A Comparative Overview of C#

Author: Ben Albahari
Company: Genamics
Date: Initially released 31 July 2000, updated 10 August 2000.
Acknowledgements, alphabetically: Don Box, C.R. Manning, Joe Nalewabau, John Osborn, Thomas Rhode & Daryl Richter for their feedback and support

This article will focus on the new ways of programming C# offers, and how it intends to improve upon its two closest neighbors, Java and C++. C# improves on C++ in a similar way to Java in many respects, so I'm not going to be re-explaining things like the benefits of a single-rooted object hierarchy. This article begins with a brief summary of the similarities between C# and Java, and then goes into exploring the new C# features.

Background

In June 2000, Microsoft announced both the .NET platform and a new programming language called C#. C# is a strongly-typed object-oriented language designed to give the optimum blend of simplicity, expressiveness, and performance. The .NET platform is centered around a Common Language Runtime (similar to a JVM) and a set of libraries which can be exploited by a wide variety of languages which are able to work together by all compiling to an intermediate language (IL). C# and .NET are a little symbiotic: some features of C# are there to work well with .NET, and some features of .NET are there to work well with C# (though .NET aims to work well with many languages). This article is mostly concerned with C#, but sometimes it is useful to discuss .NET too. The C# language was built with the hindsight of many languages, but most notably Java and C++. It was co-authored by Anders Hejlsberg (who is famous for the design of the Delphi language), and Scott Wiltamuth.

Contents

1. C# and Java

Below is a list of features C# and Java share, which are intended to improve on C++. These features are not the focus of this article, but it is very important to be aware of the similarities.

- Compiles into machine-independent language-independent code which runs in a managed execution environment.
- Garbage Collection coupled with the elimination of pointers (in C# restricted use is permitted within code marked unsafe)
- Powerful reflection capabilities
- No header files, all code scoped to packages or assemblies, no problems declaring one class before another with circular dependencies
- Classes all descend from object and must be allocated on the heap with new keyword
- Thread support by putting a lock on objects when entering code marked as locked/synchronized
- Interfaces, with multiple-inheritance of interfaces, single inheritance of implementations
- Inner classes
- No concept of inheriting a class with a specified access level
- No global functions or constants, everything belongs to a class
- Arrays and strings with lengths built-in and bounds checking
- The "." operator is always used, no more ->, :: operators
- null and boolean/bool are keywords
- All values are initialized before use
- Can’t use integers to govern if statements
- Try Blocks can have a finally clause

2. Properties

Properties will be a familiar concept to Delphi and Visual Basic users. The motivation is for the language to formalize the concept of getter/setter methods, which is an extensively used pattern, particularly in RAD (Rapid Application Development) tools.

This is typical code you might write in Java or C++:

```
foo.setSize (getSize () + 1);
label.getFont().setBold (true);
```

The same code you would write like this in C#:

```
foo.size++;
label.font.bold = true;
```

The C# code is immediately more readable by those who are using foo and label. There is similar simplicity when implementing properties:

Java/C++:

```
public int getSize() {
        return size;
}

public void setSize (int value) {
        size = value;
}
```

C#:

```
public int Size {
        get {return size;
        }
        set {size = value;
        }
}
```

Particularly for read/write properties, C# provides a cleaner way of handling this concept. The relationship between a get and set method is inherent in C#, while has to be maintained in Java or C++. There are many benefits of this approach. It encourages programmers to think in terms of properties, and whether that property is most natural as read/write vs read only, or whether it really shouldn't be a property at all. If you wish to change the name of your property, you only have one place to look (I've seen getters and setter several hundred lines away from each other). Comments only have to be made once, and won't get out of sync with each other. It is feasible that an IDE could help out here (and in fact I suggest they do), but one should remember an essential principle in programming is to try to make abstractions which model our problem space well. A language which supports properties will reap the benefits of that better abstraction.

One possible argument against this being a benefit is that you don't know if you're manipulating a field or a property with this syntax. However, almost all classes with any real complexity designed in Java (and certainly in C#) do not have public fields anyway. Fields typically have a reduced access level (private/protected/default) and are only exposed through getter/setters, which means one may as well have the nicer syntax. It is also totally feasible an IDE could parse the code, highlighting properties with a different color, or provide code completion information indicating if it is a property or not. It should also be noted that if a class is designed well, then a user of that class should only worry about the specification of that class, and not its implementation. Another possible argument is that it is less efficient. As a matter a fact, good compilers can in-line the default getter which merely returns a field, making it just as fast as field. Finally, even if using a field is more efficient that a getter/setter, it is a good thing to be able to change the field to a property later without breaking the source code which relies on the property.

