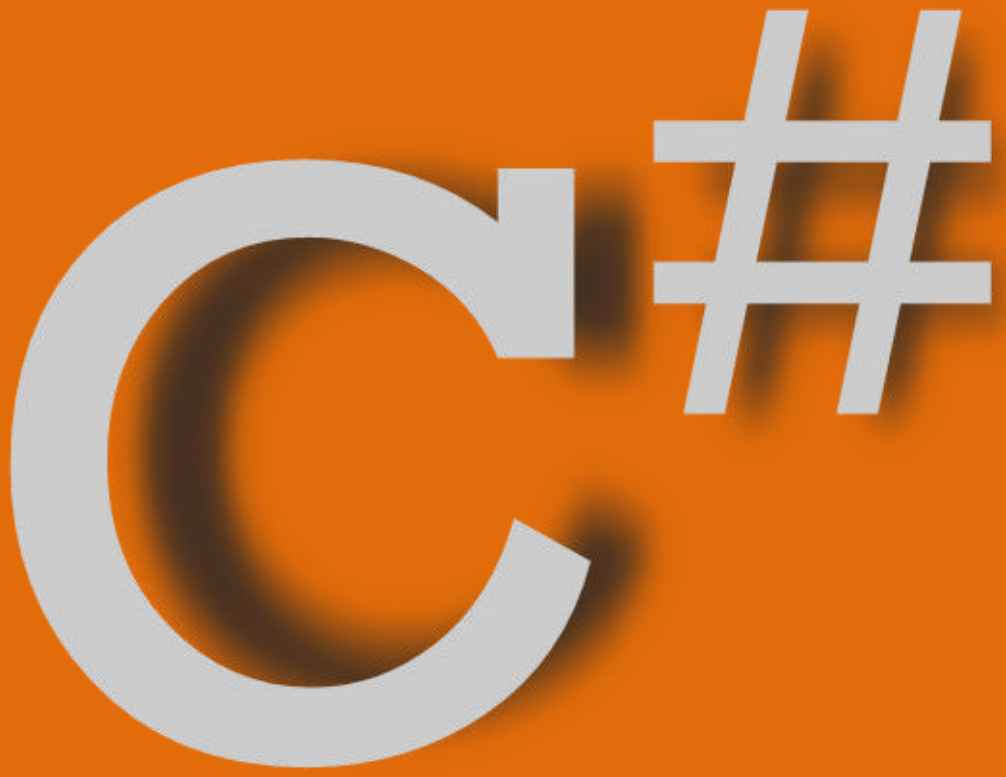
# C# From Java



**Rob Miles**

**Edition 3.1 2008-2009**

**Department of Computer Science**
**University of Hull**

# Contents

 The author can be contacted at:

The Department of Computer Science,
Robert Blackburn Building
The University of Hull,
Cottingham Road
HULL
HU6 7RX
UK

Email: rob@robmiles.com
Blog: www.robmiles.com

If you find a mistake in the text please report the error to foundamistake@robmiles.com and I will take a look.

Vsn. 3.1
1/30/2009 9:45:00 AM

# Introduction

These notes should help you convert a program from Java to C#. I assume that the reader is an experienced Java programmer. The notes also document decisions which I have made which may or may not be sensible. I've written them in FAQ form as that seems to be the most useful way to present the information.

I hope that those that know less than me will find them useful. I hope that those who know more than me won't find too many things wrong…

# Big Project Blues

This section deals with project management/creation issues. We have a fairly large project, in this case around forty or so class files arranged in a number of packages. It might not be big to you, but it is to us.

## *What do I do for development tools?*

One useful addition to the Visual Studio range is Visual Studio Express, which is presently available for free. It involves a download of around 300 MB, but it is fully functional. It is of particular interest to students, who could perform all the coursework on our first year with this product.

For the first part of the year I advise students to make use of the command line compiler, and edit their programs in text files using notepad. There are two reasons for this.

The first is that the complexity of the Visual Studio environment and the need to create a project can be daunting for a newcomer to programming.

The second reason is that I want the students to have a solid understanding of the compile/execute process, and the fact that the compiler is a program that converts text based source code into an executable version. Seeing the .exe files is a very good way of making this point. In Visual Studio it is not immediately clear where the program file runs from, and this can be confusing.

## *How do I break a system up into chunks?*

My friend Kevin, who knows about these things, says that you break your projects up by using, er, projects. A *project* is a subcomponent of a *solution*. A project can be placed in a separate directory with all its class files. Visual Studio solution can contain lots of projects.

The output of a project is an *assembly.* This is analogous to a Java Archive (jar) file in that it can contain classes and other resources (forms, bitmaps, sound samples etc). An assembly can be either an executable program (.exe) or a Dynamic Link Library (.dll). A dll assembly just contains classes that can be picked up by the master asembly in your solution which produces the executable. This is the class that contains the Main method (see below).

