A Taste of C#

Barry Cornelius
Computing Services, University of Oxford
Date: 24th February 2002; first created: 11th March 2001
http://users.ox.ac.uk/~barry/papers/
mailto:barry.cornelius@oucs.ox.ac.uk

1 Introduction

Microsoft have hit back: having fallen out with Sun over Java, they have now developed a rival product. Whereas the Java technology has produced a single language that is portable across many platforms, Microsoft's .NET Framework provides a number of languages that interoperate, initially only for most varieties of Microsoft Windows.

Microsoft's Visual Studio.NET product not only makes it easy to produce code for standalone programs (such as console applications and windows forms applications) but also makes it easy to produce code that can be executed by their web server software (IIS). The latter possibility not only allows the creation of dynamically generated WWW pages, but also the ability to offer *web services*, i.e., to provide methods that can be called by external clients.

This document:

- considers the key aspects of the .NET Framework;
- provides an introduction to C#;
- gives some examples of some standalone programs written in C#.

The document assumes the reader has some knowledge of the Java programming language.

The use of C# to support web forms and web services are dealt with in another document ([2]).

2 The .NET Framework

2.1 Overview of the .NET Framework

Put simply, the .NET Framework consists of three aspects:

1. the Common Language Runtime (CLR);
2. a comprehensive set of class libraries;
3. class libraries associated with particular kinds of applications:
   (a) console applications;
   (b) windows forms applications;
   (c) web form applications;
   (d) web services.

2.2 The Common Language Runtime

In the past, compiler writers have put code to support the execution of programs into a runtime system. Instead of providing a different runtime system for each programming language, the .NET Framework provides a runtime system that is used by all of the languages that are targetted at the .NET Framework. This is called the *Common Language Runtime* (or *CLR*). Code that targets the CLR is called *managed code*.

Microsoft are providing several .NET compilers: Managed C++, Visual Basic.NET, JScript and C#. In addition, other people/companies are producing .NET compilers for other languages including COBOL, Eiffel, Haskell, ML, Perl, Python, Scheme and Smalltalk.

A .NET compiler writer can rely on the CLR for a large number of tasks, including:

- creating new types;
- creating and initializing of objects;
- tracking references to objects and providing garbage collection;
- handling the calling of methods (including virtual methods);
- managing the access to array elements;
- providing support for exceptions and exception handling.

All of the .NET languages have compilers that generate code written in an intermediate language called *MSIL* (or *IL*). A file containing MSIL instructions can be run on any platform so long as the operating system for that platform hosts the CLR engine. Currently, a CLR engine is available for Windows XP, Windows 2000, Windows NT 4.0, Windows 98 and Windows Me.

2.3 The Common Type System

Besides providing the functionality normally expected from a runtime system, the CLR also defines a *Common Type System* (*CTS*) which must be supported by all .NET languages. The CTS says that each language must provide value types (primitive types, struct types, enumerations) and reference types (class types, interface types, array types, delegate types).

Thai and Lam ([9], p39) say: 'The CLR provides full support for object-oriented concepts (such as encapsulation, inheritance, and polymorphism) and class features (such as methods, fields, static members, visibility, accessibility, nested types, and so forth). In addition, the CLR supports new features that are nonexistent in many traditional object-oriented programming languages, including properties, indexers and events.'

For efficiency reasons, the CTS has value types as well as reference types. So an `int` or a value of some struct type will be stored on the stack or inline, whereas an instance of some class type will be stored on the heap (and pointed to by some variable). However, any value (that is of some value type) can automatically be wrapped into an object by a process known as *boxing*. So a value (of some value type) can be used in contexts where an object is expected:

```
tArrayList.Add(27);
```

2.4 The primitive types

As well as providing a type system that is common to all .NET languages, the CLR also provides a set of primitives types that is common to all .NET languages. The .NET primitive types include:

```
               size   C#        VB.NET
System.Boolean    8   bool      Boolean
System.Byte       8   byte      Byte
System.Int16     16   short     Short
System.Int32     32   int       Integer
System.Int64     64   long      Long
System.Float     32   float     Single
System.Double    64   double    Double
System.Char      16   char      Char
System.Decimal  128   decimal   Decimal
```

2.5 Language interoperability

Given that the compilers for each .NET language generate the same intermediate language, use the same runtime, build the same kind of types and use the same primitive types, it is possible to build programs where the code is written in different .NET languages.

For example, a Visual Basic.NET programmer can create a class that derives from a C# class and overrides some of its virtual methods; or a C# programmer can handle an exception thrown by a method being applied to an object of an Eiffel class; and so on.

2.6 Tool support

Because there is a CLR, debuggers can support programs where the code has been written in different .NET languages, and IDEs (such as Visual Studio.NET) can use the CLR to provide information to the programmer. For example, if you have declared some variable:

```
ArrayList tArrayList;
```

then, when you start to type the code to apply a method to `tArrayList`:

```
tArrayList.
```

as soon as you type the dot, the IDE can provide you with a pop-up window displaying a list of methods that can be applied to `tArrayList`.

