Mastering the Visual Basic Language: Procedures, Error Handling, Classes, and Objects

Today, we’re going to look at some crucial aspects of the Visual Basic language: procedures such as Sub procedures and functions, procedure scope, and exception (runtime error) handling. We’ll also get an introduction to a topic that’s become central to Visual Basic: classes and objects.

Now that our code is growing larger, it’s good to know about procedures, which allow us to break up our code into manageable chunks. In fact, in Visual Basic, all executable code must be in procedures. There are two types of procedures: Sub procedures and functions. In Visual Basic, Sub procedures do not return values when they terminate, but functions do.
If you declare variables in your new procedures, those variables might not be accessible from outside the procedure, and that fact is new also. The area of your program in which a data item is visible and can be accessed in code is called scope, and we’ll try to understand scope—a crucial aspect of object-oriented programming—in this chapter.

We’ll also look at handling runtime errors today. In Visual Basic, a runtime error is the same as an exception (that’s not true in all languages), so we’re going to look at exception handling. We’ll see that there are two ways of heading off errors that happen at runtime before they become problems.

Finally, we’ll get an introduction to classes and objects in this chapter. Visual Basic .NET programming is object-oriented programming (OOP), a fact you need to understand in depth to be a Visual Basic programmer. Today, we’ll start by discussing classes and objects in preparation for our later work (such as Day 9, “Object-Oriented Programming,” which is all about OOP). Here’s an overview of today’s topics:

- Creating Sub procedures and functions
- Passing arguments to procedures
- Returning data from functions
- Preserving data values between procedure calls
- Understanding scope
- Using unstructured exception handling
- Using structured exception handling with Try/Catch
- Using exception filtering in Catch blocks
- Using multiple Catch statements
- Throwing an exception
- Throwing a custom exception
- Understanding classes and objects
- Supporting properties and methods in objects

All these topics are powerful ones, and they’re all related. And today, the best place to start is with Sub procedures.

**Sub Procedures**

Procedures give you a way to break up your Visual Basic code, which is invaluable as that code grows longer and longer. Ideally, each procedure should handle one discrete
task. That way, you break up your code by task; having one task per procedure makes it
easier to keep in mind what each procedure does.

You can place a set of Visual Basic statements in a procedure, and when that procedure is
called, those statements will be run. You can pass data to procedures for that code to
work on and read that data in your code. The two types of procedures in Visual Basic are
Sub procedures and functions, and both can read the data you pass them (the name Sub
procedure comes from the programming term subroutine). However, only one type, func-
tions, can also return data.

In fact, we’ve been creating Sub procedures in our code already (not surprisingly,
because all Visual Basic code has to be in a procedure). All the code we developed yes-
terday went into the Sub procedure named Main, created with the keyword Sub:

Module Module1
    Sub Main()
        Console.WriteLine("Hello there!")
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
    End Sub
End Module

This Main Sub procedure is special because when a console application starts, Visual
Basic calls Main automatically to start the program. When Main is called, the code is run
as we wanted.

You can also create your own Sub procedures, giving them your own names. Those
names should give an indication of the procedure’s task. For example, to show the “Hi
there!” message, you might create a new Sub procedure named ShowMessage by simply
typing this text into the code designer:

Module Module1
    Sub Main()
        End Sub
    End Module

    Sub ShowMessage()
        End Sub
End Module
In the `ShowMessage` Sub procedure, you place the code you want to execute, like this code to display the message:

```vbnet
Module Module1

Sub Main()
    End Sub

Sub ShowMessage()
    Console.WriteLine("Hi there!")
End Sub

End Module
```

How do you make the code in the `ShowMessage` Sub procedure run? You can do that by calling it; to do so, just insert its name, followed by parentheses, into your code:

```vbnet
Module Module1

Sub Main()
    ShowMessage()  
    Console.WriteLine("Press Enter to continue...")
    Console.ReadLine()
    End Sub

Sub ShowMessage()
    Console.WriteLine("Hi there!")
End Sub

End Module
```

And that's it! Now when you run this code, Visual Basic will call the `Main` Sub procedure, which in turn will call the `ShowMessage` Sub procedure, giving you the same result as before:

```
Hi there!
Press Enter to continue...
```

If you want to, you can use a Visual Basic `Call` statement to call a Sub procedure like this: `Call ShowMessage()`. This usage is still supported, although it goes back to the earliest days of Visual Basic, and there's no real reason to use it here.