Don't confuse these files with old school .exe and .dll files which have been part of Windows for years. These must be processed by the .NET runtime system which performs Just In Time compilation on them to make them run, just as the Java Virtual Machine (JVM) runs Java class files. Fortunately most versions of Windows XP and all versions of Vista now have the .NET runtimes built in.

### *Where does the Main method go in C#?*

Java doesn't make a distinction between library files and executable files, a java source file compiles into a class file. A familiar Java trick is the inclusion of a **`main`** method in all classes. This provides a way of performing isolated tests on each class simply by running it. Some Java programmers like this, others loath it. C# and Visual Studio are quite unambiguous on this one. Only a project which produces a .exe file is allowed to have a **`Main`** method (which starts with a capital letter).

Unlike Java you can use one of a number of different **`Main`** method signatures, depending on whether or not you are interested in the result produced by your program or want to feed it parameters.

If your solution contains multiple project that produce executables (see above) each of the projects can contain a **`Main`** method which will be entered when the program starts. You can nominate one of the projects in the solution as the one which is started by Visual Studio when you press the run button. This means that you can have a set of business classes and then multiple executable classes (a CLI version, a GUI version and perhaps a test harness) all in the same solution, and pick which one you want to work with.

### *What is the Automatic Conversion tool like?*

I've had a brief fling with the automatic conversion system. My advice is don't bother unless you are after something very quick and very dirty, and you happen to have written the code in Visual J++; neither of which apply to me.

My gut feeling is that if you want to make proper use of all that C# offers you will need to convert the application by hand. The good news is that this will probably result in smaller, faster and clearer code. The bad news is that you have to do it.

## All Quiet in the Studio

This section deals with how you use Visual Studio (a **really** nice tool) to write your programs.

### *How do I set a bookmark in the source using Visual Studio?*

You set a bookmark by using the sequence CTRL+K CTRL+K (that is CTRL+K twice). This is sort of like old style Borland C (or Wordstar for very old people) except that the second character needs to have control held down as well. This toggles the bookmark on and off.

You jump to a bookmark by using CTRL+K CTRL+N. If you have lots of bookmarks this sequence moves you to the next one in the sequence.

### *How do I get my source code laid out for me?*

Visual Studio makes an attempt at laying out your code, but it sometimes gets mixed up. If you want to force it to lay the code out for you the commands are:

CTRL+A (select all the text in the edit window) CTRL+K CTRL+F

### *Why do class members start with capital letters?*

Java programmers will love this. The convention in C# is that public members of classes have a capital initial letter in their name. This goes directly against the Java

convention. Only parameters, local variables and private members have lower case characters as the first letters of their identifiers.

## How do I create a new class in a Visual Studio Project?

Visual Studio will let you make a new class but, unlike Java, it doesn't have a strict mapping between class names and filenames. Indeed you can put classes from several different namespaces (the C# equivalent for packages – almost) in files of any name.

I like to have the names of the classes match the files (which is probably a bit sad) and also to have a directory hierarchy which matches the namespace one. However, C# does not insist on this, and lately I've began to relax about this and put files where they make the most sense.

## How do I put the documentation comments into my source

Java has Javadoc. C# doesn't. C# has a commenting system which is very similar, where you put keys into the source code which are picked up and used to generate program documentation. Most Java development environments have a neat way of creating the comment structures which drive the Java documentation process. Visual Studio didn't seem to have any of this until I moaned to a Microsoft guy (step forward George Conard) about this terrible limitation. Then he showed me how to do it. It is very simple, very easy to use, and very neat.

Just above the start of your method press forward slash three times:

```
/// <summary>
///
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
public int AddOneTo ( int value )
{
      return value + 1;
}
```

All the comments are generated automatically. All you have to do now is fill in the text in the empty spaces to produce the program documentation. Note how you can enter a description of the method, information about the parameters and also the return information just by filling in text in the appropriate positions. Note also that when you do this the very nice IDE puts in leading **///** sequences to make sure that your code is a proper comment.

One thing to remember is that this automatic behaviour only works if Visual Studio is working with a project. If you just have source file open you will find that /// doesn't do anything.

To get the actual documentation you can use a tool like ndoc:

**http://ndoc.sourceforge.net/**

to build the web pages.

## How do I get full screen source code edit in Visual Studio?

You can select this from the view menu, but I use ALT+SHIFT+ENTER. You can do most of your work (including compiling and debugging) using this nice large view, but it is best if you make sure that all the sources files you want to use are open before you switch to this view because opening extra ones is a bit of a pain.

