Programming in Visual Basic .NET^{TM}

Rex Jaeschke

Programming in Visual Basic .NET

© 2001–2002, 2004–2005, 2007, 2009 Rex Jaeschke. All rights reserved.

Edition: 2.0

The training materials associated with this book are available for license. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
+1 (703) 860-3008 (fax)
www.RexJaeschke.com
rex@RexJaeschke.com

Preface

Throughout this book, we will look at the statements and constructs of the Visual Basic .NET programming language. Each statement and construct will be introduced by example with corresponding explanations, and, except where errors are intentional, the examples will be complete programs or procedures that are error-free. I encourage you to run and modify these examples, because the only realistic way to master a language is to write programs in it, run them, and debug them.

This book has been written with teaching in mind and has evolved from an earlier series on C, C++, Java, and C#. It has been written for use both in a classroom environment as well as for self-paced learning.

Visual Basic .NET is a robust, general-purpose, high-level language that supports object-oriented programming (OOP). It contains significant extensions to its very popular predecessors, Visual Basic, versions 1–6.

Visual Basic .NET supports the 16-bit character set ISO 10646 UCS-2—commonly known as Unicode, of which ASCII is a proper subset. (Character codes 0–127 represent the same characters in both sets.) As such, it does not suffer from many of the difficulties faced by other languages that were designed primarily to support a "USA-English" mode of programming.

With the .NET extensions, the language encourages a structured, modular approach to programming through the use of classes, structures, and modules containing callable procedures. Classes, structures, and modules can be compiled separately, with external references being resolved at runtime.

Visual Basic .NET is an architecturally neutral language that when compiled, results in the generation of Common Intermediate Language (CIL).[1] During program execution, this intermediate language is compiled to native code, which is then executed. CIL is not language-specific, so a program can be made up from modules written in any language that can produce CIL.[2]

There is a fine (and subjective) line between writing code that is terse and code that is cryptic. It is easy to write code that is unreadable and, therefore, unmaintainable. However, with care and a small dose of discipline, you can produce nicely formatted code that is easy to read and maintain. But remember, good code doesn't happen automatically—you have to work at it. Throughout the book, I will make numerous comments and suggestions regarding style. Perhaps the best advice I can give is "Remember that the poor fool who has to read your code in the future just might be you!" Above all, be overt and be consistent.

Numerous "Tips" and "Style Tips" have been added throughout to highlight important suggestions. Considerable attention has been paid to using and teaching a consistent, popular, and overt programming style. Both kinds of tips are highlighted so they stand out. For example:

[1] Microsoft's current implementation calls this language MSIL. CIL is the name given to it by the Ecma committee that standardized it.

[2] Currently, compilers for more than 15 different languages produce CIL. These include Microsoft's own Visual Basic .NET, Visual C++, Visual C#, and JScript.

Tip: Avoid initializing enumerators explicitly or making duplicates unless you have good reason to do so. Assigning a range of values that is non-continuous or that contains duplicates significantly reduces the set of operations that can meaningfully be performed.

Style Tip: Use liberal amounts of white space to improve program readability. The compiler discards all white space, so its presence has no effect on program execution. Apart from separating tokens, white space exists solely for the benefit of the reader. If you can't read the code, you surely won't be able to understand it.

Visual Basic .NET is not all things to all people; nor does it claim to be. For many applications, assembly language, Pascal, COBOL, FORTRAN, and C++, for example, will do just fine. In any event, compared to other high-level languages, Visual Basic .NET is a relatively inexpensive language to learn and master. It is certainly much less expensive to master—and far less error-prone—than C or C++. Learning Visual Basic .NET is comparable to learning Java or C#.

Visual Basic .NET's Design Goals

The Visual Basic .NET language, library, and run-time environment were designed to deliver the following:

- Code reuse and reduced development effort via object-oriented programming support.
- Extremely broad portability by the elimination of almost all implementation-defined and unspecified behavior, by the use of an architecturally neutral language.
- Strong type checking.
- Support for threading.
- Easy to learn, read, and write.
- Support for internationalization.
- Support for developing software components.
- Permit easy integration with other languages and operating system environments.

Reader Assumptions

I assume that you know how to use your particular text editor, Visual Basic .NET compiler, and debugger. Comments on the use of these utility programs will be limited to points of particular interest to the Visual Basic .NET programmer.