Note the parentheses at the end of the call to `ShowMessage` like this: `ShowMessage()`. You use those parentheses to pass data to a procedure, and we’ll take a look at that task next.
Passing Data to Procedures

Say you want to pass the message text you want to display to the ShowMessage Sub procedure, allowing you to display whatever message you want. You can do that by passing a text string to ShowMessage, like this:

```vbnet
Module Module1
    Sub Main()
        ShowMessage("Hi there!")
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
    End Sub

    Sub ShowMessage()
    End Sub
End Module
```

A data item you pass to a procedure in parentheses this way is called an argument. Now in ShowMessage, you must declare the type of the argument passed to this procedure in the procedure’s argument list:

```vbnet
Module Module1
    Sub Main()
        ShowMessage("Hi there!")
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
    End Sub

    Sub ShowMessage(ByVal Text As String)
    End Sub
End Module
```

This creates a new string variable, `Text`, which you’ll be able to access in the procedure’s code. The `ByVal` keyword here indicates that the data is being passed by value, which is the default in Visual Basic (you don’t even have to type `ByVal`, just `Text As String` here, and Visual Basic will add `ByVal` automatically).

Passing data by value means a copy of the data will be passed to the procedure. The other way of passing data is by reference, where you use the `ByRef` keyword. Passing by reference (which was the default in VB6) meant that the location of the data in memory will be passed to the procedure. Here’s an important point to know: Because...
objects can become very large in Visual Basic, making a copy of an object and passing that copy can be very wasteful of memory, so objects are automatically passed by reference. We’ll discuss passing by value and passing by reference in more detail in a page or two.

Visual Basic automatically fills the Text variable you declared in the argument list in this example with the string data passed to the procedure. This means you can access that data as you would the data in any other variable, as you see in the SubProcedures project in the code for this book, as shown in Listing 3.1.

**Listing 3.1 Passing Data to a Sub Procedure (SubProcedures project, Module1.vb)**

```vbnet
Module Module1

Sub Main()
    ShowMessage("Hi there!")
    Console.WriteLine("Press Enter to continue...")
    Console.ReadLine()
End Sub

Sub ShowMessage(ByVal Text As String)
    Console.WriteLine(Text)
End Sub

End Module
```

And that’s all you need! Now you’re passing data to Sub procedures and retrieving that data in the procedure’s code. You can pass more than one argument to procedures as long as you declare each argument in the procedure’s argument list. For example, say you want to pass the string to show and the number of times to show it to ShowMessage; that code might look like this:

```vbnet
Module Module1

Sub Main()
    ShowMessage("Hi there!", 3)
    Console.WriteLine("Press Enter to continue...")
    Console.ReadLine()
End Sub

Sub ShowMessage(ByVal Text As String, ByVal Times As Integer)
    For intLoopIndex As Integer = 1 To Times
```

Procedures, Error Handling, Classes, and Objects

Here's the result of this code:

Hi there!
Hi there!
Hi there!
Press Enter to continue...

If you pass arguments by reference, using the ByRef keyword, Visual Basic passes the memory location of the passed data to the procedure (which gives the code in that procedure access to that data). You can read that data just as you do when you pass arguments by value:

```vbnet
Sub ShowMessage(ByRef Text As String, ByRef Times As Integer)
    For intLoopIndex As Integer = 1 To Times
        Console.WriteLine(Text)
    Next intLoopIndex
End Sub
```

The code in the procedure has access to the data's location in memory, however, and that's something to keep in mind. So far, we've passed two literals ("Hello there!" and 3) to ShowMessage, and literals don't correspond to memory locations. But see what happens if you pass a variable by reference, like this:

```vbnet
Dim NumberOfTimes As Integer = 3
ShowMessage("Hi there!", NumberOfTimes)
```

The code in the procedure has access to that variable, and if you change the value of the passed argument, you'll also change the value in the original variable:

```vbnet
Dim NumberOfTimes As Integer = 3
ShowMessage("Hi there!", NumberOfTimes)
```
Sub ShowMessage(ByRef Text As String, ByRef Times As Integer)
    For intLoopIndex As Integer = 1 To Times
        Console.WriteLine(Text)
    Next intLoopIndex
    Times = 24
End Sub

After this code is finished executing, for example, the variable NumberOfTimes will be
left holding 24. This side effect is not unintentional; it’s intentional. Being able to change
the value of arguments is a primary reason to pass arguments by reference.