## How do I link a C# solution?

With Java you don't have to think about linking. If the compiler is happy your program will work (leaving aside potential nasties with the class path). In C# and Visual Studio things are different. You have to make sure that the compiler can find stuff. You do this by using **`using`**; for example if you want to use the **`Forms`** library in a class you have to put:

```
using System.Windows.Forms;
```

- at the top. This is directly analogous to Java **`import`**.

However, unlike Java you also have to tell the Visual Studio project that you are using this set of resources. To do this you go into the Solution Explorer and right click on the References leaf just underneath the tree entry for the project that you want to use the particular assembly. Then select Add Reference from the context menu which pops up and pick up your reference.

You can add references to .NET resources, assemblies in your project or COM objects. A sign of problems with your references is when the auto completion stops working. This will be followed by a whole bunch of build errors.

If your solution makes use of multiple assemblies these will be referenced by the executable and you must make sure that all the assemblies are distributed, otherwise your program will file just like a Java application would if it was shipped with some class files missing.

## How do I make a distribution of my program?

You need to add a project to do this for you. Open the Solution Explorer and right click on the Solution item at the very top. Select Add and then Add New Project when it opens up. From the New Project dialogue box select the tree item "Setup and Deployment Projects" and then pick Setup Project. Give the project a name and sensible location.

You can now build a setup, but it won't do anything (I actually did this – oh how I laughed). You need to add some files to the setup project to get it to work. Right click on the project in the Solution Explorer and select add from the Context Menu and then move on to Project Output. This means that you will actually get your program deployed. To build the setup you need to right click on the project and select build.

Note that the build behaviour is controlled by the Release/Debut selection in main window. The output is actually placed in the directory you specified for the project, with separate directories for build and release. You can copy the MSI file onto another machine and run it to install the program. You don't get a shortcut for the desktop or anything, but the program can be made to run from the appropriate Program Files directory.

# Programming Matters

These bits deal with flotsam and jetsam which I've picked up just writing the code.

## How do I store the date and time?

The **Date** class in Java has been replaced with **System.DateTime**. The good news is that this class is actually able to perform date arithmetic for you. Again, the bad news is that you have to perform the conversion.

## How do I get the current date and time?

You can use the static **Now** property of the **System.DateTime** class:

```
CurrentTime.Text = System.DateTime.Now.ToLongTimeString();
```

## How do I copy a *DateTime* value?

You need to convert a date into a **Long** value of ticks which can be passed into the constructor to make a **DateTime** instance:

```
NewTime = new System.DateTime(OldTime.Ticks);
```

## How do I do date comparison?

If you subtract one **System.DateTime** instance from another you get an instance of the **System.TimeSpan** class as a result. You can compare these directly to find out if a given time has elapsed:

```
// initialise our timer
TStart = new System.DateTime(System.DateTime.Now.Ticks);

// let some time go by

System.TimeSpan WarningLength = new System.TimeSpan(0,9,0);

System.TimeSpan CurrentLength = System.DateTime.Now - TStart;

if (CurrentLength > WarningLength) {
      // get here if more than 9 minutes have gone by
}
```

The above code will obey the line shown when more than 9 minutes have elapsed. The **WarningLength** timespan has been set to 0 hours, 9 minutes and 0 seconds.

## How do I get a random number?

Random numbers are quite easy. You have to make an instance of a **System.Random** object and then use that to produce numbers for you:

```
System.Random r = new System.Random();
```

The constructor for this class is overloaded. There is a version you can call with an integer seed if you always want the same pseudo-random sequence. If you call the version with no parameters you get the generator seeded from the system clock.

Once you have your random object you can ask it to give you random numbers:

```
int coin = r.Next(1); // give me a value between 0 and 1
int diceSpots = r.Next(1,7) ; // give me a value between 1 and 6
int big = r.Next() ; // give me a value between 0 and maxint
```

Note that in the call of the Next method the upper limit is exclusive, in that random numbers produced will not include that value.

There are also methods which can give you an array full of random numbers, and also floating point numbers. Remember though that these random number sequences are not cryptographically strong, but they are fine for things like games. For heavy duty randomness you should take a look at the **`RandomNumberGenerator`** class in the **`System.Security.Cryptography`** namespace.

### *How do I terminate the program with extreme prejudice?*

Sometimes your program just has to die. To perform such a mercy killing Java has **`System.exit(n)`** where **`n`** is the last message value from your dying program. C# has something virtually identical:

```
System.Environment.Exit(0);
```

The parameter given to the **`Exit`** method will be passed back to the operating system.

## Through a Window darkly

These are items to do with the windowing output.

