C# Quick Ref

Review notes for the most often forgotten things

Compiled by Bruce Hantover -- August 2016 (ver 823)

Contents

1. History

C# 2.0 features (2005)

► Generics, Partial types, Anonymous methods, Iterators, Nullable types, Getter/setter separate accessibility, Method group conversions (delegates), Static classes, Delegate inference

C# 3.0 features (2007)

► Language Integrated Queries (LINQ), Implicitly typed local variables, Object and collection initializers, Auto-Implemented properties, Anonymous types, Extension methods, Query expressions, Lambda expressions, Expression trees, Partial methods

C# 4.0 features (2010)

► Dynamic programming, Named and optional parameters, Covariance and Contravariance

C# 5.0 features (2012)

► Asynchronous methods, Caller info attributes

C# 6.0 features (July 2015, .Net 4.6)

► Exception filters, finally, interpolated strings, auto-property initializers, accessors with different visibility, nameof operator Dictionary initializer

C# 7.0 features (planned)

► Tuples, local functions (defined within a method)

2. Control Structures

```
if (condition)
{
  // Do something
}
else if
{
  // Do something else
}
else
{
  // Do something else
}
```

```
for (int i = 0; i < 100; i++)
{
    // Do something
}
```

```
foreach (string item in itemsToWrite)
{
    // Do something
}
```

// Break exits from for, foreach, while, do .. while loops

```
for (i = 0; i < 10; i++) // see the comparison, i < 10
{
    if (i >= 3)
    {
        break;  // leaves the loop instantly
    }
}
```

// continue skips rest of loop, goes back to condition test

```
while (condition) {  Do something }
```

► do ... while does first execution *before* comparison

```
do { Do something }
while (condition);
```

```
try
{
   // Do something
}
catch(Exception e)
{
    Console.WriteLine(e.Message);
}
finally
{
   // Do something
}

throw new Exception("blah, blah, blah ...");
```

```
switch (myString)
{
   case "abc":
      // Do something
      break;
   case "def":
      // Do something
      break;
   default:
      // Do something
      break;
}
```

Conditional Operator

t ? x : y – if test t is true, then evaluate and return x;
otherwise, evaluate and return y.

(continued next page)

3. Data Structures

Value types (stored in the **stack**)

The value types in the .NET framework are usually small, frequently used types. The benefit of using them is that the type requires very little resources to get up and running by the CLR. Value types do not require memory to be allocated on the heap and therefore will not cause garbage collection. However, in order to be useful, the value types (or types derived from it) should remain small - ideally below 16 bytes of data.

►Value types are always copied (intrinsically) before being passed to a method. Changes to this new object will not be reflected back in the original object passed into the method.
►Value types do not /need/ you to call their constructor. They are automatically initialized.
►Value types **always initialize their fields** to 0 or null.
►Value types can NEVER be assigned a value of null (unless they are declared as Nullable Yypes)
►Value types sometimes need to be boxed (wrapped inside an object), allowing their values to be used like objects.

Reference types (stored in the **heap**)

Reference types are managed very differently by the CLR. All reference types consist of two parts: A pointer to the heap (which contains the object), and the object itself. Reference types are slightly heavier weight because of the management behind the scenes needed to keep track of them. Reference types are nullable.

When an object is initialized, by use of the constructor, and is of a reference type, the CLR must perform four operations:

(1) The CLR calculates the amount of memory required to hold the object on the heap. **(2)** The CLR inserts the data into the newly created memory space. **(3)** The CLR marks where the end of the space lies, so that the next object can be placed there. **(4)** The CLR returns a reference to the newly created space.

This occurs every single time an object is created. However the assumption is that there is infinite memory, therefore some maintenance needs to take place - and that's where the garbage collector comes in.

