").ExtractToFile("test_sub.txt", true);

    // You can also use the Entries property to find files
    archive.Entries.FirstOrDefault(f => f.Name ==
"test.txt")?.ExtractToFile("test_extracted_linq.txt", true);

    // This will throw a System.ArgumentNullException because the file cannot be found
    archive.GetEntry("nonexistingfile.txt").ExtractToFile("fail.txt", true);
}
```

Any of these methods will produce the same result.

Section 19.3: Updating a ZIP file

To update a ZIP file, the file has to be opened with ZipArchiveMode.Update instead.

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
```

```
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Update))
{
    // Add file to root
    archive.CreateEntryFromFile("test.txt", "test.txt");

    // Add file to subfolder
    archive.CreateEntryFromFile("test.txt", "symbols/test.txt");
}
```

There is also the option to write directly to a file within the archive:

```
var entry = archive.CreateEntry("createentry.txt");
using(var writer = new StreamWriter(entry.Open()))
{
    writer.WriteLine("Test line");
}
```

Chapter 20: Managed Extensibility Framework

Section 20.1: Connecting (Basic)

See the other (Basic) examples above.

```
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;

namespace Demo
{
    public static class Program
    {
        public static void Main()
        {
            using (var catalog = new ApplicationCatalog())
            using (var exportProvider = new CatalogExportProvider(catalog))
            using (var container = new CompositionContainer(exportProvider))
            {
                exportProvider.SourceProvider = container;

                UserWriter writer = new UserWriter();

                // at this point, writer's userProvider field is null
                container.ComposeParts(writer);

                // now, it should be non-null (or an exception will be thrown).
                writer.PrintAllUsers();
            }
        }
    }
}
```

As long as something in the application's assembly search path has `[Export(typeof(IUserProvider))]`, `UserWriter`'s corresponding import will be satisfied and the users will be printed.

Other types of catalogs (e.g., `DirectoryCatalog`) can be used instead of (or in addition to) `ApplicationCatalog`, to look in other places for exports that satisfy the imports.

Section 20.2: Exporting a Type (Basic)

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel.Composition;

namespace Demo
{
    [Export(typeof(IUserProvider))]
    public sealed class UserProvider : IUserProvider
    {
        public ReadOnlyCollection<User> GetAllUsers()
        {
            return new List<User>
            {
                new User(0, "admin"),
                new User(1, "Dennis"),
```

```
                new User(2, "Samantha"),
            }.AsReadOnly();
        }
    }
}
```

This could be defined virtually anywhere; all that matters is that the application knows where to look for it (via the ComposablePartCatalogs it creates).

Section 20.3: Importing (Basic)

```
using System;
using System.ComponentModel.Composition;

namespace Demo
{
    public sealed class UserWriter
    {
        [Import(typeof(IUserProvider))]
        private IUserProvider userProvider;

        public void PrintAllUsers()
        {
            foreach (User user in this.userProvider.GetAllUsers())
            {
                Console.WriteLine(user);
            }
        }
    }
}
```

This is a type that has a dependency on an `IUserProvider`, which could be defined anywhere. Like the previous example, all that matters is that the application knows where to look for the matching export (via the ComposablePartCatalogs it creates).

Chapter 21: SpeechRecognitionEngine class to recognize speech

`LoadGrammar`: Parameters	Details
grammar	The grammar to load. For example, a `DictationGrammar` object to allow free text dictation.
`RecognizeAsync`: Parameters	**Details**
mode	The `RecognizeMode` for the current recognition: `Single` for just one recognition, `Multiple` to allow multiple.
`GrammarBuilder.Append`: Parameters	**Details**
choices	Appends some choices to the grammar builder. This means that, when the user inputs speech, the recognizer can follow different "branches" from a grammar.
`Choices` constructor: Parameters	**Details**
choices	An array of choices for the grammar builder. See `GrammarBuilder.Append`.
`Grammar` constructor: Parameter	**Details**
builder	The `GrammarBuilder` to construct a `Grammar` from.

Section 21.1: Asynchronously recognizing speech based on a restricted set of phrases

```
SpeechRecognitionEngine recognitionEngine = new SpeechRecognitionEngine();
GrammarBuilder builder = new GrammarBuilder();
builder.Append(new Choices("I am", "You are", "He is", "She is", "We are", "They are"));
builder.Append(new Choices("friendly", "unfriendly"));
recognitionEngine.LoadGrammar(new Grammar(builder));
recognitionEngine.SpeechRecognized += delegate(object sender, SpeechRecognizedEventArgs e)
{
    Console.WriteLine("You said: {0}", e.Result.Text);
};
recognitionEngine.SetInputToDefaultAudioDevice();
recognitionEngine.RecognizeAsync(RecognizeMode.Multiple);
```

Section 21.2: Asynchronously recognizing speech for free text dictation

```
using System.Speech.Recognition;

// ...

SpeechRecognitionEngine recognitionEngine = new SpeechRecognitionEngine();
recognitionEngine.LoadGrammar(new DictationGrammar());
recognitionEngine.SpeechRecognized += delegate(object sender, SpeechRecognizedEventArgs e)
{
    Console.WriteLine("You said: {0}", e.Result.Text);
};
recognitionEngine.SetInputToDefaultAudioDevice();
recognitionEngine.RecognizeAsync(RecognizeMode.Multiple);
```

Chapter 22: System.Runtime.Caching.MemoryCache (ObjectCache)

Section 22.1: Adding Item to Cache (Set)

Set function inserts a cache entry into the cache by using a CacheItem instance to supply the key and value for the cache entry.

This function Overrides `ObjectCache.Set(CacheItem, CacheItemPolicy)`

```
private static bool SetToCache()
{
    string key = "Cache_Key";
    string value = "Cache_Value";

    //Get a reference to the default MemoryCache instance.
    var cacheContainer = MemoryCache.Default;

    var policy = new CacheItemPolicy()
    {
        AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(DEFAULT_CACHE_EXPIRATION_MINUTES)
     };
     var itemToCache = new CacheItem(key, value); //Value is of type object.
     cacheContainer.Set(itemToCache, policy);
}
```

Section 22.2: System.Runtime.Caching.MemoryCache (ObjectCache)

This function gets existing item form cache, and if the item don't exist in cache, it will fetch item based on the valueFetchFactory function.

```
    public static TValue GetExistingOrAdd<TValue>(string key, double minutesForExpiration,
Func<TValue> valueFetchFactory)
    {
        try
        {
            //The Lazy class provides Lazy initialization which will evaluate
            //the valueFetchFactory only if item is not in the cache.
            var newValue = new Lazy<TValue>(valueFetchFactory);

            //Setup the cache policy if item will be saved back to cache.
            CacheItemPolicy policy = new CacheItemPolicy()
            {
                AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(minutesForExpiration)
            };

            //returns existing item form cache or add the new value if it does not exist.
            var cachedItem = _cacheContainer.AddOrGetExisting(key, newValue, policy) as
Lazy<TValue>;

            return (cachedItem ?? newValue).Value;
        }
        catch (Exception excep)
        {
```

```
            return default(TValue);
        }
    }
```

Chapter 23: System.Reflection.Emit namespace

Section 23.1: Creating an assembly dynamically

```
using System;
using System.Reflection;
using System.Reflection.Emit;

class DemoAssemblyBuilder
{
    public static void Main()
    {
        // An assembly consists of one or more modules, each of which
        // contains zero or more types. This code creates a single-module
        // assembly, the most common case. The module contains one type,
        // named "MyDynamicType", that has a private field, a property
        // that gets and sets the private field, constructors that
        // initialize the private field, and a method that multiplies
        // a user-supplied number by the private field value and returns
        // the result. In C# the type might look like this:
        /*
        public class MyDynamicType
        {
            private int m_number;

            public MyDynamicType() : this(42) {}
            public MyDynamicType(int initNumber)
            {
                m_number = initNumber;
            }

            public int Number
            {
                get { return m_number; }
                set { m_number = value; }
            }

            public int MyMethod(int multiplier)
            {
                return m_number * multiplier;
            }
        }
        */

        AssemblyName aName = new AssemblyName("DynamicAssemblyExample");
        AssemblyBuilder ab =
            AppDomain.CurrentDomain.DefineDynamicAssembly(
                aName,
                AssemblyBuilderAccess.RunAndSave);

        // For a single-module assembly, the module name is usually
        // the assembly name plus an extension.
        ModuleBuilder mb =
            ab.DefineDynamicModule(aName.Name, aName.Name + ".dll");

        TypeBuilder tb = mb.DefineType(
            "MyDynamicType",
             TypeAttributes.Public);
```

```
        // Add a private field of type int (Int32).
        FieldBuilder fbNumber = tb.DefineField(
            "m_number",
            typeof(int),
            FieldAttributes.Private);

        // Next, we make a simple sealed method.
        MethodBuilder mbMyMethod = tb.DefineMethod(
            "MyMethod",
            MethodAttributes.Public,
            typeof(int),
            new[] { typeof(int) });

        ILGenerator il = mbMyMethod.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this - always the first argument of any instance method
        il.Emit(OpCodes.Ldfld, fbNumber);
        il.Emit(OpCodes.Ldarg_1); // Load the integer argument
        il.Emit(OpCodes.Mul); // Multiply the two numbers with no overflow checking
        il.Emit(OpCodes.Ret); // Return

        // Next, we build the property. This involves building the property itself, as well as the
        // getter and setter methods.
        PropertyBuilder pbNumber = tb.DefineProperty(
            "Number", // Name
            PropertyAttributes.None,
            typeof(int), // Type of the property
            new Type[0]); // Types of indices, if any

        MethodBuilder mbSetNumber = tb.DefineMethod(
            "set_Number", // Name - setters are set_Property by convention
            // Setter is a special method and we don't want it to appear to callers from C#
            MethodAttributes.PrivateScope | MethodAttributes.HideBySig | MethodAttributes.Public |
MethodAttributes.SpecialName,
            typeof(void), // Setters don't return a value
            new[] { typeof(int) }); // We have a single argument of type System.Int32

        // To generate the body of the method, we'll need an IL generator
        il = mbSetNumber.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this
        il.Emit(OpCodes.Ldarg_1); // Load the new value
        il.Emit(OpCodes.Stfld, fbNumber); // Save the new value to this.m_number
        il.Emit(OpCodes.Ret); // Return

        // Finally, link the method to the setter of our property
        pbNumber.SetSetMethod(mbSetNumber);

        MethodBuilder mbGetNumber = tb.DefineMethod(
            "get_Number",
            MethodAttributes.PrivateScope | MethodAttributes.HideBySig | MethodAttributes.Public |
MethodAttributes.SpecialName,
            typeof(int),
            new Type[0]);

        il = mbGetNumber.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this
        il.Emit(OpCodes.Ldfld, fbNumber); // Load the value of this.m_number
        il.Emit(OpCodes.Ret); // Return the value

        pbNumber.SetGetMethod(mbGetNumber);

        // Finally, we add the two constructors.
        // Constructor needs to call the constructor of the parent class, or another constructor in
```

```
the same class
        ConstructorBuilder intConstructor = tb.DefineConstructor(
            MethodAttributes.Public, CallingConventions.Standard | CallingConventions.HasThis,
new[] { typeof(int) });
        il = intConstructor.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Call, typeof(object).GetConstructor(new Type[0])); // call parent's
constructor
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Ldarg_1); // our int argument
        il.Emit(OpCodes.Stfld, fbNumber); // store argument in this.m_number
        il.Emit(OpCodes.Ret);

        var parameterlessConstructor = tb.DefineConstructor(
            MethodAttributes.Public, CallingConventions.Standard | CallingConventions.HasThis, new
Type[0]);
        il = parameterlessConstructor.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Ldc_I4_S, (byte)42); // load 42 as an integer constant
        il.Emit(OpCodes.Call, intConstructor); // call this(42)
        il.Emit(OpCodes.Ret);

        // And make sure the type is created
        Type ourType = tb.CreateType();

        // The types from the assembly can be used directly using reflection, or we can save the
assembly to use as a reference
        object ourInstance = Activator.CreateInstance(ourType);
        Console.WriteLine(ourType.GetProperty("Number").GetValue(ourInstance)); // 42

        // Save the assembly for use elsewhere. This is very useful for debugging - you can use e.g.
ILSpy to look at the equivalent IL/C# code.
        ab.Save(@"DynamicAssemblyExample.dll");
        // Using newly created type
        var myDynamicType = tb.CreateType();
        var myDynamicTypeInstance = Activator.CreateInstance(myDynamicType);

        Console.WriteLine(myDynamicTypeInstance.GetType()); // MyDynamicType

        var numberField = myDynamicType.GetField("m_number", BindingFlags.NonPublic |
BindingFlags.Instance);
        numberField.SetValue (myDynamicTypeInstance, 10);

        Console.WriteLine(numberField.GetValue(myDynamicTypeInstance)); // 10
    }
}
```

Chapter 24: .NET Core

.NET Core is a general purpose development platform maintained by Microsoft and the .NET community on GitHub. It is cross-platform, supporting Windows, macOS and Linux, and can be used in device, cloud, and embedded/IoT scenarios.

When you think of .NET Core the following should come to mind (flexible deployment, cross-platform, command-line tools, open source).

Another great thing is that even if it's open source Microsoft is actively supporting it.

Section 24.1: Basic Console App

```
public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("\nWhat is your name? ");
        var name = Console.ReadLine();
        var date = DateTime.Now;
        Console.WriteLine("\nHello, {0}, on {1:d} at {1:t}", name, date);
        Console.Write("\nPress any key to exit...");
        Console.ReadKey(true);
    }
}
```

Chapter 25: ADO.NET

ADO(ActiveX Data Objects).Net is a tool provided by Microsoft which provides access to data sources such as SQL Server, Oracle, and XML through its components. .Net front-end applications can retrieve, create, and manipulate data, once they are connected to a data source through ADO.Net with appropriate privileges.

ADO.Net provides a connection-less architecture. It is a secure approach to interact with a database, since, the connection doesn't have to be maintained during the entire session.

Section 25.1: Best Practices - Executing Sql Statements

```
public void SaveNewEmployee(Employee newEmployee)
{
    // best practice - wrap all database connections in a using block so they are always closed &
disposed even in the event of an Exception
    // best practice - retrieve the connection string by name from the app.config or web.config
(depending on the application type) (note, this requires an assembly reference to
System.configuration)
    using(SqlConnection con = new
SqlConnection(System.Configuration.ConfigurationManager.ConnectionStrings["MyConnectionName"].Conne
ctionString))
    {
        // best practice - use column names in your INSERT statement so you are not dependent on the
sql schema column order
        // best practice - always use parameters to avoid sql injection attacks and errors if
malformed text is used like including a single quote which is the sql equivalent of escaping or
starting a string (varchar/nvarchar)
        // best practice - give your parameters meaningful names just like you do variables in your
code
        using(SqlCommand sc = new SqlCommand("INSERT INTO employee (FirstName, LastName,
DateOfBirth /*etc*/) VALUES (@firstName, @lastName, @dateOfBirth /*etc*/)", con))
        {
            // best practice - always specify the database data type of the column you are using
            // best practice - check for valid values in your code and/or use a database constraint,
if inserting NULL then use System.DbNull.Value
            sc.Parameters.Add(new SqlParameter("@firstName", SqlDbType.VarChar, 200){Value =
newEmployee.FirstName ?? (object) System.DBNull.Value});
            sc.Parameters.Add(new SqlParameter("@lastName", SqlDbType.VarChar, 200){Value =
newEmployee.LastName ?? (object) System.DBNull.Value});

            // best practice - always use the correct types when specifying your parameters, Value
is assigned to a DateTime instance and not a string representation of a Date
            sc.Parameters.Add(new SqlParameter("@dateOfBirth", SqlDbType.Date){ Value =
newEmployee.DateOfBirth });

            // best practice - open your connection as late as possible unless you need to verify
that the database connection is valid and won't fail and the proceeding code execution takes a long
time (not the case here)
            con.Open();
            sc.ExecuteNonQuery();
        }

        // the end of the using block will close and dispose the SqlConnection
        // best practice - end the using block as soon as possible to release the database connection
    }
}

// supporting class used as parameter for example
public class Employee
```

```
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
}
```

Best practice for working with ADO.NET

- Rule of thumb is to open connection for minimal time. Close the connection explicitly once your procedure execution is over this will return the connection object back to connection pool. Default connection pool max size = 100. As connection pooling enhances the performance of physical connection to SQL Server.Connection Pooling in SQL Server
- Wrap all database connections in a using block so they are always closed & disposed even in the event of an Exception. See using Statement (C# Reference) for more information on using statements
- Retrieve the connection strings by name from the app.config or web.config (depending on the application type)
    - This requires an assembly reference to `System.configuration`
    - See Connection Strings and Configuration Files for additional information on how to structure your configuration file
- Always use parameters for incoming values to
    - Avoid sql injection attacks
    - Avoid errors if malformed text is used like including a single quote which is the sql equivalent of escaping or starting a string (varchar/nvarchar)
    - Letting the database provider reuse query plans (not supported by all database providers) which increases efficiency
- When working with parameters
    - Sql parameters type and size mismatch is a common cause of insert/ updated/ select failure
    - Give your Sql parameters meaningful names just like you do variables in your code
    - Specify the database data type of the column you are using, this ensures the wrong parameter types is not used which could lead to unexpected results
    - Validate your incoming parameters before you pass them into the command (as the saying goes, "garbage in, garbage out"). Validate incoming values as early as possible in the stack
    - Use the correct types when assigning your parameter values, example: do not assign the string value of a DateTime, instead assign an actual DateTime instance to the value of the parameter
    - Specify the size of string-type parameters. This is because SQL Server can re-use execution plans if the parameters match in type *and* size. Use -1 for MAX
    - Do not use the method AddWithValue, the main reason is it is very easy to forget to specify the parameter type or the precision/scale when needed. For additional information see Can we stop using AddWithValue already?
- When using database connections
    - Open the connection as late as possible and close it as soon as possible. This is a general guideline when working with any external resource
    - Never share database connection instances (example: having a singleton host a shared instance of type `SqlConnection`). Have your code always create a new database connection instance when needed and then have the calling code dispose of it and "throw it away" when it is done. The reason for this is
        - Most database providers have some sort of connection pooling so creating new managed connections is cheap
        - It eliminates any future errors if the code starts working with multiple threads

Section 25.2: Executing SQL statements as a command

```
// Uses Windows authentication. Replace the Trusted_Connection parameter with
```

```
// User Id=...;Password=...; to use SQL Server authentication instead. You may
// want to find the appropriate connection string for your server.
string connectionString =
@"Server=myServer\myInstance;Database=myDataBase;Trusted_Connection=True;"

string sql = "INSERT INTO myTable (myDateTimeField, myIntField) " +
    "VALUES (@someDateTime, @someInt);";

// Most ADO.NET objects are disposable and, thus, require the using keyword.
using (var connection = new SqlConnection(connectionString))
using (var command = new SqlCommand(sql, connection))
{
    // Use parameters instead of string concatenation to add user-supplied
    // values to avoid SQL injection and formatting issues. Explicitly supply datatype.

