OOP using VB.NET

Barry Cornelius
Computing Services, University of Oxford
Date: 23rd October 2004; first created: 18th October 2004
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 also have an *integrated development environment* (*IDE*) called *Visual Studio.NET*. This 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 IIS (Microsoft's web server software). 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 programs running elsewhere in the world.

Microsoft are providing .NET compilers for many languages including the latest version of Visual Basic called *Visual Basic.NET*. The aim of this document is to introduce the ideas of object-oriented programming using Visual Basic.NET.

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 classes;
3. classes 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

2.2.1 The role of 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 .NET compilers for several programming languages: Managed C++, Visual Basic.NET, JScript, J# and C#. In addition, other people/companies are producing .NET compilers for other languages including COBOL, Eiffel, Fortran, 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 instructions coded 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.

There is a project called *Mono* that is building an open-source implementation of the .NET Framework, and Microsoft are working on an implementation for FreeBSD.

2.2.2 The Common Type System

Besides providing the functionality normally expected from a runtime system, the CLR also defines a *Common Type System* (*CTS*). In their book, .NET Framework Essentials, Thai and Lam 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 `Integer` or a value of some structure 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*.

The aspects of the CTS that must be supported by all .NET languages is defined by the *Common Language Specification* (*CLS*). The CLS says that each language must provide value types (simple types, structure types, enumeration types) and reference types (class types, interface types, array types, delegate types).

2.2.3 The simple types

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

	size	*C#*	*Managed C++*	*Visual Basic.NET*
`System.Boolean`	8	`bool`	`bool`	`Boolean`
`System.Byte`	8	`byte`	`char`	`Byte`
`System.Int16`	16	`short`	`short`	`Short`
`System.Int32`	32	`int`	`int` or `long`	`Integer`
`System.Int64`	64	`long`	`__int64`	`Long`
`System.Single`	32	`float`	`float`	`Single`
`System.Double`	64	`double`	`double`	`Double`
`System.Char`	16	`char`	`__wchar_t`	`Char`
`System.Decimal`	128	`decimal`	`System::Decimal`	`Decimal`

2.3 A comprehensive class library

One of the significant features of the .NET Framework is that it comes with an enormous class library called the *Framework Class Library* (*FCL*). The types provided by the FCL can be used in any .NET language.

To begin with, you may want to become familiar with the following types:

```
System.Object
System.ValueType
System.String
System.IComparable
System.IEnumerable
System.Collections.ArrayList
System.Collections.Hashtable
System.Collections.IEnumerator
```

There are also types for accessing databases, using XML, and many more besides these!

2.4 Types associated with various applications

There are four main kinds of applications: console applications, windows forms applications, web form applications (ASP.NET pages) and web services. Each of these has associated types.

For example, if you are developing a console application, you will need the type:

```
System.Console
```

And, if you are developing a windows forms application, you will find the following types useful:

```
System.Windows.Forms.Form
System.Windows.Forms.Application
System.Windows.Forms.Button
System.Windows.Forms.TextBox
System.Windows.Forms.Label
System.EventArgs
System.EventHandler
System.Drawing.Point
System.Drawing.Size
System.Windows.Forms.MainMenu
System.Windows.Forms.MenuItem
```

Later, we will look in more detail at console applications and windows forms applications.

2.5 Other benefits of the .NET Framework

2.5.1 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 simple types, the .NET Framework makes it easy to build programs where different bits of the code are 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.

2.5.2 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, suppose you have declared some variable:

```
ArrayList tArrayList;
```

and 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`.

2.5.3 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.

2.5.4 Versioning

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 more than one DLL with the same name.

3 VB using just the .NET Framework

The .NET Framework can be downloaded free of charge. In the future, it will be included as part of the Windows operating system.

Note: At the present time, there are two .NET Framework downloads: there is a *.NET Framework Redistributable* which software developers can give away with their products, and there is also a *.NET Framework SDK* which includes the compilers for Visual Basic.NET and C#. Although both of these downloads can currently be downloaded for free, I am unclear as to whether both will form part of future versions of the Windows operating system.

Rant: It is interesting to note that Microsoft are now giving away compilers: this once again makes it difficult for other software developers to offer competing products.

