Introduction

In this preliminary chapter, we introduce a couple of topics we'll be using throughout the book. First, we discuss how to use classes and object-oriented programming (OOP) to aid in the development of data structures and algorithms. Using OOP techniques will make our algorithms and data structures more general and easier to modify, not to mention easier to understand.

The second part of this Introduction familiarizes the reader with techniques for performing timing tests on data structures and, most importantly, the different algorithms examined in this book. Running timing tests (also called benchmarking) is notoriously difficult to get exactly right, and in the .NET environment, it is even more complex than in other environments. We develop a Timing class that makes it easy to test the efficiency of an algorithm (or a data structure when appropriate) without obscuring the code for the algorithm or data structures.

Developing Classes

This section provides the reader with a quick overview of developing classes in VB.NET. The rationale for using classes and for OOP in general is not discussed here. For a more thorough discussion of OOP in VB.NET, see McMillan (2004).

One of the primary uses of OOP is to develop user-defined data types. To aid our discussion, and to illustrate some of the fundamental principles of OOP,

we will develop two classes for describing one or two features of a geometric data processing system: the Point class and the Circle class.

Data Members and Constructors

The data defined in a class, generally, are meant to stay hidden within the class definition. This is part of the principle of encapsulation. The data stored in a class are called data members, or alternatively, fields. To keep the data in a class hidden, data members are usually declared with the Private access modifier. Data declared like this cannot be accessed by user code.

The Point class will store two pieces of data—the *x* coordinate and the *y* coordinate. Here are the declarations for these data members:

```
Public Class Point

  Private x As Integer
  Private y As Integer

  'More stuff goes here'

End Class
```

When a new class object is declared, a constructor method should be called to perform any initialization that is necessary. Constructors in VB.NET are named New by default, unlike in other languages where constructor methods are named the same as the class.

Constructors can be written with or without arguments. A constructor with no arguments is called the *default* constructor. A constructor with arguments is called a parameterized constructor. Here are examples of each for the Point class:

```
Public Sub New()
  x = 0
  y = 0
End Sub

Public Sub New(ByVal xcor As Integer, ByVal ycor As _
               Integer)
  x = xcor
  y = ycor
End Sub
```

Property Methods

After the data member values are initialized, the next set of operations we need to write involves methods for setting and retrieving values from the data members. In VB.NET, these methods are usually written as *Property methods*.

A Property method provides the ability to both set and retrieve the value of a data member within the same method definition. This is accomplished by utilizing a Get clause and a Set clause. Here are the property methods for getting and setting *x*-coordinate and *y*-coordinate values in the Point class:

```
Public Property Xval() As Integer
  Get
    Return x
  End Get
  Set(ByVal Value As Integer)
    x = Value
  End Set
End Property

Public Property Yval() As Integer
  Get
    Return y
  End Get
  Set(ByVal Value As Integer)
    y = Value
  End Set
End Property
```

When you create a Property method using Visual Studio.NET, the editor provides a template for the method definition like this:

```
Public Property Xval() As Integer
  Get

  End Get
  Set(ByVal Value As Integer)

  End Set
End Property
```

Other Methods

Of course, constructor methods and Property methods aren't the only methods we will need in a class definition. Just what methods you'll need depend on the application. One method included in all well-defined classes is a ToString method, which returns the current state of an object by building a string that consists of the data member's values. Here's the ToString method for the Point class:

```
Public Overrides Function ToString() As String
  Return x & "," & y
End Function
```

Notice that the ToString method includes the modifier Overrides. This modifier is necessary because all classes inherit from the Object class and this class already has a ToString method. For the compiler to keep the methods straight, the Overrides modifier indicates that, when the compiler is working with a Point object, it should use the Point class definition of ToString and not the Object class definition.

One additional method many classes include is one to test whether two objects of the same class are equal. Here is the Point class method to test for equality:

```
Public Function Equal(ByVal p As Point) As Boolean
  If (Me.x = p.x) And (Me.y = p.y) Then
    Return True
  Else
    Return False
  End If
End Function
```

Methods don't have to be written as functions; they can also be subroutines, as we saw with the constructor methods.