Changing the value of arguments passed by reference is one way to pass data from a pro-
cedure back to the calling code, but it can be troublesome. You can easily change an
argument’s value unintentionally, for example. A more structured way of passing data
back from procedures is to use functions, which is the next topic.

You should also know that an Exit Sub statement, if you use one, causes an immediate
exit from a Sub procedure in case you want to leave before executing all code. For exam-
ple, say you have a Sub procedure that displays reciprocals of numbers you pass to it, but
you want to avoid trying to find the reciprocal of 0. You could display an error message
and exit the procedure like this if 0 is passed to the procedure:

Sub Reciprocal(ByVal dblNumber As Double)
    If dblNumber = 0 Then
        Console.WriteLine("Cannot find the reciprocal of 0.")
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
    Exit Sub
End If
    Console.WriteLine("The reciprocal is " & 1 / dblNumber)
    Console.WriteLine("Press Enter to continue...")
    Console.ReadLine()
End Sub

Sub Procedure Syntax
Like other Visual Basic statements, Sub procedures require a formal declaration. You
declare Sub procedures with the Sub statement:

[ <attrlist> ] [{ Overloads | Overrides | Overridable | NotOverridable | MustOverride | Shadows | Shared }]
[{ Public | Protected | Friend | Protected Friend | Private }]
Sub name [(arglist)]
[ Implements interface.definedname ]
[ statements ]
Procedures, Error Handling, Classes, and Objects

[ Exit Sub ]
[ statements ]
End Sub

And like other Visual Basic statements, many of the keywords here won’t make sense at this point, so you can treat this information as reference material to come back to later. (Many of the keywords here deal with OOP, but we can’t cover OOP in the detail needed here before knowing how to work with procedures, so it’s impossible to avoid slightly circular definitions.) The parts of this statement are as follows:

- **attrlist**—This is an advanced (and optional) topic; this is a list of attributes for use with this procedure. Attributes can add more information about the procedure, such as copyright data and so on. You separate multiple attributes with commas.

- **Overloads**—Specifies that this Sub procedure overloads one (or more) procedures defined with the same name in a base class. An overloaded procedure has multiple versions, each with a different argument list, as we’ll see in Day 9. The argument list must be different from the argument list of every procedure that is to be overloaded. You cannot specify both Overloads and Shadows in the same procedure declaration.

- **Overrides**—Specifies that this Sub procedure overrides (replaces) a procedure with the same name in a base class. The number and data types of the arguments must match those of the procedure in the base class.

- **Overridable**—Specifies that this Sub procedure can be overridden by a procedure with the same name in a derived class.

- **NotOverridable**—Specifies that this Sub procedure may not be overridden in a derived class.

- **MustOverride**—Specifies that this Sub procedure is not implemented. This procedure must be implemented in a derived class.

- **Shadows**—Makes this Sub procedure a shadow of an identically named programming element in a base class. You can use Shadows only at module, namespace, or file level (but not inside a procedure). You cannot specify both Overloads and Shadows in the same procedure declaration.

- **Shared**—Specifies that this Sub procedure is a shared procedure. As a shared procedure, it is not associated with a specific object, and you can call it using the class or structure name.

- **Public**—Procedures declared Public have public access. There are no restrictions on the accessibility of public procedures.
• **Protected**—Procedures declared **Protected** have protected access. They are accessible only from within their own class or from a derived class. You can specify **Protected** access only for members of classes.

• **Friend**—Procedures declared **Friend** have friend access. They are accessible from within the program that contains their declaration and from anywhere else in the same assembly.

• **Protected Friend**—Procedures declared **Protected Friend** have both protected and friend accessibility. They can be used by code in the same assembly, as well as by code in derived classes.

• **Private**—Procedures declared **Private** have private access. They are accessible only within the element in which they’re declared.

• **name**—Specifies the name of the Sub procedure.

• **arglist**—Lists expressions representing arguments that are passed to the Sub procedure when it is called. You separate multiple arguments with commas.

• **Implements** *interface.definedname*—Indicates that this Sub procedure implements an interface. We’ll see interfaces, which allow you to derive one class from several others, in Day 9.

• **statements**—Specifies the block of statements to be executed within the Sub procedure.

In addition, each argument in the argument list, **arglist**, has this syntax:

```plaintext
[ <attrlist> ] [ Optional ] [{ ByVal | ByRef }] [ ParamArray ] argname[() ] [ As argtype ] [ = defaultvalue ]
```

Here are the parts of **arglist**:

• **attrlist**—Lists (optional) attributes that apply to this argument. Multiple attributes are separated by commas.