So you just need the .NET Framework in order to compile and execute Visual Basic.NET programs or C# programs. The compilers are in files called `vbc.exe` and `csc.exe`. On my laptop these files are in the directory:

```
E:\WINDOWS\Microsoft.NET\Framework\v1.1.4322
```

So, in a command shell window, add the appropriate directory to the path:

```
path=E:\WINDOWS\Microsoft.NET\Framework\v1.1.4322;%path%
```

Then use your favourite text editor to produce a program in VB.NET. For example, suppose you use a text editor to produce a file called `Module1.vb` that contains the following text. Note: ignore the line numbers that appear in this document: they do not form part of the programs:

```
0001: Imports System
0002: Module Module1
0003:
0004:     Sub Main()
0005:         Console.Write("Centigrade value: ")
0006:         Dim tCentigradeString As String = Console.ReadLine()
0007:         Dim tCentigrade As Double = Double.Parse(tCentigradeString)
0008:         Dim tFahrenheit As Double = 32 + tCentigrade * 9 / 5
0009:         Console.WriteLine("Fahrenheit value: " & tFahrenheit)
0010:     End Sub
0011:
0012: End Module
```

This program can then be compiled using the command line:

```
vbc Module1.vb
```

This produces a file called `Module1.exe` that can be executed in the usual way:

```
Module1
```

4 VB using Visual Studio.NET

Visual Studio.NET is Microsoft's integrated development environment for producing programs. It can be used to create programs written in Visual Basic.NET, C#, Managed C++, J# and JScript.

There are many ways in which Visual Studio.NET makes programming easier:

- It does syntax checking as you type.
- It provides help about the identifiers that you are using as you type them (called *intellisense*).
- It provides a help system (which besides other things documents the FCL).
- It provides wizards that automatically generate the code of typical programs (including a wizard for generating a GUI).
- It has a built-in debugger.
- When using VB.NET, it usually adds the end of a statement after you have typed its start.

5 Object-oriented programming

5.1 Shared methods

The program given above includes four calls of *methods*. Two of these are calls of *subroutines*: these are the calls of `Console.Write` and `Console.WriteLine`. The other two calls are calls of *functions*: these are the calls of `Console.ReadLine` and `Double.Parse`.

Related methods can be grouped together using a *class* (or a *structure*). For example, the class `Console` contains methods associated with the keyboard/screen. You can deduce from the above program that it has methods called `Write`, `WriteLine` and `ReadLine`. And you can see that the structure `Double` has a method called `Parse`.

Another class that provides a lot of methods is a class called `Math`:

```
Double x = 2.0;
Double y = Math.Sqrt(x);
```

You call these methods using the notation:

```
ClassOrStructureName.MethodName(parameters)
```

In Visual Basic.NET, such methods are called *shared methods*: elsewhere they are called *static methods* or *class methods*. There is nothing new here: since the beginning of computing, we have had libraries of methods. So, shared methods are not exciting, and they have nothing to do with object-oriented programming.

5.2 Using types other than the simple types

One approach to writing a program is to identify the real-world objects (of the problem) that you need to represent in the program. Each real-world object can be in a number of states (i.e., may possess one of a number of values) and has a set of operations that can be performed on it.

Some of these real-world objects can be represented in a program by variables each of which is declared to be one of the simple types. For example, if our problem needs to represent the number of students in a lecture theatre, we can use an Integer in a VB.NET program in order to represent this real-world object. Or we might use a variable of type Double in order to represent the temperature of the air outside.

However, there are many real-world objects that cannot be represented in a VB.NET program by a value that is of a simple type. For example, we might want to represent a date in history, a point in two-dimensional space, the ISBN of a book, an e-mail address, the URL of a WWW page, and so on.

In most programming languages, a programmer can introduce new types to model these real-world objects. One way of doing this in a .NET language is to introduce a *class declaration*: we could introduce a class declaration for each of these classes of real-world objects.

However, we need not get into the details of writing class declarations straightaway: any .NET language can use any of the large number of classes defined in the Framework Class Library: maybe the FCL has a class that we can use.

5.3 Creating objects

Suppose you want to store a collection of strings. One way of doing this is to use a class of the FCL called ArrayList. This class allows you to create a flexible array of objects, an array which increases in size whenever necessary.

