Learning C# from Java and C++

By Jonathan White
March 24, 2009
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>2</td>
</tr>
<tr>
<td>Introduction</td>
<td>4</td>
</tr>
<tr>
<td>Explanation of how this document is intended for people familiar with Java and C++</td>
<td>4</td>
</tr>
<tr>
<td>Explanation of how this document won’t be much help of to people who aren’t familiar with Java and C++</td>
<td>4</td>
</tr>
<tr>
<td>Recommendation to read the basic language features section, and read the other sections as needed</td>
<td>4</td>
</tr>
<tr>
<td>Quick introduction to Visual Studio</td>
<td>4</td>
</tr>
<tr>
<td>Installing Visual Studio Express Edition</td>
<td>4</td>
</tr>
<tr>
<td>A comparison with Eclipse</td>
<td>5</td>
</tr>
<tr>
<td>Basic language features</td>
<td>5</td>
</tr>
<tr>
<td>Overview of basic language features</td>
<td>5</td>
</tr>
<tr>
<td>An overview of the virtual machine</td>
<td>5</td>
</tr>
<tr>
<td>An overview of memory management</td>
<td>5</td>
</tr>
<tr>
<td>An overview of preprocessor directives</td>
<td>6</td>
</tr>
<tr>
<td>An overview of assemblies</td>
<td>6</td>
</tr>
<tr>
<td>Explanation of statements and program structure are the in C#</td>
<td>7</td>
</tr>
<tr>
<td>A comparison of Java and C# variable types</td>
<td>7</td>
</tr>
<tr>
<td>An explanation of the decimal type</td>
<td>8</td>
</tr>
<tr>
<td>An explanation of C# flow control</td>
<td>8</td>
</tr>
<tr>
<td>An explanation of C# conditionals</td>
<td>8</td>
</tr>
<tr>
<td>An explanation of C# loops</td>
<td>8</td>
</tr>
<tr>
<td>An explanation of C# methods</td>
<td>8</td>
</tr>
<tr>
<td>Passing by reference/value in C#</td>
<td>8</td>
</tr>
<tr>
<td>Using Classes, Interfaces and Objects</td>
<td>9</td>
</tr>
<tr>
<td>Using Enumerations</td>
<td>10</td>
</tr>
<tr>
<td>Differences between packages and namespaces</td>
<td>10</td>
</tr>
</tbody>
</table>
Introduction

**Explanation of how this document is intended for people familiar with Java and C++**

I intend this document for people who are familiar with both Java and C++. Those not familiar with these languages, and in particular Java, will find little use in this document.

My goal is to introduce C# concisely. I want to give an introduction so they can get to know the language quickly. I do not intend to answer all of your questions here, and if you are doing something more serious with the language, I recommend getting any one of a number of good books on the topic.

**Explanation of how this document won't be much help of to people who aren’t familiar with Java and C++**

I intend this document to explain the C# programming language in terms of C++ and Java. If you do not know C++, you will probably be able to get plenty of use out of this document; however, if you do not know Java, this document will be of little use to you.

**Recommendation to read the basic language features section, and read the other sections as needed.**

I am writing this with the intention of giving a quick background in the C# language. The basic language features section will give you that quick introduction. I have also documented some parts of the .NET framework that I have used the most; however, some may only apply to you in some circumstances. I recommend reading those sections only if you need them. I wrote those sections assuming you have already read the basic language features or already have an understanding of the material in that section.

**Quick introduction to Visual Studio**

**Installing Visual Studio Express Edition**


For the things explained in this document, you will not need SQL server, but you can install it if you like.

Also, consider installing MSDN Documentation Express. This includes the documentation for the C# programming language and the .NET class library. This information is online at [http://www.msdn.com/](http://www.msdn.com/); however, I find it much quicker to have a local copy of the documentation.

I also recommend taking some time to get familiar with Visual Studio. You should learn some keyboard shortcuts that will make you more productive. The most important are F5, which starts
debugging, and CTRL+F5, which runs the program without the debugger, but also prompts you to press enter after your program terminates.

**A comparison with Eclipse**

Many basic features of Visual Studio are similar to Eclipse. For example, you work with projects in Visual Studio as you would in Eclipse; however, in Eclipse you must establish a workbench and create projects within that workbench. In Visual Studio, you create a solution, and create projects within that solution. You do not necessarily have to have a solution in order to be able to work with a project.