    // System.Data.SqlDbType is an enumeration. See Note1
    command.Parameters.Add("@someDateTime", SqlDbType.DateTime).Value = myDateTimeVariable;
    command.Parameters.Add("@someInt", SqlDbType.Int).Value = myInt32Variable;

    // Execute the SQL statement. Use ExecuteScalar and ExecuteReader instead
    // for query that return results (or see the more specific examples, once
    // those have been added).

    connection.Open();
    command.ExecuteNonQuery();
}
```

Note 1: Please see SqlDbType Enumeration for the MSFT SQL Server-specific variation.

Note 2: Please see MySqlDbType Enumeration for the MySQL-specific variation.

Section 25.3: Using common interfaces to abstract away vendor specific classes

```
var providerName = "System.Data.SqlClient";    //Oracle.ManagedDataAccess.Client, IBM.Data.DB2
var connectionString = "{your-connection-string}";
//you will probably get the above two values in the ConnectionStringSettings object from .config file

var factory = DbProviderFactories.GetFactory(providerName);
using(var connection = factory.CreateConnection()) {    //IDbConnection
    connection.ConnectionString = connectionString;
    connection.Open();

    using(var command = connection.CreateCommand()) {    //IDbCommand
        command.CommandText = "{query}";

        using(var reader = command.ExecuteReader()) {    //IDataReader
            while(reader.Read()) {
                ...
            }
        }
    }
}
```

Chapter 26: Dependency Injection

Section 26.1: How Dependency Injection Makes Unit Testing Easier

This builds on the previous example of the `Greeter` class which has two dependencies, `IGreetingProvider` and `IGreetingWriter`.

The actual implementation of `IGreetingProvider` might retrieve a string from an API call or a database. The implementation of `IGreetingWriter` might display the greeting in the console. But because `Greeter` has its dependencies injected into its constructor, it's easy to write a unit test that injects mocked versions of those interfaces. In real life we might use a framework like Moq, but in this case I'll write those mocked implementations.

```
public class TestGreetingProvider : IGreetingProvider
{
    public const string TestGreeting = "Hello!";

    public string GetGreeting()
    {
        return TestGreeting;
    }
}

public class TestGreetingWriter : List<string>, IGreetingWriter
{
    public void WriteGreeting(string greeting)
    {
        Add(greeting);
    }
}

[TestClass]
public class GreeterTests
{
    [TestMethod]
    public void Greeter_WritesGreeting()
    {
        var greetingProvider = new TestGreetingProvider();
        var greetingWriter = new TestGreetingWriter();
        var greeter = new Greeter(greetingProvider, greetingWriter);
        greeter.Greet();
        Assert.AreEqual(greetingWriter[0], TestGreetingProvider.TestGreeting);
    }
}
```

The behavior of `IGreetingProvider` and `IGreetingWriter` are not relevant to this test. We want to test that `Greeter` gets a greeting and writes it. The design of `Greeter` (using dependency injection) allows us to inject mocked dependencies without any complicated moving parts. All we're testing is that `Greeter` interacts with those dependencies as we expect it to.

Section 26.2: Dependency Injection - Simple example

This class is called `Greeter`. Its responsibility is to output a greeting. It has two *dependencies*. It needs something that will give it the greeting to output, and then it needs a way to output that greeting. Those dependencies are both described as interfaces, `IGreetingProvider` and `IGreetingWriter`. In this example, those two dependencies are "injected" into `Greeter`. (Further explanation following the example.)

```
public class Greeter
{
    private readonly IGreetingProvider _greetingProvider;
    private readonly IGreetingWriter _greetingWriter;

    public Greeter(IGreetingProvider greetingProvider, IGreetingWriter greetingWriter)
    {
        _greetingProvider = greetingProvider;
        _greetingWriter = greetingWriter;
    }

    public void Greet()
    {
        var greeting = _greetingProvider.GetGreeting();
        _greetingWriter.WriteGreeting(greeting);
    }
}

public interface IGreetingProvider
{
    string GetGreeting();
}

public interface IGreetingWriter
{
    void WriteGreeting(string greeting);
}
```

The `Greeting` class depends on both `IGreetingProvider` and `IGreetingWriter`, but it is not responsible for creating instances of either. Instead it requires them in its constructor. Whatever creates an instance of `Greeting` must provide those two dependencies. We can call that "injecting" the dependencies.

Because dependencies are provided to the class in its constructor, this is also called "constructor injection."

A few common conventions:

- The constructor saves the dependencies as `private` fields. As soon as the class is instantiated, those dependencies are available to all other non-static methods of the class.
- The `private` fields are `readonly`. Once they are set in the constructor they cannot be changed. This indicates that those fields should not (and cannot) be modified outside of the constructor. That further ensures that those dependencies will be available for the lifetime of the class.
- The dependencies are interfaces. This is not strictly necessary, but is common because it makes it easier to substitute one implementation of the dependency with another. It also allows providing a mocked version of the interface for unit testing purposes.

Section 26.3: Why We Use Dependency Injection Containers (IoC Containers)

Dependency injection means writing classes so that they do not control their dependencies - instead, their dependencies are provided to them ("injected.")

This is not the same thing as using a dependency injection framework (often called a "DI container", "IoC container", or just "container") like Castle Windsor, Autofac, SimpleInjector, Ninject, Unity, or others.

A container just makes dependency injection easier. For example, suppose you write a number of classes that rely on dependency injection. One class depends on several interfaces, the classes that implement those interfaces depend on other interfaces, and so on. Some depend on specific values. And just for fun, some of those classes

implement `IDisposable` and need to be disposed.

Each individual class is well-written and easy to test. But now there's a different problem: Creating an instance of a class has become much more complicated. Suppose we're creating an instance of a `CustomerService` class. It has dependencies and its dependencies have dependencies. Constructing an instance might look something like this:

```
public CustomerData GetCustomerData(string customerNumber)
{
    var customerApiEndpoint = ConfigurationManager.AppSettings["customerApi:customerApiEndpoint"];
    var logFilePath = ConfigurationManager.AppSettings["logwriter:logFilePath"];
    var authConnectionString =
ConfigurationManager.ConnectionStrings["authorization"].ConnectionString;
    using(var logWriter = new LogWriter(logFilePath ))
    {
        using(var customerApiClient = new CustomerApiClient(customerApiEndpoint))
        {
            var customerService = new CustomerService(
                new SqlAuthorizationRepository(authorizationConnectionString, logWriter),
                new CustomerDataRepository(customerApiClient, logWriter),
                logWriter
            );

            // All this just to create an instance of CustomerService!
            return customerService.GetCustomerData(string customerNumber);
        }
    }
}
```

You might wonder, why not put the whole giant construction in a separate function that just returns `CustomerService`? One reason is that because the dependencies for each class are injected into it, a class isn't responsible for knowing whether those dependencies are `IDisposable` or disposing them. It just uses them. So if a we had a `GetCustomerService()` function that returned a fully-constructed `CustomerService`, that class might contain a number of disposable resources and no way to access or dispose them.

And aside from disposing `IDisposable`, who wants to call a series of nested constructors like that, ever? That's a short example. It could get much, much worse. Again, that doesn't mean that we wrote the classes the wrong way. The classes might be individually perfect. The challenge is composing them together.

A dependency injection container simplifies that. It allows us to specify which class or value should be used to fulfill each dependency. This slightly oversimplified example uses Castle Windsor:

```
var container = new WindsorContainer()
container.Register(
    Component.For<CustomerService>(),
    Component.For<ILogWriter, LogWriter>()
        .DependsOn(Dependency.OnAppSettingsValue("logFilePath", "logWriter:logFilePath")),
    Component.For<IAuthorizationRepository, SqlAuthorizationRepository>()
        .DependsOn(Dependency.OnValue(connectionString,
ConfigurationManager.ConnectionStrings["authorization"].ConnectionString)),
    Component.For<ICustomerDataProvider, CustomerApiClient>()
         .DependsOn(Dependency.OnAppSettingsValue("apiEndpoint",
"customerApi:customerApiEndpoint"))
);
```

We call this "registering dependencies" or "configuring the container." Translated, this tells our `WindsorContainer`:

- If a class requires `ILogWriter`, create an instance of `LogWriter`. `LogWriter` requires a file path. Use this value from `AppSettings`.

- If a class requires `IAuthorizationRepository`, create an instance of `SqlAuthorizationRepository`. It requires a connection string. Use this value from the `ConnectionStrings` section.
- If a class requires `ICustomerDataProvider`, create a `CustomerApiClient` and provide the string it needs from `AppSettings`.

When we request a dependency from the container we call that "resolving" a dependency. It's bad practice to do that directly using the container, but that's a different story. For demonstration purposes, we could now do this:

```
var customerService = container.Resolve<CustomerService>();
var data = customerService.GetCustomerData(customerNumber);
container.Release(customerService);
```

The container knows that `CustomerService` depends on `IAuthorizationRepository` and `ICustomerDataProvider`. It knows what classes it needs to create to fulfill those requirements. Those classes, in turn, have more dependencies, and the container knows how to fulfill those. It will create every class it needs to until it can return an instance of `CustomerService`.

If it gets to a point where a class requires a dependency that we haven't registered, like `IDoesSomethingElse`, then when we try to resolve `CustomerService` it will throw a clear exception telling us that we haven't registered anything to fulfill that requirement.

Each DI framework behaves a little differently, but typically they give us some control over how certain classes are instantiated. For example, do we want it to create one instance of `LogWriter` and provide it to every class that depends on `ILogWriter`, or do we want it to create a new one every time? Most containers have a way to specify that.

What about classes that implement `IDisposable`? That's why we call `container.Release(customerService);` at the end. Most containers (including Windsor) will step back through all of the dependencies created and `Dispose` the ones that need disposing. If `CustomerService` is `IDisposable` it will dispose that too.

Registering dependencies as seen above might just look like more code to write. But when we have lots of classes with lots of dependencies then it really pays off. And if we had to write those same classes *without* using dependency injection then that same application with lots of classes would become difficult to maintain and test.

This scratches the surface of *why* we use dependency injection containers. *How* we configure our application to use one (and use it correctly) is not just one topic - it's a number of topics, as the instructions and examples vary from one container to the next.

Chapter 27: Platform Invoke

Section 27.1: Marshaling structs

Simple struct

C++ signature:

```
typedef struct _PERSON
{
    int age;
    char name[32];
} PERSON, *LP_PERSON;

void GetSpouse(PERSON person, LP_PERSON spouse);
```

C# definition

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct PERSON
{
    public int age;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
    public string name;
}

[DllImport("family.dll", CharSet = CharSet.Auto)]
public static extern bool GetSpouse(PERSON person, ref PERSON spouse);
```

Struct with unknown size array fields. Passing in

C++ signature

```
typedef struct
{
    int length;
    int *data;
} VECTOR;

void SetVector(VECTOR &vector);
```

When passed from managed to unmanaged code, this

The `data` array should be defined as IntPtr and memory should be explicitly allocated with `Marshal.AllocHGlobal()` (and freed with `Marshal.FreeHGlobal()` afterwords):

```
[StructLayout(LayoutKind.Sequential)]
public struct VECTOR : IDisposable
{
    int length;
    IntPtr dataBuf;

    public int[] data
    {
        set
        {
            FreeDataBuf();
```

```
            if (value != null && value.Length > 0)
            {
                dataBuf = Marshal.AllocHGlobal(value.Length * Marshal.SizeOf(value[0]));
                Marshal.Copy(value, 0, dataBuf, value.Length);
                length = value.Length;
            }
        }
    }
    void FreeDataBuf()
    {
        if (dataBuf != IntPtr.Zero)
        {
            Marshal.FreeHGlobal(dataBuf);
            dataBuf = IntPtr.Zero;
        }
    }
    public void Dispose()
    {
        FreeDataBuf();
    }
}

[DllImport("vectors.dll")]
public static extern void SetVector([In]ref VECTOR vector);
```

Struct with unknown size array fields. Receiving

C++ signature:

```
typedef struct
{
    char *name;
} USER;

bool GetCurrentUser(USER *user);
```

When such data is passed out of unmanaged code and memory is allocated by the unmanaged functions, the managed caller should receive it into an `IntPrt` variable and convert the buffer to a managed array. In case of strings there is a convenient `Marshal.PtrToStringAnsi()` method:

```
[StructLayout(LayoutKind.Sequential)]
public struct USER
{
    IntPtr nameBuffer;
    public string name { get { return Marshal.PtrToStringAnsi(nameBuffer); } }
}

[DllImport("users.dll")]
public static extern bool GetCurrentUser(out USER user);
```

Section 27.2: Marshaling unions

Value-type fields only

C++ declaration

```
typedef union
{
    char c;
```

```
    int i;
} CharOrInt;
```

C# declaration

```
[StructLayout(LayoutKind.Explicit)]
public struct CharOrInt
{
    [FieldOffset(0)]
    public byte c;
    [FieldOffset(0)]
    public int i;
}
```

Mixing value-type and reference fields

Overlapping a reference value with a value type one is not allowed so you cannot simply use the `FieldOffset(0) text; FieldOffset(0) i;` will not compile for

```
typedef union
{
    char text[128];
    int i;
} TextOrInt;
```

and generally you would have to employ custom marshaling. However, in particular cases like this simpler technics may be used:

```
[StructLayout(LayoutKind.Sequential)]
public struct TextOrInt
{
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 128)]
    public byte[] text;
    public int i { get { return BitConverter.ToInt32(text, 0); } }
}
```

Section 27.3: Calling a Win32 dll function

```
using System.Runtime.InteropServices;

class PInvokeExample
{
    [DllImport("user32.dll", CharSet = CharSet.Auto)]
    public static extern uint MessageBox(IntPtr hWnd, String text, String caption, int options);

    public static void test()
    {
        MessageBox(IntPtr.Zero, "Hello!", "Message", 0);
    }
}
```

Declare a function as `static extern` stting `DllImportAttribute` with its `Value` property set to .dll name. Don't forget to use `System.Runtime.InteropServices` namespace. Then call it as an regular static method.

The Platform Invocation Services will take care of loading the .dll and finding the desired finction. The P/Invoke in most simple cases will also marshal parameters and return value to and from the .dll (i.e. convert from .NET datatypes to Win32 ones and vice versa).

Section 27.4: Using Windows API

Use pinvoke.net.

Before declaring an `extern` Windows API function in your code, consider looking for it on pinvoke.net. They most likely already have a suitable declaration with all supporting types and good examples.

Section 27.5: Marshalling arrays

Arrays of simple type

```
[DllImport("Example.dll")]
static extern void SetArray(
    [MarshalAs(UnmanagedType.LPArray, SizeConst = 128)]
    byte[] data);
```

Arrays of string

```
[DllImport("Example.dll")]
static extern void SetStrArray(string[] textLines);
```

Chapter 28: NuGet packaging system

Section 28.1: Uninstalling a package from one project in a solution

```
PM> Uninstall-Package -ProjectName MyProjectB EntityFramework
```

Section 28.2: Installing a specific version of a package

```
PM> Install-Package EntityFramework -Version 6.1.2
```

Section 28.3: Adding a package source feed (MyGet, Klondike, ect)

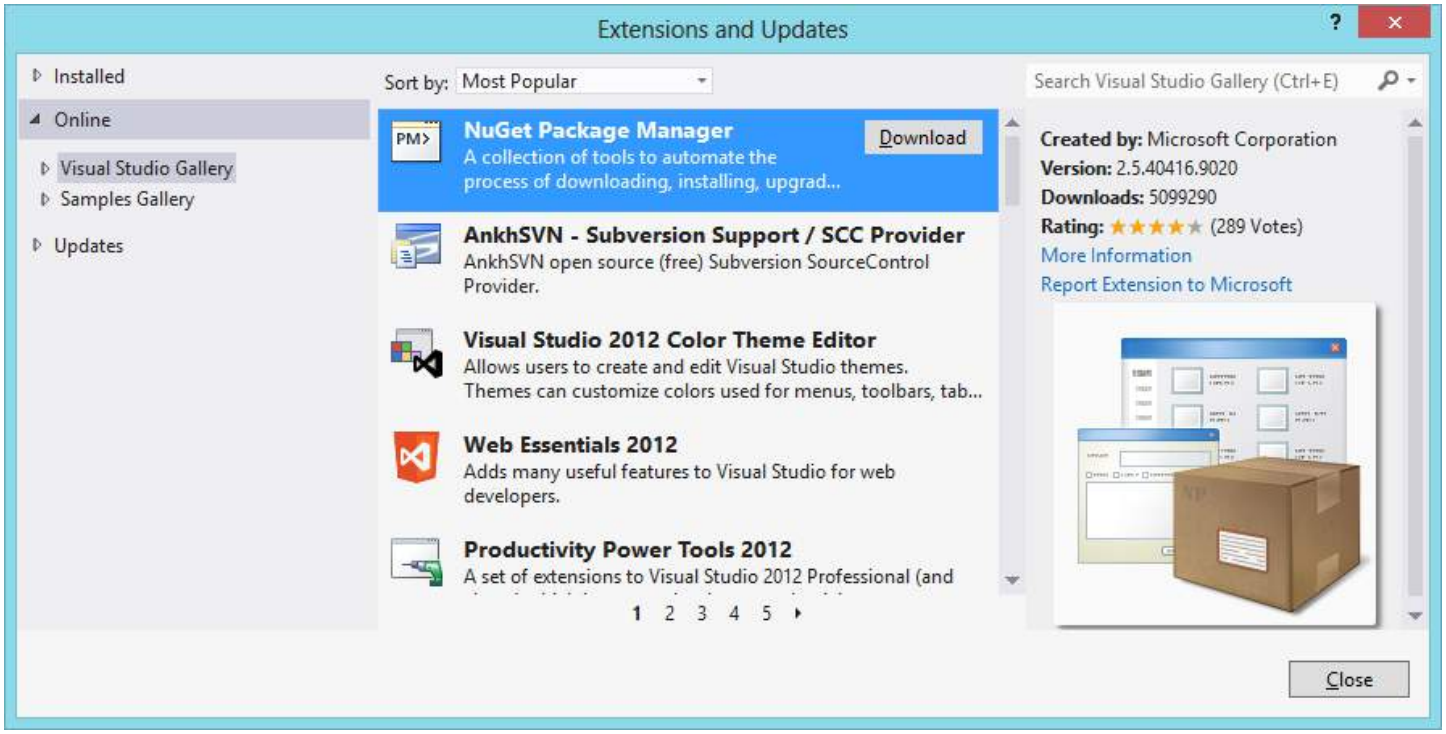
```
nuget sources add -name feedname -source http://sourcefeedurl
```

Section 28.4: Installing the NuGet Package Manager

In order to be able to manage your projects' packages, you need the NuGet Package Manager. This is a Visual Studio Extension, explained in the official docs: Installing and Updating NuGet Client.

Starting with Visual Studio 2012, NuGet is included in every edition, and can be used from: Tools -> NuGet Package Manager -> Package Manager Console.