The ArrayList class provides several *constructors*. One of these allows you to specify the initial capacity of the ArrayList. Using this constructor, we can create an object using `New`:

```
        Dim tLanguages As ArrayList = New ArrayList(2)
```

This makes the variable tLanguages point to an ArrayList object. It is a bit silly to give the ArrayList a capacity of 2, but this silliness will demonstrate that the ArrayList automatically increases in size when necessary.

The main purpose of a constructor is to give some initial values for aspects of the object being created.

5.4 Instance methods

The ArrayList class provides a method called Add that can be used to add an object to the end of an ArrayList. Here it is in action:

```
        tLanguages.Add("Visual_Basic")
        tLanguages.Add("C#")
        tLanguages.Add("Managed_C++")
        tLanguages.Add("JScript")
```

These calls of Add are unlike the calls of Write, WriteLine, Readline and Parse. Because, here before the dot, we put the name of the object to which we want the Add method applied. We might have several ArrayList objects in our program: however, in these statements, we want the Add method applied to the tLanguages object.

This kind of method is called an *instance method* because the method is being applied to a specific instance of the type.

The call of an instance method has the form:

```
VariableName.MethodName(parameters)
```

Suppose in our program we include a method to output an ArrayList of Strings:

```
    Sub OutputStringList(ByVal pArrayList As ArrayList)
        For Each tString As String In pArrayList
            Console.Write(tString & " ")
        Next
        Console.WriteLine()
    End Sub
```

This method can then be called in the following way:

```
        OutputStringList(tLanguages)
```

This produces the output:

```
Visual_Basic C# Managed_C++ JScript
```

The ArrayList class also provides a method for reversing the elements of an ArrayList. This is another instance method. Here is a use of it:

```
        tLanguages.Reverse()
```

If we call OutputStringList again:

```
        OutputStringList(tLanguages)
```

we will get the output:

```
JScript Managed_C++ C# Visual_Basic
```

The ArrayList class provides several other instance methods. Here is a use of Insert and Sort:

```
        tLanguages.Insert(2, "J#")
        OutputStringList(tLanguages)
        tLanguages.Sort()
        OutputStringList(tLanguages)
```

These statements produce the output:

```
JScript Managed_C++ J# C# Visual_Basic
C# J# JScript Managed_C++ Visual_Basic
```

All of these calls are calls of instance methods: a method is being applied to an instance of the class.

5.5 Namespaces and Imports statements

So far we have used Console, Double, String and ArrayList. (Three of these types are classes; the other one is a structure.) Unless the organisation of names is more complicated, we would quickly run out of names. To avoid name clashes, each name is allocated to a *namespace*.

For example, Console, Double and String belong to the System namespace.

If we wish, we can fully qualify all the names we use:

```
Dim tCentigradeString As System.String = System.Console.ReadLine()
```

Some namespaces have *subnamespaces*. For example, the System namespace has a subnamespace called Collections. And ArrayList belongs to this subnamespace. So you could write:

```
Dim tLanguages As System.Collections.ArrayList = New System.Collections.ArrayList(2)
```

Obviously, it is tedious to use fully qualified names: they can be avoided by using an *Imports statement*. So, if your program starts with the lines:

```
Imports System
Imports System.Collections
Module Module1
```

you can use any of the classes of the System namespace or the System.Collections namespace without fully qualifying their names.

If you are using Visual Studio.NET (rather than using the vbc command), some Imports statements can be omitted. When using Visual Studio.NET, you do not need:

```
Imports System
Imports System.Collections
Imports System.Data
Imports System.Diagnostics
Imports System.Drawing
Imports System.Windows.Form
Imports Microsoft.VisualBasic
```

5.6 Providing our own types

We cannot just rely on others to produce all the types we need: we have to do some of this work ourselves. This is because some of the real-world objects that we will want to represent will be specific to whatever we are doing. We may need to represent people, bank accounts, grid references, ISBNs, audio CDs, and so on. And there are no types in the FCL for any of these.

In .NET languages, there are two kinds of types: *value types* and *reference types*.

A value of a value type is stored in a variable whereas for a reference type a variable points to the object in which the value is stored.

So the type Double is a value type whereas the type ArrayList is a reference type.

In .NET languages, there are two kinds of value types: *structure types* and *enumeration types*; and there are four kinds of reference types: *class types*, *interface types*, *array types* and *delegate types*.

Note: all of the simple types (e.g., Integer and Double) are structure types.