### *How do I create a modal window?*

Modal windows are the useful ones which stop everything else while the user fills in a form. You determine whether or not a window is modal by the way that you show it. Assuming that you have an instance of a **`Form`** called **`MyForm`** from **`System.Windows.Forms`** you can create a modal version by going:

```
MyForm.ShowDialog();
```

Or

```
MyForm.ShowDialog(ParentForm);
```

- if you have a reference to the parent form. If you want a **`Form`** which is not modal, and therefore harder to program, you can use the **`Show`** method:

```
MyForm.Show ();
```

## How do I stop a window from closing?

Sometimes you want to get control as a window is being closed, for example you may want to insist that the user saves their file before exiting from their edit. Alternatively you may want to stop the user from closing a form if they have not done some action.

In C# you do this by adding a delegate to the closing action of the form:

```
AuthorForm.Closing +=
      new System.ComponentModel.CancelEventHandler(CloseHit);
```

**AuthorForm** is a reference to a class derived from **System.Windows.Forms.Form**. The class containing this line of code also contains a method called **CloseHit**:

```
private void CloseHit (object sender,
      System.ComponentModel.CancelEventArgs e)
{
      if (AuthorCompletedOK)
      {
            e.Cancel=false;
            return;
      }

      // not saved OK - must ask if it is OK to shut down

      string message = "OK to shut down?" ;
      string caption = "Authoring";
      MessageBoxButtons buttons = MessageBoxButtons.YesNo;
      DialogResult result;
      result = MessageBox.Show(message, caption, buttons);

      if (result == DialogResult.No)
      {
            AuthorForm.Show(); // because my window is modal
            e.Cancel = true;
      }
      else
      {
            e.Cancel = false;
      }
}
```

I signal that I don't want the close to complete by setting to true the **Cancel** property of the **CancelEventArgs** instance which is passed in to the method call. Note that I also need to call a **Show** of the form. I need to do this because my **AuthorForm** happens to be a modal dialog, and these are hidden just before they are closed. If it is OK to close the form I set the **Cancel** property of the **CancelEventArgs** instance to **false**.

Note that my class contains a **bool** flag called **AuthorCompletedOK**. This is set to **true** by me if it is OK to close the window. I needed to do this because the behaviour of the closing event is slightly different from that in Java Swing. This closing event will be fired when the form closes, irrespective of whether this is a

close request from the close button on the top line or a call of the **Dispose** method to close the form down "naturally". My dispose code looks like this:

```
private void SavePressed(object sender, System.EventArgs e)
{
      // save the document
      DoSave ();
      AuthorCompletedOK = true;
      AuthorForm.Dispose();
}
```

This method is bound to the pressed event for the save button. It sets the flag and then disposes of the form. When the form is disposed my closing event runs, but because the flag has been set to say that all is well it doesn't show the confirm dialog.

## How do I ask a user yes or no?

You need to make a **MessageBox**:

```
string message = "OK to shut down?" ;
string caption = "File Save Request";
MessageBoxButtons buttons = MessageBoxButtons.YesNo;
DialogResult result;
result = MessageBox.Show(message, caption, buttons);
if (result == DialogResult.No)
{
      // stuff you do if the answer is no
}
else
{
      // stuff you do if the answer is yes
}
```

The **MessageBox** and **MessageBoxButtons** are in the **System.Windows.Forms.** namespace.

## How do I give a title to a Windows Form?

If you want to give a title to your form you use the **Text** property of the form instance:

```
MyForm.Text = "User Edit";
```

## Where have the layout managers gone?

There are no layout managers in Visual Studio/Windows Forms. This would be bad news except that there are some features which are almost as useful. You can anchor components to sides of your window so that they move and/or grow when the window changes shape. You can also dock components to the sides of your window so that they change in size along that edge.

If you want to have just one component in your window change size when you move things life is good. If you want to have several of them (and control how

much of the extra space that each component gets as with Java's much maligned **GridBagLayout**) then your life is about to take a downturn just like mine has.

# Class Acts

Things you have to do which are different in terms of Java and C# classes and the way they work.

## *What is the relationship between const, final and static?*

In Java I make my defined constants **final** and **static**. In C# you just use **const**. It implies **static** so you have to edit all these out of the source when you convert it. I guess this makes sense, but it is a bit of a pain.

## *Where has the Vector class gone?*

There is no **Vector** collection class in C#. This means that you have to use an **ArrayList** instead. One nice thing about C# collections is that they can be indexed. This means that I can add subscripts to an instance of a collection to get particular elements out of it.

The latest version of C# has generics, which means that you can create a typesafe **List** of references to any class.