To fully understand and exploit the material, you should be conversant with the following topics:

- The basic purpose of a compiler and linker.
- Number system theory.
- Bit operations such as AND, inclusive-OR, exclusive-OR, complement, and left- and right-shift.
- Data representation.
- Communication between procedures by passing arguments and/or by returning a value, as well as via global variables.

- Use of single- and multi-dimensional arrays.
- Creation and use of sequential files and how to do formatted and unformatted I/O.
- Basic data structures such as linked lists.

Many of Visual Basic .NET's more powerful capabilities, particularly exception handling, inheritance, and threads, require advanced programming experience before they can be understood and exploited fully.

Despite the many similarities between Visual Basic .NET, C#, C, C++, and Java, you do *not* need to know any of these other languages to use this book. If you do know one or more of them, simply read quickly over those sections that seem familiar to you. I say "read quickly" rather than "skip" because you may well find that Visual Basic .NET defines things more fully or a little differently than do C#, C/C++ or Java.

If you already know C#, C++, or Java, the Visual Basic .NET learning curve should be short and not too steep. If you know C, you have all the OO stuff to learn as well. If you know some OO language other than C#, C++, or Java you'll already be familiar with the OO concepts, but not the syntax. And if your programming background is in some procedural language such as Pascal, FORTRAN, or BASIC (including early versions of Visual Basic), you'll need to read the whole book closely and do all the exercises. As for those of you having only a COBOL background, you'll have some catching-up to do with respect to thinking in binary versus decimal when it comes to data representation, as well as the use of procedures and their associated argument passing and value returning.

Note that the transition to Visual Basic .NET from Visual Basic involves more than a little work. A number of keywords are no longer supported, new keywords exist, whole new constructs have been added, an alternate way of doing I/O has been added, and the huge .NET class library has been made available. Even a simple call to the humble square root library procedure has changed. Beware, too much knowledge of Visual Basic might get in the way, since Visual Basic .NET does business a whole new way when compared to its predecessor.

Limitations

This book covers almost all of the Visual Basic .NET language. It also introduces the core class library. However, a very small percentage of library facilities are mentioned or covered in any detail. The common library used by Visual Basic .NET (and any other language that targets CIL) contains so many classes and procedures that whole books have been written about that subject alone.

Although Visual Basic .NET was designed for use in building software components, this book is directed at teaching the Visual Basic .NET language proper, by writing simple stand-alone applications, since without a thorough knowledge of the language, you will not be able to understand and implement other, more advanced, solutions.

GUI, calling non-Visual Basic .NET routines, threading, inter-process communications, and a number of other advanced features are covered in a separate text.

Presentation Style

The approach used in this book is different from that used in many other books and training courses. Having developed and delivered programming language training courses for some 15 years, I have found that the best approach for my students is an incremental one. I avoid introducing things that are unnecessary at any given time, thus making examples small, simple, and well focused. Specifically, I introduce the basic elements and constructs of the language using procedural programming examples. Once those fundamentals have been mastered, I move on to object-oriented concepts and syntax. Then come the more advanced language topics and library usage. Many books on object-oriented languages use objects, inheritance, exception handling, GUI, and numerous non-trivial library facilities in the first few examples, and certainly in the first chapter. I do not care for this approach, either as a reader or as an educator. Instead, I prefer the student to have an excellent chance of understanding and absorbing small amounts of new material, and reinforcing it with lab sessions, as they progress. The intent here is to eliminate any chance of their being overwhelmed, provided, of course, they meet the prerequisites.

Different styles work for different teachers and different students. I do not suggest that my approach is better than any other; I simply know that my approach works well, and has formed the basis of my successful seminar business for more than a decade.

Exercises and Solutions

The programs shown in the text are available electronically in a directory tree named Source, where each chapter has its own subdirectory. By convention, the names of Visual Basic .NET source files end in `.vb`.

Each chapter contains exercises, some of which have the character ***** following their number. For each exercise so marked, a solution is provided in a directory tree named Labs, in which each chapter has its own subdirectory.[1] Exercises that are not so marked have no general solution and require experimentation or research in an implementation's documentation. Numerous exercises contain a statement of the form "(See lab directory *xx*.)"' This indicates the corresponding solution or test file in the Labs subdirectory.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

The Status of Visual Basic .NET

Microsoft announced the Visual Basic .NET (and .NET platform) in July 2000. The first full implementation was called Visual Basic .NET 2002, the second, 2005, and the third, 2008. Unless stated otherwise in this book, a feature was present from the beginning; that is, in the 2002 release.