You do so through the Tools menu of Visual Studio, clicking Extensions and Updates:



This installs both the GUI:

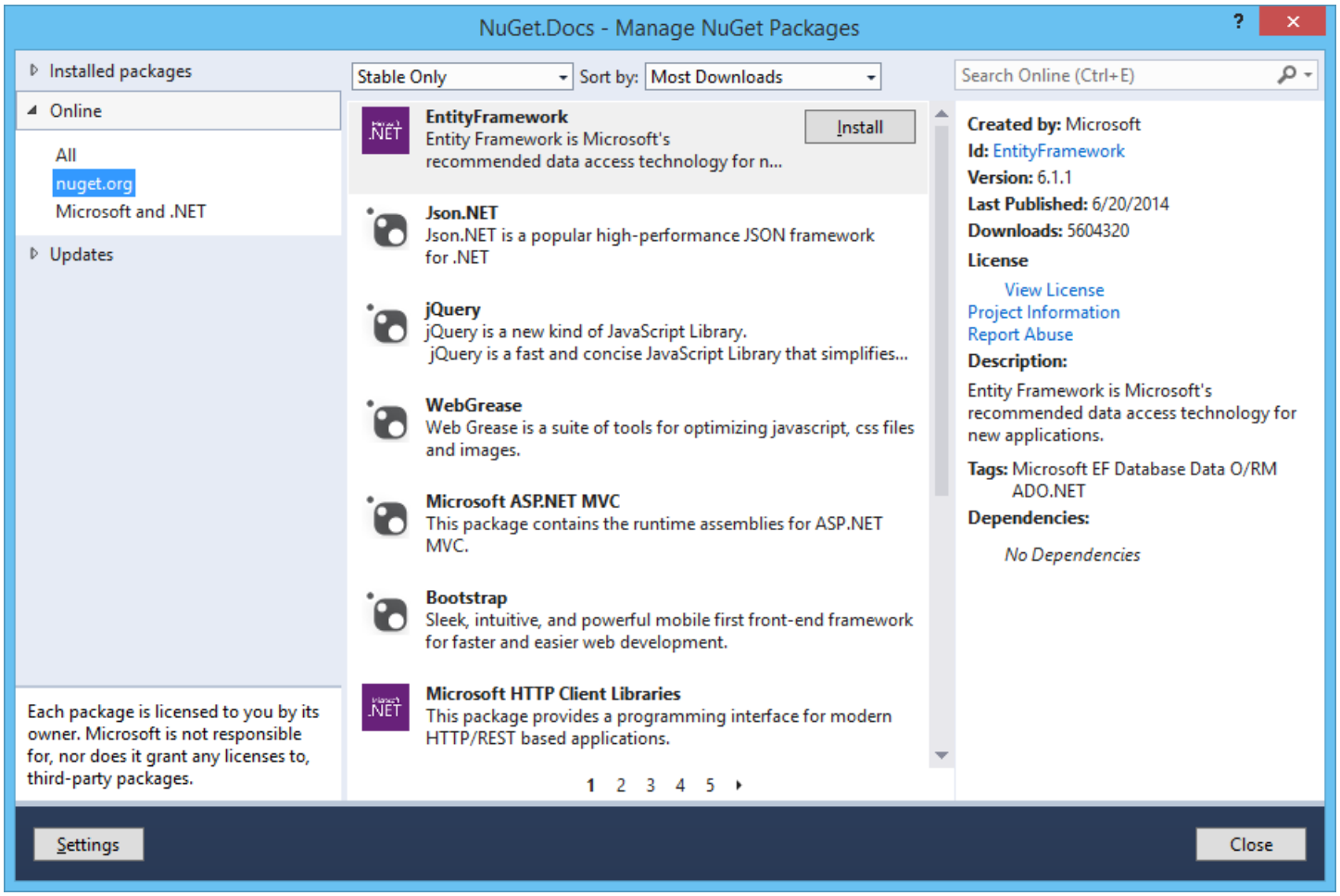
- Available through clicking "Manage NuGet Packages..." on a project or its References folder

And the Package Manager Console:

- Tools -> NuGet Package Manager -> Package Manager Console.

Section 28.5: Managing Packages through the UI

When you right-click a project (or its References folder), you can click the "Manage NuGet Packages..." option. This shows the Package Manager Dialog.



Section 28.6: Managing Packages through the console

Click the menus Tools -> NuGet Package Manager -> Package Manager Console to show the console in your IDE. Official documentation here.

Here you can issue, amongst others, `install-package` commands which installs the entered package into the currently selected "Default project":

```
Install-Package Elmah
```

You can also provide the project to install the package to, overriding the selected project in the "Default project" dropdown:

```
Install-Package Elmah -ProjectName MyFirstWebsite
```

Section 28.7: Updating a package

To update a package use the following command:

```
PM> Update-Package EntityFramework
```

where EntityFramework is the name of the package to be updated. Note that update will run for all projects, and so is different from `Install-Package EntityFramework` which would install to "Default project" only.

You can also specify a single project explicitly:

```
PM> Update-Package EntityFramework -ProjectName MyFirstWebsite
```

Section 28.8: Uninstalling a package

```
PM> Uninstall-Package EntityFramework
```

Section 28.9: Uninstall a specific version of package

```
PM> uninstall-Package EntityFramework -Version 6.1.2
```

Chapter 29: Globalization in ASP.NET MVC using Smart internationalization for ASP.NET

Section 29.1: Basic configuration and setup

1. Add the I18N nuget package to your MVC project.
2. In web.config, add the `i18n.LocalizingModule` to your **`<httpModules>`** or **`<modules>`** section.

```
<!-- IIS 6 -->
<httpModules>
  <add name="i18n.LocalizingModule" type="i18n.LocalizingModule, i18n" />
</httpModules>

<!-- IIS 7 -->
<system.webServer>
  <modules>
    <add name="i18n.LocalizingModule" type="i18n.LocalizingModule, i18n" />
  </modules>
</system.webServer>
```

3. Add a folder named "locale" to the root of your site. Create a subfolder for each culture you wish to support. For example, `/locale/fr/`.
4. In each culture-specific folder, create a text file named `messages.po`.
5. For testing purposes, enter the following lines of text in your `messages.po` file:

```
#: Translation test
msgid "Hello, world!"
msgstr "Bonjour le monde!"
```

6. Add a controller to your project which returns some text to translate.

```
using System.Web.Mvc;

namespace I18nDemo.Controllers
{
    public class DefaultController : Controller
    {
        public ActionResult Index()
        {
            // Text inside [[[triple brackets]]] must precisely match
            // the msgid in your .po file.
            return Content("[[[Hello, world!]]]");
        }
    }
}
```

7. Run your MVC application and browse to the route corresponding to your controller action, such as http://localhost:[yourportnumber]/default.
   Observe that the URL is changed to reflect your default culture, such as http://localhost:[yourportnumber]/en/default.
8. Replace `/en/` in the URL with `/fr/` (or whatever culture you've selected.) The page should now display the translated version of your text.
9. Change your browser's language setting to prefer your alternate culture and browse to `/default` again. Observe that the URL is changed to reflect your alternate culture and the translated text appears.

10. In web.config, add handlers so that users cannot browse to your `locale` folder.

```
<!-- IIS 6 -->
<system.web>
  <httpHandlers>
    <add path="*" verb="*" type="System.Web.HttpNotFoundHandler"/>
  </httpHandlers>
</system.web>

<!-- IIS 7 -->
<system.webServer>
  <handlers>
    <remove name="BlockViewHandler"/>
   <add name="BlockViewHandler" path="*" verb="*" preCondition="integratedMode"
type="System.Web.HttpNotFoundHandler"/>
  </handlers>
</system.webServer>
```

Chapter 30: System.Net.Mail

Section 30.1: MailMessage

Here is the example of creating of mail message with attachments. After creating we send this message with the help of `SmtpClient` class. Default 25 port is used here.

```
public class clsMail
{
    private static bool SendMail(string mailfrom, List<string>replytos, List<string> mailtos,
List<string> mailccs, List<string> mailbccs, string body, string subject, List<string> Attachment)
    {
        try
        {
            using(MailMessage MyMail = new MailMessage())
            {
                MyMail.From = new MailAddress(mailfrom);
                foreach (string mailto in mailtos)
                    MyMail.To.Add(mailto);

                if (replytos != null && replytos.Any())
                {
                    foreach (string replyto in replytos)
                        MyMail.ReplyToList.Add(replyto);
                }

                if (mailccs != null && mailccs.Any())
                {
                    foreach (string mailcc in mailccs)
                        MyMail.CC.Add(mailcc);
                }

                if (mailbccs != null && mailbccs.Any())
                {
                    foreach (string mailbcc in mailbccs)
                        MyMail.Bcc.Add(mailbcc);
                }

                MyMail.Subject = subject;
                MyMail.IsBodyHtml = true;
                MyMail.Body = body;
                MyMail.Priority = MailPriority.Normal;

                if (Attachment != null && Attachment.Any())
                {
                    System.Net.Mail.Attachment attachment;
                    foreach (var item in Attachment)
                    {
                        attachment = new System.Net.Mail.Attachment(item);
                        MyMail.Attachments.Add(attachment);
                    }
                }

                SmtpClient smtpMailObj = new SmtpClient();
                smtpMailObj.Host = "your host";
                smtpMailObj.Port = 25;
                smtpMailObj.Credentials = new System.Net.NetworkCredential("uid", "pwd");

                smtpMailObj.Send(MyMail);
                return true;
```

```
            }
        }
        catch
        {
            return false;
        }
    }
}
```

Section 30.2: Mail with Attachment

`MailMessage` represents mail message which can be sent further using `SmtpClient` class. Several attachments (files) can be added to mail message.

```
using System.Net.Mail;

using(MailMessage myMail = new MailMessage())
{
     Attachment attachment = new Attachment(path);
     myMail.Attachments.Add(attachment);

     // further processing to send the mail message

}
```

Chapter 31: Using Progress<T> and IProgress<T>

Section 31.1: Simple Progress reporting

`IProgress<T>` can be used to report progress of some procedure to another procedure. This example shows how you can create a basic method that reports its progress.

```
void Main()
{
    IProgress<int> p = new Progress<int>(progress =>
    {
        Console.WriteLine("Running Step: {0}", progress);
    });
    LongJob(p);
}

public void LongJob(IProgress<int> progress)
{
    var max = 10;
    for (int i = 0; i < max; i++)
    {
        progress.Report(i);
    }
}
```

Output:

```
Running Step: 0
Running Step: 3
Running Step: 4
Running Step: 5
Running Step: 6
Running Step: 7
Running Step: 8
Running Step: 9
Running Step: 2
Running Step: 1
```

Note that when you this code runs, you may see numbers be output out of order. This is because the `IProgress<T>.Report()` method is run asynchronously, and is therefore not as suitable for situations where the progress must be reported in order.

Section 31.2: Using IProgress<T>

It's important to note that the `System.Progress<T>` class does not have the `Report()` method available on it. This method was implemented explicitly from the `IProgress<T>` interface, and therefore must be called on a `Progress<T>` when it's cast to an `IProgress<T>`.

```
var p1 = new Progress<int>();
p1.Report(1); //compiler error, Progress does not contain method 'Report'

IProgress<int> p2 = new Progress<int>();
p2.Report(2); //works
```

```
var p3 = new Progress<int>();
((IProgress<int>)p3).Report(3); //works
```

Chapter 32: JSON Serialization

Section 32.1: Deserialization using System.Web.Script.Serialization.JavaScriptSerializer

The `JavaScriptSerializer.Deserialize<T>(input)` method attempts to deserialize a string of valid JSON into an object of the specified type **<T>**, using the default mappings natively supported by `JavaScriptSerializer`.

```
using System.Collections;
using System.Web.Script.Serialization;

// ...

string rawJSON = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";

JavaScriptSerializer JSS = new JavaScriptSerializer();
Dictionary<string, object> parsedObj = JSS.Deserialize<Dictionary<string, object>>(rawJSON);

string name = parsedObj["Name"].toString();
ArrayList numbers = (ArrayList)parsedObj["Numbers"]
```

Note: The `JavaScriptSerializer` object was introduced in .NET version 3.5

Section 32.2: Serialization using Json.NET

```
[JsonObject("person")]
public class Person
{
    [JsonProperty("name")]
    public string PersonName { get; set; }
    [JsonProperty("age")]
    public int PersonAge { get; set; }
    [JsonIgnore]
    public string Address { get; set; }
}

Person person = new Person { PersonName = "Andrius", PersonAge = 99, Address = "Some address" };
string rawJson = JsonConvert.SerializeObject(person);

Console.WriteLine(rawJson); // {"name":"Andrius","age":99}
```

Notice how properties (and classes) can be decorated with attributes to change their appearance in resulting json string or to remove them from json string at all (JsonIgnore).

More information about Json.NET serialization attributes can be found here.

In C#, public identifiers are written in *PascalCase* by convention. In JSON, the convention is to use *camelCase* for all names. You can use a contract resolver to convert between the two.

```
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    [JsonIgnore]
```

```
    public string Address { get; set; }
}

public void ToJson() {
    Person person = new Person { Name = "Andrius", Age = 99, Address = "Some address" };
    var resolver = new CamelCasePropertyNamesContractResolver();
    var settings = new JsonSerializerSettings { ContractResolver = resolver };
    string json = JsonConvert.SerializeObject(person, settings);

    Console.WriteLine(json); // {"name":"Andrius","age":99}
}
```

Section 32.3: Serialization-Deserialization using Newtonsoft.Json

Unlike the other helpers, this one uses static class helpers to serialize and deserialize, hence it is a little bit easier than the others to use.

```
using Newtonsoft.Json;

var rawJSON      = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";
var fibo         = JsonConvert.DeserializeObject<Dictionary<string, object>>(rawJSON);
var rawJSON2     = JsonConvert.SerializeObject(fibo);
```

Section 32.4: Deserialization using Json.NET

```
internal class Sequence{
    public string Name;
    public List<int> Numbers;
}

// ...

string rawJSON = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";

Sequence sequence = JsonConvert.DeserializeObject<Sequence>(rawJSON);
```

For more information, refer to the Json.NET official site.

Note: Json.NET supports .NET version 2 and higher.

Section 32.5: Dynamic binding

Newtonsoft's Json.NET allows you to bind json dynamically (using ExpandoObject / Dynamic objects) without the need to create the type explicitly.

Serialization

```
dynamic jsonObject = new ExpandoObject();
jsonObject.Title   = "Merchent of Venice";
jsonObject.Author  = "William Shakespeare";
Console.WriteLine(JsonConvert.SerializeObject(jsonObject));
```

De-serialization

```
var rawJson = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";
dynamic parsedJson = JObject.Parse(rawJson);
```

```
Console.WriteLine("Name: " + parsedJson.Name);
Console.WriteLine("Name: " + parsedJson.Numbers.Length);
```

Notice that the keys in the rawJson object have been turned into member variables in the dynamic object.

This is useful in cases where an application can accept/ produce varying formats of JSON. It is however suggested to use an extra level of validation for the Json string or to the dynamic object generated as a result of serialization/ de-serialization.

Section 32.6: Serialization using Json.NET with JsonSerializerSettings

This serializer has some nice features that the default .net json serializer doesn't have, like Null value handling, you just need to create the `JsonSerializerSettings` :

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings { NullValueHandling
= NullValueHandling.Ignore});
   return result;
}
```

Another serious serializer issue in .net is the self referencing loop. In the case of a student that is enrolled in a course, its instance has a course property and a course has a collection of students that means a `List<Student>` which will create a reference loop. You can handle this with `JsonSerializerSettings` :

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
ReferenceLoopHandling = ReferenceLoopHandling.Ignore});
   return result;
}
```

You can put various serializations option like this:

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings { NullValueHandling
= NullValueHandling.Ignore, ReferenceLoopHandling = ReferenceLoopHandling.Ignore});
   return result;
}
```

Chapter 33: JSON in .NET with Newtonsoft.Json

The NuGet package `Newtonsoft.Json` has become the defacto standard for using and manipulating JSON formatted text and objects in .NET. It is a robust tool that is fast, and easy to use.

Section 33.1: Deserialize an object from JSON text

```
var json = "{\"Name\":\"Joe Smith\",\"Age\":21}";
var person = JsonConvert.DeserializeObject<Person>(json);
```

This yields a `Person` object with Name "Joe Smith" and Age 21.

Section 33.2: Serialize object into JSON

```
using Newtonsoft.Json;

var obj = new Person
{
    Name = "Joe Smith",
    Age = 21
};
var serializedJson = JsonConvert.SerializeObject(obj);
```

This results in this JSON: `{"Name":"Joe Smith","Age":21}`

Chapter 34: XmlSerializer

Section 34.1: Formatting: Custom DateTime format

```
public class Dog
{
    private const string _birthStringFormat = "yyyy-MM-dd";

    [XmlIgnore]
    public DateTime Birth {get; set;}

    [XmlElement(ElementName="Birth")]
    public string BirthString
    {
        get { return Birth.ToString(_birthStringFormat); }
        set { Birth = DateTime.ParseExact(value, _birthStringFormat, CultureInfo.InvariantCulture);
}
    }
}
```

Section 34.2: Serialize object

```
public void SerializeFoo(string fileName, Foo foo)
{
    var serializer = new XmlSerializer(typeof(Foo));
    using (var stream = File.Open(fileName, FileMode.Create))
    {
        serializer.Serialize(stream, foo);
    }
}
```

Section 34.3: Deserialize object

```
public Foo DeserializeFoo(string fileName)
{
    var serializer = new XmlSerializer(typeof(Foo));
    using (var stream = File.OpenRead(fileName))
    {
        return (Foo)serializer.Deserialize(stream);
    }
}
```

Section 34.4: Behaviour: Map array name to property (XmlArray)

```
<Store>
    <Articles>
        <Product/>
        <Product/>
    </Articles>
</Store>
```

```
public class Store
{
    [XmlArray("Articles")]
```

```
    public List<Product> Products {get; set; }
}
```

Section 34.5: Behaviour: Map Element name to Property

```
<Foo>
    <Dog/>
</Foo>
```

```
public class Foo
{
    // Using XmlElement
    [XmlElement(Name="Dog")]
    public Animal Cat { get; set; }
}
```

Section 34.6: Efficiently building multiple serializers with derived types specified dynamically

Where we came from

Sometimes we can't provide all of the required metadata needed for the XmlSerializer framework in attribute. Suppose we have a base class of serialized objects, and some of the derived classes are unknown to the base class. We can't place an attribute for all of the classes which are not know at the design time of the base type. We could have another team developing some of the derived classes.

What can we do

We can use `new XmlSerializer(type, knownTypes)`, but that would be a O(N^2) operation for N serializers, at least to discover all of the types supplied in arguments:

```
// Beware of the N^2 in terms of the number of types.
var allSerializers = allTypes.Select(t => new XmlSerializer(t, allTypes));
var serializerDictionary = Enumerable.Range(0, allTypes.Length)
    .ToDictionary (i => allTypes[i], i => allSerializers[i])
```

In this example, the the Base type is not aware of it's derived types, which is normal in OOP.

Doing it efficiently

Luckily, there is a method which addresses this particular problem - supplying known types for multiple serializers efficiently:

System.Xml.Serialization.XmlSerializer.FromTypes Method (Type[])

> The FromTypes method allows you to efficiently create an array of XmlSerializer objects for processing an array of Type objects.