2.7 Deployment

One of the problems with Windows applications is that they can be difficult to install. Besides providing the files, the application may want to change the registry or provide shortcuts. It is also difficult and sometimes impossible to uninstall applications. With .NET, all the code and any information needed to run an application are provided in a collection of files. In order to install an application, you just need to create a directory containing these files, and removing this directory uninstalls the application.

One of the main problems with installing a Windows application is that the installation may overwrite a DLL used by some other application, and overwriting it causes the other application no longer to work. This is because the two applications require different versions of the DLL file. Because in .NET, DLLs can be signed with a public key and a version number, it is possible for the cache of DLLs to have two DLLs with the same name.

3 The `SumProg` program

Here is a C# program to read in two integers and output their sum:

```
namespace first
{
    using System;
    using System.Threading;
    public class SumProg
    {
        public static void Main()
        {
            Console.Write("Type in the first number: ");
            string tFirstString = Console.ReadLine();
            int tFirst = int.Parse(tFirstString);
            Console.Write("Type in the second number: ");
            string tSecondString = Console.ReadLine();
            int tSecond = int.Parse(tSecondString);
            int tSum = tFirst + tSecond;
            Console.WriteLine("The sum is " + tSum);
            Thread.Sleep(3000);     // wait for 3 seconds
        }
    }
}
```

4 Namespaces, `using` and `Main`

4.1 Namespace declarations and using directives

The first line:

```
namespace first
```

is the header of a *namespace declaration*. This header is similar to Java's package declaration. Essentially the class `SumProg` belongs to a namespace called `first`.

The *using directive*:

```
using System;
```

means that the classes of the namespace `System` can be used in the subsequent code without qualification. So we can refer to the `Write` method of the `System.Console` class by `Console.Write`. The method `Sleep` belongs to the class `Thread` of the namespace `System.Threading`. Hence we can use:

```
using System.Threading;
...
Thread.Sleep(3000);
```

4.2 An alternative approach

There are two kinds of using directives: *using-namespace directives* and *using-alias directives*. A using-namespace directive like:

```
using System;
```

means that any of the classes of the `System` namespace can be used in the subsequent code without qualification. It is similar to Java's:

```
import java.util.*;
```

If you have a lot of these kind of using directives, then, when you look at the code that follows, it is difficult to detect from which namespace a class belongs.

An alternative approach is to use a using-alias directive to give an alias for a class:

```
using Console = System.Console;
using Thread  = System.Threading.Thread;
...
Console.WriteLine("Hello world");
Thread.Sleep(3000);
```

Using this approach, you would need one using directive for each class that is used in the code.

4.3 The `Main` method

The `Main` method may have one of the following signatures:

```
public static void Main()
public static int  Main()
public static void Main(string[] pArgs)
public static int  Main(string[] pArgs)
```

If its return type is `int`, a return statement should be used to terminate the program's execution, e.g.:

```
return 0;
```

5 Types

5.1 Introduction

In C#, there are three kinds of types: *value types*, *reference types* and *pointer types*. As a pointer type can only be used in code marked as `unsafe`, we will ignore pointer types.

As in Java, a reference type is a class type, an interface type or an array type. C# has one other kind of reference type: the *delegate type* (which will be considered later).

A value type is either a *struct type* or an *enumeration type*.

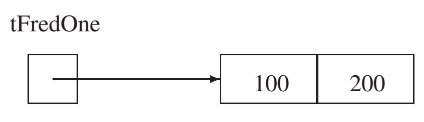
5.2 Struct types

As well as class types, C# also has struct types. Struct types are declared and used like class types.

If `Fred` is a class type, then, in both Java and C#, an instance of `Fred` can be created by:

```
Fred tFredOne = new Fred(100, 200);
```

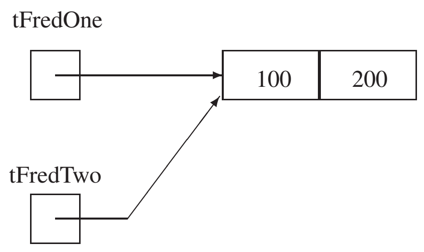
This produces:



If we now do:

```
Fred tFredTwo = tFredOne;
```

we get:



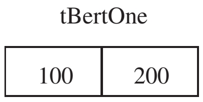
A declaration of a struct type looks similar to that of a class type:

```
public struct Bert
{
    private int iX;
    private int iY;
    public Bert(int pX, int pY)
    {
        iX = pX;
        iY = pY;
    }
    ...
}
```

However, the declaration:

```
Bert tBertOne = new Bert(100, 200);
```

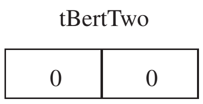
produces:



and:

```
Bert tBertTwo = new Bert(0, 0);
```

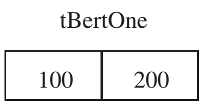
produces:

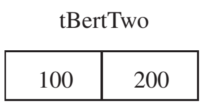


If we now do:

```
tBertTwo = tBertOne;
```

we get:





Like any other local variable (e.g., a local variable of type `int`), a local variable that is of a struct type comes into being when the block starts and ceases to exist when the block is exited. This is quite different from an object (that is of some class, and is pointed to by some reference variable): the object is created by the use of `new` and ceases to exist when the garbage collector says so. In implementation terms, a local variable that is of a struct type is allocated on the stack whereas an object is allocated on the heap.