Nullable Types -- If x.HasValue is true, it is not null

```
int? x = 10;      x = null;      int?[] arr = new int?[10];
int y = x;              //-- will not compile
int y = (int)x;         //-- compiles, but throws exception if x null
int y = x.Value;        //-- same as above
```

►If you perform an operation with null, the result will be null

?? operator -- defines default assignment if target is null:

```
int d = c ?? -1; // d = c, unless c is null, then d = -1
int g = e ?? f ?? -1 // g = e, if e null, then f, or -1
```

The built-in C# type aliases and their equivalent .NET Framework types follow:

Integers

C# Alias	Type	(bits)	Range
sbyte	SByte	8	-128 to 127
byte	Byte	8	0 to 255
short	Int16	16	-32,768 to 32,767
ushort	UInt16	16	0 to 65,535
char	Char	16	unicode 0 to 65,535
int	Int32	32	- + 2 billion
uint	UInt32	32	0 to 4 billion
long	Int64	64	- + 9 quintillion to
ulong	UInt64	64	0 to 18 quintillion

Floating-point

C# Alias	.NET Type	(bits)	Precision	Range
float	Single	32	7 digits	E -45 to 38
double	Double	64	15-16 digits	E -324, 308
decimal	Decimal	128	28-29 decimal	- + E28

Other predefined types

C# Alias	.NET Type	(bits)	Range
bool	Boolean	32	true or false
object	Object	32/64	a pointer to an object
string	String	16*length	unicode, no upper bound.

enum Weekday { Monday, Tuesday, ... Sunday };
Weekday day = Weekday.Monday;

struct is a light-weight object (value type), classes are ref types. Structs work best if < 17 bytes of data. If in doubt, use classes.

```
struct Person
{
    public string name;
    public int height;
}
```

Person dana = new Person(); dana.name = "Dana Dev";

```
struct Person // with constructor
 {
    string name;   int height;

    public Person(string name, int height)
    {
       this.name = name;
       this.height = height;
    }
 }
```

```
public class StructWikiBookSample
{
   public static void Main()
   {
      Person dana = new Person("Dana Developer", 60) ;
   }
}
```

4. Collections

Arrays --size can be set by variable but can't change

```
int[] integers = new int[20];
double[] doubles = new double[myVariable];
```

► **Multidimensional**

```
int[,] test = new int[2, 3];
```

► **Initialized at time of creation**

```
int[] arr = new int[] { 24, 2, 13, 47, 45 };
```

► Arrays are passed by reference, not by value
► Arrays size will be arr.Length

```
// example: to copy first 3 elements of array:
var sourceArray = new int[] { 11, 12, 3, 5, 2, 9, 28, 17 };
var destinationArray= new int[3];
Array.Copy(sourceArray, 0, destinationArray, 0, 3 );
```

Array List

► ArrayList **before generics** used boxing / unboxing. This was slow & mix-and-match was not good practice.

```
ArrayList myAL = new ArrayList();
myAL.Add(1);
myAL.Add("foo");
```

List (generic) (IEnumerable, code used inside a method)

```
List<string> list = new List<string>();
list.Add("Bruce is cool");
list.Add("Yes he is");
string s = list[1];
int len = list.Count;
```

List methods include: Sort, IndexOf, Insert, Reverse, AddRange, RemoveRange, RemoveAt, etc

//-- create Array, convert to List, convert back

```
string[] myArr = { "dog", "zebra", "ant", "parrot" };
List<string> list = new List<string>(myArr);
string[] myArr2 = list.ToArray() as string[];
```

//-- sort Array and List

```
Array.Sort(myArr);
list.Sort();
```

(Continued from 4. Collections)

Generic classes (Better to use generics)

Generic class	Non-generic counterpart	Meaning
Collection<T>	CollectionBase	The basis for the collections
Comparer<T>	Comparer	Compares two objects for equality
Dictionary<K,V>	Hashtable	A collection of name-value pair
List<T>	ArrayList	A dynamic resizable list of items
Queue<T>	Queue	FIFO list
Stack<T>	Stack	LILO list

LinkedList -- allows constant time for insert or remove, but only sequential access to elements

```
LinkedList<string> LL = new LinkedList<string>();

LL.AddLast("aaa");
LL.AddFirst("bbb");
LL.AddLast("ccc");
LL.AddLast("ddd");

LinkedListNode<string> node = LL.Find("ccc");
LL.AddBefore(node, "hello there beautiful");
```

Other methods include: AddAfter, Average, Contains, Distinct, Find, FindLast, Max, Min, OrderBy, Remove, ToList, etc

Queue -- First In, First Out (FIFO)

```
Queue<string> q = new Queue<string>();
q.Enqueue("aaa");
q.Enqueue("bbb");
q.Enqueue("ccc");
WriteLine(q.Dequeue()); //-- removes and returns
WriteLine(q.Peek()); //-- returns, does not remove
```

Stack -- Last In, First Out (LIFO)

```
Stack<string> stack = new Stack<string>();
stack.Push("xxx");
stack.Push("yyy");
stack.Push("zzz");
WriteLine(stack.Pop());   //-- removes and returns
WriteLine(stack.Peek()); //-- return, don't remove
```

Dictionary -- key/value pairs (keys are unique)

► A dictionary is a hash table where the details and complexity are hidden. Hash tables are faster than search trees or other lookup structures. Why is this?