5.7 Providing a new class type in VB.NET

For an example of a class declaration, we now look at how we might represent a point in 2D space.

Note: although the FCL provides a type (`System.Drawing.Point`) for representing points in 2D space, for the sake of this exercise, we will pretend that it does not exist.

When we design a class declaration, we should think long-term: not about what we need in our current program but about what would be useful in this class so that it could also be used by other programs in the future.

There are two main characteristics to a type:

- a type has a set of values associated with it;
- a type has a set of operations that are permitted on those values.

So when we are choosing how to represent something (e.g., points in 2D space) we need to consider these two characteristics.

5.8 A class type for representing 2D points

Here is a class type that can be used for representing 2D points:

```
0037: Public Class Point
0038:     Private iX As Integer
0039:     Private iY As Integer
0040:     Public Sub New(ByVal pX As Integer, ByVal pY As Integer)
0041:         iX = pX
0042:         iY = pY
0043:     End Sub
0044:     Public Property X() As Integer
0045:         Get
0046:             Return iX
0047:         End Get
0048:         Set(ByVal Value As Integer)
0049:             iX = Value
0050:         End Set
0051:     End Property
0052:     Public Function Distance() As Double
0053:         Return Math.Sqrt(iX * iX + iY * iY)
0054:     End Function
0055:     Public Overloads Overrides Function Equals(ByVal pObject As Object) As Boolean
0056:         If (pObject Is Nothing Or Not (Me.GetType() Is pObject.GetType())) Then
0057:             Return False
0058:         End If
0059:         Dim tPoint As Point = DirectCast(pObject, Point)
0060:         Return Me.iX = tPoint.iX And Me.iY = tPoint.iY
0061:     End Function
0062:     Public Overrides Function ToString() As String
0063:         Return iX & ":" & iY
0064:     End Function
0065: End Class
```

This class declaration needs to be put in a file called Point.vb.

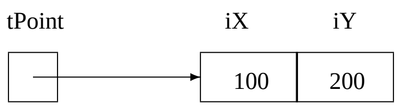
In Visual Studio.NET, you add a new class (called `Point`) to a project by selecting *Add New Item* from the *File* menu, selecting the *Class* icon, changing the contents of the textbox to `Point.vb`, and clicking on *Open*.

Having created the class `Point`, how do we use it in a program? How do we create objects of this class, and use them?

Well, the class Point provides a constructor. It is shown on lines 40 to 43. So, in a program (e.g., Module1), an instance of the `Point` class can be created by:

```
0068:         Dim tPoint As Point = New Point(100, 200)
```

This produces:



The class Point also provides an instance method called ToString. So in our program we can call ToString as follows:

```
0069:         Dim tPointString As String = tPoint.ToString()
0070:         Console.WriteLine(tPointString)
```

This will output:

```
100:200
```

However, if a class declares a method called ToString, the ToString method will be called if we pass an object of that class to Write/WriteLine. So the above two statements can be abbreviated to:

```
0071:         Console.WriteLine(tPoint)
```

When you produce a class, it is very useful to provide a method called ToString.

In the following statements:

```
Dim tDistance As Double = tPoint.Distance()
Console.WriteLine(tDistance)
```

there is a call of an instance method called Distance. This will output:

```
223.606797749979
```

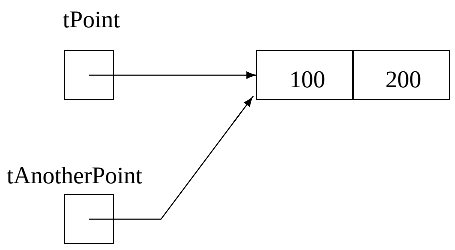
Suppose we now execute the statements:

```
Dim tAnotherPoint As Point = tPoint
Console.WriteLine(tAnotherPoint)
```

we will get the output:

```
100:200
```

We now have:



i.e., both variables are pointing to the same object.
This can be tested using the Is operator:

```
Console.WriteLine("Is: " & (tPoint Is tAnotherPoint))
```

This outputs:

```
Is: True
```

The class Point also provides an instance method called Equals. This can be called as follows:

```
Console.WriteLine("Equals: " & (tPoint.Equals(tAnotherPoint)))
```