3. Indexers

C# provides indexers allow objects to be treated like arrays, except that like properties, each element is exposed with a get and/or set method.

```
public class Skyscraper
{
```

```
    Story[] stories;
    public Story this [int index] {
         get {
               return stories [index];
         }
         set {
              if (value != null) {
                   stories [index] = value;
              }
         }
    }
    ...
}

Skyscraper empireState = new Skyscraper (...);
empireState [102] = new Story ("The Top One", ...);
```

4. Delegates

A delegate can be thought of as a type-safe object-oriented function pointer, which is able to hold multiple methods rather than just one. Delegates handle problems which would be solved with function pointers in C++, and interfaces in Java. It improves on the function pointer approach by being type safe and being able to hold multiple methods. It improves on the interface approach by allowing the invocation of a method without the need for inner-class adapters or extra code to handle multiple-method invocations. The most important use of delegates is for event handling, which is in the next section (which gives an example of how delegates are used).

5. Events

C# provides direct support for events. Although event handling has been a fundamental part of programming since programming began, there has been surprisingly little effort made by most languages to formalize this concept. If you look at how today's mainstream frameworks handle events, we've got examples like Delphi's function pointers (called closures), Java's inner class adaptors, and of course, the Windows API's message system. C# uses delegates along with the event keyword to provide a very clean solution to event handling. I thought the best way to illustrate this was to give an example showing the whole process of declaring, firing, and handling an event:

```
// The Delegate declaration which defines the signature of methods which can be invo
public delegate void ScoreChangeEventHandler (int newScore, ref bool cancel);

// Class which makes the event
public class Game {
    // Note the use of the event keyword
    public event ScoreChangeEventHandler ScoreChange;

    int score;

        // Score Property
    public int Score {
        get {
            return score;
            }
        set {
            if (score != value) {
                bool cancel = false;
                ScoreChange (value, ref cancel);
```

```
                if (! cancel)
                    score = value;
            }
        }
    }
}

// Class which handles the event
public class Referee
{
    public Referee (Game game) {
        // Monitor when a score changes in the game
        game.ScoreChange += new ScoreChangeEventHandler (game_ScoreChange);
    }

    // Notice how this method signature matches the ScoreChangeEventHandler's signat
    private void game_ScoreChange (int newScore, ref bool cancel) {
        if (newScore < 100)
            System.Console.WriteLine ("Good Score");
        else {
            cancel = true;
            System.Console.WriteLine ("No Score can be that high!");
        }
    }
}

// Class to test it all
public class GameTest
{
    public static void Main () {
        Game game = new Game ();
        Referee referee = new Referee (game);
        game.Score = 70;
        game.Score = 110;
    }
}
```

In the GameTest we make a game, make a referee who monitors the game, then change the game's score to see what the referee does in response to this. With this system, the Game has no knowledge of the Referee at all, and simply lets any class listen and act on changes made to its score. The event keyword hides all the delegate's methods apart from += and -= to classes other than the class it is declared in. These operators allow you to add (and remove) as many event handlers as you want to the event.

You will probably first encounter this system in GUI frameworks, where the Game would be analogous to UI widgets which fire events according to user input, and the referee would be analogous to a form, which would process the events.

Delegates were first introduced in Microsoft's Visual J++ also designed by Anders Hejlsberg, and were a cause of much technical and legal dispute between Sun and Microsoft. James Gosling, the man who designed Java made a condescending though humorous comment about Anders Hejlsberg, saying his attachment to Delphi made him "Mr. Method Pointers". After examining the arguments against delegates made by Sun, I believe it would be fair to call Gosling "Mr. Everything's-a-Class". In the last few years of programming "make abstractions which try to model reality well" has been replaced by many people with "Reality is object-oriented, so we should model it with object oriented abstractions".