Let's assume you want to fill up a library with books and not just stuff them in there, but you want to be able to easily find them again when you need them. So, you decide that if the person that wants to read a book knows the title of the book, the person, with the aid of the librarian, should be able to find the book easily and quickly.

So, how can you do that? Well, obviously you can keep some kind of list of where you put each book, but then you have the same problem as searching the library, you need to search the list. Granted, the list would be smaller and easier to search, but still you don't want to search sequentially from one end of the library (or list) to the other.

You want something that, with the title of the book, can give you the right spot at once, so all you have to do is just stroll over to the right shelf, and pick up the book. So you devise a clever method: You take the title of the book, run it through a small computer program, which spits out a shelf number and a slot number on that shelf. This is where you place the book.

```
Dictionary<string, string> dict
                = new Dictionary<string, string>();

dict.Add("de", "Germany");
dict.Add("es", "Spain");
dict.Add("uk", "Britain");
dict.Add("fr", "France");
dict.Add("cn", "China");

WriteLine(dict["cn"]);  //-- shows China
WriteLine(dict["fr"]);  //-- shows France
```

► Below are examples of what you can do to filter, sort and view dictionary keys and values.

(continued from Dictionary)

```
//--- show key/value pairs

foreach (KeyValuePair<string, string> kvp in dict)
{
    WriteLine(kvp.Key + ", " + kvp.Value);
}

//--- Create lists using dict keys or values

var list1 = dict.Keys.ToList();
var list2 = dict.Values.ToList();
foreach (string s in list1) { WriteLine (s); }
foreach (string s in list2) { WriteLine (s); }
list1.Sort(); //-- easy to sort

//-- Sort using LINQ
var items = from x in dict orderby x.Value select x;
foreach(KeyValuePair<string, string> pair in items)
{ WriteLine (pair.Key + ", " + pair.Value); }

//-- Sort using Lamda
foreach (var pair in dict.OrderBy(p => p.Key))
{ WriteLine(pair); }

//-- update value for a given key
dict["cn"] = "People's Republic of China";

//-- xx
x
```

Don't bother with:

(1) any of the non-generic collections or
(2) SortedDictionary, SortedList, SortedSet or
(3) Concurrent versions

► Some of these are similar to dictionary as they make use of key-value pairs, but are sorted or do other things a bit different.

► The non-generic stuff should be considered deprecated. Don't use unless necessary for maint work on old code.

► Also--don't bother with Concurrent versions of these collections unless you are using multiple threads

5. Classes

► Fields can be assigned in class declarations (compiler will automatically move initialization to constructor)
► Below is class with **constructors**

```
public class Employee
{
   public Employee()
   {
      Console.WriteLine("Constructed without parameters");
   }

   public Employee(string strText)
   {
      Console.WriteLine(strText);
   }
}
```

► Constructors can call each other:

```
public class Employee
{
   public Employee(string text, int nbr)   { ... }

   // calls above with user text and default nbr
   public Employee(string text) : this(text, 1234) { ... }

   // calls above with default text
   public Employee() : this("hello")     { ... }
}
```

this is used to refer to member of the current instance of class. It cannot refer to static field or method or be inside a static class. It is often not required and may be infered by the compiler, but it clarifies intent to human readers.

base is used in constructors of a child class to refer to the instance of the parent class

```
class A {public string myStr = "hello"; }

class B : A
{
     public string myStr = "world";
     public void MyWrite()
     { Console.WriteLine( base.myStr + " " + this.myStr ) }
}
```

```
public class B : A
{
     // constructor for class B (constructor for A executed first)
     public B () : base () {  }
}
```

static vs. instantiated

► fields, methods and classes can be static
► static class can't be instantiated or derived from
► static class can contain only static methods and fields
► static is good enough if the class holds no state info
► a static method can't access non-static class members

```
Car ford = new Car(); // instantiating a non-static class
MyStaticClass.MyMethod(); // invoking method from static class
```

encapsulation -- restrict access for safety --

public / private -- private methods and fields can only be accessed from other class methods, not from outside the class. Classes, methods and fields are **private** by default.
Enums and structs are **public** by default.

