C# for Delphi Developers

By Glenn Stephens
glenn@glennstephens.com.au
http://www.glennstephens.com.au

Introduction

C# has really gathered momentum in the last few years. As a powerful OO language it combines the best features of many programming languages. As the C# language was designed by Anders Hejlsberg, who was one of the original architects on C#, Delphi style themes can be seen in throughout C# language and the .NET framework.

This means many Delphi developers will find themselves comfortable with C# and the .NET framework after they learn the differences between the two languages. This paper/session will teach the Delphi VCL developer what it takes to start to move to C# and .NET.

This paper will not go into depth with the each class in the .NET Framework, but instead will cover some of the C# ways that you are used to doing in Delphi. Also this paper doesn't show every equivalent of Delphi's language, but it is intended to take you as far as you need to go in order to start building apps with C#.

So why should I bother with C#?

Good question. Not everyone will want to learn C#, but there are a few trends that are emerging. One reason is that many of the samples in .NET developer articles and journals are using C#, so you will need to understand C# a little to understand the techniques that are being used.

Another reason is that many of the component vendors are only writing their .NET components in C#, so if you want to extend the functionality of your components when you buy source you may need to get your hands dirty and code some C#.

There are plenty of reasons to use C# from understanding samples to just playing with it because it's new. Either way it's a good language that you will find useful.

Language differences/similarities

You may say ‘C# is just Java’. It’s kind of a true-ism. It sounds true but it’s not really. I have programmed Java for a while and when I first got my hands on C#, I just wrote simple Java code and it seemed to work.

In reality though, there are many differences. C# and Java are kind of like the siblings that don’t get along. Yeah the might look the same, but they have their own individualities. So those of you who know Java, you will probably find working with C# pretty easy.

But if you’re reading this paper, chances are you’re a Delphi Developer. So if you know a little C, C++ or Java you will find this pretty easy.

For Delphi developers, the language is different. You will need to know the ways to transpose the code from Delphi-ism into C#-isms. Once you are familiar with the ‘isms’, you should be able to do pretty much what you like. Much of this paper will show you the Delphi way of doing things and then show you the C# way (see Listing x), and will discuss any gotcha’s along the way.

One of the main differences that you will find with C# is that the language is case sensitive. This means MyVar, myvar, MYVAR and myVAR are all different variables. One of the things I have always liked about Delphi is that you don’t have to remember the case. With C# you tend to rely on the code completion features a little more so you don’t stuff up the variable names.

A Typical C# File

Just like Delphi has its own semantics on how you set up a file, C# has its own way. Let’s look at a simple C# file.

```
using System;

namespace Project4
{
	public enum FavouriteMovie
	{
		PrincessBride = 1,
		DudeWheresMyCar = 2,
		HarryPotter2
	}

	public class MyDemoClass
	{
		protected int SomeValue = 12;

		public MyDemoClass()
		{
		}

		public void SomeMethod()
		{
```

```
                        //Do something
                }

                public int AddSomeNumbers(int a, int b)
                {
                        return a + b;
                }
        }
}
```

Basically you have the assemblies that you will be **using** at the top. This is like the **uses** clause in Delphi. You will then have the **namespace** declaration, which defines what namespace you code belongs to. Within the namespace you then have your information that this source file might contain such as type declarations (see the FavouriteMovie enumeration) or class declarations such as the MyDemoClass. You can just follow the convention of having the code that does all the work between the namespace declarations.

The Language

From here we will deal with working with the C# language, covering things such as comments, variables, data types and flow control, and what you need to transfer your skills over to the C# world. So lets gets started with comments.

Comments

Comments are easy enough. If you have worked with Java or C++ then you know the comment style.