## *How do I use enum in C#?*

In C# enums are back (nobody is quite sure why they went away in Java). Using them is quite easy. First you declare them in the class that you want to use them:

```
public enum FlowTypeValue
{
      flowToRight,
      belowFlow
};
```

Remember that if you want other classes to use the **enum** you have to make it **public** or **protected**. Otherwise the **enum** can only be used in the class it is declared within. If the **emum** is public, in another class you can do things like:

```
DataEdit.DataEditItem.FlowTypeValue x =
      DataEdit.DataEditItem.FlowTypeValue.belowFlow;
```

- to create a variable of the **enum** type and then pick up one of the values. I forgot to mention that I created the **FlowtypeValue** in a class called **DataEditItem** in an assembly called **DataEdit**. Note that all the rules about linking should be adhered to. See "Come back linking, all is forgiven" for more details.

## How do I call a constructor in a parent class?

The protocol for calling a super constructor (i.e. calling the constructor in a class you are overloading) is different. Rather than use **super** you use **base**, and you put it in a rather funny place. Rather than:

```
public MyClass (String name, String otherstuff) {
      super (name); // pass name into the super constructor
      // do things with otherstuff here
}
```

You need to write:

```
public MyClass (string name, string otherstuff) :
      base (name) // call the super here
{
      // do stuff with otherstuff here
}
```

I'm not sure about my layout, but it is probably a good idea in that it emphasises the fact that the call is made at the very start of the constructor execution.

## How do I call another constructor in the same class?

If you call another constructor in the same class you still use **this**, but you use the same strange layout as for calling a parent constructor:

```
public MyClass (string name, string otherstuff) {
// full blown constructor here
}

public MyClass (string name ) :
      this (name, "Default Otherstuff Value")
{
}
```

## Where have implements/extends gone?

When you extend a parent class you don't say **extends**. Instead you use the multi-purpose : (colon) character. This character also replaces the **implements** Java keyword, so in a class header you might get:

```
class NewOne : Fred, Jim, Ethel, Nigel {
```

The first class name in the list you give must be the one you are extending, but you are not forced to extend a class. This means that **Fred** could be a class or an interface. There is a kind of convention that interface names are preceded with **I**:

```
class NewOne : Fred, IJim, IEthel, INigel {
```

Note that I would never create things with such stupid names, and that you shouldn't either.

## Why can't the compiler find my implementation of an interface?

If you implement a method in an interface that method is implicitly **public** in the interface definition, even if you don't ask for it. This means that your implementation of the method must also be made **public**, otherwise the compiler won't find it.

Note that you can get around this by using a really neat feature of C#, *explicit interface implementation*. This gets around the hassle in Java when you don't know from a method declaration whether or not it is there to override a parent method, implement an interface or on a whim. In C# you can explicitly state that the method implements a particular interface:

```
bool UserManagement.IMenuProviderCheck.CanUseMenu (
      UserManagement.UserStore store, UserManagement.User user )
```

The method **CanUseMenu** is specified in the interface **IMenuProviderCheck**.. The code above is in a class which implements this interface. I use the interface name in the name of the method to indicate exactly what this method is here for.

## Why won't my inner classes work properly?

I *love* inner classes. I particularly like the way that they have access to all the members of enclosing class. This makes them super for edit interfaces. You make an inner class which extends a windows component (either a panel or a form) and then let it twiddle with the innards of your object. Add a factory method to make an instance of this edit component (which can be authenticated at that point) and you have a really neat way to edit your objects.

The only things that work against this approach is that you may have difficulty creating the component using the GUI editor (my tip is to create it as a new external class and then block copy the class source inside) and that some programming purists insist that it breaks good coupling and cohesion policy (you can just ignore them – it works).

When you move to C# from Java you find that things are a little different. In Java an instance of an inner class is part of an enclosing class, i.e. you make the inner instance straight from the enclosing one. Then you do weird things with **this** to get hold of the members of the enclosing instance.

In C# there is a subtle change which irritated me greatly until I decided that it is really much better. Inner classes in C# are always **static**. This means that you can't get hold of the enclosing instance because there isn't one. This drove me up the wall because for a while it looked like there was no way of doing my edit trick. The answer is very simple; just pass the constructor of your inner class a reference to an instance of the enclosing one. Then twiddle with the contents of this reference. It is neater because it means that the code is much clearer. Hopefully this code example below will make it much clearer:

```
class DataClass {

      // lots of private members I want to edit
      private string MyData;

      // my edit form for the class
      private class EditorClass : Form
      {
            // reference to what we are editing
            DataClass ThisClass;
            public EditorClass ( DataClass thisClass )
            {
                  // copy a reference to the enclosing instance
                  this.ThisClass = thisClass;
                  // use it to get my properties to change
                  ThisClass.MyData = "new value";
            }
      }

      // the edit method for my class
      public void DoEdit ()
      {
            // pass a reference to myself into the edit class
            // constructor
            new EditorClass(this);
      }
}
```

Note that I wouldn't normally do the editing in the constructor; I'd probably call **ShowDialog** as the last line the constructor so that the editing starts instantly. If you have a call of **Dispose** tied to the Exit and Cancel buttons in your form this means that when the editing is finished the **DoEdit** method will return.