protected -- like private except derived classes can access

internal -- restricted, only members of the class can access. An *internal class* is not accessible from an external program. (This is only needed in larger, more complex programs)

protected internal -- combination of protected OR internal

sealed -- class that cannot be derived from

inheritance (**derived** class B **inherits** from **parent** A)

```
Class A { ... }
```

```
Class B : A { ... }
```

example:

```
public class BaseClass
{
      public string HelloMessage = "Hello, World!";
}
```

```
public class SubClass : BaseClass
{
      public string ArbitraryMessage = "Uh, Hi!";
}
```

```
public class Test
{
   static void Main()
   {
        SubClass subClass = new SubClass();
        Console.WriteLine(subClass.HelloMessage);
   }
}
```

polymorphism -- overriding what you Inherited

```
public class Animal
{
   public virtual string talk() { return "Hi"; }
   public string sing() { return "lalala"; }
}
```

```
public class Cat : Animal
{
   public override string talk() { return "Meow!"; }
}
```

```
public class Dog : Animal
{
   public override string  talk() { return "Woof!"; }
   public new string sing() { return "woofa woofa woooof"; }
}
```

```
public class Polymorphism
{
   public Polymorphism()
   {
      write(new Cat());
      write(new Dog());
   }
   public void write(Animal a)
   {
      WriteLine(a.talk());
   }
}
```

another example

```
public interface Animal
{
  string Name { get; }
}
```

```
public class Dog : Animal
{
  public string Name { get { return "Dog"; } }
}
```

```
public class Cat : Animal
{
  public string Name { get { return "Cat"; } }
}
```

```
public class Test
{   static void Main()
    {
        Animal animal = new Dog();
        Animal animal2 = new Cat();
        Console.WriteLine(animal.Name);
        Console.WriteLine(animal2.Name);
    }
}
```

(Continued from 5. Classes)

get or **set** class properties using **accessors**

► The variable below is too easy to access from outside the class, this lacks discipline, invites trouble.

```
class Culture
{
   public string Language;
}
```

► Accessors below can be used as gatekeepers or modify read/write, maintain maximum control, etc.

```
class Culture
{
   public string Language { get; set; }
}
```

-- or --

```
class Culture
{
   private string lang;
   public string Language
   {
      get { return "my " + lang; }
      set { lang = value; }
   }
}
```

protected set //-- permission only to derived classes

► As a principle, anything that you can keep from allowing other code to touch, should be kept that way. If other classes will need to view something in your class, you'll need to expose a getter, but not a setter. They also allow you to perform calculations or validations before returning or setting data. For example, if you have some integer value that should be within some range, in your setter you can check to make sure this condition is met before actually updating your value.

► **Virtual** methods are subject to **override**

```
class A
{
   public virtual void Act() { }
}
```

```
class B : A
{
   public override void Act()
   {  Console.WriteLine("Perl.Act"); }
}
```

► **Abstract** classes require a derived class, and cannot be directly instantiated. They can contain abstract methods, which require override methods.

```
abstract class A
{
   public int ok;
   public abstract void MyMethod();
}
```

```
class B : A
{
   public override void MyMethod()
   {
         Console.WriteLine("this is cool");
         base.ok++;
   }
}
```

interface -- Similar to abstract class. Contains classes, methods and properties (but not fields). Widely used (behind the scenes) to implement work with IEnumerable and foreach. Don't worry about it.

Extension methods -- Custom extension methods are very cool ! You create a **static** class that uses the **this** keyword, and it acts like an instance method for whatever you want to extend. See example below.

Generally, you will want to store your extension method class in a separate source file, such as "ExtensionMethods.cs" in your project.

```
string s = "dot net perls";
s = s.value.UppercaseFirstLetter();
```

(Continued next column)

(continued from Extension Methods)

```
public static class ExtensionMethods
{
   public static string UppercaseFirstLetter(this string value)
   {
         if (value.Length > 0)
         {
            char[] array = value.ToCharArray();
            array[0] = char.ToUpper(array[0]);
            return new string(array);
         }
         return value;
   }
}
```

partial -- partial class can be split across more than one file

nested -- like it sounds, one class nested inside another

```
class A
{
   public int v1;

   public class B  { public v2; }
}
```

```
class Program
{
   static void Main()
   {
         A a = new A();
         a.v1++;

         A.B ab = new A.B();
         ab.v2++;
   }
}
```