```
var allSerializers = XmlSerializer.FromTypes(allTypes);
var serializerDictionary = Enumerable.Range(0, allTypes.Length)
    .ToDictionary(i => allTypes[i], i => allSerializers[i]);
```

Here is a complete code sample:

```
using System;
using System.Collections.Generic;
using System.Xml.Serialization;
using System.Linq;
using System.Linq;

public class Program
{
    public class Container
    {
        public Base Base { get; set; }
    }

    public class Base
    {
        public int JustSomePropInBase { get; set; }
    }

    public class Derived : Base
    {
        public int JustSomePropInDerived { get; set; }
    }

    public void Main()
    {
        var sampleObject = new Container { Base = new Derived() };
        var allTypes = new[] { typeof(Container), typeof(Base), typeof(Derived) };

        Console.WriteLine("Trying to serialize without a derived class metadata:");
        SetupSerializers(allTypes.Except(new[] { typeof(Derived) }).ToArray());
        try
        {
            Serialize(sampleObject);
        }
        catch (InvalidOperationException e)
        {
            Console.WriteLine();
            Console.WriteLine("This error was anticipated,");
            Console.WriteLine("we have not supplied a derived class.");
            Console.WriteLine(e);
        }
        Console.WriteLine("Now trying to serialize with all of the type information:");
        SetupSerializers(allTypes);
        Serialize(sampleObject);
        Console.WriteLine();
        Console.WriteLine("Slides down well this time!");
    }

    static void Serialize<T>(T o)
    {
        serializerDictionary[typeof(T)].Serialize(Console.Out, o);
    }

    private static Dictionary<Type, XmlSerializer> serializerDictionary;

    static void SetupSerializers(Type[] allTypes)
    {
        var allSerializers = XmlSerializer.FromTypes(allTypes);
        serializerDictionary = Enumerable.Range(0, allTypes.Length)
            .ToDictionary(i => allTypes[i], i => allSerializers[i]);
    }
```

```
}
```

Output:

```
Trying to serialize without a derived class metadata:
<?xml version="1.0" encoding="utf-16"?>
 System.InvalidOperationException: The type Program+Derived was not expected. Use the XmlInclude
or SoapInclude attribute to specify types that are not known statically.
at Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write2_Base(String n,
String ns, Base o, Boolean isNullable, Boolean needType)
at Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write3_Container(String
n, String ns, Container o, Boolean isNullable, Boolean needType)
at Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write4_Container(Object
o)
at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle, String id)
--- End of inner exception stack trace ---
at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle, String id)
at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle)
at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces)
at Program.Serialize[T](T o)
at Program.Main()
Now trying to serialize with all of the type information:
<?xml version="1.0" encoding="utf-16"?>


0
0


Slides down well this time!
```

What's in the output

This error message recommends what we tried to avoid (or what we can not do in some scenarios) - referencing derived types from base class:

`Use the XmlInclude or SoapInclude attribute to specify types that are not known statically.`

This is how we get our derived class in the XML:

```
<Base xsi:type="Derived">
```

`Base` corresponds to the property type declared in the `Container` type, and `Derived` being the type of the instance actually supplied.

Here is a working example fiddle

Chapter 35: VB Forms

Section 35.1: Hello World in VB.NET Forms

To show a message box when the form has been shown:

```
Public Class Form1
    Private Sub Form1_Shown(sender As Object, e As EventArgs) Handles MyBase.Shown
        MessageBox.Show("Hello, World!")
    End Sub
End Class
To show a message box before the form has been shown:

Public Class Form1
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        MessageBox.Show("Hello, World!")
    End Sub
End Class
```

Load() will be called first, and only once, when the form first loads. Show() will be called every time the user launches the form. Activate() will be called every time the user makes the form active.

Load() will execute before Show() is called, but be warned: calling msgBox() in show can cause that msgBox() to execute before Load() is finished. **It is generally a bad idea to depend on event ordering between Load(), Show(), and similar.**

Section 35.2: For Beginners

Some things all beginners should know / do that will help them have a good start with VB .Net:

Set the following Options:

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

End Class
```

Use &, not + for string concatenation. Strings should be studied in some detail as they are widely used.

Spend some time understanding Value and Reference Types.

Never use Application.DoEvents. Pay attention to the 'Caution'. When you reach a point where this seems like something you must use, ask.

The documentation is your friend.

Section 35.3: Forms Timer

The Windows.Forms.Timer component can be used to provide the user information that is **not** time critical. Create a form with one button, one label, and a Timer component.

For example it could be used to show the user the time of day periodically.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        Timer1.Interval = 60 * 1000 'one minute intervals
        'start timer
        Timer1.Start()
        Label1.Text = DateTime.Now.ToLongTimeString
    End Sub

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        Label1.Text = DateTime.Now.ToLongTimeString
    End Sub
End Class
```

But this timer is not suited for timing. An example would be using it for a countdown. In this example we will simulate a countdown to three minutes. This may very well be one of the most boringly important examples here.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        ctSecs = 0 'clear count
        Timer1.Interval = 1000 'one second in ms.
        'start timers
        stpw.Reset()
        stpw.Start()
        Timer1.Start()
    End Sub

    Dim stpw As New Stopwatch
    Dim ctSecs As Integer

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        ctSecs += 1
        If ctSecs = 180 Then 'about 2.5 seconds off on my PC!
            'stop timing
            stpw.Stop()
            Timer1.Stop()
            'show actual elapsed time
            'Is it near 180?
            Label1.Text = stpw.Elapsed.TotalSeconds.ToString("n1")
        End If
    End Sub
End Class
```

After button1 is clicked, about three minutes pass and label1 shows the results. Does label1 show 180? Probably not. On my machine it showed 182.5!

The reason for the discrepancy is in the documentation, "The Windows Forms Timer component is single-threaded, and is limited to an accuracy of 55 milliseconds." This is why it shouldn't be used for timing.

By using the timer and stopwatch a little differently we can obtain better results.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        Timer1.Interval = 100 'one tenth of a second in ms.
        'start timers
        stpw.Reset()
        stpw.Start()
        Timer1.Start()
    End Sub

    Dim stpw As New Stopwatch
    Dim threeMinutes As TimeSpan = TimeSpan.FromMinutes(3)

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        If stpw.Elapsed >= threeMinutes Then '0.1 off on my PC!
            'stop timing
            stpw.Stop()
            Timer1.Stop()
            'show actual elapsed time
            'how close?
            Label1.Text = stpw.Elapsed.TotalSeconds.ToString("n1")
        End If
    End Sub
End Class
```

There are other timers that can be used as needed. This search should help in that regard.

Chapter 36: JIT compiler

JIT compilation, or just-in-time compilation, is an alternative approach to interpretation of code or ahead-of-time compilation. JIT compilation is used in the .NET framework. The CLR code (C#, F#, Visual Basic, etc.) is first compiled into something called Interpreted Language, or IL. This is lower level code that is closer to machine code, but is not platform specific. Rather, at runtime, this code is compiled into machine code for the relevant system.

Section 36.1: IL compilation sample

Simple Hello World Application:

```
using System;

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

Equivalent IL Code (which will be JIT compiled)

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 4.6.1055.0
//  Copyright (c) Microsoft Corporation.  All rights reserved.



// Metadata version: v4.0.30319
.assembly extern mscorlib
{
.publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z\V.4..
.ver 4:0:0:0
}
.assembly HelloWorld
{
.custom instance void
[mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.ctor(int32) = ( 01 00
08 00 00 00 00 00 )
.custom instance void
[mscorlib]System.Runtime.CompilerServices.RuntimeCompatibilityAttribute::.ctor() = ( 01 00 01 00
54 02 16 57 72 61 70 4E 6F 6E 45 78   // ....T..WrapNonEx
63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )       // ceptionThrows.

// --- The following custom attribute is added automatically, do not uncomment -------
//  .custom instance void [mscorlib]System.Diagnostics.DebuggableAttribute::.ctor(valuetype
[mscorlib]System.Diagnostics.DebuggableAttribute/DebuggingModes) = ( 01 00 07 01 00 00 00 00 )

.custom instance void [mscorlib]System.Reflection.AssemblyTitleAttribute::.ctor(string) = ( 01 00
0A 48 65 6C 6C 6F 57 6F 72 6C 64 00 00 )    // ...HelloWorld..
.custom instance void [mscorlib]System.Reflection.AssemblyDescriptionAttribute::.ctor(string) = (
01 00 00 00 00 )
.custom instance void [mscorlib]System.Reflection.AssemblyConfigurationAttribute::.ctor(string) =
( 01 00 00 00 00 )
.custom instance void [mscorlib]System.Reflection.AssemblyCompanyAttribute::.ctor(string) = ( 01
00 00 00 00 )
```

```
.custom instance void [mscorlib]System.Reflection.AssemblyProductAttribute::.ctor(string) = ( 01
00 0A 48 65 6C 6C 6F 57 6F 72 6C 64 00 00 )    // ...HelloWorld..
.custom instance void [mscorlib]System.Reflection.AssemblyCopyrightAttribute::.ctor(string) = ( 01
00 12 43 6F 70 79 72 69 67 68 74 20 C2 A9 20   // ...Copyright ..
20 32 30 31 37 00 00 )                         //  2017..
.custom instance void [mscorlib]System.Reflection.AssemblyTrademarkAttribute::.ctor(string) = ( 01
00 00 00 00 )
.custom instance void [mscorlib]System.Runtime.InteropServices.ComVisibleAttribute::.ctor(bool) =
( 01 00 00 00 00 )
.custom instance void [mscorlib]System.Runtime.InteropServices.GuidAttribute::.ctor(string) = ( 01
00 24 33 30 38 62 33 64 38 36 2D 34 31 37 32   // ..$308b3d86-4172
2D 34 30 32 32 2D 61 66 63 63 2D 33 66 38 65 33   // -4022-afcc-3f8e3
32 33 33 63 35 62 30 00 00 )                   // 233c5b0..
.custom instance void [mscorlib]System.Reflection.AssemblyFileVersionAttribute::.ctor(string) = (
01 00 07 31 2E 30 2E 30 2E 30 00 00 )          // ...1.0.0.0..
.custom instance void [mscorlib]System.Runtime.Versioning.TargetFrameworkAttribute::.ctor(string)
= ( 01 00 1C 2E 4E 45 54 46 72 61 6D 65 77 6F 72 6B   // ....NETFramework
2C 56 65 72 73 69 6F 6E 3D 76 34 2E 35 2E 32 01   // ,Version=v4.5.2.
00 54 0E 14 46 72 61 6D 65 77 6F 72 6B 44 69 73   // .T..FrameworkDis
70 6C 61 79 4E 61 6D 65 14 2E 4E 45 54 20 46 72   // playName..NET Fr
61 6D 65 77 6F 72 6B 20 34 2E 35 2E 32 )       // amework 4.5.2
.hash algorithm 0x00008004
.ver 1:0:0:0
}
.module HelloWorld.exe
// MVID: {2A7E1D59-1272-4B47-85F6-D7E1ED057831}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00020003    //  ILONLY 32BITPREFERRED
// Image base: 0x0000021C70230000


// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit HelloWorld.Program
extends [mscorlib]System.Object
{
.method private hidebysig static void  Main(string[] args) cil managed
{
.entrypoint
// Code size       13 (0xd)
.maxstack  8
IL_0000:  nop
IL_0001:  ldstr      "Hello World"
IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
IL_000b:  nop
IL_000c:  ret
} // end of method Program::Main

.method public hidebysig specialname rtspecialname
instance void  .ctor() cil managed
{
// Code size       8 (0x8)
.maxstack  8
IL_0000:  ldarg.0
IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
IL_0006:  nop
IL_0007:  ret
} // end of method Program::.ctor

} // end of class HelloWorld.Program
```

Generated with MS ILDASM tool (IL disassembler)

Chapter 37: CLR

Section 37.1: An introduction to Common Language Runtime

The **Common Language Runtime (CLR)** is a virtual machine environment and part of the .NET Framework. It contains:

- A portable bytecode language called **Common Intermediate Language** (abbreviated CIL, or IL)
- A Just-In-Time compiler that generates machine code
- A tracing garbage collector that provides automatic memory management
- Support for lightweight sub-processes called AppDomains
- Security mechanisms through the concepts of verifiable code and trust levels

Code that runs in the CLR is referred to as *managed code* to distinguish it from code running outside the CLR (usually native code) which is referred to as *unmanaged code*. There are various mechanisms that facilitate interoperability between managed and unmanaged code.

Chapter 38: TPL Dataflow

Section 38.1: Asynchronous Producer Consumer With A Bounded BufferBlock

```
var bufferBlock = new BufferBlock<int>(new DataflowBlockOptions
{
    BoundedCapacity = 1000
});

var cancellationToken = new CancellationTokenSource(TimeSpan.FromSeconds(10)).Token;

var producerTask = Task.Run(async () =>
{
    var random = new Random();

    while (!cancellationToken.IsCancellationRequested)
    {
        var value = random.Next();
        await bufferBlock.SendAsync(value, cancellationToken);
    }
});

var consumerTask = Task.Run(async () =>
{
    while (await bufferBlock.OutputAvailableAsync())
    {
        var value = bufferBlock.Receive();
        Console.WriteLine(value);
    }
});

await Task.WhenAll(producerTask, consumerTask);
```

Section 38.2: Posting to an ActionBlock and waiting for completion

```
// Create a block with an asynchronous action
var block = new ActionBlock<string>(async hostName =>
{
    IPAddress[] ipAddresses = await Dns.GetHostAddressesAsync(hostName);
    Console.WriteLine(ipAddresses[0]);
});

block.Post("google.com"); // Post items to the block's InputQueue for processing
block.Post("reddit.com");
block.Post("stackoverflow.com");

block.Complete(); // Tell the block to complete and stop accepting new items
await block.Completion; // Asynchronously wait until all items completed processingu
```

Section 38.3: Linking blocks to create a pipeline

```
var httpClient = new HttpClient();

// Create a block the accepts a uri and returns its contents as a string
var downloaderBlock = new TransformBlock<string, string>(
```

```
    async uri => await httpClient.GetStringAsync(uri));

// Create a block that accepts the content and prints it to the console
var printerBlock = new ActionBlock<string>(
    contents => Console.WriteLine(contents));

// Make the downloaderBlock complete the printerBlock when its completed.
var dataflowLinkOptions = new DataflowLinkOptions {PropagateCompletion = true};

// Link the block to create a pipeline
downloaderBlock.LinkTo(printerBlock, dataflowLinkOptions);

// Post urls to the first block which will pass their contents to the second one.
downloaderBlock.Post("http://youtube.com");
downloaderBlock.Post("http://github.com");
downloaderBlock.Post("http://twitter.com");

downloaderBlock.Complete(); // Completion will propagate to printerBlock
await printerBlock.Completion; // Only need to wait for the last block in the pipeline
```

Section 38.4: Synchronous Producer/Consumer with BufferBlock<T>

```
public class Producer
{
    private static Random random = new Random((int)DateTime.UtcNow.Ticks);
    //produce the value that will be posted to buffer block
    public double Produce ( )
    {
        var value = random.NextDouble();
        Console.WriteLine($"Producing value: {value}");
        return value;
    }
}

public class Consumer
{
    //consume the value that will be received from buffer block
    public void Consume (double value) => Console.WriteLine($"Consuming value: {value}");
}

class Program
{
    private static BufferBlock<double> buffer = new BufferBlock<double>();
    static void Main (string[] args)
    {
        //start a task that will every 1 second post a value from the producer to buffer block
        var producerTask = Task.Run(async () =>
        {
            var producer = new Producer();
            while(true)
            {
                buffer.Post(producer.Produce());
                await Task.Delay(1000);
            }
        });
        //start a task that will recieve values from bufferblock and consume it
        var consumerTask = Task.Run(() =>
        {
            var consumer = new Consumer();
            while(true)
```

```
            {
                consumer.Consume(buffer.Receive());
            }
        });

        Task.WaitAll(new[] { producerTask, consumerTask });
    }
}
```

Chapter 39: Threading

Section 39.1: Accessing form controls from other threads

If you want to change an attribute of a control such as a textbox or label from another thread than the GUI thread that created the control, you will have to invoke it or else you might get an error message stating:

> "Cross-thread operation not valid: Control 'control_name' accessed from a thread other than the thread it was created on."

Using this example code on a system.windows.forms form will cast an exception with that message:

```
private void button4_Click(object sender, EventArgs e)
{
    Thread thread = new Thread(updatetextbox);
    thread.Start();
}

private void updatetextbox()
{
    textBox1.Text = "updated"; // Throws exception
}
```

Instead when you want to change a textbox's text from within a thread that doesn't own it use Control.Invoke or Control.BeginInvoke. You can also use Control.InvokeRequired to check if invoking the control is necessary.

```
private void updatetextbox()
{
    if (textBox1.InvokeRequired)
        textBox1.BeginInvoke((Action)(() => textBox1.Text = "updated"));
    else
        textBox1.Text = "updated";
}
```

If you need to do this often, you can write an extension for invokeable objects to reduce the amount of code necessary to make this check:

```
public static class Extensions
{
    public static void BeginInvokeIfRequired(this ISynchronizeInvoke obj, Action action)
    {
        if (obj.InvokeRequired)
            obj.BeginInvoke(action, new object[0]);
        else
            action();
    }
}
```

And updating the textbox from any thread becomes a bit simpler:

```
private void updatetextbox()
{
    textBox1.BeginInvokeIfRequired(() => textBox1.Text = "updated");
}
```

Be aware that Control.BeginInvoke as used in this example is asynchronous, meaning that code coming after a call to Control.BeginInvoke can be run immedeately after, whether or not the passed delegate has been executed yet.

If you need to be sure that textBox1 is updated before continuing, use Control.Invoke instead, which will block the calling thread until your delegate has been executed. Do note that this approach can slow your code down significantly if you make many invoke calls and note that it will deadlock your application if your GUI thread is waiting for the calling thread to complete or release a held resource.

Chapter 40: Process and Thread affinity setting

Parameter	Details
affinity	integer that describes the set of processors on which the process is allowed to run. For example, on a 8 processor system if you want your process to be executed only on processors 3 and 4 than you choose affinity like this : 00001100 which equals 12

Section 40.1: Get process affinity mask

```
    public static int GetProcessAffinityMask(string processName = null)
    {
        Process myProcess = GetProcessByName(ref processName);

        int processorAffinity = (int)myProcess.ProcessorAffinity;
        Console.WriteLine("Process {0} Affinity Mask is : {1}", processName,
FormatAffinity(processorAffinity));

        return processorAffinity;
    }

    public static Process GetProcessByName(ref string processName)
    {
        Process myProcess;
        if (string.IsNullOrEmpty(processName))
        {
            myProcess = Process.GetCurrentProcess();
            processName = myProcess.ProcessName;
        }
        else
        {
            Process[] processList = Process.GetProcessesByName(processName);
            myProcess = processList[0];
        }
        return myProcess;
    }

    private static string FormatAffinity(int affinity)
    {
        return Convert.ToString(affinity, 2).PadLeft(Environment.ProcessorCount, '0');
    }
}
```

Example of usage :

```
private static void Main(string[] args)
{
    GetProcessAffinityMask();

    Console.ReadKey();
}
// Output:
// Process Test.vshost Affinity Mask is : 11111111
```

Section 40.2: Set process affinity mask

```
    public static void SetProcessAffinityMask(int affinity, string processName = null)
```

```
    {
        Process myProcess = GetProcessByName(ref processName);

        Console.WriteLine("Process {0} Old Affinity Mask is : {1}", processName,
FormatAffinity((int)myProcess.ProcessorAffinity));

        myProcess.ProcessorAffinity = new IntPtr(affinity);
        Console.WriteLine("Process {0} New Affinity Mask is : {1}", processName,
FormatAffinity((int)myProcess.ProcessorAffinity));
    }
```

Example of usage :

```
private static void Main(string[] args)
{
    int newAffinity = Convert.ToInt32("10101010", 2);
    SetProcessAffinityMask(newAffinity);

    Console.ReadKey();
}
// Output :
// Process Test.vshost Old Affinity Mask is : 11111111
// Process Test.vshost New Affinity Mask is : 10101010
```

Chapter 41: Parallel processing using .Net framework

This Topic is about Multi core programming using Task Parallel Library with .NET framework. The task parallel library allows you to write code which is human readable and adjusts itself with the number of Cores available. So you can be sure that your software would auto-upgrade itself with the upgrading environment.

Section 41.1: Parallel Extensions

Parallel extensions have been introduced along with the Task Parallel Library to achieve data Parallelism. Data parallelism refers to scenarios in which the same operation is performed concurrently (that is, in parallel) on elements in a source collection or array. The .NET provides new constructs to achieve data parallelism by using Parallel.For and Parallel.Foreach constructs.