Sun's and Microsoft's arguments for and against delegates can be found at:

- http://www.Javasoft.com/docs/white/delegates.html
- http://msdn.microsoft.com/visualj/technical/articles/delegates/truth.asp

6. Enums

In case you don't know C, Enums let you specify a group of objects, e.g.:

Declaration:

```
public enum Direction {North, East, West, South};
```

Usage:

```
Direction wall = Direction.North;
```

It's a nice construct, so perhaps the question is not why did C# decide to have them, but rather, why did Java choose to omit them? In Java, you would have to go:

Declaration:

```
public class Direction {
        public final static int NORTH = 1;
        public final static int EAST = 2;
        public final static int WEST = 3;
        public final static int SOUTH = 4;
}
```

Usage:

```
int wall = Direction.NORTH;
```

Despite the fact the Java version seems to express more, it doesn't, and is less type-safe, by allowing you to accidentally assign wall to any int value without the compiler complaining. To be fair, in my experience of Java programming I haven't wasted much time on writing a few extra tokens and tracking down an error because of lack of type-safety here, but nonetheless, this is nice to have. One benefit of C# is the nice surprise you get when debugging - if you put a break point on an enumeration which holds combined enumerations, it will automatically decode direction to give you a human readable output, rather than a number you have to decode:

Declaration:

```
public enum Direction {North=1, East=2, West=4, South=8};
Usage:
Direction direction = Direction.North | Direction.West;
if ((direction & Direction.North) != 0)
    ....
```

If you put a breakpoint on the if statement, you'll get the human readable version for direction rather than the number 5.

The mostly likely reason why enums are absent in Java is that you can get by using classes instead. As I mentioned in previous sections, with classes alone we are not able to express a feature in the world as well as we could with another construct. What are the benefits of the Java philosophy of "if it can be done by a class, then don't introduce a new construct"? It would seem simplicity would be the biggest benefit - a shorter learning curve and the prevention of programmers having to think of multiple ways of doing things. Indeed the Java language has improved on C++ in many ways by aiming for simplicity, such as the elimination of pointers, the elimination of header files, and a single-rooted object hierarchy. However, a common aspect of all these simplifications is they actually make coding - uh - simpler. Leaving out constructs, we've looked at enums, properties and events so far, makes your coding more complicated.

7. Collections and the Foreach Statement

C# provides a shorthand for for-loops, which also encourages increased consistency for collections classes:

In Java or C++:

```
1. while (! collection.isEmpty()) {
        Object o = collection.get();
        collection.next();
        ...
2. for (int i = 0; i < array.length; i++)...
```

In C#:

```
1. foreach (object o in collection)...
2. foreach (int i in array)...
```

The C# for-loop will work on collection objects (arrays implement a collection). Collection objects have a GetEnumerator() method which returns an Enumerator object. An Enumerator object has a MoveNext method and a Current property.

8. Structs

It is helpful to view C# structs as a construct which makes the C# type system work elegantly, rather than just "a way to write really efficient code if you need to".

In C++, both structs and classes (objects) can be allocated on the stack/in-line or on the heap. In C#, structs are always created on the stack/in-line, and classes (objects) are always created on the heap. Structs indeed allow more efficient code to be created:

```
public struct Vector {
    public float direction;
    public int magnitude;
}

Vector[] vectors = new Vector [1000];
```

This will allocate the space for all 1000 vectors in one lump, which is much more efficient that had we declared Vector as a class and used a for-loop to instantiate 1000 individual vectors. What we've effectively done is declared the array like we would declare an int array in C# or Java:

```
int[] ints = new ints [1000];
```

C# simply allows you to extend the primitive set of types built in to the language. In fact, C# implements all the primitive types as structs. The int type merely aliases System.Int32 struct, the long type aliases System.Int64 struct etc. These primitive types are of course able to be treated specially by the compiler, but the language itself does not make such a distinction. The next section shows how C# takes advantage of this.

9. Type Unification

Most languages have primitive types (int, long, etc), and higher level types which are ultimately composed of primitive types. It is often useful to be able to treat primitive types and higher level types in the same way. For instance, it is useful to have collections which can take both ints as well as strings. Smalltalk achieves this by sacrificing some efficiency and treating ints and longs as types like String or Form. Java tries to avoid sacrificing this efficiency, and treats primitive types like in C or C++, but provides corresponding wrapper classes for each primitive - int is wrapped by Integer, double is wrapped by Double. C++'s templates allow code to be written which takes any type, so long as the operations done on that type are provided by that type.