Other points:

1. It is faster to access a field of a struct than the field of a class.
2. Passing a struct (as a value parameter) to a method will be slower than passing a value of a class type.

3. Each struct type is derived from the type object.
4. A struct type can implement one or more interfaces.
5. Like class types, a struct type can declare fields, properties, constructors, indexers, methods and operators.
6. A struct type is *sealed* meaning that you cannot derive a new type from a struct type.

5.3 The simple types

C# provides a set of predefined types called the *simple types*. Some examples are: `byte`, `short`, `int`, `long`, `char`, `float`, `double` and `bool`. So C# has all the primitive types of Java. In addition, C# has unsigned/signed variants of the integer types: they are `sbyte`, `ushort`, `uint` and `ulong`. C# also has a simple type called `decimal` which can be used for the exact representation of monetary values.

The names of these simple types are keywords (*reserved words*). All of them are simply aliases for some predefined struct types. It is as if we had written:

```
using char   = System.Char;
using double = System.Double;
using int    = System.Int32;
```

Because a simple type (e.g., `int`) is just an alias for a struct type (e.g., `System.Int32`), every simple type has the members that the struct type has. For example:

```
int tLargest = int.MaxValue;
```

is the same as:

```
int tLargest = System.Int32.MaxValue;
```

And the two calls of `ToString` in:

```
int    tSomeInt      = 123;
string tFirstString  = tSomeInt.ToString();
string tSecondString = 123.ToString();
```

both access the `ToString` method of the class `System.Int32`.

5.4 Enumeration types

An *enum declaration* introduces a new type. It is declared in terms of an underlying type which if it is omitted defaults to `int`:

```
enum Days: int
{
    Sunday = 1,
    Monday, Tuesday, Wednesday, Thursday, Friday,
    Saturday
}
```

Each enum member has an associated value which is either given explicitly or it is one more than the associated value of the previous member.

Each enum declaration introduces a new type. An explicit conversion has to be done to convert between an enum type and an integer type:

```
Days tDays = (Days)3;
int tSomeInt = (int)tDays;
```

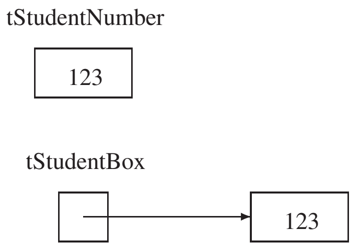
5.5 Boxing and unboxing

As mentioned previously, the value types are struct types and enumeration types, where the struct types include the predefined simple types. When appropriate, an object is automatically created from a variable that is of some value type. This is known as *boxing*.

For example:

```
int tStudentNumber = 123;
object tStudentBox = tStudentNumber;
```

results in:



Boxing is a two-stage process: create an object and then copy the value of the variable into the object.

Unboxing is achieved as follows:

```
int tStudentNumberAgain = (int) tStudentBox;
```

An `InvalidCastException` exception is thrown if you attempt to cast to an inappropriate type.

Automatic boxing and unboxing are not present in Java. Instead, in Java, you have to do these tasks explicitly using the wrapper types, e.g.:

```
int tStudentNumber = 123;
Object tStudentBox = new Integer(tStudentNumber);
int tStudentNumberAgain = ((Integer)tStudentBox).intValue();
```

Microsoft's marketing refers to C#'s *unified type system*. The phrase *Everything is an object* is used: this means that, because boxing and unboxing are automatically performed, everything can be treated as an object even if it is a value of a value type.

Here is a more convincing example. Like Java, C# has a collection class called `ArrayList`. This class has a method called `Add` which is declared as:

```
public int Add(object pObject);
```

So, like Java, we can do:

```
ArrayList tArrayList = new Arraylist();
tArrayList.Add("Hello world");
Date tDate = new Date(2001, 3, 13);
tArrayList.Add(tDate);
```

However, in C#, we can also do:

```
int tStudentNumber = 123;
tArrayList.Add(tStudentNumber);
```

In Java, you would have to write:

```
int tStudentNumber = 123;
tArrayList.add(new Integer(tStudentNumber));
```

6 Methods

6.1 value, ref and out parameters

In Java, all parameters are *value parameters*. A value parameter acts like a local variable of the method whose initial value is the value of the argument used in the call.

C# also has *ref parameters* and *out parameters*. If a method has a ref/out parameter, then that parameter represents the same variable as the variable given as the argument.

Here is an example of a ref parameter:

```
private static void iIncrease(ref int pCount)
{
    pCount++;
}
public static void Main()
{
    int tCount = 1;
    iIncrease(ref tCount);
    Console.WriteLine(tCount);
    // 2 would be output
}
```

Note that the keyword `ref` appears both in the parameter list and in the argument list.

An out parameter is used in a similar way to a ref parameter. The differences between a ref parameter and an out parameter are:

- the argument for a ref parameter must have a value when the method is called;
- the argument for an out parameter must have a value when the method is exited.