```
//Sequential version

foreach (var item in sourcecollection){

Process(item);

}

// Parallel equivalent

Parallel.foreach(sourcecollection, item => Process(item));
```

The above mentioned Parallel.ForEach construct utilizes the multiple cores and thus enhances the performance in the same fashion.

Chapter 42: Task Parallel Library (TPL)

Section 42.1: Basic producer-consumer loop (BlockingCollection)

```
var collection = new BlockingCollection<int>(5);
var random = new Random();

var producerTask = Task.Run(() => {
    for(int item=1; item<=10; item++)
    {
        collection.Add(item);
        Console.WriteLine("Produced: " + item);
        Thread.Sleep(random.Next(10,1000));
    }
    collection.CompleteAdding();
    Console.WriteLine("Producer completed!");
});
```

It is worth noting that if you do not call `collection.CompleteAdding();`, you are able to keep adding to the collection even if your consumer task is running. Just call `collection.CompleteAdding();` when you are sure there are no more additions. This functionality can be used to make a Multiple Producer to a Single Consumer pattern where you have multiple sources feeding items into the BlockingCollection and a single consumer pulling items out and doing something with them. If your BlockingCollection is empty before you call complete adding, the Enumerable from `collection.GetConsumingEnumerable()` will block until a new item is added to the collection or BlockingCollection.CompleteAdding(); is called and the queue is empty.

```
var consumerTask = Task.Run(() => {
    foreach(var item in collection.GetConsumingEnumerable())
    {
        Console.WriteLine("Consumed: " + item);
        Thread.Sleep(random.Next(10,1000));
    }
    Console.WriteLine("Consumer completed!");
});

Task.WaitAll(producerTask, consumerTask);

Console.WriteLine("Everything completed!");
```

Section 42.2: Parallel.Invoke

```
var actions = Enumerable.Range(1, 10).Select(n => new Action(() =>
{
    Console.WriteLine("I'm task " + n);
    if((n & 1) == 0)
        throw new Exception("Exception from task " + n);
})).ToArray();

try
{
    Parallel.Invoke(actions);
}
catch(AggregateException ex)
{
    foreach(var inner in ex.InnerExceptions)
        Console.WriteLine("Task failed: " + inner.Message);
```

```
}
```

Section 42.3: Task: Returning a value

Task that return a value has return type of `Task< TResult >` where TResult is the type of value that needs to be returned. You can query the outcome of a Task by its Result property.

```
Task<int> t = Task.Run(() =>
    {
        int sum = 0;

        for(int i = 0; i < 500; i++)
            sum += i;

        return sum;
    });

Console.WriteLine(t.Result); // Outuput 124750
```

If the Task execute asynchronously than awaiting the Task returns it's result.

```
public async Task DoSomeWork()
{
    WebClient client = new WebClient();
    // Because the task is awaited, result of the task is assigned to response
    string response = await client.DownloadStringTaskAsync("http://somedomain.com");
}
```

Section 42.4: Parallel.ForEach

This example uses `Parallel.ForEach` to calculate the sum of the numbers between 1 and 10000 by using multiple threads. To achieve thread-safety, `Interlocked.Add` is used to sum the numbers.

```
using System.Threading;

int Foo()
{
    int total = 0;
    var numbers = Enumerable.Range(1, 10000).ToList();
    Parallel.ForEach(numbers,
        () => 0, // initial value,
        (num, state, localSum) => num + localSum,
        localSum => Interlocked.Add(ref total, localSum));
    return total; // total = 50005000
}
```

Section 42.5: Parallel.For

This example uses `Parallel.For` to calculate the sum of the numbers between 1 and 10000 by using multiple threads. To achieve thread-safety, `Interlocked.Add` is used to sum the numbers.

```
using System.Threading;

int Foo()
{
    int total = 0;
    Parallel.For(1, 10001,
```

```
        () => 0, // initial value,
        (num, state, localSum) => num + localSum,
        localSum => Interlocked.Add(ref total, localSum));
    return total; // total = 50005000
}
```

Section 42.6: Task: basic instantiation and Wait

A task can be created by directly instantiating the `Task` class...

```
var task = new Task(() =>
{
    Console.WriteLine("Task code starting...");
    Thread.Sleep(2000);
    Console.WriteLine("...task code ending!");
});

Console.WriteLine("Starting task...");
task.Start();
task.Wait();
Console.WriteLine("Task completed!");
```

...or by using the static `Task.Run` method:

```
Console.WriteLine("Starting task...");
var task = Task.Run(() =>
{
    Console.WriteLine("Task code starting...");
    Thread.Sleep(2000);
    Console.WriteLine("...task code ending!");
});
task.Wait();
Console.WriteLine("Task completed!");
```

Note that only in the first case it is necessary to explicitly invoke `Start`.

Section 42.7: Task.WhenAll

```
var random = new Random();
IEnumerable<Task<int>> tasks = Enumerable.Range(1, 5).Select(n => Task.Run(() =>
{
    Console.WriteLine("I'm task " + n);
    return n;
}));

Task<int[]> task = Task.WhenAll(tasks);
int[] results = await task;

Console.WriteLine(string.Join(",", results.Select(n => n.ToString())));
// Output: 1,2,3,4,5
```

Section 42.8: Flowing execution context with AsyncLocal

When you need to pass some data from the parent task to its children tasks, so it logically flows with the execution, use `AsyncLocal` class:

```
void Main()
{
```

```
    AsyncLocal<string> user = new AsyncLocal<string>();
    user.Value = "initial user";

    // this does not affect other tasks - values are local relative to the branches of execution flow
    Task.Run(() => user.Value = "user from another task");

    var task1 = Task.Run(() =>
    {
        Console.WriteLine(user.Value); // outputs "initial user"
        Task.Run(() =>
        {
            // outputs "initial user" - value has flown from main method to this task without being
changed
            Console.WriteLine(user.Value);
        }).Wait();

        user.Value = "user from task1";

        Task.Run(() =>
        {
            // outputs "user from task1" - value has flown from main method to task1
            // than value was changed and flown to this task.
            Console.WriteLine(user.Value);
        }).Wait();
    });

    task1.Wait();

    // outputs "initial user" - changes do not propagate back upstream the execution flow
    Console.WriteLine(user.Value);
}
```

Note: As can be seen from the example above `AsynLocal.Value` has `copy on read` semantic, but if you flow some reference type and change its properties you will affect other tasks. Hence, best practice with `AsyncLocal` is to use value types or immutable types.

Section 42.9: Parallel.ForEach in VB.NET

```
For Each row As DataRow In FooDataTable.Rows
    Me.RowsToProcess.Add(row)
Next

Dim myOptions As ParallelOptions = New ParallelOptions()
myOptions.MaxDegreeOfParallelism = environment.processorcount

Parallel.ForEach(RowsToProcess, myOptions, Sub(currentRow, state)
                                               ProcessRowParallel(currentRow, state)
                                           End Sub)
```

Section 42.10: Task: WaitAll and variable capturing

```
var tasks = Enumerable.Range(1, 5).Select(n => new Task<int>(() =>
{
    Console.WriteLine("I'm task " + n);
    return n;
})).ToArray();

foreach(var task in tasks) task.Start();
Task.WaitAll(tasks);
```

```
foreach(var task in tasks)
    Console.WriteLine(task.Result);
```

Section 42.11: Task: WaitAny

```
var allTasks = Enumerable.Range(1, 5).Select(n => new Task<int>(() => n)).ToArray();
var pendingTasks = allTasks.ToArray();

foreach(var task in allTasks) task.Start();

while(pendingTasks.Length > 0)
{
    var finishedTask = pendingTasks[Task.WaitAny(pendingTasks)];
    Console.WriteLine("Task {0} finished", finishedTask.Result);
    pendingTasks = pendingTasks.Except(new[] {finishedTask}).ToArray();
}

Task.WaitAll(allTasks);
```

Note: The final `WaitAll` is necessary becasue `WaitAny` does not cause exceptions to be observed.

Section 42.12: Task: handling exceptions (using Wait)

```
var task1 = Task.Run(() =>
{
    Console.WriteLine("Task 1 code starting...");
    throw new Exception("Oh no, exception from task 1!!");
});

var task2 = Task.Run(() =>
{
    Console.WriteLine("Task 2 code starting...");
    throw new Exception("Oh no, exception from task 2!!");
});

Console.WriteLine("Starting tasks...");
try
{
    Task.WaitAll(task1, task2);
}
catch(AggregateException ex)
{
    Console.WriteLine("Task(s) failed!");
    foreach(var inner in ex.InnerExceptions)
        Console.WriteLine(inner.Message);
}

Console.WriteLine("Task 1 status is: " + task1.Status); //Faulted
Console.WriteLine("Task 2 status is: " + task2.Status); //Faulted
```

Section 42.13: Task: handling exceptions (without using Wait)

```
var task1 = Task.Run(() =>
{
    Console.WriteLine("Task 1 code starting...");
    throw new Exception("Oh no, exception from task 1!!");
});

var task2 = Task.Run(() =>
```

```
{
    Console.WriteLine("Task 2 code starting...");
    throw new Exception("Oh no, exception from task 2!!");
});

var tasks = new[] {task1, task2};

Console.WriteLine("Starting tasks...");
while(tasks.All(task => !task.IsCompleted));

foreach(var task in tasks)
{
    if(task.IsFaulted)
        Console.WriteLine("Task failed: " +
            task.Exception.InnerExceptions.First().Message);
}

Console.WriteLine("Task 1 status is: " + task1.Status); //Faulted
Console.WriteLine("Task 2 status is: " + task2.Status); //Faulted
```

Section 42.14: Task: cancelling using CancellationToken

```
var cancellationTokenSource = new CancellationTokenSource();
var cancellationToken = cancellationTokenSource.Token;

var task = new Task((state) =>
    {
        int i = 1;
        var myCancellationToken = (CancellationToken)state;
        while(true)
        {
            Console.Write("{0} ", i++);
            Thread.Sleep(1000);
            myCancellationToken.ThrowIfCancellationRequested();
        }
    },
    cancellationToken: cancellationToken,
    state: cancellationToken);

Console.WriteLine("Counting to infinity. Press any key to cancel!");
task.Start();
Console.ReadKey();

cancellationTokenSource.Cancel();
try
{
    task.Wait();
}
catch(AggregateException ex)
{
    ex.Handle(inner => inner is OperationCanceledException);
}

Console.WriteLine($"{Environment.NewLine}You have cancelled! Task status is: {task.Status}");
//Canceled
```

As an alternative to `ThrowIfCancellationRequested`, the cancellation request can be detected with `IsCancellationRequested` and a `OperationCanceledException` can be thrown manually:

```
//New task delegate
```

```
int i = 1;
var myCancellationToken = (CancellationToken)state;
while(!myCancellationToken.IsCancellationRequested)
{
    Console.Write("{0} ", i++);
    Thread.Sleep(1000);
}
Console.WriteLine($"{Environment.NewLine}Ouch, I have been cancelled!!");
throw new OperationCanceledException(myCancellationToken);
```

Note how the cancellation token is passed to the task constructor in the `cancellationToken` parameter. This is needed so that the task transitions to the `Canceled` state, not to the `Faulted` state, when `ThrowIfCancellationRequested` is invoked. Also, for the same reason, the cancellation token is explicitly supplied in the constructor of `OperationCanceledException` in the second case.

Section 42.15: Task.WhenAny

```
var random = new Random();
IEnumerable<Task<int>> tasks = Enumerable.Range(1, 5).Select(n => Task.Run(async() =>
{
    Console.WriteLine("I'm task " + n);
    await Task.Delay(random.Next(10,1000));
    return n;
}));

Task<Task<int>> whenAnyTask = Task.WhenAny(tasks);
Task<int> completedTask = await whenAnyTask;
Console.WriteLine("The winner is: task " + await completedTask);

await Task.WhenAll(tasks);
Console.WriteLine("All tasks finished!");
```

Chapter 43: Task Parallel Library (TPL) API Overviews

Section 43.1: Perform work in response to a button click and update the UI

This example demonstrates how you can respond to a button click by performing some work on a worker thread and then update the user interface to indicate completion

```
void MyButton_OnClick(object sender, EventArgs args)
{
    Task.Run(() => // Schedule work using the thread pool
        {
            System.Threading.Thread.Sleep(5000); // Sleep for 5 seconds to simulate work.
        })
    .ContinueWith(p => // this continuation contains the 'update' code to run on the UI thread
    {
        this.TextBlock_ResultText.Text = "The work completed at " + DateTime.Now.ToString()
    },
    TaskScheduler.FromCurrentSynchronizationContext()); // make sure the update is run on the UI
thread.

}
```

Chapter 44: Synchronization Contexts

Section 44.1: Execute code on the UI thread after performing background work

This example shows how to update a UI component from a background thread by using a `SynchronizationContext`

```
void Button_Click(object sender, EventArgs args)
{
    SynchronizationContext context = SynchronizationContext.Current;
    Task.Run(() =>
    {
        for(int i = 0; i < 10; i++)
        {
            Thread.Sleep(500); //simulate work being done
            context.Post(ShowProgress, "Work complete on item " + i);
        }
    }
}

void UpdateCallback(object state)
{
    // UI can be safely updated as this method is only called from the UI thread
    this.MyTextBox.Text = state as string;
}
```

In this example, if you tried to directly update `MyTextBox.Text` inside the `for` loop, you would get a threading error. By posting the `UpdateCallback` action to the `SynchronizationContext`, the text box is updated on the same thread as the rest of the UI.

In practice, progress updates should be performed using an instance of `System.IProgress<T>`. The default implementation `System.Progress<T>` automatically captures the synchronisation context it is created on.

Chapter 45: Memory management

Section 45.1: Use SafeHandle when wrapping unmanaged resources

When writing wrappers for unmanaged resources, you should subclass `SafeHandle` rather than trying to implement `IDisposable` and a finalizer yourself. Your `SafeHandle` subclass should be as small and simple as possible to minimize the chance of a handle leak. This likely means that your SafeHandle implementation would an internal implementation detail of a class which wraps it to provide a usable API. This class ensures that, even if a program leaks your `SafeHandle` instance, your unmanaged handle is released.

```
using System.Runtime.InteropServices;

class MyHandle : SafeHandle
{
    public override bool IsInvalid => handle == IntPtr.Zero;
    public MyHandle() : base(IntPtr.Zero, true)
    { }

    public MyHandle(int length) : this()
    {
        SetHandle(Marshal.AllocHGlobal(length));
    }

    protected override bool ReleaseHandle()
    {
        Marshal.FreeHGlobal(handle);
        return true;
    }
}
```

Disclaimer: This example is an attempt to show how to guard a managed resource with `SafeHandle` which implements `IDisposable` for you and configures finalizers appropriately. It is very contrived and likely pointless to allocate a chunk of memory in this manner.

Section 45.2: Unmanaged Resources

When we talk about the GC and the "heap", we're really talking about what's called the *managed heap*. Objects on the *managed heap* can access resources not on the managed heap, for example, when writing to or reading from a file. Unexpected behavior can occur when, a file is opened for reading and then an exception occurs, preventing the file handle from closing as it normally would. For this reason, .NET requires that unmanaged resources implement the `IDisposable` interface. This interface has a single method called `Dispose` with no parameters:

```
public interface IDisposable
{
    Dispose();
}
```

When handling unmanaged resources, you should make sure that they are properly disposed. You can do this by explicitly calling `Dispose()` in a `finally` block, or with a `using` statement.