C# provides a different solution to this problem. In the previous section, I introduced C# structs, explaining how primitive types like int were merely aliases for structs. This allows code like this to be written, since structs have all the methods that the object class does:

```
int i = 5;
System.Console.WriteLine (i.ToString());
```

If we want to use a struct as an object, C# will box the struct in an object for you, and unbox the struct when you need it again:

```
Stack stack = new Stack ();
stack.Push (i); // box the int
int j = (int) stack.Pop(); // unbox the int
```

Apart from a type-cast required when unboxing structs, this is a seamless way to handle the relationship between structs and classes. You should bare in mind that boxing does entail the creation of a wrapper object, though the CLR may provide additional optimization for boxed objects.

The designers of C# must have considered templates during their design process. I suspect there were two main reasons for not using templates. The first is messiness - templates can be difficult to mesh with object oriented features, they open up too many (confusing) design possibilities for programmers, and they are difficult to work with reflection. The second is that templates wouldn't be very useful unless the .NET libraries like the collection classes used them. However, if the .NET classes used them, then the 20+ languages which use the .NET classes would have to work with templates too, which may be technically very difficult to achieve.

It is interesting to note that templates (generics) are being considered for inclusion in the Java language by the Java Community Process. It may be that each company starts singing each other's tunes when Sun says that ".NET suffers from lowest common denominator syndrome", and Microsoft says "Java doesn't have cross-language support".

(Amended 10 August) From reading an interview with Anders Hejlsberg, it appears templates are on the horizon, but not for the first release, for the difficulties which were suggested above. It was very interesting to see that the IL specification was written so that the IL code could represent templates (in a non-destructive way so that reflection works well), while byte-code was not. I've also included a link to the Java Community Process for considering generics in java:

- Interview With Anders Hejlsberg and Tony Goodhew
- Java Community Process for adding Generics to Java

10. Operator Overloading

Operator overloading allows programmers to build types which feel as natural to use as simple types (int, long, etc.). C# implements a stricter version of operator overloading in C++, but it allows classes such as the quintessential example of operator-overloading, the complex number class, to work well.

In C#, the == operator is a non-virtual (operators can't be virtual) method of the object class which compares by reference. When you build a class, you may define your own == operator. If you are using your class with collections, then you should implement the IComparable interface. This interface has one method to implement, called the CompareTo (object) method, which should return positive, negative, or 0 if "this" is greater, less than, or the same value as the object. You may choose to define <, <=, >=, > methods if you want users of your class to have a nicer syntax. The numeric types (int, long, etc) implement the IComparable interface.

Here's a simple example of how to deal with equality and comparisons:

```
public class Score : IComparable
{
    int value;

    public Score (int score) {
        value = score;
    }

    public static bool operator == (Score x, Score y) {
        return x.value == y.value;
    }

    public static bool operator != (Score x, Score y) {
        return x.value != y.value;
    }

    public int CompareTo (object o) {
        return value - ((Score)o).value;
    }
}

Score a = new Score (5);
Score b = new Score (5);
Object c = a;
Object d = b;
```

To compare a and b by reference:

```
System.Console.WriteLine ((object)a == (object)b; // false
```

To compare a and b by value:

```
System.Console.WriteLine (a == b); // true
```

To compare c and d by reference:

```
System.Console.WriteLine (c == d); // false
```

To compare c and d by value:

```
System.Console.WriteLine (((IComparable)c).CompareTo (d) == 0); // true
```

You could also add the <, <=, >=, > operators to the score class. C# ensures at compile time that operators which are logically paired (!= and ==, > and <, >= and <=), must both be defined.

11. Polymorphism

Virtual methods allow object oriented languages to express polymorphism. This means a derived class can write a method with the same signature as a method in its base class, and the base class will call the derived class's method. By default in Java, all methods are virtual. In C#, like C++, the virtual keyword must be used so that the method will be called by the base class.

In C#, it is also required that the override keyword is needed to specify that a method should override a method (or implement an abstract method) of its base class.

```
Class B {
    public virtual void foo () {}
}

Class D : B {
    public override void foo () {}
}
```

Attempting to override a non-virtual method will result in a compile-time error unless the "new" keyword is added to the declaration, indicating the method is intentionally hiding the base class's method.

```
Class N : D {
    public new void foo () {}
}

N n = new N ();
n.foo(); // calls N's foo
((D)n).foo(); // calls D's foo
((B)n).foo(); // calls D's foo
```

In contrast to both C++ and Java, requiring the override keyword makes it more clear what methods are overridden when looking at source code. However, requiring the use of the virtual method has its pros and cons. The first pro is that is the slightly increased execution speed from avoiding virtual methods. The second pro is to make clear what methods are intended to be overridden. However, this pro can also be a con. Compare the default option of leaving out a final modifier in Java vs leaving out a virtual modifier in C++. The default option in Java may make your program slightly less efficient, but in C++ it may prevent extendibility, albeit unforeseen, by the implementer of the base class.