6.2 params parameters

A *params parameter* may appear as the last parameter of a parameter list: it permits a variable number of arguments. Here is an example:

```
private static int iHowMany(int pTestValue, params int[] pValues)
{
    int tResult = 0;
    for (int tIndex = 0; tIndex<pValues.Length; tIndex++)
    {
        if (pValues[tIndex]==pTestValue)
        {
            tResult++;
        }
    }
    return tResult;
}
...
int tNumberFound = iHowMany(3, 1, 3, 3, 2);
// tNumberFound now has the value 2
```

6.3 No throws clause

Unlike Java, it is not possible in C# to document that a method may throw an exception. So a method's header does not have a *throws clause*.

7 Statements

The statements of C# are much like those of Java. Here we consider three differences.

7.1 foreach statements

With a *foreach statement*, you can visit each element of a collection. For example, the above for statement:

```
for (int tIndex = 0; tIndex<pValues.Length; tIndex++)
{
    if (pValues[tIndex]==pTestValue)
    ...
```

can instead be written as:

```
foreach (int tValueFound in pValues)
{
    if (tValueFound==pTestValue)
    ...
```

A foreach statement can be used for any type that implements the `IEnumerable` interface:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

In Java, the equivalent method to `GetEnumerator` is called `iterator`. The `IEnumerator` interface is:

```
public interface IEnumerator
{
    bool MoveNext();
    object Current{ get; }
    void Reset();
}
```

In Java, the equivalent interface to `IEnumerator` is called `Iterator`.

The following types implement the `IEnumerable` interface:

- any array;
- any of the collection classes (that are in the `System.Collections` namespace).

You can also produce your own types that implement this interface.

7.2 switch statement

C#'s *switch statement* has two 'improvements' over that of Java's:

1. It is not possible to fall through from one arm to the next. So:

   ```
   case 0:
       i = 1;
   case 1:
       j = 2;
   ```

   is illegal. An arm must end in a statement that transfers control. Four examples of such statements are:

   ```
   break;
   goto case 1;
   goto default;
   return;
   ```

2. A string may be used as a *selector*:

   ```
   switch ( tCommand )
   {
       case "add":
           ...
           break;
       case "remove":
           ...
           break;
   }
   ```

7.3 try statements

C#'s *try statement* is similar to that of Java. However, a catch clause can omit the name of its parameter as in:

```
catch(IndexOutOfRangeException)
{
    ...
}
```

If you wish to catch any exception, the type of the parameter and the parentheses may also be omitted:

```
catch
{
    ...
}
```

8 Interfaces, classes and structs

8.1 A class declaration for dates

Chapter 11 of the 'C# Language Specification' ([6]) says that 'Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs.' So, if we want to represent dates in a program, it would probably be desirable to provide a struct type. However, we will be Java-luddites and use a class type.

So here is a class declaration for dates:

```
namespace utility
{
    using IComparable = System.IComparable;
    public class Date: IComparable
    {
        private int iYear;
        private int iMonth;
        private int iDay;
        public Date(int pYear, int pMonth, int pDay)
        {
            iYear  = pYear;  iMonth = pMonth;  iDay = pDay;
        }
        public int Year
        {
            get { return iYear;   }
            set { iYear = value;  }
        }
        public int Month
        {
            get { return iMonth;  }
            set { iMonth = value; }
        }
        public int Day
        {
```

```
            get { return iDay;    }
            set { iDay = value;   }
        }
        ...
        public override string ToString()
        {
            return iYear + "-" + iMonth/10 + iMonth%10
                         + "-" +   iDay/10 +   iDay%10;
        }
    }
}
```

The first line of the class declaration is:

```
public class Date: IComparable
```

If a colon is present, a comma-separated list of names should be given following the colon. If you wish to derive a class from a particular *base class*, the first name should be the name of this base class; otherwise the class will be derived from the class `object`. The remaining names should be the names of interfaces that are implemented by this class. So the above line says that the `Date` class is derived from the class `object` and that it implements the `IComparable` interface.

The `Date` class declaration provides three private fields, a constructor, three properties and a `ToString` method. These can be used as follows:

```
Date tDate = new Date(2001, 3, 13);
int  tYear = tDate.Year;
tDate.Year = 2002;
Console.WriteLine(tDate);
```

The first occurrence of `tDate.Year` will use the *get accessor* for `Year` and the assignment to `tDate.Year` uses its *set accessor*.

There are three kinds of properties:

- *read-write* has both get and set accessors;
- *read-only* only has a get accessor;
- *write-only* only has a set accessor.

Curiously, `get`, `set` and `value` are not keywords: they are tokens that have a special meaning in the context of a property.

The default access for a member of a class is private and so the three occurrences of the `private` keyword in the above class declaration can be removed.

The `override` keyword has to be provided in the declaration of `ToString` because `ToString` is a *virtual method* of the class `object` (which is `Date`'s base class). There is more information about this topic later.

8.2 Providing an `Equals` method

8.2.1 Just like Java

By default, both:

```
tDate.Equals(tToday)
```

and:

```
tDate==tToday
```

will provide reference semantics: they ask whether `tDate` and `tToday` point to the same object.