(continued next page)

6. LINQ and Lambda

► **This section is not complete, or very good. If I revise these notes--this section will be my priority.**

LINQ

---- examples ----

-- 1 --

```
List<int> elements = new List<int>() { 10, 20, 31, 40 };
// ... Find index of first odd element.
int oddIndex = elements.FindIndex(x => x % 2 != 0);
```

-- 2 --

```
string[] a = new string[] { "US", "China", "Peru", "UK", "Spain" };
var sort = from s in a orderby s select s;
foreach (string c in sort) { Console.WriteLine(c); }
```

-- Cars example --

```
List<Car> cars=new List<Car>()
{
{new Car() {id=100, color="blue", speed=20, make="Honda"}},
{new Car() {id=101, color="red", speed=30, make="Ford"}},
{new Car() {id=102, color="green", speed=40, make="Honda"}},
{new Car() {id=103, color="blue", speed=40, make="Ford"}},
...
{new Car() {id=111, color="green", speed=10, make="Ford"}}
};

 var subset=    from rows in cars
                where rows.make != "Toyota"
                && rows.speed > 10
                orderby rows.color, rows.speed descending
                select rows;

 List<Car> cars2=new List<Car>();
 foreach (Car car in subset) { cars2.Add(car); }
```

```
public class Car
{
    public int id;
    public string color;
    public int speed;
    public string make;
}
```

-- 4 -- Using Lambda (need better example)

```
public struct Word_Freq { public string word; public int freq; }
List<Word_Freq> wfList = new List<Word_Freq>();
// .. add to and update the wfList ...
wfList.Sort(  (x, y) =>
                ((1000 - x.freq).ToString() + x.word).CompareTo(
                ((1000 - y.freq).ToString() + y.word))
            );
```

Some Lambda notes: **(need something better here)**

A lambda expression is an anonymous function and it is mostly used to create delegates in LINQ. Simply put, it's a method without a declaration (i.e., access modifier, return value declaration, and name). Convenience. It's a shorthand that allows you to write a method in the same place you are going to use it.

//-- Lambda with no inputs: ()=>

```
delegate int del(int i);

static void Main(string[] args)
{
      del del_1 = y => y * y;
      int j = del_1(5);
      WriteLine(j);

      del del_2 = x => x + 2;
      j = del_2(4);
      WriteLine(j);
}
```

More LINQ examples here:

(1) (MSDN) LINQ query examples (a good start)
(includes joins and all kinds of things)
https://msdn.microsoft.com/en-us/library/gg509017.aspx

(2) http://studyoverflow.com/linq/

(3) 50 LINQ Examples, Tips and How To's
http://www.dotnetcurry.com/linq/727/linq-examples-tips-tricks

(4) 101 LINQ SAMPLES
https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b

(5) LINQ 101 Samples - Lambda Style
http://linq101.nilzorblog.com/linq101-lambda.php

(6) Wikipedia
Lots of info about LINQ to XML, SQL and data sets
https://en.wikipedia.org/wiki/Language_Integrated_Query

7. Misc Stuff

Anonymous Methods

-- define functionality of a delegate (such as an event) inline rather than as a separate method.

See: http://csharp-station.com/Tutorial/CSharp/Lesson21

Anonymous Types -- provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first:

```
var v = new { Amount = 108, Message = "Hello" };
Console.WriteLine(v.Amount + v.Message);
```

(Behind the scenes, they are actually classes of type object)

You can return an anonymous object from your function

```
public static object FunctionWithUnknowReturnValues ()
{
    /// anonymous object
    return new { a = 1, b = 2 };
}

dynamic x = FunctionWithUnknowReturnValues();
Console.WriteLine(x.a);
Console.WriteLine(x.b);
```

String functions

► examples: Length, Substring, IndexOf, ToLower, ToUpper, Insert, Split, Replace, Remove, Trim, etc

To sort chars in string

```
static string SortMyString(string s)
{
      char[] myChars = s.ToCharArray();
      Array.Sort(myChars);
      return new string(myChars);
}
```

To replace a section of a string

```
string s = "One step at a time, Bruce will make this easy";

//-- method # 1
int spot = s.IndexOf("Bruce");
int sLen = "Bruce".Length;
string section1 = s.Substring(0, spot);
string section2 = s.Substring(spot + sLen);
string s2 = section1 + "Heather" + section2;

//-- method # 2
string s3 = s.Replace("Bruce", "Heather");
```