12. Interfaces

Interfaces in C# are similar to Java interfaces, but can be used with greater flexibility. A class may optionally "explicitly" implement an interface:

```
public interface ITeller
{
    void Next ();
}

public interface IIterator
{
    void Next ();
}

public class Clark : ITeller, IIterator
{
    void ITeller.Next () {
    }
    void IIterator.Next () {
    }
}
```

This gives a class two benefits. First, a class can implement several interfaces without having to worry about naming conflicts. Second, it allows a class to "hide" a method if it is not useful for general users of the class. Explicitly implemented methods are invoked by type-casting a class to the interface required:

```
Clark clark = new Clark ();
((ITeller)clark).Next();
```

13. Versioning

Solving versioning issues has been a major consideration in the .NET framework. Most of these considerations apply to assemblies. There are some quite impressive capabilities such as running multiple versions of the same assembly in the same process.

The C# language prevents software failures when new versions of code (most notably the .NET libraries) are made. The C# Language Reference explains this in detail, but I've summarized the problem with a highly condensed example:

In Java, supposing we deploy a class called D which derives a class distributed with the VM, called B. Class D has a method called foo which B does not have at the time of distribution. Later, an update is made to class B, which now includes a foo method, and the new VM is installed on the computer running software which uses class D. The software using class D may malfunction because the new implementation of class B will make a virtual call to class D, performing an action with an implementation totally unintended by class B. In C#, class D's foo would have been declared without the override modifier (which expresses the programmer's intention), so the runtime knows to make Class D's foo hide Class B's foo, rather than override it.

An interesting quote from the C# reference manual was: "C# addresses this versioning problem by requiring developers to clearly state their intent". Although the use of the word override is one way to state intention, it could also be automatically generated by the compiler by checking to see if a method performs (rather than declares) an override at the time of compilation. This means you can still have Java-like languages (which don't use virtual and override keywords) and still cope with versioning correctly.

Also see Field Modifiers

14. Parameter Modifiers

"ref" parameter modifier

C# (as opposed to Java) lets you pass parameters by reference. The most obvious example to illustrate the point of this is the general purpose swap method. Unlike in C++, you must specify when calling as well as when declaring a method which takes ref parameters:

```
public class Test
{
    public static void Main ()
    {
        int a = 1;
        int b = 2;
        swap (ref a, ref b);
    }

    public static void swap (ref int a, ref int b) {
       int temp = a;
       a = b;
       b = temp;
    }
}
```

"out" parameter modifier

There is also an "out" keyword, which is a natural complement to the ref parameter modifier. While the ref modifier requires that a value is definitely assigned before passed into a method, the out modifier requires that the method's implementation definitely assigns the parameter a value before returning.

"params" parameter modifier

The params modifier may be added to the last parameter of a method so that the method accepts any number of parameters of a particular type. For example:

```
public class Test
{
    public static void Main () {
        Console.WriteLine (add (1, 2, 3, 4).ToString());
    }

    public static int add (params int[] array) {
       int sum = 0;
       foreach (int i in array)
           sum += i;
       return sum;
    }
```

```
}
```

One of the most surprising things when you're learning Java is not being able to pass by reference. It turns out though that after a little while you seldom miss this functionality, and write code which doesn't use it it. When I went over the C# specification for the first time, I often thought "Why have they got this or that functionality, I can code without it". With a little introspection, I realized this wasn't really an argument to show some functionality wasn't useful, but was more an argument to show that you get conditioned into getting by without it.

Java did a good thing when it simplified how parameters could be passed when you consider what C++ does. In C++, parameters of a method and a method call pass in values or references, which in addition to having pointers, can lead to unnecessarily complicated code. C# makes passing by reference explicit for both the method and the method call, which greatly alleviates any confusion, thus achieving the same goal as Java, but with greater expressiveness. It is clearly a theme of C# to not hedge programmers into situations where they require a round-about way of achieving something. Remember those Java tutorials that suggest that to overcome the pass-by-reference problem, you should pass a 1-element array to holding your value, or make another class to hold that value?