The default version of the `Equals` method is provided by a (virtual) method declared in the class `object`. In the class `Date`, we can override `object`'s `Equals` by declaring:

```
public override bool Equals(object pObject)
{
    if ( pObject==null || GetType()!=pObject.GetType() )
    {
        return false:
    }
    Date tDate = (Date)pObject;
    return iYear ==tDate.iYear  &&
           iMonth==tDate.iMonth &&
           iDay  ==tDate.iDay;
}
```

This version of `Equals` provides value semantics: so:

```
tDate.Equals(tToday)
```

asks whether `tDate` and `tToday` point to two objects that have the same value.

So far, all of this is the same as what you would do in Java.

8.2.2 Overloading the == operator

However, in C#, you can also overload many of the operators. For example, `Date` could provide declarations for the == and `!=` operators. So instead of the above declaration of `Equals`, we could provide:

```
public static bool operator ==(Date pLeft, Date pRight)
{
    return pLeft.iYear ==pRight.iYear  &&
           pLeft.iMonth==pRight.iMonth &&
           pLeft.iDay  ==pRight.iDay;
}
public static bool operator !=(Date pLeft, Date pRight)
{
    return ! (pLeft==pRight);
}
public override bool Equals(object pObject)
{
    if ( pObject==null || GetType()!=pObject.GetType() )
    {
        return false:
    }
    Date tDate = (Date)pObject;
    return this==tDate;
}
```

Note: if you do provide a declaration for ==, you must also provide one for `!=`. Here are some statements showing a use of each of the above three declarations:

```
bool tTest1 = tFirstDate==tSecondDate;
bool tTest2 = tFirstDate!=tSecondDate;
bool tTest3 = tFirstDate.Equals(tSecondDate);
```

Now `Equals` and == are both providing value semantics.

As a use of == looks nicer than a call of `Equals`, you may be tempted not to bother with declaring `Equals`. However, this may be unwise. Albahari *et al* ([1], p46) say that declaring `Equals` 'provides compatibility with other .NET languages that don't overload operators'. What this means is as follows: when using such a language, you will always be able to call the `Equals` method of your C# component even if your language does not allow you to use the == operator of your component.

8.2.3 == and `Equals` for value types

When the == operator is used with two values that are of some value type, the == operator will deliver true if the two values are bitwise equal. For most value types, this means that == has value semantics.

All value types are derived from `System.ValueType`. This type defines `Equals` to have the same meaning as the == operator.

8.2.4 Problems

There are a number of problems:

1. The definition of == is left to the designer of a type: it could be providing value semantics or it might not be defined in which case you would get reference semantics for class types and bitwise equality for struct types. So you need to search for the class/struct declaration to find out the answer. Similarly for the `Equals` method. In Java, there is not so much uncertainty: when used with two reference variables, the == operator always provides reference semantics and the recommendation of the designers of Java is that a class should provide an `equals` method which implements value semantics.

2. The other problem is that the overloading of == does not apply to interface variables. If you want to use an interface declaration to form the contract for a new type, surely it would be desirable to allow operator declarations in an interface.

8.3 Providing `GetHashCode`

`GetHashCode` is used in much the same way as `hashCode` in Java. Visual Studio.NET gives a warning if a class overrides `Equals` but does not override `GetHashCode`.

The `Date` class could provide:

```
public override int GetHashCode()
{
    return 0;
}
```

In my opinion, it would have been better if a call of `object`'s `Equals` method always produces an exception. This would ensure that you would soon detect if your code calls `object`'s `Equals`: this is rarely what you want to do as it does not provide you with an appropriate result. I also believe that it would have better if a call of `object`'s `GetHashCode` method returns the value 0. If a class fails to provide `GetHashCode`, then `object`'s `GetHashCode` will be used instead: returning 0 would at least do something sensible. Similar comments apply to Java.

8.4 Providing `CompareTo`

As `Date` says it implements the `IComparable` interface, it needs to provide a `CompareTo` method:

```
public int CompareTo(object pObject)
{
    Date tDate = (Date)pObject;
    int tResult = iYear - tDate.iYear;
    if (tResult==0)
    {
        tResult = iMonth - tDate.iMonth;
        if (tResult==0)
        {
            tResult = iDay - tDate.iDay;
        }
    }
    return tResult;
}
```

Gunnerson (3, p249) says 'If a class has an ordering that is expressed in `IComparable`, it may also make sense to overload the other relational operators. As with == and `!=`, other operators must be declared as pairs, with < and > being one pair, and >= and <= being the other pair.'

9 Inheritance

As far as inheritance is concerned, there are two main differences between Java and C#.

9.1 Syntactical changes

Some of the syntax of inheritance is closer to C++ than Java:

1. a colon is used instead of `extends`;
2. special syntax is employed to invoke a constructor from a constructor;
3. `base` is used instead of `super`.