## *Why won't my overriding method run?*

If you want to override a method you must say so. If you leave it out you might not get the effect that you want. As an example, consider **ToString**, the C# version of Java **toString**. Like Java it returns the string representation of the object. Like Java you override the parent method to get your own behaviour. Unlike Java you have to explicitly override it using the override keyword:

```
public override string ToString ()
{
      return ItemName;
}
```

If you miss out the **override** modifier the program doesn't call your **ToString** method. And it compiles OK. This can be very confusing. Note that for overriding to work at all you have to declare the method you are overriding as **virtual**.

## How do I use delegates?

Delegates are nice. They are as good as pointers to functions in C++ but without the heartache. They get seriously fun when you pass them into and out of methods in parameters. You declare a delegate as a type:

```
public delegate bool CheckResult ( string newValue );
```

Now I have a delegate type called **CheckResult** which I can use to create references to methods which fit the signature (and no others). Note that I have made this type **public**, so that I can create these outside of the class. In other words I can do things like:

```
CheckResult CheckMethod;
```

Note I haven't got a method yet, just a delegate reference which can refer to methods which are of type **bool** and accept a **string** as a parameter.

So if I have a method as follows:

```
public bool EmptyStringCheck ( string newValue )
{
      if (newValue.Trim().Length == 0)
      {
            return false;
      }
      else
      {
            return true;
      }
}
```

It is a method which fits the bill so I can go:

```
CheckMethod = new CheckResult (EmptyStringCheck);
```

Note that we are creating a new delegate instance which contains the method that we want to use; delegates do not function as references themselves.

Now I can call the method which the delegate presently refers to by simply calling it:

```
CheckMethod ( "Fred");
```

This would result in a call to **EmptyStringCheck** if the assignment had been performed.

Big fun comes when I use a delegate as a parameter to a method call:

```
public DoStuffThatNeedsACheckMethod( CheckResult CheckMethod )
{
      string thingWeWorkWith;
      // do stuff with strings and when we need to check them:
      if (CheckMethod (thingWeWorkWith ) )
      {
            // pass the test
      }
}
```

We can pass into the method a delegate which specifies the thing that needs to be invoked to perform the verification:

```
CheckResult MyCheckMethod;
// assign MyCheckMethod to the desired method
DoStuffThatNeedsACheckMethod (MyCheckMethod);
```

You can have lots of method calls attached to a single delegate. You can use the **+=** operator (which has been suitably overridden) to add new ones:

```
CheckMethod += new CheckResult (InvalidStringCheck);
```

Now when the delegate is invoked both **InvalidStringCheck** and **EmptyStringCheck** will be called. If you don't like overloaded operators you can use

If you want to remove a method from a delegate you use **-=** operator:

```
CheckMethod -= new CheckResult (EmptyStringCheck);
```

This would mean that only the call of **InvalidStringCheck** would be used. I don't particularly like this format, in that it implies that you have to create a new thing just to get rid of it – but this is how it works.

Other delegate stuff I've discovered:

Delegate's methods are called in the order that they have been added to the delegate. In the above sequence, when we have two delegates assigned the method in the sequence as given above **EmptyStringCheck** would be called first.

If the delegate methods return a value (in the case our example above we return a **bool**) then the value of the most last call (i.e. the method which was added most recently) is returned to the caller, in the case of the example above this means that the result of **InvalidStringCheck** would be returned.

If you remove all the methods from a delegate you will get a **NullReferenceException** thrown if you are stupid enough to call the delegate.

## How do I use properties?

Properties are luverly. They are extremely useful and make your code a lot cleaner. Essentially you can have code like:

```
x.Width = 99;
```

This looks like an assignment to a member of a class, but it can be much more than that, and result in code running. The **Width** property could be managed like this:

```
class ThingWithWidth {
      private int WidthValue;
      public int Width
      {
      get
      {
            return WidthValue;
      }

      set
      {
            WidthValue = value;
      }
   }
}
```

This very simple minded exercise doesn't do much, but you should get the picture from it. When I perform the assignment to the property the set portion runs. The keyword **value** is set to the value of the incoming property. In the code above I do a simple assignment, but you could validate the value and throw an exception if you don't like it.

I've decoupled the names of the property value from the value of the property (my convention is to put the word value on the end of the name of the internal value). Of course you don't actually have to have a value inside the class; you could calculate a result rather than return a member.

When it comes to setting the value you can get lumps of code to run when you change the value of your property. This makes creating state machines quite fun. Furthermore, you don't have to implement both a getter and a setter, you can have just one so that you can create write only (or read only) properties. You can have lots of getters for the same property; perhaps monsieur would like the speed in KPH as well as MPH.