15. Attributes

Both C# and Java contain information such as the access level of a field in the compiled code. C# generalizes this ability, so you can compile custom information about just about any element of code such as a class, method, field and even individual parameters. This information can be retrieved at run-time. Here is a very simple example of a class which makes use of attributes:

```
[AuthorAttribute ("Ben Albahari")]
class A
{
    [Localizable(true)]
    public String Text {
        get {return text;
        }
        ...
    }
}
```

Java uses a combination of /** */ and @tag comments to include additional information about classes and methods, but this information (apart from the @deprecated) is not built into the byte code. C# uses the pre-defined attributes ObsoleteAttribute so the compiler can warn you against obsolete code (like @deprecated), and the ConditionalAttribute to enable conditional compilation. Microsoft's new XML libraries utilize attributes to express how fields should be serialized into XML, which means you can easily turn a class into XML and then reconstruct it again. Another apt use for attributes is for the creation of a really powerful class browsing tool. The C# Language Reference explains exactly how to create and use attributes.

16. Selection Statements

C# lets you govern a switch statement with an integral type, char, enum or (unlike C++ or Java) a string. In Java and C++ if you leave out a break after each case statement you risk having another case statement being executed. I have no idea why this rarely needed error-prone behavior was made the default behavior in Java and C++, and I'm glad to see C# doesn't do this.

17. Predefined Types

The C# primitive types are basically the same as the ones in Java except they add unsigned types to. We've got sbyte, byte, short, ushort, int, uint, long, ulong, char, float & double. The only surprise here is the 12-byte "decimal" floating point number which can take advantage of the latest processors.

18. Field Modifiers

Again, the field modifiers are basically the same as the ones in Java. To express fields which can't be modified, C# uses the const and readonly modifiers. A const field modifier is like the Java final field modifier, and compiles so that the actually value is part of the IL code. A readonly modifier compiles so that at run-time the value is evaluated. This allows an upgrade, say to one one of the standard C# libraries, without breaking your deployed code.

19. Jump Statements

Not many surprises here, apart from perhaps the infamous goto. However, it's really a distant relative of the evil goto statement we remember from basic 20 years ago. A goto statement must point to a label or one of the options in a switch statement. The first usage of pointing to a label is similar to the usage of a continue : label statement in Java, except with a little more freedom. Such a goto statement may point anywhere within its scope, which restricts it to the same method, or finally block if it is declared within one. It may not jump into a loop statement which it is not within, and it cannot leave a try block before the enclosing finally block(s) are executed. The continue statement in C# is equivalent to the continue statement in Java, except it can't point to a label.

20. Assemblies, Namespaces & Access Levels

In C#, you can organize the components of your source-code (classes, structs, delegates, enums, etc.) into files, namespaces & assemblies.

Namespaces are nothing but syntactic sugar for long class names. For instance, rather than calling a class Genamics.WinForms.Grid you can declare the class as Grid and enclose the class with:

```
namespace Genamics.WinForms {
    public class Grid {
        ....
    }
}
```

For classes which use Grid, you can import it using the "using" keyword instead of referring to its full class name Genamics.WinForms.Grid.

Assemblies are .exes or .dlls generated from compiling a project of files. The .NET runtime uses the configurable attributes and versioning rules built into assemblies to greatly simplify deployment - no more hacking the registry - just copy the assembly into a directory and it goes. Assemblies also form a type-boundary to deal with type-name collisions, to the extent that multiple versions of an assembly can co-exist in the same process. Each file can contain multiple classes and multiple namespaces. A namespace may also be spread across several assemblies, so there is high level of freedom with this system.

There are five access levels in C#, private, internal, protected, internal protected, and public. Private and public have the same meanings as in Java, noting that in C# the default access level is private, rather than package. Internal access is scoped to assemblies rather than namespaces (which would be more analogous to Java). Internal protected access is equivalent to Java's protected access, and protected access is equivalent to Java's private protected access made obsolete in Java some time ago.

21. Pointer Arithmetic

Pointer arithmetic can be performed in C# within methods marked with the unsafe modifier. When pointers point to garbage collected objects, the compiler enforces the use of the fixed word to pin the object. This is because garbage collectors rely on moving objects around to reclaim memory, but if this happens when you're dealing with raw pointers you'll be pointing at garbage. The choice of the word "unsafe" I believe is well chosen since it discourages developers from using pointers unless they really need to.