There is an important distinction between a solution and a project. A project contains all source code and resources that will compile into a single assembly, while a solution is the set of all the projects that will create your program or library.

**Basic language features**

**Overview of basic language features**

For the most part, you can assume that most basic language features of C# work the same as in Java. Throughout the following section, I will quickly introduce the C# language.

**An overview of the virtual machine**

Like Java, C# uses a virtual machine to execute. There are a few differences, however, between the Java and .NET virtual machines. First is the overall structure of .NET differs slightly from Java. First, C# is compiled to a language called MSIL, short for Microsoft Intermediate Language. This is similar to Java byte code, and you can think of this as an object oriented assembly language.

The .NET virtual machine executes MSIL. Java usually uses an interpreter to execute bytecode, and in some cases uses a just-in-time compiler. The .NET virtual machine uses a just-in-time compiler in all cases. It compiles the byte code of each assembly at a time, and caches the generated executable code for later use.

**An overview of memory management**

.NET uses a mark and sweep garbage collector, similar to Java. That means when any object in the heap looses all of its references to it, it becomes a candidate for garbage collection. It is important to know that the garbage collector will not reclaim it until the virtual machine runs out of memory and the .NET runtime invokes the garbage collector. Because of this, you cannot know when exactly when the garbage collector will destroy your objects. For this reason, you must use constructs such as the **using** construct, to ensure that you property clean up things like file streams.

.NET creates all objects on the heap, and it allocates primitive types such as int and double on the stack, unless they are a member of an object.

---

An overview of preprocessor directives

C# supports a number of preprocessor directives, just like C++. These, however, in C# do completely different things. They have a similar syntax to C/C++ preprocessor directives (i.e. they begin with a #). The supported preprocessor directives are as follows:

- **#define, #undef** – The first is just like its counterpart in C++. It defines a variable for use with the conditional preprocessor directives. #undef also allows you to undefine these variables.

- **#if, #elif, #else, #endif** – These constitute the conditional preprocessor directives in C#. You can use ==, !=, &&, ||, and () to build expressions for use in conditional compilation.

- **#region, #endregion** – These are purely for use with the Visual Studio IDE. Many times, you have blocks of methods in a class that go together, and many times are only in the way when working on other parts of a class. You can surround these methods with the #region/#endregion directives to collapse that section of the code as you would a method.

- **#warning, #error** – These generate either a warning or error when you try to compile it. It gives, as an error message, the text on that same line that follows. These are particularly useful when there is a part of the code that you will need to update later. If you place either a warning or an error in that part of the code, it will serve as a good reminder when you compile that you need to come back to that part of the code to update some that part of the code.

An overview of assemblies

An assembly is a .NET unit of deployment. They can consist of a single dll or exe, but can consist of multiple files. For the purposes of this document, you should know that they each contain a manifest and they use that manifest to resolve any possible versioning issues that may arise.

Remember that a project will produce a single assembly, be it an executable or dll. The assembly can contain multiple classes and even resources such as images that you can include inside of the dll or exe. This is a major difference from java, which produces a class file for every class. Java requires the creation of jar files to merge class files and resources into a single file.

---

3 Jones, et al. 25
**Explanation of statements and program structure are the in C#**

Programs written in C# and Java share a very similar structure. All code must be in methods written within classes. Each executable program must include a `Main` method, where execution starts. The method signature is as follows:

```
public static void Main(string[] args) {...}
```

Notice that you capitalize `Main` and that `string` is lowercase. Now is a good time for the typical hello world program in C#:

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

**A comparison of Java and C# variable types**

Here is a quick comparison of Java and C# primitive variable types.⁴

<table>
<thead>
<tr>
<th>Java</th>
<th>C#</th>
<th>Size (bytes)</th>
<th>Other important info:</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>bool</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>sbyte</td>
<td>1</td>
<td>Note that this is signed.</td>
</tr>
<tr>
<td>-</td>
<td>byte</td>
<td>1</td>
<td>This is an unsigned byte.</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
<td>2</td>
<td>*</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>4</td>
<td>*</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>8</td>
<td>*</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>char</td>
<td>2</td>
<td>This stores a single Unicode character.</td>
</tr>
<tr>
<td>-</td>
<td>decimal</td>
<td>16</td>
<td>Consists of a decimal number with 28 significant digits.</td>
</tr>
</tbody>
</table>

*Note: C# has no unsigned keyword as Java does. To make `short`, `int`, and `long` unsigned, simply pre-pend the letter `u`, so they become `ushort`, `uint`, `ulong`. |

⁴ Mok, Heng Ngee *From Java to C# - A Developer’s Guide*. (Great Britan: Addison Wesley. 2003), 131
An explanation of the decimal type

The decimal data type is a 128 bit type that can represent numbers from $1.0 \times 10^{-28}$ to $1.0 \times 10^{28}$. It has a greater precision than float or double, but a smaller power. This prevents implicit casting to and from the decimal type from float and double.