These points are illustrated by the following classes:

```
public class Figure
{
    private int iX;
    private int iY;
    public Figure()
        : this(0, 0)                              // 2
    {
    }
    public Figure(int pX, int pY)
    {
        iX = pX;  iY = pY;
    }
    ...
    public override string ToString()
    {
        return iX + ":" + iY;
    }
}
public class Circle: Figure                       // 1
{
    private int iRadius;
    public Circle()
        : this(0, 0, 0)                           // 2
    {
    }
    public Circle(int pX, int pY, int pRadius)
        : base(pX, pY)                            // 2, 3
    {
```

```
        iRadius = pRadius;
    }
    ...
    public override string ToString()
    {
        return base.ToString() + ":" + iRadius;   // 3
    }
}
```

9.2 Overriding virtual methods

Suppose we have:

```
Figure tF = new Figure(100, 200);
tF.Fred();
```

In both Java and C#, it is the `Fred` method of the `Figure` class that will get called.

However, what happens if we have:

```
Figure tC = new Circle(100, 200, 50);
tC.Fred();
```

Assuming that `Figure` and `Circle` both declare a method called `Fred`, whose `Fred` method gets executed: is it `Figure`'s `Fred` or `Circle`'s `Fred`?

In Java, it is `Circle`'s `Fred` that gets executed because `tC` is pointing to a `Circle`. This is known as *dynamic binding*.

However in C# (and also in C++), what happens depends on whether the declaration of `Fred` in the base class (`Figure`) includes the keyword `virtual`, i.e., does it use:

```
public void Fred() ...
```

or:

```
public virtual void Fred() ...
```

If `virtual` is absent, i.e., `Fred` is a non-virtual method, the declaration of `Fred` in `Circle` must include the keyword `new`, i.e.:

```
public new void Fred() ...
```

and the call `tC.Fred()` calls `Figure`'s `Fred`.

If `virtual` is present, i.e., `Fred` is a virtual method, then:

1. *either* the declaration of `Circle`'s `Fred` includes the keyword `new`, i.e.:

   ```
   public new void Fred() ...
   ```

   and `tC.Fred()` calls `Figure`'s `Fred`;

2. *or* the declaration of `Circle`'s `Fred` includes the keyword `override`, i.e.:

   ```
   public override void Fred() ...
   ```

   and `tC.Fred()` calls `Circle`'s `Fred`.

Summary:

```
call of           virtual in       non-virtual in          absent
tC.Fred           base class         base class        in base class

absent in           warning            warning              error
derived class

new in              Figure             Figure              warning
derived class

override in         Circle              error               error
derived class
```

10 Delegates

In its simplest form, a *delegate* is a type representing the signature of a method. For example:

```
delegate int Massage(string s);
```

declares a new type called `Massage` that is the type of methods that take a `string` and return an `int`. Perhaps:

```
delegate int:(string s) Massage;
```

would have been better syntax.

If we now write:

```
private static void iProcess(Massage pMassage)
{
    string tString = Console.ReadLine();
    int tInt = pMassage(tString);
    Console.WriteLine(tInt);
}
```

then what a call of `iProcess` such as:

```
iProcess(tMassage);
```

will do depends on the value of the delegate variable `tMassage`. We could have:

```
Massage tMassage = new Massage(StringLength);
iProcess(tMassage);
```

where `StringLength` is declared as:

```
private static int StringLength(string pString)
{
    return pString.Length;
}
```

Here `tMassage` is made to point to the `StringLength` method.

A method like `iProcess` is sometimes called a *higher order method* as it is written in terms of a pointer to a method (which is passed as a parameter).

If a delegate's return type is void, a delegate variable can be assigned a value that represents (not just one method but) a list of methods to be called:

```
delegate void Display(string s);
private static void iProcess(Display pDisplay)
{
    string tString = Console.ReadLine();
    pDisplay(tString);
}
private static void All(string pString)
{
    Console.WriteLine(pString);
}
private static void FirstTwo(string pString)
{
    Console.WriteLine(pString.Substring(0, 2));
}
public static void Main()
{
    Display tDisplay = new Display(All) + new Display(FirstTwo);
    iProcess(tDisplay);
}
```

Here `tDisplay` is assigned a list of methods, and so when it gets called:

```
pDisplay(tString)
```

each method of the list will get executed in turn. This use of delegates is called a *multicast*.

Although delegates are not part of Java, they appeared in Microsoft's Visual J++. Sun Microsystems' criticisms of delegates are given at [8].

11 Events

11.1 Using delegates for event handling

One of the main uses of delegates is for handling events.

Suppose we have some object of some class `X`, and we want to offer the ability to register methods that will be called when the object changes. Suppose that each of these methods has a header like:

```
void MethodName()
```

In C#, we can introduce a delegate type to describe this:

```
delegate void Handler();
```

In the class `X` we can declare a public field to be of this type. Here is an example:

```
public class X
{
    private int iValue = 0;
    public Handler uHandler = null;
    public void inc()
    {
        iValue++;
        uHandler();
    }
}
```

In a client class, we can then do:

```
X tX = new X();
tX.uHandler += new Handler(Fred);
tX.uHandler += new Handler(Bert);
```

where `Fred` and `Bert` are appropriate methods.

If we do this, then whenever there is a call of `inc`, the methods `Fred` and `Bert` will also be called.