Formated String -- example

```
string      v1    = "Dot Net Perls";
int         v2    = 10000;
DateTime v3    = new DateTime(2016, 8, 16);

string result1   = string.Format("{0}, {1}, {2}", v1, v2, v3);
string result2   = string.Format
                      ("{0}: {1:0.0}, {2:yyyy-MM-dd}", v1, v2, v3);
Console.WriteLine(result1);
Console.WriteLine(result2);

Result:
   Dot Net Perls, 10000, 8/16/2016 12:00:00 AM
   Dot Net Perls: 10000.0, 2016-08-16
```

String split example

```
string[] words
     = text.Split(' ', '.', ',', '-');
```

StringBuilder -- A more mutable form of string, eliminates auto creation, copy and reassign each time you modify it.

File I/O (reading and writing text file)

```
//-- true in statement below is the flag for Append
using (StreamWriter sw
                  = new StreamWriter(filePathAndName, true))
{
     foreach (string s in myText) { sw.WriteLine(s); }
}

List<string> myText = new List<string>();
using (StreamReader sr = new
StreamReader(filePathAndName))
{
     string line;
     while ((line = sr.ReadLine()) != null)
     {
          myText.Add(line);
     }
}
```

IEnumerable and **yield** -- Indicates that method (or get accessor) that contains this is an iterator, and that state needs to be stored. Iterator is consumed with foreach or a LINQ query. Return type must be IEnumerable or IEnumerator (with or without <T>).

```
public class PowersOf2
{
   static void Main()
   {
      // Display powers of 2 up to the exponent of 8:
      // Output will be: 2 4 8 16 32 64 128 256
      foreach (int i in Power(2, 8))
      {   Console.Write("{0} ", i);  }
   }

   public static IEnumerable<int> Power(int nbr, int exponent)
   {
      int result = 1;
      for (int i = 0; i < exponent; i++)
      {
         result = result * nbr;
         yield return result;
      }
   }
}
```

Another example (using nested classes)

```
public static class GalaxyClass
{
   public static void ShowGalaxies()
   {
      var theGalaxies = new Galaxies();
      foreach (Galaxy theGalaxy in theGalaxies.NextGalaxy)
      {
         WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears.ToString());
      }
   } // ShowGalaxies
```

```
   public class Galaxies
   {
      public IEnumerable<Galaxy> NextGalaxy
      {
         get
         {
            yield return new Galaxy { Name = "Tadpole", MegaLightYears = 400 };
            yield return new Galaxy { Name = "Pinwheel", MegaLightYears = 25 };
            yield return new Galaxy { Name = "Milky Way", MegaLightYears = 0 };
            yield return new Galaxy { Name = "Andromeda", MegaLightYears = 3 };
         }
      } // Next galaxy
   } // Galaxies
```

```
   public class Galaxy
   {
      public String Name { get; set; }
      public int MegaLightYears { get; set; }
   }
} // GalaxyClass
```

unsafe {} // for use with pointers

Constraints -- (on type parameters)

```
public class GenericList<T> where T : Employee
```

The constraint enables the generic class to use the Employee.Name property because all items of type T are guaranteed to be either an Employee object or an object that inherits from Employee.

Delegate -- similar to pointer to function in C or C++. It is a ref type variable pointing to a method, and can be changed at runtime. Used in such things as binding events to event handlers, like if you click a button. Or, more generally, when you want to send a method as a parameter to another method.

example: myList.Foreach(i => i.DoSomething());

see: "C# delegates and equivalent Java constructs" here:
https://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Java

Tuples -- are cool

```
var tuple = new Tuple<string, int, bool, MyClass>
                        ("foo", 123, true, new MyClass());

List<Tuple<int, string>> list = new List<Tuple<int, string>>();
list.Add(new Tuple<int, string>(2, "foo"));
list.Add(new Tuple<int, string>(1, "bar"));
list.Add(new Tuple<int, string>(3, "qux"));

//sort based on the string element
list.Sort((a, b) => a.Item2.CompareTo(b.Item2));

foreach (var element in list)
{
   Console.WriteLine(element);
}

// Output:
// (1, bar)
// (2, foo)
// (3, qux)
```

P/Invoke -- Platform Invoke used on .Net Compact Framework is used to invoke functions in external DLL's.