An explanation of C# flow control

Flow control in C# almost the same as in Java. I will outline the differences in the following sections.

An explanation of C# conditionals

Conditionals in C# are virtually identical to Java. C# uses the same true/false keywords as Java and their conditionals, such as if, else, and switch.

An explanation of C# loops

Loops in C# act the same as in Java, with one exception. In Java, there is an enhanced for loop, which allows you to iterate over every element in a collection. In C#, there is a similar feature that allows iteration over a collection data structure or an array. The syntax is as follows:

```csharp
foreach (type item in collection) { /* ... */ }
```

You use this to iterate over each element in the collection. The collection must implement the IEnumerable interface, which defines a method, which returns an iterator, which .NET uses to iterate over each element in the collection.

An explanation of C# methods

Methods are the same in C# as in Java, with a few subtle differences. First is that class methods are not virtual by default. You must declare them virtual, similar to C++. This makes a big difference when you use polymorphism. To declare a method as virtual, you must use the virtual keyword in the same way as you would in C++.

Passing by reference/value in C#

Java always passes primitives by value and objects by reference. C# passes objects by reference, and primitives by value as a default, but that can change. C# uses two keywords, ref and out, to indicate that you want to pass a primitive value by reference.

Functionally, ref and out are equivalent, but in the meanings they convey to the programmer they are different. When you use the ref keyword, it tells the user that the value is being passed by reference and may be modified by the method being called. The out keyword indicates that that particular parameter is one of multiple return values, such as you would see when you pass multiple pointers in C or C++.

It is important to note that you must include these keywords when you both define and call a particular method. An example using the ref keyword is as follows:

```csharp
public static void Main(string[] args)
```
```csharp
{  
    int i = 5;
    test(ref i);
    Console.WriteLine(i);
}

public static void test(ref int i)
{
    i = 10;
}

This example prints out the 10, as you might guess. An example of the out keyword is as follows:

```csharp
public static void Main(string[] args)
{
    int i = 5;
    if (test(out i))
    {
        Console.WriteLine(i);
    }
}

public static bool test(out int i)
{
    i = 10;
    return true;
}
```

The out keyword does not require multiple return values. However, I wanted to show here how you could use this when there are multiple return values.

**Using Classes, Interfaces and Objects**

Classes, interfaces and objects work almost the same in C# as in Java. You do need to be familiar with just a few syntactic and behavioral differences.

First, C#, like Java, supports single inheritance and can implement multiple interfaces. To declare a sub class or a class that implements an interface, it uses syntax similar to C++. For example:

```csharp
public class SuperClass { }
public interface ExampleInterface { }
public class Program: SuperClass, ExampleInterface
{
    //...
}
```

Notice that the super class and interfaces follow the class name declaration on the same line as a comma separated list. You do not have to specify a super class, nor do you have to specify interfaces here. You must ensure, though, that the super class comes before any interfaces.

The second major difference comes with virtual methods in C#. Refer to the section on virtual methods in C#.
Using Enumerations

C# includes support for enumerations similar to Java and C++. In C#, however, you can specify specific values for each value or even a primitive integral type, such as `long`, `short` or `byte` that you want to use as the background data type for the enumeration. To see how to do this, refer to a book or the MSDN documentation. A simple example is as follows:

```csharp
enum FoodType
{
    American,
    Italian,
    Chinese,
    Mexican,
    Other
}
```

Differences between packages and namespaces

In Java, you use packages to help avoid name collisions among classes. In C#, you use a similar structure called namespaces. These are similar to C++ namespaces in declaration, and similar to Java packages in use. To declare a namespace, you simply enclose your class in a namespace block, like the following example:

```csharp
using System;
using System.Collections.Generic;
namespace HelloWorld
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var list = new List<string>();
            System.Text.StringBuilder sb1 = new System.Text.StringBuilder();
            var sb2 = new System.Text.StringBuilder();
        }
    }
}
```

There are a few things to notice in this example. First, you can see the namespace block that encloses the class `Program`. This also allows you to have multiple namespaces in the same file. This works as one would expect it. Second, C# uses the `using` keyword instead of the `import` keyword. Along with that, Java allows you to import specific classes using the `import` keyword, but C# only allows you to import entire namespaces at a time. In the case that there is a naming conflict, the compiler will raise an error, and you will have to specify the full class name including the namespace.

Notice how I used the `System.Text.StringBuilder` to refer to the `StringBuilder` class (This performs the same function as the `StringBuilder` class in Java). You can use fully qualified names to refer to classes in C# just as you would in Java. Also, notice the use of
the `var` keyword. It is a shortcut for not having to write out the fully qualified type name. It assigns the type of the object to be the type of whatever is on the right.

**C# Style and how it differs from Java**

You have probably already noticed a few elements of C# style. First is that Visual Studio automatically formats your code in a certain way. Although you may not like the style it uses, it does encourage a consistent style between all C# code.

The second has to do with casing. Java uses Pascal casing (Pascal casing is where you capitalize the first letter of every word in the identifier, for example: `ThisIsPascalCasing`) for class names, and camel casing (Camel casing is where you capitalize the first letter of every word, with the exception of the first letter in the identifier, for example: `thisIsCamelCasing`) for method and variable names. C# uses Pascal casing for class, property and method names, while local and member variables use camel casing. In some cases, programmers prepend an underscore to the names of private variables.

In addition, when naming interfaces, you should begin the interface name with a capital I. So, if your interface would be called `Callable` in Java, it should be called `ICallable` in C#.

**How constructors work in C#**

Constructors in C# function virtually the same in Java. You should be aware of one difference between C# and C++ constructors: primitive member variables always have an initial value of 0 in C#, while they are uninitialized in C++. For that reason, you may not even need a constructor in C#.

If you do not define a constructor, the C# compiler generates a default constructor with no arguments and the same visibility as the class.

**The differences between virtual methods in C# and Java**

This can be a source of headaches for new C# developers coming from Java. In Java, all methods are virtual by default, but in C#, methods are not virtual by default. This follows the C++ convention. In order to have a virtual method that allows polymorphic behavior, you must include the `virtual` and `override` keywords in the method declarations. Refer to the following example:

```csharp
public class A
{
    public void a()
    {
        Console.WriteLine("A.a");
    }

    public virtual void b()
    {
        Console.WriteLine("A.b");
    }
}
public class B : A
```
```csharp
{ 
    public void a()
    {
        Console.WriteLine("B.a");
    }
    public override void b()
    {
        Console.WriteLine("B.b");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        A a = new A();
        B b = new B();
        A c = new B();
        a.a();
        a.b();
        b.a();
        b.b();
        c.a();
        c.b();
    }

    // This produces the following output:
    // A.a
    // A.b
    // B.a
    // B.b
    // A.a
    // B.b
}
```

There are some important things to notice here. First is that when I declared class A, I declared my virtual method with the `virtual` keyword, while in class B I had to use the `override` keyword to give it polymorphic behavior. The use of the override keyword prevents bugs where you mistype the name of the method you are supposed to be overriding, invoking the incorrect method.

**Differences in access modifiers between C# and Java/C++**

C# uses the `public`, `protected` and `private` access modifiers just like Java and C++. They act the same in C# as in Java and C++. C#, however, also supports two more access modifiers, `internal` and `protected internal`. When you specify a member as `internal`, the member is visible only within its containing assembly. If you specify a member as `protected internal`, it is visible to any code within its containing assembly and to code in any derived class.

---

5 Nagel, et al. 104
An explanation of C# properties

C# supports a feature that neither C++ nor Java support, properties. Properties take the place of the getter/setter code that exists in C++ and Java. A simple example would be as follows:

```csharp
public class Program
{
    public int A { get; set; }

    public void display()
    {
        Console.WriteLine(A);
    }

    public static void Main(string[] args)
    {
        var a = new Program();
        a.A = 1;
        a.display();
        a.A = 5;
        if (a.A == 5)
        {
            Console.WriteLine("a.A is 5");
        }
    }
}
```

First, consider the declaration of property A in class Program. This declares a simple property where you can simply get and set the property A just as you would a public member in Java. In many cases, you want to do some correctness checking or computation in either the getting or setting code. In actuality, the previous example is syntactic sugar for something more powerful. You can change the declaration of A as follows to some checking:

```csharp
int _A;
public int A {
    get {
        if (_A > 0)
            return _A;
        else
            return 0;
    }
    
    private set {
        if (value > _A)
            _A = A;
    }
}
```

First, note that we had to declare another variable in which to store our value. This is because the form seen in the previous version generated a variable internally to store the value, and we did not have to deal with it. Note that in the setting clause, we refer to the value we are assigning with the value keyword. Second, we can change the visibility of the setting, so in this case
have a public get case and a private setting case. In addition, you can entirely remove the get or set case if you want.

Remember that you can put any code you want in the getting and setting code. You can even have properties that have no setting code and just compute the value when you call get.

A comparison of C# and C++ structs

C# supports the use of structs. You can think of them as classes without polymorphism. You can still declare methods in them, and almost everything you would in a class. However, these exist for performance reasons, so you should keep them small. An example follows:

```csharp
struct Person
{
    public int age;
    public string name;
    public int weight;
    public int heightInInches;
    public double calculateBMI()
    {
        return (weight * 703) / (Math.Pow(heightInInches, 2));
    }
}
public class Program
{
    public static void Main(string[] args)
    {
        Person p = new Person();
        p.age = 10;
        p.name = "john";
        p.weight = 160;
        p.heightInInches = 78;
        Console.WriteLine(p.calculateBMI());
    }
}
```

Notice how I declared and used the struct in almost in the same way as a class, just being that they use the keyword `struct` instead of `class`.

Using C# object initializers

C# provides some syntactic sugar for when you want to declare a class with multiple properties set to various values. An example of this and its syntax is as follows:

```csharp
class Person
{
    public int Age { get; set; }
    public string Name { get; set; }
    public int Weight { get; set; }
    public int HeightInInches { get; set; }
    public double calculateBMI()
    {
        return (Weight * 703) / (Math.Pow(HeightInInches, 2));
    }
}
public class Program
{
    public static void Main(string[] args)
    {
        Person p = new Person()
        {
            Age = 10,
            Name = "john",
            Weight = 160,
            HeightInInches = 160
        };
        Console.WriteLine(p.calculateBMI());
    }
}

Notice how you just place a brace enclosed block following the declaration of the object, containing the comma separated list of Property = value assignments.

**Using C# exceptions**

C# provides an exception mechanism very similar to the one found in Java. You use the same try-catch-finally blocks in the same way as you would in Java, with one small difference. If you want to have a catch block that catches all exceptions and you do not reference that object, you do not have to declare the exception as in the following example:

```csharp
try {
    // do something...
}
catch (StackOverflowException e) {
    // handle this...
}
catch {
    // handle the general case
}
finally {
    // clean up
}
```

To throw an exception, simply throw a new instance of some error object. Consider the following example:

```csharp
public void NotImplemented()
{
    throw new NotImplementedException(
        "This hasn't been implemented");
}
```
In C#, you do not have to declare that a method throws an exception, but you do have to create a new instance of that object to throw, unlike Java.

**A comparison of C# and Java Generics**

C# supports generics like in Java. These are also similar to C++ templates. In C#, you can create generic Java classes, interfaces, methods and delegates. There are a few important things to understand about generics. First, unlike C++, MSIL supports generics, and when you use a generic class with a new type, it does not create a new copy of that class for that type as it does in C++.

So far, we have seen some examples of generics, particularly when using collections. The syntax is similar to generics syntax in Java. An example:

```csharp
List<int> lst = new List<int>();
```

The syntax to declare generics, however, is not that different from Java.

An example of a generic class is as follows:

```csharp
public class GenericClass<T>
{
    T one;
    T two;
    public GenericClass(T _one, T _two)
    {
        one = _one;
        two = _two;
    }
}
```

An example of a generic method:

```csharp
public void GenericSwap<N>(ref N one, ref N two)
{
    N temp;
    temp = one;
    one = two;
    two = temp;
}
```

You can also specify constraints for your generic classes. To do so, use the following syntax:

```csharp
public void GenericSwap<N>(ref N one, ref N two)
    where N : constraint
{
    N temp;
    temp = one;
    one = two;
    two = temp;
}
```
Note that constraint is in italics, because you should replace it with an actual constraint. You can replace it with any of the following.\(^6\)