Visual Basic .NET 2005 added the following new features: XML comments; the signed byte type `SByte`; the unsigned integer types `UShort`, `UInteger`, and `ULong`; different accessibility for a property's setter and getter; partial classes, the Continue statement; generic procedures, the Using statement, the `IsNot` operator, the `TryCast` statement; operator overloading, and the keyword `Global`.

Most of the features added in Visual Basic .NET 2008 were in support of *Language-Integrated Query* (LINQ) and XML processing, neither of which is covered by this book.

Rex Jaeschke, September 2009

[1] The solutions are only available to licensees of these materials when they are used in formal training scenarios.

1. The Basics

In this chapter, we will learn about a number of fundamental constructs and language elements. The main topics covered include basic program structure, user-defined names, basic formatted output, data types, literals, constants, operator precedence, type conversion, arithmetic overflow, and the use of basic library facilities.

1.1 Basic Program Structure

A correct source program is made up of an acceptable combination of tokens, where a *token* is the smallest possible unit of source. Visual Basic .NET has a number of different kinds of tokens: identifiers, keywords, literals, operators, and separators.

With respect to source code format, Visual Basic .NET allows more freedom than some languages, but less than others. While space between tokens is usually optional, in some cases, something is needed between tokens so they can be recognized the way they were intended. White space performs this function. *White space* consists of one or more consecutive space and/or horizontal tab characters. Under certain conditions, *newline* characters are required or permitted. The newline character is entered by pressing the RETURN or ENTER key. In any source file, an arbitrary amount of white space is permitted before the first token, between any two adjacent tokens, or after the last token.

Style Tip: Use liberal amounts of white space to improve program readability. The compiler discards all white space, so its presence has no effect on program execution. Apart from separating tokens, white space exists solely for the benefit of the reader. If you can't read the code, you surely won't be able to understand it.

Let's look at the basic structure of a Visual Basic .NET program (see directory Ba01):

```
' Ba01.vb - A sample Visual Basic .NET program
'
REM In case 1 below, the text "Welcome to Visual Basic .NET"
REM is written to the standard output device.
'
' In case 2, the Main procedure returns to its caller.
```

```
Public Module Ba01
   Public Sub Main()       ' program starts here
'1
        System.Console.WriteLine("Welcome to Visual Basic .NET")
'2
        Return             ' redundant
   End Sub
End Module
```

The output produced is

```
Welcome to Visual Basic .NET
```

A program may contain textual comments. A comment begins with a single quote character (`'`) and runs to the end of that physical source line. As such, a comment may occupy a whole physical source line or the trailing part of a such a line. No white space is needed between the quote and the text that follows. An alternate way of starting a comment is to use the keyword REM (an historical abbreviation for "Remark"). For this keyword to be recognized as such, it must be separated from the text of the comment by white space. Throughout this book we will use the single quote form.

Visual Basic .NET 2005 added the capability to put documentation at certain places in source code, in the form of XML. This capability is called *XML Comments*. Such comments begin with three single quote characters (`'''`) , and are not discussed further in this book.

Note the unusual comments, `'1` and `'2`, in the example above. Throughout this book, such comments are used to give source lines pseudo-labels, so they can be referenced directly in the narrative. In production code, this approach can also be used, as follows: A comment prior to a procedure provides a general introduction and describes the steps used to implement the solution. By giving each step a label, we can place a comment containing that label prior to the first statement that implements that step. For example, in the program above, the introductory comment mentions step 1, and the comment `'1` shows where that step is implemented. (Steps are referred to in the narrative as cases.)

A Visual Basic .NET program consists of one or more *procedures* that can be defined in one or more source files. (A procedure corresponds to what some other languages refer to as a method, a function, or a subroutine.) A program must contain at least one procedure, called `Main`.[1] This specially named procedure indicates where the program is to begin execution.

Each procedure can have one or more *modifiers*; `Main` has one, `Public`. Names having this modifier are visible outside their parent module. And since the procedure `Main` needs to be accessed by the program execution machinery—which is outside the module `Ba01`—the `Public` modifier is needed. The keyword Sub preceding the procedure name `Main` indicates that this procedure is a *subroutine*; that is, it does not return a value. (`Main` is permitted to return an integer value to the execution environment. That is, it is permitted to be a *function*. This is discussed in §3.7.)