Although this will work, the `uHandler` field of `X` is very vulnerable. Any client can do anything it wants to it, e.g.:

```
tX.uHandler = null;
```

To prevent this, `uHandler` should be declared with the keyword `event` as in:

```
public event Handler uHandler = null;
```

If you do this, clients can only execute the `+=` or `-=` operators on `uHandler`.

11.2 Passing information from the object to each method

Although the `Handler` type has been declared without parameters, it is useful to pass information from the source of the event to each method. It is conventional for the delegate type to have two parameters: one being a pointer to the object causing the event, and the other containing information about the event.

So usually the delegate type is like:

```
delegate void Handler(object pSource, XEventArgs pXEventArgs);
```

where `XEventArgs` is a class derived from `System.EventArgs`.

11.3 Common uses of events

One common use of events is for registering methods to be executed when an event such as a click of a button in a GUI occurs.

Suppose we have:

```
using Button = System.Windows.Forms.Button;
```

A class can then declare a button using:

```
private Button iAddButton;
```

The constructor for the class can execute statements like:

```
iAddButton.BackColor = Color.Wheat;
iAddButton.Font      = new Font("Times New Roman", 8);
iAddButton.Text      = "Click Here";
```

where `Color` and `Font` are classes from the `System.Drawing` namespace.

The `Button` class also has an event property called `Click`. So if we want a method called `iAddClick` to be executed whenever there is a click on this button, we should also get the constructor to do:

```
iAddButton.Click += new EventHandler(iAddClick);
```

where `EventHandler` is a class of the `System` namespace.

12 Other points

12.1 Collections

C# has a number of `Collection` classes including:

```
interface      class
IList          ArrayList
IList          StringCollection
IDictionary    Hashtable
IDictionary    SortedList
```

Here are some comparisons with classes from Java's Collections API:

```
C#             Java
ArrayList      ArrayList
N/A            LinkedList
N/A            HashSet
N/A            TreeSet
Hashtable      HashMap
SortedList     TreeMap
```

12.2 Documentation comments

In Java, an ad-hoc format is used for comments that can be used to produce documentation. C#'s *documentation comments* uses an XML notation:

```
/// <summary>
/// The inc method is used to increase the balance.
/// </summary>
/// <param name="amount">
///    The amount by which the balance is increased.
/// </param>
public void inc(int amount)
...
```

12.3 Templates

Unlike C++, C# does not have *templates*. Templates (also known as *generics* or *parameterized types*) are likely to appear in a future revision of C# (and Java).

12.4 Const, final and virtual

Keywords like `const` and `virtual` have many different meanings in C++. Similarly, in Java the keyword `final` has many uses. Some of these are not possible in C#:

```
                         JDK 1.0.2   JDK 1.1+        C#
constant field             final       final        const
constant local variable    N/A         final         N/A
constant parameter         N/A         final         N/A
non-overridable method     final       final      [default]
    overridable method   [default]   [default]     virtual
non-overridable class      final       final       sealed
```

13 Four kinds of .NET applications

As mentioned earlier, the .NET Framework (and Microsoft's Visual Studio.NET) makes it easy to produce console applications, windows forms applications, web form applications and web services. Examples of standalone programs that are console applications or windows forms applications will now be given. Although these examples are coded using C#, each of these examples could just as easily have been coded in any other .NET language, such as Visual Basic.NET ([4]).

Examples of C# being used to support web forms and web services are dealt with in another document ([2]).

13.1 Console applications

Here is a program that reads in a temperature given in degrees Centigrade and outputs the corresponding value in degrees Fahrenheit:

```
namespace ConsoleConvert
{
    using System;
    using System.Threading;
    public class MyClass
    {
        public static void Main()
        {
            Console.Write("Centigrade value: ");
            string tCentigradeString = Console.ReadLine();
            double tCentigrade = double.Parse(tCentigradeString);
            double tFahrenheit = 32 + tCentigrade*9/5;
            Console.WriteLine("Fahrenheit value: " + tFahrenheit);
            Thread.Sleep(10000);
        }
    }
}
```

13.2 Windows Forms applications

The .NET Framework includes a number of classes that can be used to provide an application driven by a GUI. Some of these classes are used in the following program:

```
namespace WindowsConvert
{
    using System;
    using System.Drawing;
    using System.Windows.Forms;
    public class MyForm : Form
    {
        private TextBox tTextBox;
        private Button tButton;
        private Label tLabel;
        public MyForm()
        {
            tTextBox = new System.Windows.Forms.TextBox();
            tTextBox.Location = new System.Drawing.Point(64, 32);
            tTextBox.Size = new System.Drawing.Size(120, 20);
            Controls.Add(tTextBox);
            tButton = new System.Windows.Forms.Button();
            tButton.Location = new System.Drawing.Point(64, 64);
            tButton.Size = new System.Drawing.Size(120, 20);
            tButton.Text = "Get Fahrenheit";
            tButton.Click += new EventHandler(iHandleClick);
            Controls.Add(tButton);
            tLabel = new System.Windows.Forms.Label();
            tLabel.Location = new System.Drawing.Point(64, 104);
            Controls.Add(tLabel);
            Text = "WindowsConvert";
        }
        protected void iHandleClick(object sender, System.EventArgs e)
        {
            double tCentigrade = double.Parse(tTextBox.Text);
            double tFahrenheit = 32 + tCentigrade*9/5;
            tLabel.Text = tFahrenheit.ToString();
        }
        public static void Main()
        {
            MyForm tMyForm = new MyForm();
            Application.Run(tMyForm);
        }
    }
}
```

The class `MyForm` is derived from `System.Windows.Forms.Form`. Its constructor creates a `TextBox` object, a `Button` object and a `Label` object. It adds each of these to the `Controls` property of the `Form`.