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct</td>
<td>Indicates the type must be a value type</td>
</tr>
<tr>
<td>class</td>
<td>Indicates the type must be a reference type</td>
</tr>
<tr>
<td>IFoo</td>
<td>Indicates the type must implement the interface IFoo.</td>
</tr>
<tr>
<td>Foo</td>
<td>Indicates the type must be either Foo or derive from Foo</td>
</tr>
<tr>
<td>new()</td>
<td>Indicates the type must have a default constructor.</td>
</tr>
<tr>
<td>T2</td>
<td>Indicates the type must derive from a generic type T2.</td>
</tr>
</tbody>
</table>

**A comparison of javadoc and C# XML documentation**

In java, you can write javadoc to generate documentation. C# uses XML documentation to document classes. To start, take a class or method you want to document, and type ///. This will then generate a XML documentation skeleton for you to document that particular class or method. Visual Studio will display the documentation you write in the autocompletion dialogs that appear when referencing this class or method in the future.

**Using delegates and events**

Delegates are a type safe function pointer. An example of the declaration and of a delegate is as follows:

```csharp
public class Program
{
    private delegate int GetAnInt();
    public intGetInt()
    {
        return 1;
    }
    public static void Main(string[] args)
    {
        Program prog = new Program();
        GetAnInt gas = new GetAnInt(prog.GetInt);
        Console.WriteLine(gas());
    }  
}
```

There are a few important things to know about delegates. First, you must declare a delegate type. This is because C# internally creates a class that will represent the delegate. As a result, you can only declare the delegate in places where you can declare a class, such as inside of a namespace or another class. Second, you create and use a delegate as you would a class. In the preceding example, I declared an instance of the GetAnInt delegate, and did it in the same way as a class. In its constructor, I simply gave it the name of the method I wanted to call. Note that I can specify the name of the method being the name of a method bound to a particular object. To invoke a delegate, I simply have to write it as I would a method call.

\(^6\) Nagel, 233
You can also create anonymous delegates, and lambda expressions, however, these are outside of the scope of this document. Refer to http://msdn.microsoft.com/en-us/library/0yw3tz5k.aspx for an explanation of anonymous methods and http://msdn.microsoft.com/en-us/library/bb397687.aspx for an explanation about lambda expressions.

When working with Windows Forms, each control has a set of events that it supports. For each event, there is a property with the += operator overloaded. These properties are collections of delegates. When an event occurs, the operating system calls each delegate. To register an event handler, simply create your delegate, and add it to the collection. When the event occurs, the operating system will call your delegate. To know the exact types of each delegate, refer to the documentation. If you are using the form builder in Visual Studio, it will generate all of this for you.

**A comparison of C++ and C# operator overloading**

One feature that C# supports, which Java does not is operator overloading. You can declare classes that support operators such as +, -, or *. An example of overloading the + operator is as follows:

```csharp
public class Program
{
    public static int operator +(Program lhs, int rhs)
    {
        return rhs + 1;
    }
    public static void Main(string[] args)
    {
        Program prog = new Program();
        Console.WriteLine(prog + 45);
    }
}
```

Note that the method where you overload the operator must be both public and static. You can use different types, and even overload a single operator with different types.

For a complete list of operators you can overload, refer to http://msdn.microsoft.com/en-us/library/8edha89s.aspx. This document also shows the syntax for overloading each individual operator.

**Basic Class Library Items**

**A quick introduction to the C# string class**

C# treats the string class in C# as a basic type, although it is a reference type. It includes a keyword, string, which maps to System.String. Because of this, you do things like string.IsNullOrEmpty(null), or refer to any other class method. (You can actually do this with any primitive type in C#, as the keywords map to classes that represent the actually data types. C# uses a form of autoboxing/unboxing similar to Java)
Strings are not mutable in C#, just like in Java. This means you are subject to the same performance issues in C# as in Java. Consider using the `StringBuilder` class as you would use the class of the same name in Java.

**An introduction to Windows forms.**

To create windows, you must utilize the classes in the `System.Windows.Forms` namespace. Visual Studio includes a form editor that you can used to create windows for your programs. When you create a program, simply select the Windows Forms project type. The best way to become familiar with Windows Forms is to experiment and see what you can do.

It is very useful to know how to display debugging information with a message dialog box. When you are using `System.Windows.Forms`, simply call

```csharp
MessageBox.Show("This is your message");
```

This can be very handy to get debugging information when you do not have a console to which you can print debugging information.