[1] This is only true for a "Console Application"; a Windows Application need not have such a procedure.

4. References, Strings, and Arrays

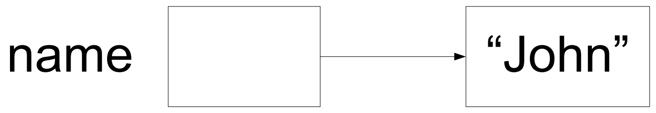
In this chapter, we will learn about reference variables, the allocation of memory at runtime using the class `String`, New, garbage collection, and array allocation and manipulation.

4.1 Introduction

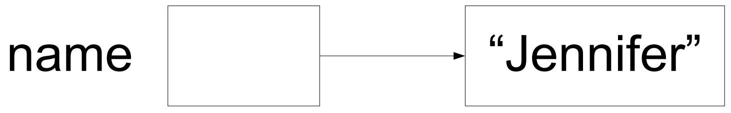
There are two categories of types in Visual Basic .NET: simple and reference. We have already seen examples of the simple types, `Boolean`, `SByte`, `Byte`, `Short`, `UShort`, `Integer`, `UInteger`, `Long`, `ULong`, `Decimal`, `Char`, `Single`, `Double`, `Date`, and Enum types.

In the definition `Dim i as Integer`, `i` is a *variable* and the name `i` designates some specific storage location in memory. We say that the value of `i` is some `Integer`. Throughout its life, that variable is always associated with the same location. (This may seem obvious based on your experience with other languages, but it's most important to understand when we look at reference types.) To help us understand reference types, let's look at the following example (see directory Rf01):

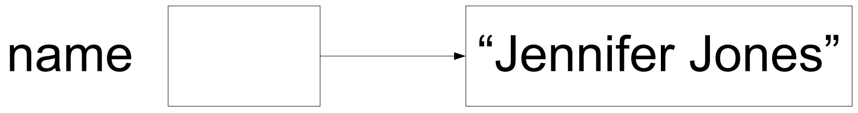
```
Public Module Rf01
    Public Sub Main()
'1
          Dim name As String = "John"
'2
          Console.WriteLine("name is >{0}<", name)
```



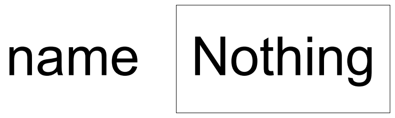
```
'3
          name = "Jennifer"
          Console.WriteLine("name is >{0}<", name)
```



```
'4
          name = "Jennifer" + " Jones"
          Console.WriteLine("name is >{0}<", name)
```



```
'5
         name = Nothing
```



```
'6
         Console.WriteLine("name is >{0}<", name)
   End Sub
End Module
```

The output produced is:

```
name is >John<
name is >Jennifer<
name is >Jennifer Jones<
name is ><
```

The keyword `String` is nothing more than an alias for the predefined library class `String.String`. Writing the keyword `String` is exactly the same as writing `System.String`, and vice versa. An object type is often called a *class type*. Unlike some languages, a string is not an array of `Byte` nor is it an array of `Char`; it is a sequence of Unicode characters. **The contents of an object of type `String` cannot be modified.**[1]

In case 1, we define a variable called `name`. At first glance we might think that `name` is a variable of type `String` and that it contains the value `"John"` directly; however, that is not the case. Instead, `name` is a reference to that string; that is, `name` does not actually contain that data, it simply points to that data, which resides elsewhere. (In the case of a string literal, the data representing that string really has no name.) We say that `name` is a *reference variable* and that its type is "reference to `String`".

In case 3, `name` is made to refer to a different string. The important thing to understand here is that `name` "has been made to point" to a different string; the value of the string `"Jennifer"` has *not* been copied anywhere.

In case 4, the two strings are concatenated resulting in a third string to which `name` is made to point. Because a string's value is read-only, the second string cannot simply be appended to the end of the first one.