This outputs:

```
Equals: True
```

If we create another point that happens to have the same value as tPoint then Is will return False whereas Equals will return True. So the statements:

```
Dim tClonePoint As Point = New Point(100, 200)
Console.WriteLine("Is: " & (tPoint Is tClonePoint))
Console.WriteLine("Equals: " & (tPoint.Equals(tClonePoint)))
```

output:

```
Is: False
Equals: True
```

Whenever you produce a class, it is important to provide a method called Equals that can be used to test whether two objects of that class have the same value.
In another demonstration that shows tPoint and tAnotherPoint are pointing to the same object, consider:

```
tAnotherPoint.X = 42
Console.WriteLine(tPoint.X)
```

These statements output:

```
42
```

Note: in the above declaration of the Point class, the iX and iY fields have been marked as Private. This means that a *client* of the class Point (such as Module1) is not allowed to access the iX and iY fields of any object. Indeed, if Module1 contains a statement like:

```
tAnotherPoint.iX = 42
```

it will not compile.

Instead, if it is necessary to allow a client to inspect/alter the value of a field, the class should provide a Property. In the above declaration of the Point class, a Property called X has been introduced that permits the value of the iX field to be *got* or *set*. By using a Property, you know that all accesses to the field will be done through the code of the Property. So you retain control. This is useful, e.g.:

- if you want to ensure that the values being assigned to the fields are in the correct range;
- if you later want to change what fields you use to represent values of the type.

If we now change the iX field of tClonePoint:

```
        tClonePoint.X = 27
        Console.WriteLine(tPoint.X)
```

the output:

```
42
```

again demonstrates that tPoint and tClonePoint are pointing to different objects.

5.9 Structure types

In C#, structure types are called *structs*. Chapter 11 of the ‘C# Language Specification’ says ‘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 good examples of structs.’ So, if we want to represent points in 2D space in a program, it would probably be more appropriate to use a structure type rather than a class type.

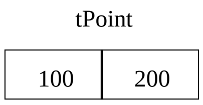
A declaration of a structure type looks similar to that of a class type:

```
Public Structure Point
    Private iX As Integer
    Private iY As Integer
    Public Sub New(ByVal pX As Integer, ByVal pY As Integer)
        iX = pX
        iY = pY
    End Sub
    Public Property X() As Integer
        Get
            Return iX
        End Get
        Set(ByVal Value As Integer)
            iX = Value
        End Set
    End Property
    Public Function Distance() As Double
        Return Math.Sqrt(iX * iX + iY * iY)
    End Function
    Public Overrides Function ToString() As String
        Return iX & ":" & iY
    End Function
End Structure
```

Suppose we alter Module1 so that instead of it using the class called `Point` it uses this structure called `Point`. This will significantly affect the meaning of some of the statements.

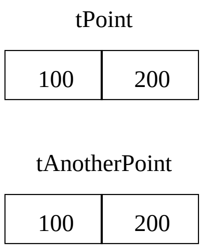
For example, a value of a structure type is stored in the variable:

```
        Dim tPoint As Point = New Point(100, 200)
```



And an assignment statement copies the value:

```
        Dim tAnotherPoint As Point = tPoint
```



5.10 Differences between value types and reference types

The main ways in which value types are different from reference types are as follows:

- A value of a value type is stored in a variable whereas a value of a reference type is pointed to by some variable.
- A local variable that is of some value type is stored on the stack whereas a value of a reference type is stored on the heap.
- A local variable that is of some value type comes into being when the block (in which it is declared) is entered, and it 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.
- If a value is passed as an argument to a method, it will be copied into the parameter (unless `ByRef` is present). This will be time-consuming if the size of the value is large.
- Each value type is derived from `System.ValueType` (which is in turn derived from `System.Object`).

As well as declaring our own value types (such as the structure type Point given in the previous section), there are hundreds of value types in the FCL. Some examples of enumeration types (from the FCL) are:

```
System.DayOfWeek                      System.IO.FileMode
System.Drawing.Drawing2D.DashStyle    System.Drawing.Drawing2D.FillMode
System.Globalization.DateTimeStyles   System.Windows.Forms.ButtonBorderStyle
```

Some examples of structure types (from the FCL) are:

```
System.DateTime                       System.TimeSpan
System.Drawing.Color                  System.Drawing.Point
System.Drawing.Rectangle              System.Drawing.Size
```

6 Turn On the Option Strict

One area where bugs often occur in programming is where variables are not typed or values are converted from one type to another. In VB.NET, it is possible to insist that all variables are typed and possible unsafe conversions are documented using DirectCast (or CType). This is done by setting the Option Strict to On.