**Using the .NET Collections**

In my opinion, the .NET collections are not yet as developed as Java collections. They do have several classes, though, that very handy. I recommend using the Generic versions of the collections found in the `System.Collections.Generic` namespace. Here is a quick listing of some of the collections I have found to be the most useful.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>List&lt;T&gt;</code></td>
<td>A linked list data structure</td>
</tr>
<tr>
<td><code>Queue&lt;T&gt;</code></td>
<td>A simple FIFO queue</td>
</tr>
<tr>
<td><code>Stack&lt;T&gt;</code></td>
<td>A simple LIFO stack</td>
</tr>
<tr>
<td><code>ArrayList&lt;T&gt;</code></td>
<td>An array list data structure</td>
</tr>
<tr>
<td><code>BitArray</code></td>
<td>An array for quick manipulation of arrays of single bits</td>
</tr>
<tr>
<td><code>SortedList&lt;Key, Value&gt;</code></td>
<td>Provides a list that is always sorted</td>
</tr>
<tr>
<td><code>Dictionary&lt;Key, Value&gt;</code></td>
<td>Provides a key/value mapping</td>
</tr>
</tbody>
</table>

Keep in mind, that like Java, these data structures are not thread safe. Refer to the .NET threading libraries section on how to synchronize access to these data structures.

**Using .NET I/O**

.NET provides a layered I/O system, very similar to Java. In this section, I will discuss the `StreamReader` and `StreamWriter` classes, which are wrappers around other streams for reading and writing text to files. An example using these classes is as follows:

```csharp
using (StreamWriter sw = new StreamWriter("test.txt"))
{
    sw.Write("test");
}
```
This should seem somewhat straightforward, except for the using construct.

**Using the using construct**

The reason why we use the using construct is the same reason why you have to close streams in Java. You do not know exactly when the garbage collector will destroy the stream, and the stream thereby closed.

Because we do not know when the garbage collector will destroy objects in C#, we cannot use object destructors as we would in C++. To solve this issue, classes such as streams implement the IDisposable interface. This interface provides a single method, called Dispose. This method, in the case of IO streams, closes the stream.

The using construct takes as its only argument the initialization of an object that implements the IDisposable interface. At the end of the block, .NET will call the Dispose method on that object, and it will fall out of scope. This ensures that you properly close your streams.

It is important to note that it will not generate an error if you do not use the using block. If you do not you must explicitly call the Close method for each stream. I strongly recommend you get in the habit of always using the using block. If it is a habit to use it with IO, you will not run into bugs with open streams.

**How to consult the MSDN documentation**

The MSDN documentation contains a huge amount of information about the .NET class library and the C# programming language. I often find it useful to read the documentation for individual classes. To do that, go the MSDN homepage at [http://www.msdn.com/](http://www.msdn.com/). Click on library, and then navigate the following path to the .NET class library directory in the tree to the left: MSDN Library → .NET Development → .NET Framework 3.5 → .NET Framework Class Library.

Searching with Google for things on the MSDN site is almost easier than searching with the MSDN search tool. If you try searching for thing and don’t have any luck, give Google a try.

**Where to go from here – Using advanced language items**

Several other useful parts of C# are outside of the scope of this document. To learn these advanced features, consider getting a book on C#. One that I own and recommend is *Professional C# 2008* by Christian Nagel, et al. In addition, MSDN is a very good resource, as is Google.
**Good resources on LINQ and functional programming**
Using functional programming and LINQ can help you write far better code, and be much more productive. There are several excellent resources on learning this. I personally recommend this site: [http://blogs.msdn.com/ericwhite/pages/FP-Tutorial.aspx](http://blogs.msdn.com/ericwhite/pages/FP-Tutorial.aspx).

**Using Unsafe code**
As you may well be aware, there are several cases when using the automated garbage collector is not what you need. You may also want to have and use C/C++ style pointers. With C#, you can declare sections of code as unsafe and gain access to these features. To learn how to do this, read [http://msdn.microsoft.com/en-us/library/aa288474(VS.71).aspx](http://msdn.microsoft.com/en-us/library/aa288474(VS.71).aspx).

**Conclusion**
C# is a powerful language that is very feature complete. It provides a level of safety and control that can help developers write very flexible, fast and bug-free code.

**References**