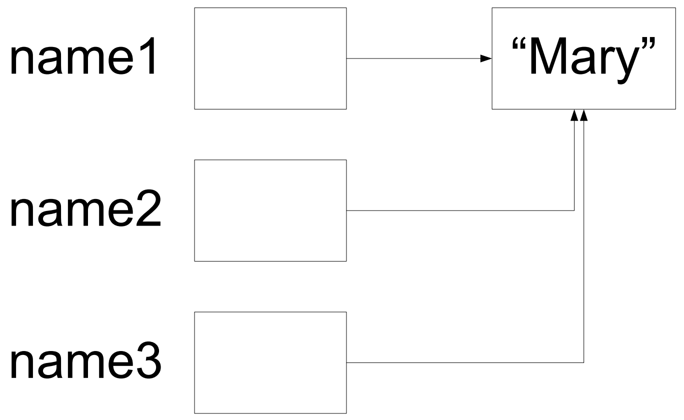
A simple variable can only ever contain a value of its type. However, a reference variable can hold either a reference to an object that is assignment compatible with the type of that variable, or the *null reference*. Case 5 introduces the name `Nothing`. By initializing a reference with the value `Nothing`, we make it point "nowhere". The output produced by case 6 shows us that when the value of a reference containing `Nothing` is used in string concatenation, an empty string results.

The default initial value of String variables is the null value `Nothing`. However, when a null-valued String reference participates in string concatenation, it is treated as an empty string.

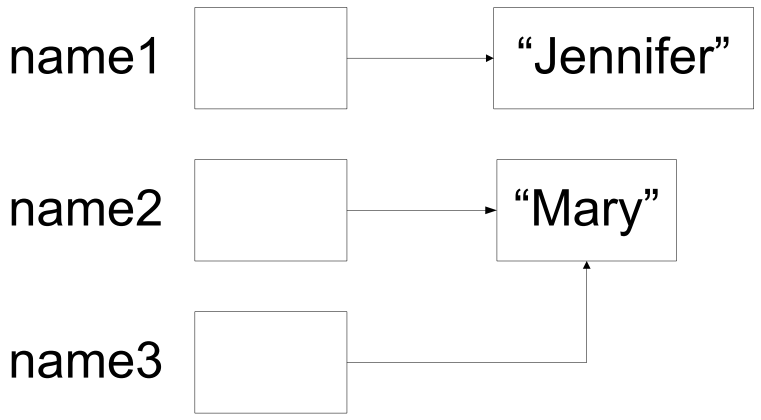
[1] The library class StringBuilder is similar to `String` except that objects of type `StringBuilder` can have their contents changed. See §4.8

There can be any number of references to the same object. For example (see directory Rf02):

```
Public Module Rf02
   Public Sub Main()
         Dim name1 As String = "Mary"
'1
         Dim name2 As String = name1
'2
         Dim name3 As String = name2
```



```
         Console.WriteLine("name1 is {0}", name1)
         Console.WriteLine("name2 is {0}", name2)
         Console.WriteLine("name3 is {0}", name3)
'3
         name1 = "Jennifer"
```



```
         Console.WriteLine(vbCrLf + "name1 is {0}", name1)
         Console.WriteLine("name2 is {0}", name2)
         Console.WriteLine("name3 is {0}", name3)
   End Sub
End Module
```

The output produced is:

5. Classes

Classes are the key foundation stone of object-oriented programming. By using classes, we can take advantage of encapsulation, data hiding, inheritance, and polymorphism, the first two of which are discussed in this chapter.

5.1 Introduction

So far, we've been creating variables of simple types, arrays of those types, and strings. And while we can do useful programming with these types, eventually we'll need more complex and sophisticated data types. For example, in a personnel system, we might want an Employee type, containing employee name, address, date of birth, veteran status, and other descriptive information. In a library catalog system, we'll probably want to keep track of each book's call number, author, title, year of publication, and the like, in some kind of a Publication type. Neither of these types can be accommodated directly by the types we have seen.

Throughout this and future sections, we'll use `Point` as our user-defined type. A Point represents a location in a two-dimensional plane. This simple type is something with which we can easily relate and it serves nicely to introduce the class support machinery. Even if you don't write graphics programs, it should be very easy to extrapolate from the principles learned here.

Let's define our new Point type (see directory Point in directory Cl01):

```
Public Class Point
    ' instance variables

'1
    Public xorigin As Integer
'2
    Public yorigin As Integer
End Class
```

Instead of using `Module`, we are using `Class`. Back in §1.11, we learned that variables defined inside a module (rather than inside a procedure) are module variables; that is, they are variables belonging to the module as a whole. Classes can also contain variables. A class can also have variables that belong to the class as a whole, provided they have the modifier `Shared`. Class variables not having this modifier are *instance variables*; `xorigin` and `yorigin` are examples. (Together, class and instance variables are known as fields.) We can create an arbitrary number of Point objects, each of which contains an x- and y-coordinate pair; that is, each instance of class Point contains a unique pair of instance variables, since each Point's representation is separate from those of all other Points, even for Points that happen to have the same coordinate values.