Inheritance and Composition

The ability to use an existing class as the basis for one or more new classes is one of the most powerful features of OOP. There are two major ways to

use an existing class in the definition of a new class: 1. The new class can be considered a subclass of the existing class (*inheritance*); and 2. the new class can be considered as at least partially made up of parts of an existing class (*composition*).

For example, we can make a Circle class using a Point class object to determine the center of the circle. Since all the methods of the Point class are already defined, we can reuse the code by declaring the Circle class to be a *derived class* of the Point class, which is called the *base class*. A derived class inherits all the code in the base class plus it can create its own definitions.

The Circle class includes both the definition of a point (*x* and *y* coordinates) as well as other data members and methods that define a circle (such as the radius and the area). Here is the definition of the Circle class:

```
Public Class Circle
  Inherits Point

  Private radius As Single

  Private Sub setRadius(ByVal r As Single)
    If (r > 0) Then
      radius = r
    Else
      radius = 0.0
    End If
  End Sub

  Public Sub New(ByVal r As Single, ByVal x As _
                 Integer, ByVal y As Integer)
    MyBase.New(x, y)
    setRadius(r)
  End Sub

  Public Sub New()
    setRadius(0)
  End Sub

  Public ReadOnly Property getRadius() As Single
    Get
      Return radius
    End Get
  End Property
```

```
    Public Function Area() As Single
      Return Math.PI * radius * radius
    End Function

    Public Overrides Function ToString() As String
      Return "Center = " & Me.Xval & "," & Me.Yval & _
             " - radius = " & radius
    End Function

End Class
```

There are a couple of features in this definition you haven't seen before. First, the parameterized constructor call includes the following line:

```
MyBase.New(x,y)
```

This is a call to the constructor for the base class (the Point class) that matches the parameter list. Every derived class constructor must include a call to one of the base classes' constructors.

The Property method getRadius is declared as a ReadOnly property. This means that it only retrieves a value and cannot be used to set a data member's value. When you use the ReadOnly modifer, Visual Studio.NET only provides you with the Get part of the method.

Timing Tests

Because this book takes a practical approach to the analysis of the data structures and algorithms examined, we eschew the use of Big O analysis, preferring instead to run simple benchmark tests that will tell us how long in seconds (or whatever time unit) it takes for a code segment to run.

Our benchmarks will be timing tests that measure the amount of time it takes an algorithm to run to completion. Benchmarking is as much of an art as a science and you have to be careful how you time a code segment to get an accurate analysis. Let's examine this in more detail.

An Oversimplified Timing Test

First, we need some code to time. For simplicity's sake, we will time a subroutine that writes the contents of an array to the console. Here's the

code:

```
Sub DisplayNums(ByVal arr() As Integer)
  Dim index As Integer
  For index = 0 To arr.GetUpperBound(0)
    Console.Write(arr(index))
  Next
End Sub
```

The array is initialized in another part of the program, which we'll examine later.

To time this subroutine, we need to create a variable that is assigned the system time just as the subroutine is called, and we need a variable to store the time when the subroutine returns. Here's how we wrote this code:

```
Dim startTime As DateTime
Dim endTime As TimeSpan
startTime = DateTime.Now
DisplayNums(nums)
endTime = DateTime.Now.Subtract(startTime)
```

Running this code on a laptop (running at 1.4 MHz on Windows XP Professional) takes about 5 seconds (4.9917 seconds to be exact). Whereas this code segment seems reasonable for performing a timing test, it is completely inadequate for timing code running in the .NET environment. Why?

First, this code measures the elapsed time from when the subroutine was called until the subroutine returns to the main program. The time used by other processes running at the same time as the VB.NET program adds to the time being measured by the test.

Second, the timing code used here doesn't take into account garbage collection performed in the .NET environment. In a runtime environment such as .NET, the system can pause at any time to perform garbage collection. The sample timing code does nothing to acknowledge garbage collection and the resulting time can be affected quite easily by garbage collection. So what do we do about this?

Timing Tests for the .NET Environment

In the .NET environment, we need to take into account the thread in which our program is running and the fact that garbage collection can occur

at any time. We need to design our timing code to take these facts into consideration.