Pascal Style	C# Style
//	//
(* *)	/* */
{ }	/* */

One thing that C# has is documentation comments. These are declared before a class, method, property, enum or other code element.

```
/// <summary>
/// This is my main form that will make you go wow!
/// </summary>
public class WinForm : System.Windows.Forms.Form
```

By using these documentation tags before you declaration, you can generate documentation based on your source code file by running the C# command line compiler **csc.exe**.

```
csc MyFile.cs /doc:MyFile.xml
```

There are also other documentation tags that you can use, and this is just scratching the surface.

begin and end

Pascal Style	C# Style
begin	{
end	}

Data Types, variables and fields

Using data types is pretty easy. The Delphi/Pascal way is to declare the name of the variables on the left and then the data type on the right under a **var** section. E.g.

```
var
  MyInteger: integer;
  SomeString: string;
```

In C#, it's the other way around, so you declare the data type and then the variables and you can also assign values to them. There is no need for a var declaration. You can declare variables anywhere. E.g.

```
int MyInteger = 5;
string SomeSting;
```

The Delphi data types and their equivalents are displayed in Table x.

Delphi Data Type	C# Data Type
integer	int
boolean	Bool
double	double
string	string
TDateTime	DateTime
float	Float
Currency	Decimal
Int64	Int64

Note: Even the primitive data types have methods on them, so you can even perform functions like Int32.Parse() to convert a string to a number.

When you declare fields for use in a class in Delphi, you will put them in the interface section of your unit where your class is, and it will be defined under the private, protected or public sections. E.g.

```
protected
  MyProtectedDouble: double;
```

In C#, you can declare the fields by declaring a variable as you would in a function and then declaring the scope before it. If no scope is defines, then it assumes the scope will be private. E.g.

```
protected double MyProtectedDouble;
```

Expressions

Arithmetic operators

Delphi Way	C# Way
`+`	`+`
`-`	`-`
`*`	`*`
`/`	`/`
`Div`	`/ (when using int variables)`
`Mod`	`%`
`Inc`	`++`
`Dec`	`--`

Logical operators

Delphi Way	C# Way
`>`	`>`
`<`	`<`
`<=`	`<=`
`>=`	`>=`
`=`	`==`
`<>`	`!=`

Methods

One of things I find frustrating in Delphi is that once you declare a method in your class you then have to rewrite the definition in the interface section of the unit. This double writing of the method is more work for us. You're probably thinking 'Why don't you use Ctrl+Shift+C to implement it for you?' That's ok when you create the method, but when you change due to a refactor or any other reason it then you still have to update it in both places. Now because in C# all methods are declared inside the class definitions, then you only have to provide the information in the one spot.

So if you have a method in Delphi such as:

```
TForm1 = class(TForm)
…
```

```
private
  procedure MyPrivateMethod(x1: integer; s: string);
end;

…
procedure TForm1.MyPrivateMethod(x1: integer; s: string)
begin
   Caption := s + ‘[‘ + inttostr(x1) + ‘]’;
end;
```

As you can see, in C# it is a little simpler.

```
public class Form1 : Form
{
      private int MyPrivateMethod(int x1, string s)
      {
            Text = s + “[“ + x1 + “]”;
      }
}
```

Now let’s look how we pass parameters into methods in C#. In Delphi/Pascal you may use var, const or out to specify the type of parameters. The table below lists the C# equivalents

Delphi parameter type	C# parameter type
var	ref
out	out

virtual, abstract and dynamic

Without virtual or abstract methods, we would have no polymorphism. That’s not a nice world to program in. Of course you will want to create classes that allow the methods to be overridden. The way you declare an abstract or virtual method is by using the abstract or virtual keyword in the method declaration.

Delphi parameter type	C# parameter type
virtual	virtual
abstract	abstract
dynamic	(use virtual instead)

The code below demonstrates using an abstract method. The Delphi way would be:

```
type
  TAddSomeNumbers = class
  public
    Function Add(a, b: integer): integer; virtual; abstract;
  end;

  TConcreteAdded = class(TAddSomeNumbers)
  public
    Function Add(a, b: integer): integer; override;
  end;

…

function TConcreteAdded.Add(a, b: integer): integer;
begin
  Result := a + b;
end;
```