The only downside is that you must be aware that substantial lumps of code can run when you perform innocent looking assignments. The way round this is to get the hang of the little icons which Visual Studio shows you in the property lists.

## How do I create an Abstract Property?

You can put an abstract property into a parent class. This forces a child class to override the property and provide their own version of the **get** or **set** actions. You do this by making the property abstract and just giving the word get or set (or both) in the property definition:

```
abstract protected string MediaName
{
      get;
}
```

This adds a property called **MediaName** which is of type string and can only be "got". To implement this property I must use **override** correctly:

```
protected override string MediaName
{
      get
      {
            return "Image";
      }
}
```

This overrides the parent.

## How do I serialize a class?

Unlike Java, where you denote a class as able to be serialised by implementing an interface, in C# you use the **[SERIALIZABLE]** attribute:

```
[SERIALIZABLE]
public class MyClass {
      public string message;
}
```

The class **MyClass** can now be written down a serialized stream. Note that if it holds references to other classes these must also be serializable or the process will fail with an exception. To write the class out I use:

```
try
{
      FileStream outStream = new FileStream(
            "myfile.dat",FileMode.OpenOrCreate,FileAccess.Write);
            IFormatter outFormatter = new BinaryFormatter();
            outFormatter.Serialize(outStream, this);
            outStream.Flush();
            outStream.Close();
}
catch ( System.Exception e )
{
      Console.Writeln("Something bad happened : " + e ) ;
}
```

This code writes out **this** to a binary stream. If I want to read it back I use:

```
public MyClass FetchClass (string Filename)
{
      FileStream inStream;
      try
      {
            inStream = new FileStream (
                  Filename, FileMode.Open, FileAccess.Read);
      }
      catch ( System.Exception e )
      {
            Console.Writeln ("Something bad happened : " + e ) ;
            return null;
      }

      IFormatter inFormatter = new BinaryFormatter();
      object newStore;
      try
      {
            newStore = inFormatter.Deserialize(inStream);
      }
      catch (System.Exception)
      {
            Console.Writeln ("More bad news : " + e ) ;
            return null;
      }
      finally {
            inStream.Close();
      }
      return newStore as MyClass;
}
```

Note that my method returns **`null`** if it doesn't get a proper object, I think it may be better if it threw an exception in real life. Note also that I use the **`as`** operator to get the object returned by the formatter into an object of type **`MyClass`**. This has the advantage that if the downcast fails (i.e. if the thing that I have just read cannot be regarded as a **`MyClass`** instance) I don't get an exception; instead the operator returns a **`null`** reference, which fits in with my error signalling mechanism.

## How do I mark a member as not to be saved by serialisation?

In Java you use the **`transient`** modifier to mark a member of a class as not for saving during serialisation. In C# you mark the members with the **`[NonSerialized]`** attribute:

```
[NonSerialized] int i;  // don't save the counter
```

## Why does my serialialization not work?

Unlike Java, where marking the parent class as implementing serializable interface will also flag all the children, in C# you seem to have to add the attribute to all the child classes as well as the parent before the save process works correctly.

## How do I find out what classes I've got?

I like writing code which automatically finds out what classes are around and then configures itself accordingly. I regard this as a constructive form of laziness, in that if I add new items they should just be picked up and used. Java is quite good for this, and C# is even better.

In Java I used to get the names of all the files in a directory, construct classes from these files and then inspect them for interfaces and parents which make them useable in particular contexts. In C# it all revolves around *Assemblies*. You get the contents of an assembly as an array of types and then inspect the types that you get. In the code example I'm looking for classes which extend the **`EntityDrawing`** class. I can then use these in menus and stuff. If I create a new entity I can just drop it into the assembly and the rest of the system will pick it up. The code we are going to write will find these for me. To take these chunks in turn:

```
Assembly FriendsAssembly ;
// Load the assembly by providing the type name.
FriendsAssembly = Assembly.Load("FriendsEnvironment");
```

This will load my assembly (which I've called **`FriendsEnvironment`** for no particular reason). I now have an instance of an **`Assembly`** object which I can fiddle with.

```
// get an array of the types in the assembly
System.Type [] FriendsTypes = FriendsAssembly.GetExportedTypes();
```

I now ask my assembly to give me an array of types which are supported by it. Once I have my list I can search through the list for compatible types.

```
System.Type EntityDrawing = null;
// first find the parent class for the EntityDrawing items
foreach ( System.Type ThisType in FriendsTypes)
{
      if (ThisType.Name.Equals("EntityDrawing") )
      {
            EntityDrawing = ThisType ;
            break;
      }
}

// if no parent found we have a broken assembly - quit
if ( EntityDrawing == null )
{
      Console.WriteLine("EntityDrawing class not found");
      System.Environment.Exit(1);
}
```