Let's start by looking at how to handle garbage collection. First, let's discuss what garbage collection is used for. In VB.NET, reference types (such as strings, arrays, and class instance objects) are allocated memory on something called the *heap*. The heap is an area of memory reserved for data items (the types previously mentioned). Value types, such as normal variables, are stored on the *stack*. References to reference data are also stored on the stack, but the actual data stored in a reference type are stored on the heap.

Variables that are stored on the stack are freed when the subprogram in which the variables are declared completes its execution. Variables stored on the heap, in contrast, are held on the heap until the garbage collection process is called. Heap data are only removed via garbage collection when there is not an active reference to those data.

Garbage collection can, and will, occur at arbitrary times during the execution of a program. However, we want to be as sure as we can that the garbage collector is not run while the code we are timing is executing. We can head off arbitrary garbage collection by calling the garbage collector explicitly. The .NET environment provides a special object for making garbage collection calls, GC. To tell the system to perform garbage collection, we simply write the following:

```
GC.Collect()
```

That's not all we have to do, though. Every object stored on the heap has a special method called a finalizer. The finalizer method is executed as the last step before deleting the object. The problem with finalizer methods is that they are not run in a systematic way. In fact, you can't even be sure an object's finalizer method will run at all, but we know that before we can be certain an object is deleted, its finalizer method must execute. To ensure this, we add a line of code that tells the program to wait until all the finalizer methods of the objects on the heap have run before continuing. The line of code is as follows:

```
GC.WaitForPendingFinalizers()
```

We have cleared one hurdle but one remains: using the proper thread. In the .NET environment, a program is run inside a process, also called an *application domain*. This allows the operating system to separate each different program running on it at the same time. Within a process, a program or a part of a

program is run inside a *thread*. Execution time for a program is allocated by the operating system via threads. When we are timing the code for a program, we want to make sure that we're timing just the code inside the process allocated for our program and not other tasks being performed by the operating system.

We can do this by using the Process class in the .NET Framework. The Process class has methods for allowing us to pick the current process (the process in which our program is running), the thread in which the program is running, and a timer to store the time the thread starts executing. Each of these methods can be combined into one call, which assigns its return value to a variable to store the starting time (a TimeSpan object). Here's the code:

```
Dim startingTime As TimeSpan
  startingTime = Process.GetCurrentProcess.Threads(0). _
                 UserProcessorTime
```

All we have left to do is capture the time when the code segment we're timing stops. Here's how it's done:

```
duration = Process.GetCurrentProcess.Threads(0). _
           UserProcessorTime.Subtract(startingTime)
```

Now let's combine all this into one program that times the same code we tested earlier:

```
Module Module1

  Sub Main()

    Dim nums(99999) As Integer
    BuildArray(nums)
    Dim startTime As TimeSpan
    Dim duration As TimeSpan
    startTime = Process.GetCurrentProcess.Threads(0). _
                UserProcessorTime
    DisplayNums(nums)
    duration = Process.GetCurrentProcess.Threads(0). _
               UserProcessorTime.Subtract (startTime)
    Console.WriteLine("Time: " & duration.TotalSeconds)

  End Sub
```

```
    Sub BuildArray(ByVal arr() As Integer)

      Dim index As Integer
      For index = 0 To 99999
        arr(index) = index
      Next
    End Sub

End Module
```

Using the new-and-improved timing code, the program returns in just 0.2526 seconds. This compares with the approximately 5 seconds return time using the first timing code. Clearly, a major discrepancy between these two timing techniques exists and you should use the .NET techniques when timing code in the .NET environment.

A Timing Test Class

Although we don't need a class to run our timing code, it makes sense to rewrite the code as a class, primarily because we'll keep our code clear if we can reduce the number of lines in the code we test.

A Timing class needs the following data members:

- startingTime—to store the starting time of the code we are testing,
- duration—the ending time of the code we are testing,

The starting time and the duration members store times and we chose to use the TimeSpan data type for these data members. We'll use just one constructor method, a default constructor that sets both the data members to 0.

We'll need methods for telling a Timing object when to start timing code and when to stop timing. We also need a method for returning the data stored in the duration data member.

As you can see, the Timing class is quite small, needing just a few methods. Here's the definition:

```
Public Class Timing

  Private startingTime As TimeSpan
  Private duration As TimeSpan

  Public Sub New()
```