22. Rectangular Arrays

C# allows both jagged and rectangular arrays to be created. Jagged arrays are pretty much the same as Java arrays. Rectangular arrays allow a more efficient and accurate representation for certain problems. An example of such an array would be:

```
int [,,] array = new int [3, 4, 5]; // creates 1 array
int [1,1,1] = 5;
```

Using jagged arrays:

```
int [][][] array = new int [3][4][5]; // creates 1+3+12=16 arrays
int [1][1][1] = 5;
```

In combination with structs, C# can provide a level of efficiency making it a good choice for areas such as graphics and mathematics.

23. Constructors and Destructors

You can specify optional constructor parameters:

```
class Test
{
    public Test () : this (0, null) {}
    public Test (int x, object o) {
    }
}
```

You can specify static constructors:

```
class Test
{
    static int[] ascendingArray = new int [100];

    static Test () {
        for (int i = 0; i < ascendingArray.Length; i++)
            ascendingArray [i] = i;
```

```
    }
}
```

Destructors are specified with the C++ naming convention using the ~ symbol. Destructors can only be made for references types and not value types, and cannot be overridden. A destructor can not be explicitly called, since it doesn't make much sense to do so when an object's lifetime is managed by the garbage collector. Each destructor in an object's hierarchy (from the most derived to the least derived) is called before the memory for the object is reclaimed.

Despite the naming similarity with C++, destructors in C# are much more like Java finalizers. This is because they are called by the garbage collector rather than explicitly called by the programmer. Furthermore, like Java finalizers, they cannot be guaranteed to be called in all circumstances (this always shocks everyone when they first discover this). If you are used to programming with deterministic finalization (you know when an object's destructor is called), then you have to adapt to a different model when moving to Java or C#. The model Microsoft recommends and implements throughout the .NET framework is the dispose pattern, whereby you define a dispose() method to for classes which manage foreign resources such as graphics handles or database connections. For distributed programming, the .NET framework provides a leased based model to improve upon DCOM reference counting.

24. Managed Execution Environments

The comparison between [C#/IL Code/CLR] and [Java/Byte-Code/JVM] is an inevitable and valid comparison to make. I thought the best way to make sense of these comparisons was to explain why these technologies were made in the first place.

With C and C++ programs, you generally compile the source code to assembly language code which will only run on a particular processor and a particular OS. The compiler needs to know which processor it is targeting, because processors vary in their instruction sets. The compiler also needs to know what OS it is targeting, because OSs vary in ways such as how executables work and how they implement some basic C/C++ constructs such as allocating memory. This C/C++ model has been very successful (most software you're using probably has been compiled like this), but has its limitations:

- Does not provide a program with a very rich interface to interact with other programs (Microsoft's COM was built to address this limitation)
- Does not allow a program to be distributed in a form which can be used as is on different platforms
- Does not allow a program's execution to be limited to a sand box of safe operations

Java addressed these problems in the same way Smalltalk had done before it by having Java compile to byte-code which then runs on a virtual machine. Byte-code maintains the basic structure of a program before it is compiled, thus making it possible for a Java program to richly interact with another program. Byte-code is also machine-independent, which means the same class file can be used on a number of platforms. Finally, the fact that the Java language did not have explicit memory manipulation (via pointers) made it well suited to writing sand-boxed programs.

Originally virtual machines had an interpreter which converted a stream of byte-code instructions to machine code on the fly, but this dreadfully slow process was never appealing to the performance conscious programmer. Today most Java Virtual Machines use a Just-In-Time compiler which basically compiles to machine-code class skeletons just before they enter scope and method bodies just before they are executed. It is also possible to convert a Java program to assembly language before it runs to eliminate the overhead in start-up time and memory of a Just-In-Time compiler. As with compiling a Visual C++ program, this process does not necessarily remove the program's dependence on a runtime. The Java runtime (also covered by the term "Java Virtual Machine") will handle many vital parts of the programs execution such as garbage collection and security. This runtime is referred to as a managed execution environment.