A `Button` object has an `Event` property called `Click` and `MyForm`'s constructor uses `+=` to add a method called `iHandleClick` to the list of methods awaiting a click of the button.

When the button is clicked, the `iHandleClick` method is executed. This takes the string stored in the `TextBox`, converts it to a double, produces the corresponding value in Fahrenheit, and stores this as a string in the `Label`.

The class `MyForm` has a `Main` method. When this program is run, it creates a `MyForm` object and passes this an argument to `Application`'s `Run` method. The program will terminate when the close button of the form is clicked.

Although it is possible to produce the above program using any text editor, Visual Studio.NET has a wizard that can be used to generate a Windows Forms application. If this is used, you can generate the program by dragging a

TextBox, a Button and a Label from the ToolBox onto the form and then adding the C# code to respond to a click of the button. Although the code Visual Studio.NET generates is more verbose than that given above, most of it is reasonably easy to understand.

14 Other examples of C# programs

There are some other examples of C# programs at:
http://www.dur.ac.uk/barry.cornelius/papers/a.taste.of.csharp/slides.baw/0052.htm

15 Some conclusions

I'll cheat. In his article for *JavaWorld* ([5]), Mark Johnson concludes as follows.

'If I were a Windows developer, I would be rejoicing at the creation of C#. It is much easier to use than C++, and yet is more full featured than Visual Basic. MFC programmers, in particular, should flock to this tool. It seems likely that C# will become a major language for developing under the Windows platform. Because of C# creator Anders Hejlsberg's excellent track record, I expect the language to live up to its promises, assuming that Microsoft delivers an implementation that does so. C# solves most of the same problems with C++ that Java solved five years ago, usually in the same way. C# also solves the business problems that Microsoft encountered when it found it could embrace and extend Java, but not extinguish it. And, if Microsoft marketing is to be believed, COM will finally be usable. C# itself is not particularly innovative: there is little in this language that has not been seen before. Still, its skillful combination of these well-understood features will provide good value to Windows programmers. Of course, those not wanting to limit themselves to Windows can still choose from among the many exceleent implementations of Java for Windows.'

'Because of its lack of platform neutrality, C# is in no way a "Java killer." Even leaving aside Sun's five-year head start, and Java's successful capture of the "gorilla" (market-owning) position among enterprise server languages, C#'s Achilles' heel is that it is tied to Windows. Of course, in theory it isn't. But widespread cross-platform implementation of C# is like widespread multivendor implementation of Win32 and COM: possible, in theory.'

'High-technology consumers today, and especially IT managers, are appropriately wary of vendor lock-in. Encoding procedural information assets in a way that ties them to a particular vendor is a Faustian bargain. The Java platform is neutral with respect to operating systems. If you don't like the service you are getting from one vendor, or if your needs change, you can find another vendor that better meets your requirements. It will be some time before that can be said of C# or .Net. In short, while C# is a fine language for Windows programming, it will be able to compete with Java only when C# is freed from its Windows dependence. For the time being, C# users still won't get to decide where they're going today.'

16 References

Microsoft have written a Reviewers Guide ([7]) for the .NET Framework and Visual Studio.NET. You may also find it useful to read an article by Mark Johnson ([5]).

1. Ben Albahari, Peter Drayton and Brad Merrill, 'C# Essentials',
O'Reilly, 2001, 0-596-00079-0.

2. Barry Cornelius, 'Web Forms and Web Services',
http://www.dur.ac.uk/barry.cornelius/java/web.forms.and.services/

3. Eric Gunnerson, 'A Programmer's Introduction to C#',
Apress, 2000, 1-893115-86-0.

4. Billy Hollis and Rockford Lhotka, 'VB.NET Programming',
Wrox Press, 2001, 1-861004-91-5.

5. Mark Johnson, 'C#: A language alternative or just J–?',
http://www.javaworld.com/javaworld/jw-11-2000/jw-1122-csharp1.html

6. Microsoft, 'C# Language Specification',
http://msdn.microsoft.com/vstudio/nextgen/technology/csharpdownload.asp

7. Microsoft, 'Visual Studio.NET and .NET Framework Reviewers Guide',
http://msdn.microsoft.com/vstudio/nextgen/evalguidedownload.asp

8. Sun Microsystems, 'About Microsoft's "DELEGATES"',
http://java.sun.com/docs/white/delegates.html

9. Thuan Thai and Hoang Q. Lam, '.NET Framework',
O'Reilly, 2001, 0-596-00165-7.