At http://www.eggheadcafe.com/articles/20030801.asp, it says: ‘Unfortunately, in bringing out the final release of Visual Studio.NET, Microsoft bowed to the cries of millions of VB programmers and left Option Strict Off by default in all our VB.NET projects! Just about every professional author and programmer will tell you that you should ALWAYS set Option Strict to "On"!’

‘Fortunately, there is a very easy way to do this so that you won’t forget. You are hereby strongly encouraged to go to the Visual Studio’s Tools | Options menu, click the Projects | VB Defaults folder and set Option Strict and Option Explicit to On. You might then spend a bit more time writing CType-s or DirectCast-s, but you will avoid spending much more time tracking down some mysterious runtime errors. Many of your programs will also run FASTER.’

That does not seem to work for projects that you have already created. For such projects, do the following: From the Project menu, choose the last option which is the name of the project followed by the word Properties (e.g., PointClassTest Properties). Then click on Build. Then change the setting of "Option Strict" to "On".

7 Windows Forms applications

Standalone programs that read and/or write to a command shell window are boring: over the years, Visual Basic has led the way in making it easier to produce programs that have a *graphical user interface* (*GUI*).

Although it is possible to construct these programs yourself with a text editor, it is a lot easier to use one of Visual Studio.NET's wizards.

If, from within Visual Studio.NET, you choose the *Windows application* template, Visual Studio.NET will present a *Form* and invite you to add GUI components to the Form from a *Toolbox*.

Suppose we want a program that records the arrivals and departures of vehicles from a carpark. To the Form, we could add:

- a TextBox for the user to type in the number plate of the vehicle;
- a Button which we can label "Arrive";
- a Button which we can label "Depart";
- a Label which the program can write to to confirm what has happened.

As you add GUI components to the Form, behind the scenes the wizard is adding code to a file called `Form1.vb`.

One key thing it does is to create variables in the program for each of the GUI components: they will be called TextBox1, Button1, Button2 and Label1.

Having added the GUI components to the Form, you can double click each button in turn. When you double click a button, Visual Studio.NET will open up the code of the Form1.vb file positioning you in the midst of the method responsible for handling clicks of that button. So if you double click the first button, it will put you in the body of a method called `Button1_Click`. You can then add the code you want executed when that button is clicked.

So I just provided lines 216 and 221 of the following code:

```
0128: Public Class Form1
0129:     Inherits System.Windows.Forms.Form
0130:
0131: #Region " Windows Form Designer generated code "
0132:
0133:     Public Sub New()
0134:         MyBase.New()
0135:
0136:         'This call is required by the Windows Form Designer.
0137:         InitializeComponent()
0138:
0139:         'Add any initialization after the InitializeComponent() call
0140:
0141:     End Sub

          ...

0212: #End Region
0213:
0214:     Private Sub Button1_Click(ByVal sender As System.Object, _
0215:             ByVal e As System.EventArgs) Handles Button1.Click
0216:         Label1.Text = TextBox1.Text + " has arrived"
0217:     End Sub
0218:
0219:     Private Sub Button2_Click(ByVal sender As System.Object, _
0220:             ByVal e As System.EventArgs) Handles Button2.Click
0221:         Label1.Text = TextBox1.Text + " has departed"
0222:     End Sub
0223:
0224: End Class
```

When we are using Visual Studio.NET, a lot of this code is hidden from us: on the screen, it is in the *region* labelled *Windows Form Designer generated code*. You can click on the + button in order to reveal the contents of this region.

You should be careful about altering the contents of this region. If you change the GUI components of the Form, it will automatically change the code of this region. Similarly, if you are clever enough to edit the code of the region to add the code for a new GUI component and it understands what you have done, it will automatically alter the contents of the Form.

8 Using an ArrayList to record the events

As well as confirming what has happened in a Label of the Form, perhaps we would like to record what has happened in a data structure. We could use an ArrayList for this.