```
StreamReader sr;
string textFromFile;
string filename = "SomeFile.txt";
try
```

```
{
    sr = new StreamReader(filename);
    textFromFile = sr.ReadToEnd();
}
finally
{
    if (sr != null) sr.Dispose();
}
```

or

```
string textFromFile;
string filename = "SomeFile.txt";

using (StreamReader sr = new Streamreader(filename))
{
    textFromFile = sr.ReadToEnd();
}
```

The latter is the preferred method, and is automatically expanded to the former during compilation.

Chapter 46: Garbage Collection

In .Net, objects created with new() are allocated on the managed heap. These objects are never explicitly finalized by the program that uses them; instead, this process is controlled by the .Net Garbage Collector.

Some of the examples below are "lab cases" to show the Garbage Collector at work and some significant details of its behavior, while other focus on how to prepare classes for proper handling by the Garbage Collector.

Section 46.1: A basic example of (garbage) collection

Given the following class:

```
public class FinalizableObject
{
    public FinalizableObject()
    {
        Console.WriteLine("Instance initialized");
    }

    ~FinalizableObject()
    {
        Console.WriteLine("Instance finalized");
    }
}
```

A program that creates an instance, even without using it:

```
new FinalizableObject(); // Object instantiated, ready to be used
```

Produces the following output:

```
<namespace>.FinalizableObject initialized
```

If nothing else happens, the object is not finalized until the program ends (which frees all objects on the managed heap, finalizing these in the process).

It is possible to force the Garbage Collector to run at a given point, as follows:

```
new FinalizableObject(); // Object instantiated, ready to be used
GC.Collect();
```

Which produces the following result:

```
<namespace>.FinalizableObject initialized
<namespace>.FinalizableObject finalized
```

This time, as soon as the Garbage Collector was invoked, the unused (aka "dead") object was finalized and freed from the managed heap.

Section 46.2: Live objects and dead objects - the basics

Rule of thumb: when garbage collection occurs, "live objects" are those still in use, while "dead objects" are those no longer used (any variable or field referencing them, if any, has gone out of scope before the collection occurs).

In the following example (for convenience, FinalizableObject1 and FinalizableObject2 are subclasses of FinalizableObject from the example above and thus inherit the initialization / finalization message behavior):

```
var obj1 = new FinalizableObject1(); // Finalizable1 instance allocated here
var obj2 = new FinalizableObject2(); // Finalizable2 instance allocated here
obj1 = null; // No more references to the Finalizable1 instance
GC.Collect();
```

The output will be:

```
<namespace>.FinalizableObject1 initialized
<namespace>.FinalizableObject2 initialized
<namespace>.FinalizableObject1 finalized
```

At the time when the Garbage Collector is invoked, FinalizableObject1 is a dead object and gets finalized, while FinalizableObject2 is a live object and it is kept on the managed heap.

Section 46.3: Multiple dead objects

What if two (or several) otherwise dead objects reference one another? This is shown in the example below, supposing that OtherObject is a public property of FinalizableObject:

```
var obj1 = new FinalizableObject1();
var obj2 = new FinalizableObject2();
obj1.OtherObject = obj2;
obj2.OtherObject = obj1;
obj1 = null; // Program no longer references Finalizable1 instance
obj2 = null; // Program no longer references Finalizable2 instance
// But the two objects still reference each other
GC.Collect();
```

This produces the following output:

```
<namespace>.FinalizedObject1 initialized
<namespace>.FinalizedObject2 initialized
<namespace>.FinalizedObject1 finalized
<namespace>.FinalizedObject2 finalized
```

The two objects are finalized and freed from the managed heap despite referencing each other (because no other reference exists to any of them from an actually live object).

Section 46.4: Weak References

Weak references are... references, to other objects (aka "targets"), but "weak" as they do not prevent those objects from being garbage-collected. In other words, weak references do not count when the Garbage Collector evaluates objects as "live" or "dead".

The following code:

```
var weak = new WeakReference<FinalizableObject>(new FinalizableObject());
GC.Collect();
```

Produces the output:

```
<namespace>.FinalizableObject initialized
```

```
<namespace>.FinalizableObject finalized
```

The object is freed from the managed heap despite being referenced by the WeakReference variable (still in scope when the Garbage collector was invoked).

Consequence #1: at any time, it is unsafe to assume whether a WeakReference target is still allocated on the managed heap or not.

Consequence #2: whenever a program needs to access the target of a Weakreference, code should be provided for both cases, of the target being still allocated or not. The method to access the target is TryGetTarget:

```
var target = new object(); // Any object will do as target
var weak = new WeakReference<object>(target); // Create weak reference
target = null; // Drop strong reference to the target

// ... Many things may happen in-between

// Check whether the target is still available
if(weak.TryGetTarget(out target))
{
    // Use re-initialized target variable
    // To do whatever the target is needed for
}
else
{
    // Do something when there is no more target object
    // The target variable value should not be used here
}
```

The generic version of WeakReference is available since .Net 4.5. All framework versions provide a non-generic, untyped version that is built in the same way and checked as follows:

```
var target = new object(); // Any object will do as target
var weak = new WeakReference(target); // Create weak reference
target = null; // Drop strong reference to the target

// ... Many things may happen in-between

// Check whether the target is still available
if (weak.IsAlive)
{
    target = weak.Target;

    // Use re-initialized target variable
    // To do whatever the target is needed for
}
else
{
    // Do something when there is no more target object
    // The target variable value should not be used here
}
```

Section 46.5: Dispose() vs. finalizers

Implement Dispose() method (and declare the containing class as IDisposable) as a means to ensure any memory-heavy resources are freed as soon as the object is no longer used. The "catch" is that there is no strong guarantee the the Dispose() method would ever be invoked (unlike finalizers that always get invoked at the end of the life of the object).

One scenario is a program calling Dispose() on objects it explicitly creates:

```
private void SomeFunction()
{
    // Initialize an object that uses heavy external resources
    var disposableObject = new ClassThatImplementsIDisposable();

    // ... Use that object

    // Dispose as soon as no longer used
    disposableObject.Dispose();

    // ... Do other stuff

    // The disposableObject variable gets out of scope here
    // The object will be finalized later on (no guarantee when)
    // But it no longer holds to the heavy external resource after it was disposed
}
```

Another scenario is declaring a class to be instantiated by the framework. In this case the new class usually inherits a base class, for instance in MVC one creates a controller class as a subclass of System.Web.Mvc.ControllerBase. When the base class implements IDisposable interface, this is a good hint that Dispose() would be invoked properly by the framework - but again there is no strong guarantee.

Thus Dispose() is not a substitute for a finalizer; instead, the two should be used for different purposes:

- A finalizer eventually frees resources to avoid memory leaks that would occur otherwise
- Dispose() frees resources (possibly the same ones) as soon as these are no longer needed, to ease pressure on overall memory allocation.

Section 46.6: Proper disposal and finalization of objects

As Dispose() and finalizers are aimed to different purposes, a class managing external memory-heavy resources should implement both of them. The consequence is writing the class so that it handles well two possible scenarios:

- When only the finalizer is invoked
- When Dispose() is invoked first and later the finalizer is invoked as well

One solution is writing the cleanup code in such a way that running it once or twice would produce the same result as running it only once. Feasibility depends on the nature of the cleanup, for instance:

- Closing an already closed database connection would probably have no effect so it works
- Updating some "usage count" is dangerous and would produce a wrong result when called twice instead of once.

A safer solution is ensuring by design that the cleanup code is called once and only once whatever the external context. This can be achieved the "classic way" using a dedicated flag:

```
public class DisposableFinalizable1: IDisposable
{
    private bool disposed = false;

    ~DisposableFinalizable1() { Cleanup(); }

    public void Dispose() { Cleanup(); }

    private void Cleanup()
```

```
    {
        if(!disposed)
        {
            // Actual code to release resources gets here, then
            disposed = true;
        }
    }
}
```

Alternately, the Garbage Collector provides a specific method SuppressFinalize() that allows skipping the finalizer after Dispose has been invoked:

```
public class DisposableFinalizable2 : IDisposable
{
    ~DisposableFinalizable2() { Cleanup(); }

    public void Dispose()
    {
        Cleanup();
        GC.SuppressFinalize(this);
    }

    private void Cleanup()
    {
        // Actual code to release resources gets here
    }
}
```

Chapter 47: Exceptions

Section 47.1: Catching and rethrowing caught exceptions

When you want to catch an exception and do something, but you can't continue execution of the current block of code because of the exception, you may want to rethrow the exception to the next exception handler in the call stack. There are good ways and bad ways to do this.

```
private static void AskTheUltimateQuestion()
{
    try
    {
        var x = 42;
        var y = x / (x - x); // will throw a DivideByZeroException

        // IMPORTANT NOTE: the error in following string format IS intentional
        // and exists to throw an exception to the FormatException catch, below
        Console.WriteLine("The secret to life, the universe, and everything is {1}", y);
    }
    catch (DivideByZeroException)
    {
        // we do not need a reference to the exception
        Console.WriteLine("Dividing by zero would destroy the universe.");

        // do this to preserve the stack trace:
        throw;
    }
    catch (FormatException ex)
    {
        // only do this if you need to change the type of the Exception to be thrown
        // and wrap the inner Exception

        // remember that the stack trace of the outer Exception will point to the
        // next line

        // you'll need to examine the InnerException property to get the stack trace
        // to the line that actually started the problem

        throw new InvalidOperationException("Watch your format string indexes.", ex);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Something else horrible happened. The exception: " + ex.Message);

        // do not do this, because the stack trace will be changed to point to
        // this location instead of the location where the exception
        // was originally thrown:
        throw ex;
    }
}

static void Main()
{
    try
    {
        AskTheUltimateQuestion();
    }
    catch
    {
        // choose this kind of catch if you don't need any information about
```

```
        // the exception that was caught

        // this block "eats" all exceptions instead of rethrowing them
    }
}
```

You can filter by exception type and even by exception properties (new in C# 6.0, a bit longer available in VB.NET (citation needed)):

Documentation/C#/new features

Section 47.2: Using a finally block

The `finally { ... }` block of a `try-finally` or `try-catch-finally` will always execute, regardless of whether an exception occurred or not (except when a `StackOverflowException` has been thrown or call has been made to `Environment.FailFast()`).

It can be utilized to free or clean up resources acquired in the `try { ... }` block safely.

```
Console.Write("Please enter a filename: ");
string filename = Console.ReadLine();

Stream fileStream = null;

try
{
    fileStream = File.Open(filename);
}
catch (FileNotFoundException)
{
    Console.WriteLine("File '{0}' could not be found.", filename);
}
finally
{
    if (fileStream != null)
    {
        fileStream.Dispose();
    }
}
```

Section 47.3: Exception Filters

Since C# 6.0 exceptions can be filtered using the `when` operator.

This is similar to using a simple `if` but does not unwind the stack if the condition inside the `when` is not met.

Example

```
try
{
  // ...
}
catch (Exception e) when (e.InnerException != null) // Any condition can go in here.
{
  // ...
}
```

The same info can be found in the C# 6.0 Features here: Exception filters

Section 47.4: Rethrowing an exception within a catch block

Within a `catch` block the `throw` keyword can be used on its own, without specifying an exception value, to *rethrow* the exception which was just caught. Rethrowing an exception allows the original exception to continue up the exception handling chain, preserving its call stack or associated data:

```
try {...}
catch (Exception ex) {
  // Note: the ex variable is *not* used
  throw;
}
```

A common anti-pattern is to instead `throw ex`, which has the effect of limiting the next exception handler's view of the stack trace:

```
try {...}
catch (Exception ex) {
  // Note: the ex variable is thrown
  //  future stack traces of the exception will not see prior calls
  throw ex;
}
```

In general using `throw ex` isn't desirable, as future exception handlers which inspect the stack trace will only be able to see calls as far back as `throw ex`. By omitting the `ex` variable, and using the `throw` keyword alone the original exception will "bubble-up".

Section 47.5: Throwing an exception from a different method while preserving its information

Occasionally you'd want to catch an exception and throw it from a different thread or method while preserving the original exception stack. This can be done with `ExceptionDispatchInfo`:

```
using System.Runtime.ExceptionServices;

void Main()
{
    ExceptionDispatchInfo capturedException = null;
    try
    {
        throw new Exception();
    }
    catch (Exception ex)
    {
        capturedException = ExceptionDispatchInfo.Capture(ex);
    }

    Foo(capturedException);
}

void Foo(ExceptionDispatchInfo exceptionDispatchInfo)
{
    // Do stuff

    if (capturedException != null)
    {
        // Exception stack trace will show it was thrown from Main() and not from Foo()
        exceptionDispatchInfo.Throw();
    }
```

```
}
```

Section 47.6: Catching an exception

Code can and should throw exceptions in exceptional circumstances. Examples of this include:

- Attempting to read past the end of a stream
- Not having necessary permissions to access a file
- Attempting to perform an invalid operation, such as dividing by zero
- A timeout occurring when downloading a file from the internet

The caller can handle these exceptions by "catching" them, and should only do so when:

- It can actually resolve the exceptional circumstance or recover appropriately, or;
- It can provide additional context to the exception that would be useful if the exception needs to be re-thrown (re-thrown exceptions are caught by exception handlers further up the call stack)

It should be noted that choosing *not* to catch an exception is perfectly valid if the intention is for it to be handled at a higher level.

Catching an exception is done by wrapping the potentially-throwing code in a `try { ... }` block as follows, and catching the exceptions it's able to handle in a `catch (ExceptionType) { ... }` block:

```
Console.Write("Please enter a filename: ");
string filename = Console.ReadLine();

Stream fileStream;

try
{
    fileStream = File.Open(filename);
}
catch (FileNotFoundException)
{
    Console.WriteLine("File '{0}' could not be found.", filename);
}
```

Chapter 48: System.Diagnostics

Section 48.1: Run shell commands

```
string strCmdText = "/C copy /b Image1.jpg + Archive.rar Image2.jpg";
System.Diagnostics.Process.Start("CMD.exe",strCmdText);
```

This is to hide the cmd window.

```
System.Diagnostics.Process process = new System.Diagnostics.Process();
System.Diagnostics.ProcessStartInfo startInfo = new System.Diagnostics.ProcessStartInfo();
startInfo.WindowStyle = System.Diagnostics.ProcessWindowStyle.Hidden;
startInfo.FileName = "cmd.exe";
startInfo.Arguments = "/C copy /b Image1.jpg + Archive.rar Image2.jpg";
process.StartInfo = startInfo;
process.Start();
```

Section 48.2: Send Command to CMD and Receive Output

This method allows a `command` to be sent to `Cmd.exe`, and returns the standard output (including standard error) as a string:

```
private static string SendCommand(string command)
{
    var cmdOut = string.Empty;

    var startInfo = new ProcessStartInfo("cmd", command)
    {
        WorkingDirectory = @"C:\Windows\System32", // Directory to make the call from
        WindowStyle = ProcessWindowStyle.Hidden,   // Hide the window
        UseShellExecute = false,                   // Do not use the OS shell to start the process
        CreateNoWindow = true,                     // Start the process in a new window
        RedirectStandardOutput = true,             // This is required to get STDOUT
        RedirectStandardError = true               // This is required to get STDERR
    };

    var p = new Process {StartInfo = startInfo};

    p.Start();

    p.OutputDataReceived += (x, y) => cmdOut += y.Data;
    p.ErrorDataReceived += (x, y) => cmdOut += y.Data;
    p.BeginOutputReadLine();
    p.BeginErrorReadLine();
    p.WaitForExit();
    return cmdOut;
}
```

Usage

```
var servername = "SVR-01.domain.co.za";
var currentUsers = SendCommand($"/C QUERY USER /SERVER:{servername}")
```

Output

> string currentUsers = "USERNAME SESSIONNAME ID STATE IDLE TIME LOGON TIME Joe.Bloggs ica-cgp#0 2

> Active 24692+13:29 25/07/2016 07:50 Jim.McFlannegan ica-cgp#1 3 Active . 25/07/2016 08:33 Andy.McAnderson ica-cgp#2 4 Active . 25/07/2016 08:54 John.Smith ica-cgp#4 5 Active 14 25/07/2016 08:57 Bob.Bobbington ica-cgp#5 6 Active 24692+13:29 25/07/2016 09:05 Tim.Tom ica-cgp#6 7 Active . 25/07/2016 09:08 Bob.Joges ica-cgp#7 8 Active 24692+13:29 25/07/2016 09:13"

On some occasions, access to the server in question may be restricted to certain users. If you have the login credentials for this user, then it is possible to send queries with this method:

```
private static string SendCommand(string command)
{
    var cmdOut = string.Empty;

    var startInfo = new ProcessStartInfo("cmd", command)
    {
        WorkingDirectory = @"C:\Windows\System32",
        WindowStyle = ProcessWindowStyle.Hidden,    // This does not actually work in conjunction
with "runas" - the console window will still appear!
        UseShellExecute = false,
        CreateNoWindow = true,
        RedirectStandardOutput = true,
        RedirectStandardError = true,

        Verb = "runas",
        Domain = "doman1.co.za",
        UserName = "administrator",
        Password = GetPassword()
    };

    var p = new Process {StartInfo = startInfo};

    p.Start();

    p.OutputDataReceived += (x, y) => cmdOut += y.Data;
    p.ErrorDataReceived += (x, y) => cmdOut += y.Data;
    p.BeginOutputReadLine();
    p.BeginErrorReadLine();
    p.WaitForExit();
    return cmdOut;
}
```

Getting the password:

```
static SecureString GetPassword()
{
    var plainText = "password123";
    var ss = new SecureString();
    foreach (char c in plainText)
    {
        ss.AppendChar(c);
    }

    return ss;
}
```

Notes

Both of the above methods will return a concatenation of STDOUT and STDERR, as `OutputDataReceived` and `ErrorDataReceived` are both appending to the same variable - `cmdOut`.

Section 48.3: Stopwatch

This example shows how `Stopwatch` can be used to benchmark a block of code.

```
using System;
using System.Diagnostics;

public class Benchmark : IDisposable
{
    private Stopwatch sw;

    public Benchmark()
    {
        sw = Stopwatch.StartNew();
    }

    public void Dispose()
    {
        sw.Stop();
        Console.WriteLine(sw.Elapsed);
    }
}

public class Program
{
    public static void Main()
    {
        using (var bench = new Benchmark())
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

Chapter 49: Encryption / Cryptography

Section 49.1: Encryption and Decryption using Cryptography (AES)

Decryption Code

```
    public static string Decrypt(string cipherText)
    {
        if (cipherText == null)
            return null;

        byte[] cipherBytes = Convert.FromBase64String(cipherText);
        using (Aes encryptor = Aes.Create())
        {
            Rfc2898DeriveBytes pdb = new Rfc2898DeriveBytes(CryptKey, new byte[] { 0x49, 0x76,
0x61, 0x6e, 0x20, 0x4d, 0x65, 0x64, 0x76, 0x65, 0x64, 0x65, 0x76 });
            encryptor.Key = pdb.GetBytes(32);
            encryptor.IV = pdb.GetBytes(16);

            using (MemoryStream ms = new MemoryStream())
            {
                using (CryptoStream cs = new CryptoStream(ms, encryptor.CreateDecryptor(),
CryptoStreamMode.Write))
                {
                    cs.Write(cipherBytes, 0, cipherBytes.Length);
                    cs.Close();
                }

                cipherText = Encoding.Unicode.GetString(ms.ToArray());
            }
        }

        return cipherText;
    }
```

Encryption Code

```
    public static string Encrypt(string cipherText)
    {
        if (cipherText == null)
            return null;

        byte[] clearBytes = Encoding.Unicode.GetBytes(cipherText);
        using (Aes encryptor = Aes.Create())
        {
            Rfc2898DeriveBytes pdb = new Rfc2898DeriveBytes(CryptKey, new byte[] { 0x49, 0x76,
0x61, 0x6e, 0x20, 0x4d, 0x65, 0x64, 0x76, 0x65, 0x64, 0x65, 0x76 });
            encryptor.Key = pdb.GetBytes(32);
            encryptor.IV = pdb.GetBytes(16);

            using (MemoryStream ms = new MemoryStream())
            {
                using (CryptoStream cs = new CryptoStream(ms, encryptor.CreateEncryptor(),
CryptoStreamMode.Write))
                {
                    cs.Write(clearBytes, 0, clearBytes.Length);
                    cs.Close();
                }
```

```
                cipherText = Convert.ToBase64String(ms.ToArray());
            }
        }
        return cipherText;
    }
```

Usage

```
var textToEncrypt = "TestEncrypt";

var encrypted = Encrypt(textToEncrypt);

var decrypted = Decrypt(encrypted);
```

Section 49.2: RijndaelManaged

Required Namespace: `System.Security.Cryptography`

```
private class Encryption {

          private const string SecretKey = "topSecretKeyusedforEncryptions";

          private const string SecretIv = "secretVectorHere";

          public string Encrypt(string data) {
            return string.IsNullOrEmpty(data) ? data :
Convert.ToBase64String(this.EncryptStringToBytesAes(data, this.GetCryptographyKey(),
this.GetCryptographyIv()));
          }

          public string Decrypt(string data) {
            return string.IsNullOrEmpty(data) ? data :
this.DecryptStringFromBytesAes(Convert.FromBase64String(data), this.GetCryptographyKey(),
this.GetCryptographyIv());
          }

          private byte[] GetCryptographyKey() {
            return Encoding.ASCII.GetBytes(SecretKey.Replace('e', '!'));
          }

          private byte[] GetCryptographyIv() {
            return Encoding.ASCII.GetBytes(SecretIv.Replace('r', '!'));
          }

          private byte[] EncryptStringToBytesAes(string plainText, byte[] key, byte[] iv) {
            MemoryStream encrypt;
            RijndaelManaged aesAlg = null;
            try {
              aesAlg = new RijndaelManaged {
                Key = key,
                IV = iv
              };
              var encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
              encrypt = new MemoryStream();
              using (var csEncrypt = new CryptoStream(encrypt, encryptor, CryptoStreamMode.Write))
{
                using (var swEncrypt = new StreamWriter(csEncrypt)) {
                  swEncrypt.Write(plainText);
                }
              }
```

```
            } finally {
              aesAlg?.Clear();
            }
            return encrypt.ToArray();
          }

          private string DecryptStringFromBytesAes(byte[] cipherText, byte[] key, byte[] iv) {
            RijndaelManaged aesAlg = null;
            string plaintext;
            try {
              aesAlg = new RijndaelManaged {
                Key = key,
                IV = iv
              };
              var decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);
              using (var msDecrypt = new MemoryStream(cipherText)) {
                using (var csDecrypt = new CryptoStream(msDecrypt, decryptor,
CryptoStreamMode.Read)) {
                  using (var srDecrypt = new StreamReader(csDecrypt))
                    plaintext = srDecrypt.ReadToEnd();
                }
              }
            } finally {
              aesAlg?.Clear();
            }
            return plaintext;
          }
        }
```

Usage

```
var textToEncrypt = "hello World";

 var encrypted = new Encryption().Encrypt(textToEncrypt); //-> zBmW+FUxOvdbpOGm9Ss/vQ==

 var decrypted = new Encryption().Decrypt(encrypted); //-> hello World
```

Note:

- Rijndael is the predecessor of the standard symmetric cryptographic algorithm AES.