In C# land you would see something like the following

```
public abstract class AddSomeNumbers
{
      public abstract int Add(int a, int b);
}

public class ConcreteAdder : AddSomeNumbers
{
      public ConcreteAdder()
      {
      }

      public override int Add(int a, int b) {
            return a + b;
      }
}
```

If the above example is you wanted to use a virtual method in the C# example, you could just replace the **abstract** keyword with **virtual**. If you have a method similar that adds some content to a StringList you may have something like the following:

```
using System;
using System.Collections.Specialized;
using System.Windows.Forms;

namespace Project4
{
      public class AddToContent
      {
            public AddToContent() { }
            public virtual void AddContent(StringCollection data, string s)
            {
                  data.Add(s);
            }
      }

      public class ConcreteAdder : AddToContent
      {
            public ConcreteAdder() : base() { }
            public override void AddContent(StringCollection data, string s)
            {
                  base.AddContent(data, s);

                  MessageBox.Show("You added "+s);
            }
      }
}
```

You will see in the ConcreteAdder.AddContent that I am calling the base.AddContent method. This is similar to using the **inherited** keyword in an overridden method.

Test Cases

One thing you will notice in C# is that all the test cases have parentheses around them. So for an if statement, a switch statement

If then else statements

The Delphi Way	The C# Way

`if <SomeCondition> then`
`begin`
`  //When the condition is met`
`end else begin`
`  //the Else clause`
`end;`
	`if (<SomeCondition>)`
	`{`
	`  //when the condition is met`
	`} else {`
	`  //The else clause`
	`}`

Case statements

The Delphi Way	The C# Way

`case <SomeOrdinalExpression> of`
`  <OrdinalValueOrRange>:`
`    begin`
`      //Code to execute`
`    end;`
`else`
`  begin`
`    //No values have been met`
`  end;`
`end;`
	`switch (<SomeExpression>)`
	`{`
	`  case <Value>:`
	`    //Do Something`
	`    break; //Compulsory to break`
	`  case <Value2>:`
	`  case <Value3>:`
	`    //Stacking the cases`
	`    break;`
	`  default:`
	`    //The else clause`
	`    break;`
	`}`

While do

The Delphi Way	The C# Way

`While <BooleanCondition> do`
`begin`
`  //Code to execute`
`end;`
	`While (<BooleanCondition>)`
	`{`
	`  //Code to execute`
	`}`

Repeat until

The Delphi Way	The C# Way

`repeat`
`  //Code to execute`
`until <Condition>;`
	`do`
	`{`
	`  // Code to execute`
	`} while (<Condition>);`

For loops

For loops are a little different in C#, whereas Delphi has a simple structure such as:

```
for counter := 0 to 120 do
begin
  Sum := Sum + counter;
end;
```

In C# this is replaced by a different for loop, where there are three things you need to do. The equivalent of the Delphi code would be the following:

```
for (int counter=0; counter <= 120; counter++)
{
  sum += counter;
}
```

You can see the structure of the for loop where there are three statements separated by semicolons. What happens is the first statement defines the initial value, the second is the test condition that it checks for and the third is the statement to perform after every iteration. This is pretty flexible because it allows you to do a range of things. For example looping to a value for the odd numbers could easily be achieved by doing the following:

```
For (int counter=0; counter <= 120; counter+=2)
{
  sum += counter;
}
```

You can see that it can be pretty flexible.

foreach loops

My favorite construct of C# is the foreach loops. We all work with countless numbers of collections in our code every day. The foreach loop works with specific elements in a Collection.

For example in Delphi if you were looping through each Item in a TListView,you may do the following:

```
var
  counter: integer;
  MyItem: TListItem;
begin
  for counter := 0 to MyListView.Items.Count – 1 do
  begin
    MyItem := MyListView.Items[counter];
    //Perform something on the ListViewItem
  end;
end;
```

In C# you can use the foreach statement instead

```
Foreach (ListViewItem MyItem in MyListView.Items)
{
      //Perform whatever you want on MyItem
}
```

The C# code feels much cleaner when you implement the foreach construct, and you never have to worry about forgetting that –1 at the end of a Delphi for-loop when looping through a collection.