I'm not particularly proud of this code, but it does work. Essentially I'm looking for a parent abstract class called **`EntityDrawing`**. This has other classes which extend it, which are the ones I want to use. I search through the list of types until I find one with this name. I can then hit against this type to find matching ones. If I don't find this class I am officially stuffed, and so I print a message and give up

(perhaps a message box would be more sense in real life..). Now I have my reference type (as it were) I can look for matches:

```
// List of references to the types I want to use
System.Collections.ArrayList DrawingTypes =
      new System.Collections.ArrayList();
// now build a list of entities which we can use
foreach ( System.Type ThisType in FriendsTypes) {
      if ( EntityDrawing.IsAssignableFrom(ThisType) &&
            !ThisType.IsAbstract )
      {
            DrawingTypes.Add(ThisType);
      }
}
```

I put references to all the available types in a collection which I can then iterate through. For each of the types that I started with I check to see if I can assign my drawing type to it. If the answer is yes I have a match. I have to also check to make sure that the type is not abstract (since I want to construct these beasts). At the end of the loop I have a collection which contains references to the types that are available:

```
foreach ( System.Type DrawType in DrawingTypes )
{
      Console.WriteLine ("Drawing type : " + DrawType.Name);
}
```

This code just prints out the name of each type as it goes.

## *How do I make an instance of one if the classes I've found?*

This follows on from the above in that once you have a type object you will want to make instances of the class that you have found. You use the assembly to do this for you:

```
foreach ( System.Type DrawType in DrawingTypes )
{
      EntityDrawing e = (EntityDrawing)
      DrawType.Assembly.CreateInstance(DrawType.FullName);
      Console.WriteLine ("Drawing type : " + e.MenuName);
}
```

This code uses the **`Assembly`** property of the type and calls the **`CreateInstance`** method on the assembly to make the instance. Note that this needs to be fed the full name of the class you are making, which you can get from the type you want to use. Note also that this is the very simple minded version of the **`CreateInstance`** method call, which pre-supposes that your class has a default constructor (i.e. one with no parameters). If this is not the case your program will throw a **`Method Not Found`** exception. There is a much more complex form of this call in which you can call a particular constructor and supply a set of parameters for it.

The code sample above just gets the **`MenuName`** property out of the **`EntityDrawing`** object that it creates. This will be used to create a menu from

which I can select this instance. If I was really obsessed with doing this properly I would have got the class from the type and then called a static method from it, i.e. I would have avoided the need for an empty constructor and I would have saved creating an instance when I don't really want one.

### How do I call a method in a type?

If you have a reference to a **Type**, the next thing you want to do is call a method in that **Type**. This method can be static, or it can be a member of an instance of the type. You call a method by getting the **MethodInfo** from the type and then calling that method:

```
System.Reflection.MethodInfo ThisMethod =
      ThisType.GetMethod("MakeEntity");
```

This gets the method info from a type reference called **ThisType**. The method I am interested in is a factory method called **MakeEntity**. After the above line the reference **ThisMethod** refers to method information for that method. I can now call the method by using **Invoke**:

```
Object Result = ThisMethod.Invoke(DrawingEntityInstance,
      new Object[] { this.Entities } );
```

The first parameter **DrawingEntityInstance** is a reference to the instance of the class whose method is to be called. If the method is **static** you can put a **null** here. I've not made the factory method **static** for reasons which are too complicated to go into here. The second parameter is an array of objects which constitute the parameters to the method call. In the case of my example code I'm passing a reference to the container which is going to contain the thing that I am making. Of course if you provide a list of objects which does not match the method signature you will get an exception at this point.

Finally I cast the object which was returned into what I really want:

```
EntityDrawing NewEntity = Result as EntityDrawing;
```

Note that I am using **as** to perform the conversion. This is better than casting, in that it doesn't through an exception if the cast fails. Instead it returns a **null** result which I can test for later (or I will get a different form of exception when I try to use it).

## Neat Stuff

Here is where I put things that I really like about .NET and Visual Studio.

### Good Dragging

You can drag items out of the class view into the program source. This is very useful if you have a bunch of constant items in one class you want to use in another. Just click on them in the Class View, hold down the left button and drag them into your source. Works a treat!

You can also drag source files in and out of directories and the system will just update where they are. This even works when you have a file open for editing.

## *Help Completion*

This is super. It makes looking for methods a breeze. The IDE will automatically look for operations which a class can perform and display them as a drop down menu. Simply type the name of a class or a reference and press the full stop and a list of what makes sense there will appear. I really like this. The only thing I have against it is that it is almost too clever.

If you type in the name of the class you get all the static members that can be quoted which is completely correct. Only if you type in the name of a variable of a particular type do you get all the members which are supported by that class.