Though somewhat obscured by terminology, the basic model .NET uses is the same as described above, though it was built to never use intepreting. Important improvements .NET will offer will come from the IL design itself, since the only way Java can match such improvements is by changing the byte-code specification which would generate serious compatibility issues. I don't want to discuss in detail these improvements - it should be left to those rare developers who understand both byte-code and IL-code. For the 99% of developers like myself who realistically aren't going to be studying the IL-Code specification, here are some of the design decisions for IL which are intended to improve upon byte-code:

- To provide greater type neutrality (helps implementing templates)
- To provide greater language neutrality
- To always be compiled to assembly language before executing, and never interpreted
- To allow additional declarative information to be added to classes, methods, etc., see 15. Attributes

Currently the CLR has yet to provide multi-OS support, but provides greater interoperability to JVMs in other areas (See 26. Interoperability)

25. Libraries

A language is pretty much useless without libraries. C# has remarkably few core libraries, but utilizes the libraries of the .NET framework (some of which were built with C#). This article is mainly concerned with addressing the C# language in particular, rather than on .NET, which is best dealt with in a separate article. Briefly, the .NET libraries come with a rich set of libraries including Threading, Collection, XML, ADO+, ASP+, GDI+ & WinForms libraries. Some of these libraries are cross-platform, while others are Windows dependent, but see the next section for a discussion on platform support.

26. Interoperability

I thought it would be useful to group interoperability into three divisions: Language interoperability, Platform interoperability, and Standards interoperability. While Java has its defining strength in platform interoperability, C# has it's strength in language interoperability. Both have strengths and weaknesses in standards interoperability.

Language Interoperability:

This is the level and ease of integration with other languages. Both the Java Virtual Machine and the Common Language Runtime allow you to write code in many different languages, so long as they compile to byte code or IL code respectively. However, the .NET platform has done much more than just allow other languages to be compiled to IL code. NET allows multiple languages to freely share and extend each others libraries to a great extent. For instance, an Eiffel or Visual Basic programmer could import a C# class, override a virtual method of that class, and the C# object would now use the Visual Basic method (polymorphism). In case you were wondering, VB.NET has been massively upgraded (at the expense of compatibility with VB6) to have modern object oriented features.

Languages written for .NET will generally plug into the Visual Studio.NET environment and use the same RAD frameworks if needed, thus overcoming the "second rate citizen" effect of using another language.

C# provides P/Invoke, which is a much simpler (no-dlls) way to interact with C code than Java's JNI. This feature is very similar to J/Direct, which is a feature of Microsoft Visual J++.

Platform Interoperability:

Generally this means OS interoperability, but over the last few years the internet browser has emerged as a platform in itself.

C# code runs in a managed execution environment, which is the most important technological step to making C# run on different operating systems. However, some of the .NET libraries are based on Windows, particularly the WinForms library which depends on the nitty gritty details of the Windows API. There is a project to port the Windows API to Unix systems, but this isn't here now and Microsoft have not given any firm indication of their intentions in this area.

However, Microsoft hasn't ignored platform interoperability. The .NET libraries provide extensive capabilities to write HTML/DHTML solutions. For solutions which can be implemented with a HTML/DHTML client, C#/.NET is a good choice. For cross-platform projects which require a more complex client interface, Java is a good choice. Kylix, a version of Delphi which allows the same code to compile to both Windows and Linux may also be a good choice for rich cross-platform solutions in the future.

Microsoft has submitted the C# specification as well as parts of the .NET specification to the ECMA standards body.

Standards Interoperability:

These are all the standards like databases systems, graphics libraries, internet protocols, and object communication standards like COM and CORBA, that the language can access. Since Microsoft owns or plays a big role in defining many of these standards, they are in a very good position to support them. They of course have business motivations (I'm not saying they are or are not justified) to provide less support for standards which compete with their own - for instance - CORBA competes with COM and OpenGL competes with DirectX. Similarly, Sun's business motivations (again I'm not saying they are or are not justified) means Java doesn't provide as good support for Microsoft standards as it could.

C# objects, since they are implemented as .NET objects, are automatically exposed as COM objects. C# thus has the ability to expose COM objects as well as to use COM objects. This will allow the huge base of COM code to be integrate with C# projects. .NET is a framework which can eventually replace COM - but there is so much deployed COM code that by the time this happens I'm sure .NET will be replaced by the next wave of technology. Anyway, expect .NET to have a long and interesting history!

27. Conclusion

I hope this has given you a feel for where C# stands in relation to Java and C++. Overall, I believe C# provides greater expressiveness and is more suited to writing performance-critical code than Java, while sharing Java's elegance and simplicity, which makes both much more appealing than C++.

Ben Albahari,
Genamics.

The Microsoft Developer .NET homepage can be found at http://msdn.microsoft.com/net