Break and Continue

No change. Woohoo.

The Delphi Way	The C# Way

`break;`
`continue;`
	`break;`
	`continue;`

Working with Files

One thing that you won't have access to in C# is the 'file of' data types such as TextFile. Instead the .NET framework provides many classes that are useful in helping you process files. The main class that you will use is System.IO.FileStream. This is not too different to working with the stream classes such as TFileStream. To start with let's look at an example where we read a line from a file passed to the application and then write it out to the console with line numbers. You will find that in .NET the combination of the FileStream and the StreamReader and StreamWriter classes should suit what you want to achieve.

Delphi way:

```
Var
  LineNumber: integer;
  MyFile: TextFile;
Begin
  AssignFile(MyFile, ParamStr(1));
  Reset(MyFile);
  try
    LineNumber := 1;
    While not Eof(MyFile) do
    Begin
      Readln(MyFile, s);
      Writeln(LineNumber, ‘: ‘, s);
      Inc(LineNumber);
    End;
  Finally
    CloseFile(MyFile);
  End;
End;
```

The C# way:

```
using System;
using System.IO;

namespace WriteStuffOut
{
	class DisplayFileWithLineNumbers
	{
		[STAThread]
		static void Main(string[] args)
		{
			if (args.Length == 0)
			{
				Console.WriteLine("You need to pass a filename “ +
					“to the application");
			} else {
				int LineNumber = 1;
				FileStream fs = new FileStream(args[0],
					FileMode.Open);
				try
				{
					StreamReader sr = new StreamReader(fs);
					try
					{
						string s;
						while ((s = sr.ReadLine()) != null)
						{
							Console.Write(LineNumber+
								": "+s+"\r\n");
							LineNumber++;
						}
					} finally {
						sr.Close();
					}
				} finally {
					fs.Close();
				}
				//Wait for something so the app sticks around
				Console.ReadLine();
			}
```

```
                }
        }
}
```

You can also use the File class's static methods to make it easy to read content. For example:

```
using (StreamReader r = File.OpenText("MyTextFile.txt"))
{
        MyTextEdit.Text = r.ReadToEnd();
}
```

The using statement

In the previous example I demonstrated using the **using** statement to create a StreamReader object. The using statement allows a block to execute and will ensure that a Dispose() call is made on the object. This is very important on certain types of objects such as files, Graphics and other objects that obtain a resource that needs to be released.

Arrays

The Delphi Way	The C# Way
`var`	`int[] I = new int[20];`
`  I: array[0..20] of integer;`	

Dynamic arrays are a little different, where you would have a dynamic array in Delphi, you should use the ArrayList class found in System.Collections. In fact you will find a lot of collection classes here that are useful.

Records

Records are useful and they contain mainly data, but in C# they are allowed to contain methods as well which work on the data contained with the record. This comes in really handy due to the fact that you no longer have to separate the data of the record from operations on the record. You will also notice that you don't have to use the static keyword for methods on C# structs.

The Delphi Way	The C# Way
`Type`	`public struct MyCube`
`  TMyCube = record`	`{`
`    Size: integer;`	`        public int Size;`
`  end;`	
	`        public int Volume()`
`function GetCubeVolume(`	`        {`
`  ACube: TMyCube): integer;`	`          return Size * Size * Size;`

`Begin`	`        }`
`  Result := ACube.Size * ACube.Size *`	`}`
`    ACube.Size;`	
`End;`	

Classes

In case you haven't noticed everything is a class in C#. Even the primitive data types have the same methods of System.Object.