We will need to refer to the ArrayList from several methods of the Form1 class and so an ArrayList variable (called `iArrayList`) can be declared as a Private variable of the class:

```
Public Class Form1
    Inherits System.Windows.Forms.Form

    Private iArrayList As ArrayList
#Region " Windows Form Designer generated code "
```

An ArrayList object has to be created and assigned to the iArrayList variable. As we want this to be done just once, an obvious place for doing this is in Form1's constructor:

```
    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call
        iArrayList = New ArrayList
    End Sub
```

And then the code of each method handling button clicks can be altered:

```
    Private Sub Button1_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles Button1.Click
        iArrayList.Add(TextBox1.Text)
        Label1.Text = TextBox1.Text + " has arrived"
    End Sub

    Private Sub Button2_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles Button2.Click
        iArrayList.Remove(TextBox1.Text)
        Label1.Text = TextBox1.Text + " has departed"
    End Sub

    Private Sub Button3_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles Button3.Click
        TextBox2.Clear()
        For Each tString As String In iArrayList
            TextBox2.AppendText(tString & vbCrLf)
        Next
    End Sub
End Class
```

9 Using a CarPark class instead

The problem with the above solution is that the code handling the carpark is mixed up with the code handling the GUI. If there is a bug with the code for the carpark, then it is difficult to find the code because of all the code to do with the GUI.

It would be better to design a class for handling a carpark. If we are going to do this, then as before there are two questions to answer:

- what set of values are associated with a carpark and how are going to represent these values in our program;
- what set of operations are permitted on a carpark.

We have already decided an answer to the first question: a collection of number plates represented using an ArrayList. There are all sorts of operations that we could provide:

- an arrival of a vehicle;
- the departure of a vehicle;
- providing a list of those vehicles in the carpark;
- providing how many vehicles are in the carpark;
- providing a list of those vehicles that have stayed in the carpark longer than some period of time;
- ...

Here is the code of one possible CarPark class:

```
Imports System.Text
Public Class CarPark
    Private iArrayList As ArrayList
    Public Sub New()
        iArrayList = New ArrayList
    End Sub
    Public Sub Arrive(ByVal pString As String)
        iArrayList.Add(pString)
    End Sub
    Public Sub Depart(ByVal pString As String)
        iArrayList.Remove(pString)
    End Sub
    Public Overrides Function ToString() As String
        Dim tStringBuilder As StringBuilder = New StringBuilder
        For Each tString As String In iArrayList
            tStringBuilder.Append(tString & vbCrLf)
        Next
        Return tStringBuilder.ToString()
    End Function
    Public Function Size() As Integer
        Return iArrayList.Count
    End Function
End Class
```

And here are the relevant parts of the Form1 class altered to use the CarPark class:

```
Public Class Form1
    Inherits System.Windows.Forms.Form

    Private iCarPark As CarPark
#Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call
        iCarPark = New CarPark
    End Sub

    ...

    Private Sub Button1_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles Button1.Click
        iCarPark.Arrive(TextBox1.Text)
        Label1.Text = TextBox1.Text + " has arrived"
    End Sub

    Private Sub Button2_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles Button2.Click
        iCarPark.Depart(TextBox1.Text)
        Label1.Text = TextBox1.Text + " has departed"
    End Sub

    Private Sub Button3_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles Button3.Click
        TextBox2.Text = iCarPark.ToString()
    End Sub
End Class
```

10 Object-oriented programming revisited

By doing this, no longer is the code of the GUI mixed up with the code of the carpark. Indeed, it would now be easier to create several carparks: we would not have to duplicate the code for adding to the ArrayList or for visiting each element of the ArrayList, and so on.

The class declaration for CarPark hides an ArrayList and provides a number of methods for *clients* to access the ArrayList indirectly. This idea of hiding fields behind access methods is called *information hiding* or *data encapsulation*.

It may seem like a complicated way of providing an object and accessing it. However, you should view the class declaration as documenting a design decision. At the moment, we have chosen to represent a carpark by an ArrayList. At a later stage, we may feel that this is wrong: we may want to use some other representation. Since we have retained control over the access to the field of the class by making it private and providing access methods, we can make changes like this with only minimal impact to the code of the program: we know that the only code that needs to be changed is located in the methods of the CarPark class.