Note that when a variable definition contains an access keyword (such as `Public`, `Private`, or `Friend`), the `Dim` keyword can be omitted.

The following program (see directory Cl01) creates and manipulates several Points:

```
Public Module Cl01
   Public Sub Main()
'3
        Dim p1 As Point = New Point()
'4
        Console.WriteLine("p1 = ({0,{1)", p1.xorigin, p1.yorigin)
'5
        p1.xorigin = 5
'6
        p1.yorigin = 7
        Console.WriteLine("p1 = ({0},{1})", p1.xorigin, p1.yorigin)
'7
        Dim p2 As New Point()
'8
        p2.xorigin += -4
'9
        p2.yorigin += 12
        Console.WriteLine("p2 = ({0},{1})", p2.xorigin, p2.yorigin)
   End Sub
End Module
```

In case 3, we allocate memory for a Point using New, just like we did for strings in §4.1. Creating an instance of a class is known as *instantiation*. By default, all instance variables take on a zero, false, or null value, depending on their type. Like strings and Points, all objects of user-defined class types must be allocated on the heap using New. To access the instance variables, we simply prefix their names with the name of their parent, as in case 4. Note that this is different to the way in which we have been accessing module (or class) variables, which use the parent module (or class) name as their prefix. A class can have both class and instance variables, as we'll see later; in fact, many of the library classes do.

It is useful to be able to change the coordinates of a Point after it has been created. This operation is often referred to as *moving* the Point, and is done in cases 5 and 6. In case 7, we define a second Point and we *translate* that Point in cases 8 and 9. (Translation involves moving by an offset rather than to an absolute new location.)

Note that case 7 uses a short-hand equivalent to case 3 with respect to the approach for memory allocation. Throughout this book, the abbreviated approach will be used.

The output produced by this program is:

```
p1 = (0,0)
p1 = (5,7)
p2 = (-4,12)
```

When we have a general-purpose class such as `Point`, we very quickly realize that it makes no sense to define our application program as a procedure within that class. Afterall, we likely will write many programs that use this class and we can't define all our programs inside that class. In any event, we can only have one procedure called

`Main` having some given signature. As a result, not only do we need to define our application and Point in different classes, these classes need to be in separate source files and assemblies.

5.2 Class-Specific Procedures

It is tedious to read and write the steps to move, translate, and display a Point each time. Also, every application relies on the fact that a Point contains an x- and a y-coordinate of type `Integer`, called `xorigin` and `yorigin`, respectively. Therefore, any change in the way that type is represented will negatively affect those applications. By making the instance variables private and adding some public procedures to class `Point`, we can deal with it in a more abstract manner. Now the Point class looks like the following (See directory Point in directory Cl02):

```
Public Class Point

   ' instance variables

   Private xorigin As Integer
   Private yorigin As Integer
```

By making the instance variables `Private`, they can only be accessed by procedures within their parent class. This is known as *data hiding*, since we hide the representation details of `Point` from all programs that use it. The main reason for data hiding is to cater for unanticipated changes. Since we can never say the representation of an object won't change and we can never say just exactly how it might or will change, we should cater for as much flexibility as possible. That is, assume everything within reason will change.

```
   ' constructors

   Public Sub New(ByVal xPos As Integer, ByVal yPos As Integer)
        xorigin = xPos
        yorigin = yPos
   End Sub

   Public Sub New()
        xorigin = 0
        yorigin = 0
   End Sub
```

In the program above, we also see an example of *encapsulation*, the process by which we associate variables and procedures by defining them in the same class. This allows us to control the access to those variables and procedures.

The subroutines called New are special. Strictly speaking, they are not really subroutines, but rather, constructors; however, for all practical purposes they look like subroutines—they just have a special name. From this we can deduce that, syntactically, a *constructor* is nothing more than a subroutine having the special name New. The first constructor expects two `Integer` arguments which represent the x- and y-coordinates, respectively, and it initializes the private fields using those values. A constructor can contain one or more `Return` statements, provided they do not contain an expression.