Section 49.3: Encrypt and decrypt data using AES (in C#)

```
using System;
using System.IO;
using System.Security.Cryptography;

namespace Aes_Example
{
    class AesExample
    {
        public static void Main()
        {
            try
            {
                string original = "Here is some data to encrypt!";

                // Create a new instance of the Aes class.
                // This generates a new key and initialization vector (IV).
                using (Aes myAes = Aes.Create())
```

```
            {
                // Encrypt the string to an array of bytes.
                byte[] encrypted = EncryptStringToBytes_Aes(original,
                                                            myAes.Key,
                                                            myAes.IV);

                // Decrypt the bytes to a string.
                string roundtrip = DecryptStringFromBytes_Aes(encrypted,
                                                              myAes.Key,
                                                              myAes.IV);

                //Display the original data and the decrypted data.
                Console.WriteLine("Original:   {0}", original);
                Console.WriteLine("Round Trip: {0}", roundtrip);
            }
        }
        catch (Exception e)
        {
            Console.WriteLine("Error: {0}", e.Message);
        }
    }

    static byte[] EncryptStringToBytes_Aes(string plainText, byte[] Key, byte[] IV)
    {
        // Check arguments.
        if (plainText == null || plainText.Length <= 0)
            throw new ArgumentNullException("plainText");
        if (Key == null || Key.Length <= 0)
            throw new ArgumentNullException("Key");
        if (IV == null || IV.Length <= 0)
            throw new ArgumentNullException("IV");

        byte[] encrypted;

        // Create an Aes object with the specified key and IV.
        using (Aes aesAlg = Aes.Create())
        {
            aesAlg.Key = Key;
            aesAlg.IV = IV;

            // Create a decrytor to perform the stream transform.
            ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key,
                                                                aesAlg.IV);

            // Create the streams used for encryption.
            using (MemoryStream msEncrypt = new MemoryStream())
            {
                using (CryptoStream csEncrypt = new CryptoStream(msEncrypt,
                                                                 encryptor,
                                                                 CryptoStreamMode.Write))
                {
                    using (StreamWriter swEncrypt = new StreamWriter(csEncrypt))
                    {
                        //Write all data to the stream.
                        swEncrypt.Write(plainText);
                    }

                    encrypted = msEncrypt.ToArray();
                }
            }
        }
```

```
            // Return the encrypted bytes from the memory stream.
            return encrypted;
        }

        static string DecryptStringFromBytes_Aes(byte[] cipherText, byte[] Key, byte[] IV)
        {
            // Check arguments.
            if (cipherText == null || cipherText.Length <= 0)
                throw new ArgumentNullException("cipherText");
            if (Key == null || Key.Length <= 0)
                throw new ArgumentNullException("Key");
            if (IV == null || IV.Length <= 0)
                throw new ArgumentNullException("IV");

            // Declare the string used to hold the decrypted text.
            string plaintext = null;

            // Create an Aes object with the specified key and IV.
            using (Aes aesAlg = Aes.Create())
            {
                aesAlg.Key = Key;
                aesAlg.IV = IV;

                // Create a decrytor to perform the stream transform.
                ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key,
                                                                    aesAlg.IV);

                // Create the streams used for decryption.
                using (MemoryStream msDecrypt = new MemoryStream(cipherText))
                {
                    using (CryptoStream csDecrypt = new CryptoStream(msDecrypt,
                                                                     decryptor,
                                                                     CryptoStreamMode.Read))
                    {
                        using (StreamReader srDecrypt = new StreamReader(csDecrypt))
                        {

                            // Read the decrypted bytes from the decrypting stream
                            // and place them in a string.
                            plaintext = srDecrypt.ReadToEnd();
                        }
                    }
                }
            }

            return plaintext;
        }
    }
}
```

This example is from MSDN.

It is a console demo application, showing how to encrypt a string by using the standard **AES** encryption, and how to decrypt it afterwards.

(**AES = Advanced Encryption Standard**, a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001 which is still the de-facto standard for symmetric encryption)

Notes:

- In a real encryption scenario, you need to choose a proper cipher mode (can be assigned to the `Mode` property by selecting a value from the `CipherMode` enumeration). **Never** use the `CipherMode.ECB` (electronic codebook mode), since this procuces a weak cypher stream
- To create a good (and not a weak) `Key`, either use a cryptographic random generator or use the example above (**Create a Key from a Password**). The recommended **KeySize** is 256 bit. Supported key sizes are available via the `LegalKeySizes` property.
- To initialize the initialization vector `IV`, you can use a SALT as shown in the example above (**Random SALT**)
- Supported block sizes are available via the `SupportedBlockSizes` property, the block size can be assigned via the `BlockSize` property

Usage: see Main() method.

Section 49.4: Create a Key from a Password / Random SALT (in C#)

```
using System;
using System.Security.Cryptography;
using System.Text;

public class PasswordDerivedBytesExample
{
    public static void Main(String[] args)
    {
        // Get a password from the user.
        Console.WriteLine("Enter a password to produce a key:");

        byte[] pwd = Encoding.Unicode.GetBytes(Console.ReadLine());

        byte[] salt = CreateRandomSalt(7);

        // Create a TripleDESCryptoServiceProvider object.
        TripleDESCryptoServiceProvider tdes = new TripleDESCryptoServiceProvider();

        try
        {
            Console.WriteLine("Creating a key with PasswordDeriveBytes...");

            // Create a PasswordDeriveBytes object and then create
            // a TripleDES key from the password and salt.
            PasswordDeriveBytes pdb = new PasswordDeriveBytes(pwd, salt);

            // Create the key and set it to the Key property
            // of the TripleDESCryptoServiceProvider object.
            tdes.Key = pdb.CryptDeriveKey("TripleDES", "SHA1", 192, tdes.IV);

            Console.WriteLine("Operation complete.");
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
        finally
        {
            // Clear the buffers
            ClearBytes(pwd);
            ClearBytes(salt);
```

```
            // Clear the key.
            tdes.Clear();
        }

        Console.ReadLine();
    }

    #region Helper methods

    /// <summary>
    /// Generates a random salt value of the specified length.
    /// </summary>
    public static byte[] CreateRandomSalt(int length)
    {
        // Create a buffer
        byte[] randBytes;

        if (length >= 1)
        {
            randBytes = new byte[length];
        }
        else
        {
            randBytes = new byte[1];
        }

        // Create a new RNGCryptoServiceProvider.
        RNGCryptoServiceProvider rand = new RNGCryptoServiceProvider();

        // Fill the buffer with random bytes.
        rand.GetBytes(randBytes);

        // return the bytes.
        return randBytes;
    }

    /// <summary>
    /// Clear the bytes in a buffer so they can't later be read from memory.
    /// </summary>
    public static void ClearBytes(byte[] buffer)
    {
        // Check arguments.
        if (buffer == null)
        {
            throw new ArgumentNullException("buffer");
        }

        // Set each byte in the buffer to 0.
        for (int x = 0; x < buffer.Length; x++)
        {
            buffer[x] = 0;
        }
    }

    #endregion
}
```

This example is taken from MSDN.

It is a console demo, and it shows how to create a secure key based on a user-defined password, and how to create a random SALT based on the cryptographic random generator.

Notes:

- The built-in function `PasswordDeriveBytes` uses the standard PBKDF1 algorithm to generate a key from the password. Per default, it uses 100 iterations to generate the key to slow down brute force attacks. The SALT generated randomly further strenghens the key.
- The function `CryptDeriveKey` converts the key generated by `PasswordDeriveBytes` into a key compatible with the specified encryption algorithm (here "TripleDES") by using the specified hash algorithm (here "SHA1"). The keysize in this example is 192 bytes, and the initialization vector IV is taken from the triple-DES crypto provider
- Usually, this mechanism is used to protect a stronger random generated key by a password, which encrypts large amount of data. You can also use it to provide multiple passwords of different users to give access to the same data (being protected by a different random key).
- Unfortunately, `CryptDeriveKey` does currently not support AES. See here.
  NOTE: As a workaround, you can create a random AES key for encryption of the data to be protected with AES and store the AES key in a TripleDES-Container which uses the key generated by `CryptDeriveKey`. But that limits the security to TripleDES, does not take advantage of the larger keysizes of AES and creates a dependency to TripleDES.

Usage: See Main() method.

Chapter 50: Work with SHA1 in C#

in this project you see how to work with SHA1 cryptographic hash function. for example get hash from string and how to crack SHA1 hash.

source compelete on github: https://github.com/mahdiabasi/SHA1Tool

Section 50.1: #Generate SHA1 checksum of a file

> first you add System.Security.Cryptography namespace to your project

```
public string GetSha1Hash(string filePath)
    {
        using (FileStream fs = File.OpenRead(filePath))
        {
            SHA1 sha = new SHA1Managed();
            return BitConverter.ToString(sha.ComputeHash(fs));
        }
    }
```

Section 50.2: #Generate hash of a text

```
  public static string TextToHash(string text)
    {
        var sh = SHA1.Create();
        var hash = new StringBuilder();
        byte[] bytes = Encoding.UTF8.GetBytes(text);
        byte[] b = sh.ComputeHash(bytes);
        foreach (byte a in b)
        {
            var h = a.ToString("x2");
            hash.Append(h);
        }
        return hash.ToString();
    }
```

Chapter 51: Unit testing

Section 51.1: Adding MSTest unit testing project to an existing solution

- Right click on the solution, Add new project
- From the Test section, select an Unit Test Project
- Pick a name for the assembly - if you are testing project `Foo`, the name can be `Foo.Tests`
- Add a reference to the tested project in the unit test project references

Section 51.2: Creating a sample test method

MSTest (the default testing framework) requires you to have your test classes decorated by a `[TestClass]` attribute, and the test methods with a `[TestMethod]` attribute, and to be public.

```
[TestClass]
public class FizzBuzzFixture
{
    [TestMethod]
    public void Test1()
    {
        //arrange
        var solver = new FizzBuzzSolver();
        //act
        var result = solver.FizzBuzz(1);
        //assert
        Assert.AreEqual("1",result);
    }
}
```

Chapter 52: Write to and read from StdErr stream

Section 52.1: Write to standard error output using Console

```
var sourceFileName = "NonExistingFile";
try
{
    System.IO.File.Copy(sourceFileName, "DestinationFile");
}
catch (Exception e)
{
    var stdErr = Console.Error;
    stdErr.WriteLine($"Failed to copy '{sourceFileName}': {e.Message}");
}
```

Section 52.2: Read from standard error of child process

```
var errors = new System.Text.StringBuilder();
var process = new Process
{
    StartInfo = new ProcessStartInfo
    {
        RedirectStandardError = true,
        FileName = "xcopy.exe",
        Arguments = "\"NonExistingFile\" \"DestinationFile\"",
        UseShellExecute = false
    },

};
process.ErrorDataReceived += (s, e) => errors.AppendLine(e.Data);
process.Start();
process.BeginErrorReadLine();
process.WaitForExit();

if (errors.Length > 0) // something went wrong
    System.Console.Error.WriteLine($"Child process error: \r\n {errors}");
```

Chapter 53: Upload file and POST data to webserver

Section 53.1: Upload file with WebRequest

To send a file and form data in single request, content should have multipart/form-data type.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
using System.Threading.Tasks;

public async Task<string> UploadFile(string url, string filename,
    Dictionary<string, object> postData)
{
    var request = WebRequest.CreateHttp(url);
    var boundary = $"{Guid.NewGuid():N}"; // boundary will separate each parameter
    request.ContentType = $"multipart/form-data; {nameof(boundary)}={boundary}";
    request.Method = "POST";

    using (var requestStream = request.GetRequestStream())
    using (var writer = new StreamWriter(requestStream))
    {
        foreach (var data in postData)
            await writer.WriteAsync( // put all POST data into request
                $"\r\n--{boundary}\r\nContent-Disposition: " +
                $"form-data; name=\"{data.Key}\"\r\n\r\n{data.Value}");

        await writer.WriteAsync( // file header
            $"\r\n--{boundary}\r\nContent-Disposition: " +
            $"form-data; name=\"File\"; filename=\"{Path.GetFileName(filename)}\"\r\n" +
            "Content-Type: application/octet-stream\r\n\r\n");

        await writer.FlushAsync();
        using (var fileStream = File.OpenRead(filename))
            await fileStream.CopyToAsync(requestStream);

        await writer.WriteAsync($"\r\n--{boundary}--\r\n");
    }

    using (var response = (HttpWebResponse) await request.GetResponseAsync())
    using (var responseStream = response.GetResponseStream())
    {
        if (responseStream == null)
            return string.Empty;
        using (var reader = new StreamReader(responseStream))
            return await reader.ReadToEndAsync();
    }
}
```

Usage:

```
var response = await uploader.UploadFile("< YOUR URL >", "< PATH TO YOUR FILE >",
    new Dictionary<string, object>
    {
        {"Comment", "test"},
        {"Modified", DateTime.Now }
```

```
    });
```

Chapter 54: Networking

Section 54.1: Basic TCP chat (TcpListener, TcpClient, NetworkStream)

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpChat
{
    static void Main(string[] args)
    {
        if(args.Length == 0)
        {
            Console.WriteLine("Basic TCP chat");
            Console.WriteLine();
            Console.WriteLine("Usage:");
            Console.WriteLine("tcpchat server <port>");
            Console.WriteLine("tcpchat client <url> <port>");
            return;
        }

        try
        {
            Run(args);
        }
        catch(IOException)
        {
            Console.WriteLine("--- Connection lost");
        }
        catch(SocketException ex)
        {
            Console.WriteLine("--- Can't connect: " + ex.Message);
        }
    }

    static void Run(string[] args)
    {
        TcpClient client;
        NetworkStream stream;
        byte[] buffer = new byte[256];
        var encoding = Encoding.ASCII;

        if(args[0].StartsWith("s", StringComparison.InvariantCultureIgnoreCase))
        {
            var port = int.Parse(args[1]);
            var listener = new TcpListener(IPAddress.Any, port);
            listener.Start();
            Console.WriteLine("--- Waiting for a connection...");
            client = listener.AcceptTcpClient();
        }
        else
        {
            var hostName = args[1];
            var port = int.Parse(args[2]);
            client = new TcpClient();
            client.Connect(hostName, port);
```

```
        }

        stream = client.GetStream();
        Console.WriteLine("--- Connected. Start typing! (exit with Ctrl-C)");

        while(true)
        {
            if(Console.KeyAvailable)
            {
                var lineToSend = Console.ReadLine();
                var bytesToSend = encoding.GetBytes(lineToSend + "\r\n");
                stream.Write(bytesToSend, 0, bytesToSend.Length);
                stream.Flush();
            }

            if (stream.DataAvailable)
            {
                var receivedBytesCount = stream.Read(buffer, 0, buffer.Length);
                var receivedString = encoding.GetString(buffer, 0, receivedBytesCount);
                Console.Write(receivedString);
            }
        }
    }
}
```

Section 54.2: Basic SNTP client (UdpClient)

See RFC 2030 for details on the SNTP protocol.

```
using System;
using System.Globalization;
using System.Linq;
using System.Net;
using System.Net.Sockets;

class SntpClient
{
    const int SntpPort = 123;
    static DateTime BaseDate = new DateTime(1900, 1, 1);

    static void Main(string[] args)
    {
        if(args.Length == 0) {
            Console.WriteLine("Simple SNTP client");
            Console.WriteLine();
            Console.WriteLine("Usage: sntpclient <sntp server url> [<local timezone>]");
            Console.WriteLine();
            Console.WriteLine("<local timezone>: a number between -12 and 12 as hours from UTC");
            Console.WriteLine("(append .5 for an extra half an hour)");
            return;
        }

        double localTimeZoneInHours = 0;
        if(args.Length > 1)
            localTimeZoneInHours = double.Parse(args[1], CultureInfo.InvariantCulture);

        var udpClient = new UdpClient();
        udpClient.Client.ReceiveTimeout = 5000;

        var sntpRequest = new byte[48];
        sntpRequest[0] = 0x23; //LI=0 (no warning), VN=4, Mode=3 (client)
```

```
        udpClient.Send(
            dgram: sntpRequest,
            bytes: sntpRequest.Length,
            hostname: args[0],
            port: SntpPort);

        byte[] sntpResponse;
        try
        {
            IPEndPoint remoteEndpoint = null;
            sntpResponse = udpClient.Receive(ref remoteEndpoint);
        }
        catch(SocketException)
        {
            Console.WriteLine("*** No response received from the server");
            return;
        }

        uint numberOfSeconds;
        if(BitConverter.IsLittleEndian)
            numberOfSeconds = BitConverter.ToUInt32(
                sntpResponse.Skip(40).Take(4).Reverse().ToArray()
                ,0);
        else
            numberOfSeconds = BitConverter.ToUInt32(sntpResponse, 40);

        var date = BaseDate.AddSeconds(numberOfSeconds).AddHours(localTimeZoneInHours);

        Console.WriteLine(
            $"Current date in server: {date:yyyy-MM-dd HH:mm:ss}
UTC{localTimeZoneInHours:+0.#;-0.#;.}");
    }
}
```

Chapter 55: HTTP servers

Section 55.1: Basic read-only HTTP file server (ASP.NET Core)

1 - Create an empty folder, it will contain the files created in the next steps.