To demonstrate this, here is a different version of the CarPark class: in this version, the events are recorded in a MySQL database. The important thing to notice that the only code that needs to be changed is the code of the CarPark class: there are no changes to the Form1 class.

```
Imports System.Text
Imports System.Data.Odbc
Public Class CarPark
    Private iConnectionString As String = _
```

```
            "DSN=Pdcl0bjc_CarParkVB;UID=dcl0bjc;PWD=XXXXXXX;"
    Public Sub New()
        Console.WriteLine("CarPark New")
    End Sub
    Public Sub Arrive(ByVal pString As String)
        Dim tSQLString As String = _
                "INSERT INTO vehicles SET number = ’" + pString + "’"
        Console.WriteLine(tSQLString)
        Dim tOdbcConnection As OdbcConnection = _
                New OdbcConnection(iConnectionString)
        Dim tOdbcCommand As OdbcCommand = _
                New OdbcCommand(tSQLString, tOdbcConnection)
        Try
            tOdbcConnection.Open()
            tOdbcCommand.ExecuteNonQuery()
            tOdbcConnection.Close()
        Catch pException As Exception
            MessageBox.Show(pException.Message)
        End Try
    End Sub
    Public Sub Depart(ByVal pString As String)
        Dim tSQLString As String = _
                "DELETE FROM vehicles WHERE number = ’" + pString + "’"
        Console.WriteLine(tSQLString)
        Dim tOdbcConnection As OdbcConnection = _
                New OdbcConnection(iConnectionString)
        Dim tOdbcCommand As OdbcCommand = _
                New OdbcCommand(tSQLString, tOdbcConnection)
        Try
            tOdbcConnection.Open()
            tOdbcCommand.ExecuteNonQuery()
            tOdbcConnection.Close()
        Catch pException As Exception
            MessageBox.Show(pException.Message)
        End Try
    End Sub
    Public Overrides Function ToString() As String
        Dim tStringBuilder As StringBuilder = New StringBuilder
        Dim tSQLString As String = "SELECT number FROM vehicles"
        Console.WriteLine(tSQLString)
        Dim tOdbcConnection As OdbcConnection = _
                New OdbcConnection(iConnectionString)
        Dim tOdbcCommand As OdbcCommand = _
                New OdbcCommand(tSQLString, tOdbcConnection)
        Try
            tOdbcConnection.Open()
            Dim tOdbcDataReader As OdbcDataReader = _
                    tOdbcCommand.ExecuteReader(CommandBehavior.SequentialAccess)
            While (tOdbcDataReader.Read())
                tStringBuilder.Append(tOdbcDataReader.GetString(0) & vbCrLf)
            End While
            tOdbcConnection.Close()
        Catch pException As Exception
            MessageBox.Show(pException.Message)
        End Try
        Return tStringBuilder.ToString()
    End Function
    Public Function Size() As Integer
        Dim tNumRows As Integer = 0
        Dim tSQLString As String = "SELECT number FROM vehicles"
        Console.WriteLine(tSQLString)
        Dim tOdbcConnection As OdbcConnection = _
                New OdbcConnection(iConnectionString)
        Dim tOdbcCommand As OdbcCommand = _
                New OdbcCommand(tSQLString, tOdbcConnection)
        Try
            tOdbcConnection.Open()
            Dim tOdbcDataReader As OdbcDataReader = _
                    tOdbcCommand.ExecuteReader(CommandBehavior.SequentialAccess)
            While (tOdbcDataReader.Read())
                tNumRows += 1
            End While
            tOdbcConnection.Close()
        Catch pException As Exception
            MessageBox.Show(pException.Message)
        End Try
        Return tNumRows
    End Function
End Class
```

The above code uses the *Open Database Connectivity* .NET Data Provider. In order to use ODBC, you will need to install an ODBC driver. As the above code connects to a MySQL server, I installed the MyODBC driver, onto the laptop from which I was running VB.NET.

11 Final thoughts

One book on Visual Basic.NET is ‘Visual Basic.NET for Dummies’ by Richard Mansfield (Wiley, 2003, 0-7645-2579-4). The Appendix of this book gives a useful summary of the issues involved in moving from Visual Basic 6 to Visual Basic.NET.

In this document, some of the ideas of object-oriented programming have been introduced. This has been done in the context of Visual Basic.NET, Microsoft’s version of VB for the .NET Framework.

Although it introduces many of the basic ideas of object-oriented programming, this document has not mentioned inheritance, dynamic binding and interfaces.