Properties

Properties are easy enough to implement in C#. The Delphi way is to declare a property in your class definition.

```
property MyProperty: string read FMyProperty write SetMyProperty;

Procedure TMyClass.SetMyProperty(Value: string);
Begin
  FMyProperty := Value;
  //Perform some other action
End;
```

In C# you would do the following

```
private string myProperty;
public string MyProperty
{
	get
	{
		return myProperty;
	}
	set
	{
		myProperty = value;
		//Perform some other action
	}
}
```

Properties are great in this way as they are set in the one spot. In Delphi you may also have an indexed array such as the following:

```
Function GetIndexedValue(Index: integer): string;
Procedure SetIndexedValue(Index: integer; Value: string);
property MyIndexedProperty[Index: integer]: string read GetIndexedValue write
SetIndexedValue;
```

In the C# world you would use what is known as an indexer.

```
public string this[int index]
{
	get
	{
```

```
            return <something>[i];
      }
      set
      {
            <something>=value;
      }
}
```

You can also have abstract properties which is pretty handy.

```
public abstract int MyAbstractInt
{
      get;
      set;
}
```

Exception Handling

try..finally

The Delphi Way	The C# Way

`try`
`  //Do Stuff`
`finally`
`  //Do this no matter what happens`
`end;`
	`try`
	`{`
	`  //Do Stuff`
	`} finally {`
	`  //Do this not matter what happens`
	`}`

try..except

The Delphi Way	The C# Way

`try`
`  //Do Stuff`
`except`
`  //Failure code`
`end;`
	`try`
	`{`
	`  //Do Stuff`
	`} catch {`
	`  //Failure code`
	`}`

`try`
`  //Do Stuff`
`except`
`  on E: Exception do`
`  begin`
`    //Deal with e`
`  end;`
`end;`
	`try`
	`{`
	`  //Do Stuff`
	`} catch (Exception e) {`
	`  //deal with e`
	`}`

One of the good things about C# is that it also has the try..except..finally block so you can do things like the following:

```
      try
      {
         Console.WriteLine("Performing the try statement.");
         throw new NullReferenceException();
      }
      catch(NullReferenceException e)
      {
         Console.WriteLine("{0} Caught a null reference exception", e);
      }
      catch
      {
         Console.WriteLine("Caught an exception");
      }
      finally
      {
         Console.WriteLine("In the finally block");
      }
```

This is a cool feature that you can use to ensure that your apps handle the errors and do the necessary cleanup in one hit.

Events and Delegates

One of the common phrases you might hear in .NET is the term delegate. You may be wondering what his new catch-phrase means. Well like most .NET technologies, it's something that we have had in Delphi for quite a while. What is it? It's basically a procedural type. For example

```
type
  TMyProc = procedure(Sender: TObject; Value: string) of object;
```

would be declared in C# using

```
public delegate void MyProc(object Sender, string value);
```

Now just like in Delphi you have to declare these procedure types. As a result the delegates normally call a constructor passing the actual method.

```
public void CallItWithAString(object Sender, string value)
{
      MessageBox.Show("You wanted: " + value);
}

public void SetupAndDelegate()
{
      MyProc delegateInstance = new MyProc(CallItWithAString);
      delegateInstance(this, “Hello Delegate World”);
}
```

So you can see it's not that difficult to set up a delegate. Once you get your head around how the delegates are created, you will find it a piece of cake. This moves us onto events.

Events

I love design patterns and one of the most powerful design patterns is the Observer pattern. This is where you can have a whole bunch of objects waiting for one action to occur. One cool thing about C# is that it has this design pattern built into the architecture of the language. Delegates are great for holding a reference to one particular method to fire, but events go that one step further an allow you to fire off all method that are listening for a particular action to occur.

```
public event MyProc myEvent;

public void SetupEvents()
{
	myEvent += new MyProc(CallItWithAString);
	myEvent += new MyProc(CallItWithAStringAgain);
}

private void button1_Click(object sender, System.EventArgs e)
{
	//Fire off the event
	 if (myEvent != null)
		myEvent(this, "Hi There");
}
```

So events are really handy. It is like having the Event notification system in Win32 Delphi, but with the power of adding as many handlers as you like.

Enumerations and Sets

Enumerations can be declared in C#, and each enum can be assigned a value as shown below.