2 - Create a file named `project.json` with the following content (adjust the port number and `rootDirectory` as appropriate):

```
{
  "dependencies": {
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
    "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final"
  },

  "commands": {
    "web": "Microsoft.AspNet.Server.Kestrel --server.urls http://localhost:60000"
  },

  "frameworks": {
    "dnxcore50": { }
  },

  "fileServer": {
    "rootDirectory": "c:\\users\\username\\Documents"
  }
}
```

3 - Create a file named `Startup.cs` with the following code:

```
using System;
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.FileProviders;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.StaticFiles;
using Microsoft.Extensions.Configuration;

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        var builder = new ConfigurationBuilder();
        builder.AddJsonFile("project.json");
        var config = builder.Build();
        var rootDirectory = config["fileServer:rootDirectory"];
        Console.WriteLine("File server root directory: " + rootDirectory);

        var fileProvider = new PhysicalFileProvider(rootDirectory);

        var options = new StaticFileOptions();
        options.ServeUnknownFileTypes = true;
        options.FileProvider = fileProvider;
        options.OnPrepareResponse = context =>
        {
            context.Context.Response.ContentType = "application/octet-stream";
            context.Context.Response.Headers.Add(
                "Content-Disposition",
                $"Attachment; filename=\"{context.File.Name}\"");
        };
```

```
        app.UseStaticFiles(options);
    }
}
```

4 - Open a command prompt, navigate to the folder and execute:

```
dnvm use 1.0.0-rc1-final -r coreclr -p
dnu restore
```

Note: These commands need to be run only once. Use `dnvm list` to check the actual number of the latest installed version of the core CLR.

5 - Start the server with: `dnx web`. Files can now be requested at `http://localhost:60000/path/to/file.ext`.

For simplicity, filenames are assumed to be all ASCII (for the filename part in the Content-Disposition header) and file access errors are not handled.

Section 55.2: Basic read-only HTTP file server (HttpListener)

Notes:

This example must be run in administrative mode.

Only one simultaneous client is supported.

For simplicity, filenames are assumed to be all ASCII (for the *filename* part in the *Content-Disposition* header) and file access errors are not handled.

```
using System;
using System.IO;
using System.Net;

class HttpFileServer
{
    private static HttpListenerResponse response;
    private static HttpListener listener;
    private static string baseFilesystemPath;

    static void Main(string[] args)
    {
        if (!HttpListener.IsSupported)
        {
            Console.WriteLine(
                "*** HttpListener requires at least Windows XP SP2 or Windows Server 2003.");
            return;
        }

        if(args.Length < 2)
        {
            Console.WriteLine("Basic read-only HTTP file server");
            Console.WriteLine();
            Console.WriteLine("Usage: httpfileserver <base filesystem path> <port>");
            Console.WriteLine("Request format: http://url:port/path/to/file.ext");
            return;
        }

        baseFilesystemPath = Path.GetFullPath(args[0]);
        var port = int.Parse(args[1]);
```

```
        listener = new HttpListener();
        listener.Prefixes.Add("http://*:" + port + "/");
        listener.Start();

        Console.WriteLine("--- Server stated, base path is: " + baseFilesystemPath);
        Console.WriteLine("--- Listening, exit with Ctrl-C");
        try
        {
            ServerLoop();
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex);
            if(response != null)
            {
                SendErrorResponse(500, "Internal server error");
            }
        }
    }

    static void ServerLoop()
    {
        while(true)
        {
            var context = listener.GetContext();

            var request = context.Request;
            response = context.Response;
            var fileName = request.RawUrl.Substring(1);
            Console.WriteLine(
                "--- Got {0} request for: {1}",
                request.HttpMethod, fileName);

            if (request.HttpMethod.ToUpper() != "GET")
            {
                SendErrorResponse(405, "Method must be GET");
                continue;
            }

            var fullFilePath = Path.Combine(baseFilesystemPath, fileName);
            if(!File.Exists(fullFilePath))
            {
                SendErrorResponse(404, "File not found");
                continue;
            }

            Console.Write("    Sending file...");
            using (var fileStream = File.OpenRead(fullFilePath))
            {
                response.ContentType = "application/octet-stream";
                response.ContentLength64 = (new FileInfo(fullFilePath)).Length;
                response.AddHeader(
                    "Content-Disposition",
                    "Attachment; filename=\"" + Path.GetFileName(fullFilePath) + "\"");
                fileStream.CopyTo(response.OutputStream);
            }

            response.OutputStream.Close();
            response = null;
            Console.WriteLine(" Ok!");
        }
    }
```

```
    static void SendErrorResponse(int statusCode, string statusResponse)
    {
        response.ContentLength64 = 0;
        response.StatusCode = statusCode;
        response.StatusDescription = statusResponse;
        response.OutputStream.Close();
        Console.WriteLine("*** Sent error: {0} {1}", statusCode, statusResponse);
    }
}
```

Chapter 56: HTTP clients

Section 56.1: Reading GET response as string using System.Net.HttpClient

`HttpClient` is available through NuGet: Microsoft HTTP Client Libraries.

```
string requestUri = "http://www.example.com";
string responseData;

using (var client = new HttpClient())
{
    using(var response = client.GetAsync(requestUri).Result)
    {
       response.EnsureSuccessStatusCode();
       responseData = response.Content.ReadAsStringAsync().Result;
    }
}
```

Section 56.2: Basic HTTP downloader using System.Net.Http.HttpClient

```
using System;
using System.IO;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

class HttpGet
{
    private static async Task DownloadAsync(string fromUrl, string toFile)
    {
        using (var fileStream = File.OpenWrite(toFile))
        {
            using (var httpClient = new HttpClient())
            {
                Console.WriteLine("Connecting...");
                using (var networkStream = await httpClient.GetStreamAsync(fromUrl))
                {
                    Console.WriteLine("Downloading...");
                    await networkStream.CopyToAsync(fileStream);
                    await fileStream.FlushAsync();
                }
            }
        }
    }

    static void Main(string[] args)
    {
        try
        {
            Run(args).Wait();
        }
        catch (Exception ex)
        {
            if (ex is AggregateException)
                ex = ((AggregateException)ex).Flatten().InnerExceptions.First();
```

```
            Console.WriteLine("--- Error: " +
                (ex.InnerException?.Message ?? ex.Message));
        }
    }
    static async Task Run(string[] args)
    {
        if (args.Length < 2)
        {
            Console.WriteLine("Basic HTTP downloader");
            Console.WriteLine();
            Console.WriteLine("Usage: httpget <url>[<:port>] <file>");
            return;
        }

        await DownloadAsync(fromUrl: args[0], toFile: args[1]);

        Console.WriteLine("Done!");
    }
}
```

Section 56.3: Reading GET response as string using System.Net.HttpWebRequest

```
string requestUri = "http://www.example.com";
string responseData;

HttpWebRequest request = (HttpWebRequest)WebRequest.Create(parameters.Uri);
WebResponse response = request.GetResponse();

using (StreamReader responseReader = new StreamReader(response.GetResponseStream()))
{
    responseData = responseReader.ReadToEnd();
}
```

Section 56.4: Reading GET response as string using System.Net.WebClient

```
string requestUri = "http://www.example.com";
string responseData;

using (var client = new WebClient())
{
    responseData = client.DownloadString(requestUri);
}
```

Section 56.5: Sending a POST request with a string payload using System.Net.HttpWebRequest

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

HttpWebRequest request = (HttpWebRequest)WebRequest.Create(requestUri)
{
  Method = requestMethod,
  ContentType = contentType,
};
```

```
byte[] bytes = Encoding.UTF8.GetBytes(requestBodyString);
Stream stream = request.GetRequestStream();
stream.Write(bytes, 0, bytes.Length);
stream.Close();

HttpWebResponse response = (HttpWebResponse)request.GetResponse();
```

Section 56.6: Sending a POST request with a string payload using System.Net.WebClient

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

byte[] responseBody;
byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBodyString);

using (var client = new WebClient())
{
    client.Headers[HttpRequestHeader.ContentType] = contentType;
    responseBody = client.UploadData(requestUri, requestMethod, requestBodyBytes);
}
```

Section 56.7: Sending a POST request with a string payload using System.Net.HttpClient

`HttpClient` is available through NuGet: Microsoft HTTP Client Libraries.

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

var request = new HttpRequestMessage
{
    RequestUri = requestUri,
    Method = requestMethod,
};

byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBodyString);
request.Content = new ByteArrayContent(requestBodyBytes);

request.Content.Headers.ContentType = new MediaTypeHeaderValue(contentType);

HttpResponseMessage result = client.SendAsync(request).Result;
result.EnsureSuccessStatusCode();
```

Chapter 57: Serial Ports

Section 57.1: Basic operation

```
var serialPort = new SerialPort("COM1", 9600, Parity.Even, 8, StopBits.One);
serialPort.Open();
serialPort.WriteLine("Test data");
string response = serialPort.ReadLine();
Console.WriteLine(response);
serialPort.Close();
```

Section 57.2: List available port names

```
string[] portNames = SerialPort.GetPortNames();
```

Section 57.3: Asynchronous read

```
void SetupAsyncRead(SerialPort serialPort)
{
    serialPort.DataReceived += (sender, e) => {
        byte[] buffer = new byte[4096];
        switch (e.EventType)
        {
            case SerialData.Chars:
                var port = (SerialPort)sender;
                int bytesToRead = port.BytesToRead;
                if (bytesToRead > buffer.Length)
                    Array.Resize(ref buffer, bytesToRead);
                int bytesRead = port.Read(buffer, 0, bytesToRead);
                // Process the read buffer here
                // ...
                break;
            case SerialData.Eof:
                // Terminate the service here
                // ...
                break;
        }
    };
```

Section 57.4: Synchronous text echo service

```
using System.IO.Ports;

namespace TextEchoService
{
    class Program
    {
        static void Main(string[] args)
        {
            var serialPort = new SerialPort("COM1", 9600, Parity.Even, 8, StopBits.One);
            serialPort.Open();
            string message = "";
            while (message != "quit")
            {
                message = serialPort.ReadLine();
                serialPort.WriteLine(message);
            }
```

```
            serialPort.Close();
        }
    }
}
```

Section 57.5: Asynchronous message receiver

```
using System;
using System.Collections.Generic;
using System.IO.Ports;
using System.Text;
using System.Threading;

namespace AsyncReceiver
{
    class Program
    {
        const byte STX = 0x02;
        const byte ETX = 0x03;
        const byte ACK = 0x06;
        const byte NAK = 0x15;
        static ManualResetEvent terminateService = new ManualResetEvent(false);
        static readonly object eventLock = new object();
        static List<byte> unprocessedBuffer = null;

        static void Main(string[] args)
        {
            try
            {
                var serialPort = new SerialPort("COM11", 9600, Parity.Even, 8, StopBits.One);
                serialPort.DataReceived += DataReceivedHandler;
                serialPort.ErrorReceived += ErrorReceivedHandler;
                serialPort.Open();
                terminateService.WaitOne();
                serialPort.Close();
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception occurred: {0}", e.Message);
            }
            Console.ReadKey();
        }

        static void DataReceivedHandler(object sender, SerialDataReceivedEventArgs e)
        {
            lock (eventLock)
            {
                byte[] buffer = new byte[4096];
                switch (e.EventType)
                {
                    case SerialData.Chars:
                        var port = (SerialPort)sender;
                        int bytesToRead = port.BytesToRead;
                        if (bytesToRead > buffer.Length)
                            Array.Resize(ref buffer, bytesToRead);
                        int bytesRead = port.Read(buffer, 0, bytesToRead);
                        ProcessBuffer(buffer, bytesRead);
                        break;
                    case SerialData.Eof:
                        terminateService.Set();
                        break;
```

```
                }
            }
        }
        static void ErrorReceivedHandler(object sender, SerialErrorReceivedEventArgs e)
        {
            lock (eventLock)
                if (e.EventType == SerialError.TXFull)
                {
                    Console.WriteLine("Error: TXFull. Can't handle this!");
                    terminateService.Set();
                }
                else
                {
                    Console.WriteLine("Error: {0}. Resetting everything", e.EventType);
                    var port = (SerialPort)sender;
                    port.DiscardInBuffer();
                    port.DiscardOutBuffer();
                    unprocessedBuffer = null;
                    port.Write(new byte[] { NAK }, 0, 1);
                }
        }

        static void ProcessBuffer(byte[] buffer, int length)
        {
            List<byte> message = unprocessedBuffer;
            for (int i = 0; i < length; i++)
                if (buffer[i] == ETX)
                {
                    if (message != null)
                    {
                        Console.WriteLine("MessageReceived: {0}",
                            Encoding.ASCII.GetString(message.ToArray()));
                        message = null;
                    }
                }
                else if (buffer[i] == STX)
                    message = null;
                else if (message != null)
                    message.Add(buffer[i]);
            unprocessedBuffer = message;
        }
    }
}
```

This program waits for messages enclosed in `STX` and `ETX` bytes and outputs the text coming between them. Everything else is discarded. On write buffer overflow it stops. On other errors it reset input and output buffers and waits for further messages.

The code illustrates:

- Asynchronous serial port reading (see `SerialPort.DataReceived` usage).
- Serial port error processing (see `SerialPort.ErrorReceived` usage).
- Non-text message-based protocol implementation.
- Partial message reading.
    - The `SerialPort.DataReceived` event may happen earlier than entire message (up to `ETX`) comes. The entire message may also not be available in the input buffer (SerialPort.Read(..., ..., port.BytesToRead) reads only a part of the message). In this case we stash the received part (`unprocessedBuffer`) and carry on waiting for further data.
- Dealing with several messages coming in one go.

- The `SerialPort.DataReceived` event may happen only after several messages have been sent by the other end.

Appendix A: Acronym Glossary

Section A.1: .Net Related Acronyms

Please note that some terms like JIT and GC are generic enough to apply to many programming language environments and runtimes.

CLR: Common Language Runtime
IL: Intermediate Language
EE: Execution Engine
JIT: Just-in-time compiler
GC: Garbage Collector
OOM: Out of memory
STA: Single-threaded apartment
MTA: Multi-threaded apartment

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

Adi Lester	Chapters 42 and 47
Adil Mammadov	Chapter 8
Adriano Repetti	Chapters 1, 2, 4 and 18
Akshay Anand	Chapter 25
Alan McBee	Chapters 12, 14 and 47
ale10ander	Chapters 16 and 35
Aleks Andreev	Chapters 10, 52 and 53
Alexander Mandt	Chapter 49
Alexander V.	Chapter 8
Alfred Myers	Chapter 47
Aman Sharma	Chapters 5 and 42
Andrew Jens	Chapter 1
Andrew Morton	Chapter 25
Andrey Shchekin	Chapter 28
Andrius	Chapter 32
Anik Saha	Chapter 28
Aphelion	Chapter 34
Arvin Baccay	Chapter 47
Arxae	Chapter 19
Ashtonian	Chapter 28
Athari	Chapter 1
avat	Chapter 46
Axarydax	Chapter 51
BananaSft	Chapter 47
Bassie	Chapter 48
Behzad	Chapter 39
Benjamin Hodgson	Chapter 8
binki	Chapter 45
Bjørn	Chapters 4 and 10
Bradley Grainger	Chapter 8
Bruno Garcia	Chapter 8
BrunoLM	Chapter 15
Carlos Muñoz	Chapter 8
CodeCaster	Chapters 8, 17, 28, 47 and 56
Daniel A. White	Chapters 1 and 25
Darrel Lee	Chapter 4
Dave R.	Chapter 47
dbasnett	Chapter 35
delete me	Chapter 18
demonplus	Chapters 30 and 49
Denuath	Chapter 15
DLeh	Chapters 31 and 33
Dmitry Egorov	Chapters 27 and 57
DoNot	Chapter 8
Dr Rob Lang	Chapter 9
Drew	Chapter 25
DrewJordan	Chapters 12 and 45

Eduardo Molteni	Chapter 8
Ehsan Sajjad	Chapter 8
Eric	Chapter 32
Felipe Oriani	Chapters 4 and 5
Filip Frącz	Chapter 17
Fredou	Chapter 48
Gajendra	Chapter 37
GalacticCowboy	Chapters 3 and 8
George Polevoy	Chapters 4, 11 and 34
Guanxi	Chapter 22
Gusdor	Chapters 43 and 44
Haney	Chapter 8
harriyott	Chapter 5
hatchet	Chapter 4
Heinzi	Chapter 25
Hogan	Chapter 4
Hywel Rees	Chapter 7
i3arnon	Chapter 38
Ian	Chapter 4
Igor	Chapter 25
Ingenioushax	Chapter 16
Jacobr365	Chapters 38, 42 and 43
Jagadisha B S	Chapter 49
JamyRyals	Chapter 42
jbtule	Chapter 8
Jigar	Chapter 10
Jim	Chapter 11
jnovo	Chapter 8
Joe Amenta	Chapters 8 and 20
John	Chapter 3
Kevin Montrose	Chapter 1
Konamiman	Chapters 8, 42, 54, 55 and 56
Krikor Ailanjian	Chapter 36
Kritner	Chapter 47
lokusking	Chapter 49
Lorenzo Dematté	Chapter 10
Luaan	Chapter 23
Lucas Trzesniewski	Chapter 9
M22an	Chapter 32
Mafii	Chapter 47
mahdi abasi	Chapter 50
MarcinJuraszek	Chapters 1 and 8
Mark C.	Chapter 5
Matas Vaitkevicius	Chapters 10 and 28
Mathias Müller	Chapter 42
Matt	Chapter 49
Matt dc	Chapter 15
matteeyah	Chapter 1
Matthew Whited	Chapter 13
McKay	Chapter 8
Mellow	Chapter 39
Mihail Stancescu	Chapter 24
Mr.Mindor	Chapter 8

MSE	Chapter 40
n.podbielski	Chapter 11
Nate Barbettini	Chapter 8
Nikola.Lukovic	Chapter 38
NikolayKondratyev	Chapter 23
Ogglas	Chapter 48
Ondřej Štorc	Chapter 48
Ozair Kafray	Chapter 28
Pavel Mayorov	Chapter 42
Pavel Voronin	Chapters 2 and 42
PedroSouki	Chapter 32
ProgramFOX	Chapter 21
Ringil	Chapter 4
Rion Williams	Chapter 1
Robert Columbia	Chapter 4
RoyalPotato	Chapter 17
Ruben Steins	Chapter 8
Salvador Rubio Martinez	Chapter 8
Sammi	Chapter 8
Scott Hannen	Chapters 26 and 29
SeeuD1	Chapter 1
Sergio Domínguez	Chapter 8
Sidewinder94	Chapter 8
smdrager	Chapter 10
starbeamrainbowlabs	Chapters 37 and 47
Steve	Chapter 30
Steven Doggart	Chapter 1
Stilgar	Chapter 11
Tanveer Badar	Chapter 58
tehDorf	Chapters 6 and 15
the berserker	Chapter 4
Theodoros Chatzigiannakis	Chapter 37
Thomas Bledsoe	Chapter 42
Thriggle	Chapter 32
toddmo	Chapter 23
Tolga Evcimen	Chapter 32
Tomáš Hübelbauer	Chapter 4
user2321864	Chapter 25
vicky	Chapter 30
wangengzheng	Chapter 11
Yahfoufi	Chapter 41
ɹoleǝz ǝɥʇ qoq	Chapter 1

You may also like