The Delphi Way	The C# Way

`type`
`  TMyEnum = (enVal1, enVal2, enVal3);`
 	
`var`
`  e: TMyEnum;`
`e := enVal1;`
	`public enum MyEnum`
	`{`
	`  Val1 = 1,`
	`  Val2,`
	`  Val3 = 100`
	`}`
	
	`MyEnum e = MyEnum.Val1;`

Sets are not really implemented in C#, but Microsoft have a set implementation class that you can use in your applications. For more information see http://msdn.microsoft.com/vcsharp/default.aspx?pull=/library/en-us/dv_vstechart/html/datastructures_guide6.asp

Type declarations

If you are familiar with the **is** and **as** directive you will find them exactly the same in C#, so that's a good thing. If you work with things such as the **class of** declaration, things are not so easy.

I prefer the way Delphi handles using classes with the **class of** directive, where you can hold a reference to a Class Type. C# has this ability as well, but it is not as clean. Essentially there is a class called **Type** which holds the information about the class. You can use the typeof function to get the type information of the class. For example:

```
Type t = typeof(MyClass);
bool isasubbie = t.IsSubclassOf(typeof(MyOtherClass));

private void button1_Click(object sender, System.EventArgs e)
{
  Type t = this.GetType();
  Form1 o = (Form1)t.InvokeMember("", BindingFlags.CreateInstance,
       null, null, null);
  o.ShowDialog();
}
```

Creating classes dynamically like you would in Delphi is more involved. You can either use the Type.Invoke to call the member of use the Activator class in order to dynamically create it. The Activator is beyond the scope of this paper, and you should consult the .NET framework help for more info on this complex but useful class.

VCL equivalents

Ok, so we've covered the language features of C#. But most of the power comes into play by using the .NET framework. You probably know the parts of the VCL that you work with back-to-front, so it may seem a little frustrating that you know need to learn what classes to use when you are trying to perform certain functions. The table listed below shows what classes to use when you are working with the .NET framework. While not exhaustive it will help you to find your way when you have something to do in .NET and you don't know which class to use

VCL Class	.NET Class	Where can I find it
TCanvas	Graphics	System.Drawing
TRegistry	Registry	Microsoft.Win32
TStringList	StringCollection	System.Collections.Specialized;
TDataModule	Component	System.ComponentModel;
TList	ArrayList	System.Collections
More VCL to FCL info coming soon –		

Supported Tools

IDE choice is important to developers, and because you are Delphi developers, I would recommend that you get started with C# by using C#Builder. There also other IDEs such as Visual Studio or #develop. But if your are a Delphi programmer (which of course you are ☺) then if you use C#Builder, you will feel right at home. The following table looks at where you can get the IDEs in question.

IDE	URL
C# Builder	Get the personal edition or trial from http://www.borland.com/products/downloads/download_csharpbuilder.html
Visual Studio	http://msdn.microsoft.com/visualstudio If you do have to use Visual Studio, you should download DPack from http://www.usysware.com/dpack which allows you to use Delphi Key Bindings. F9 for Run, F11 for object inspector etc.
#develop	http://www.icsharpcode.net/OpenSource/SD/
ASP.NET Web Matrix	Web Based IDE for C# http://www.asp.net/webmatrix/default.aspx

Conclusion

So you've learnt the basics. C# is a good language, and is worth learning. By using the techniques in this paper, you should be able to transfer your Delphi knowledge into knowledge of the C# fundamentals in a short amount of time. Also keep a look out on http://www.glennstephens.com.au/ as I will be updating this paper with any comments or additions.

Bio

Glenn Stephens designs and develops applications for various platforms, and has been programming for over 15 years. He a Borland Certified Consultant, a Microsoft Certified Solution Developer and is the author of the Tomes of Kylix – The Linux API. Lately Glenn has been writing articles for IBM's DB2 Portal on C# and a range of articles for Delphi Informant on Delphi 8 technologies such as ECO/MDA and the Borland Data Providers. Feel free to contact Glenn at glenn@glennstephens.com.au