• **Optional**—Specifies that this argument is not required when the procedure is called. If you use this keyword, all following arguments in **arglist** must also be optional and be declared using the **Optional** keyword. Every optional argument declaration must supply a **defaultvalue**. **Optional** cannot be used for any argument if you also use **ParamArray**.

• **ByVal**—Specifies passing by value. **ByVal** is the default in Visual Basic.

• **ByRef**—Specifies passing by reference, which means the procedure code can modify the value of the original variable in the calling code.
• ParamArray—Acts as the last argument in arglist to indicate that the final argument is an optional array of elements of the specified type. The ParamArray keyword allows you to pass an arbitrary number of arguments to the procedure. ParamArray arguments are always passed by value.

• argname—Specifies the name of the variable representing the argument.

• argtype—Specifies the data type of the argument passed to the procedure; this part is optional unless Option Strict is set to On. It can be Boolean, Byte, Char, Date, Decimal, Double, Integer, Long, Object, Short, Single, or String, or the name of an enumeration, structure, class, or interface.

• defaultvalue—Specifies the default value for an optional argument, required for all optional arguments. It can be any constant or constant expression that evaluates to the data type of the argument. Note that if the type is Object, or a class, interface, array, or structure, the default value must be Nothing.

That gives us what we need to know about Sub procedures, we’ll move on to functions next.

Creating Functions

You can also create functions in Visual Basic .NET. They are just like Sub procedures except that they can return a value. You declare a function in much the same way as a Sub procedure, except that you use the Function keyword instead of Sub.

Let’s look at an example. In this case, we’ll create a function named Summer that calculates the sum of two integers and returns that sum; this project is named Functions in the code for the book. To create this new function, you use the Function keyword:

```vbscript
Module Module1
    Sub Main()
    End Sub

    Function Summer(ByVal int1 As Integer, ByVal int2 As Integer) As Long
    End Function
End Module
```

This example looks just like creating a Sub procedure, except for the Function keyword and the As Long part at the end. The As Long part is there to indicate that this function
returns a Long value. (Long is a better return value choice than Integer here because two integers could be passed to Summer such that their sum exceeds the capacity of the Integer data type.)

Tip

It's worth realizing that function names can use the same type prefixes that variable names use to indicate return type, such as intSummer.

You return a value from a function with the Return statement, as here, where the code is returning the sum of the two arguments passed to the function:

```vbnet
Module Module1
    Sub Main()
        Dim intItem1 As Integer = 2
        Dim intItem2 As Integer = 3
        Console.WriteLine(intItem1 & " + " & intItem2 & " = " & Summer(intItem1, intItem2))
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
    End Sub

    Function Summer(ByVal int1 As Integer, ByVal int2 As Integer) As Long
        Return int1 + int2
    End Function
End Module
```

Now when you call Summer with two integers, like Summer(2, 3), Visual Basic will treat that function call as an expression and replace it with the value returned by the function, which is 5 here. Listing 3.2 shows how this might look in code.

**Listing 3.2  Returning Data from a Function (Functions project, Module1.vb)**

```vbnet
Module Module1
    Sub Main()
        Dim intItem1 As Integer = 2
        Dim intItem2 As Integer = 3
        Console.WriteLine(intItem1 & " + " & intItem2 & " = " & Summer(intItem1, intItem2))
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
    End Sub

    Function Summer(ByVal int1 As Integer, ByVal int2 As Integer) As Long
        Return int1 + int2
    End Function
End Module
```
When you run this code, you see this result:

2 + 3 = 5
Press Enter to continue...

**Function Syntax**

Here’s the formal syntax for functions; you use the `Function` statement:

```vbc
[ <attrlist> ] [{ Overloads | Overrides | Overridable | NotOverridable | MustOverride | Shadows | Shared }]
[{ Public | Protected | Friend | Protected Friend | Private }]
Function name{(arglist)} [{ As type }]
[ Implements interface.definedname ]
[ statements ]
[ Exit Function ]
[ statements ]
End Function
```

The various parts of this statement are the same as for Sub procedures (see the previous topic) except for the `As type` clause, which specifies the type of the return value from the function. This clause indicates the data type of the value returned by the function. That type can be `Boolean`, `Byte`, `Char`, `Date`, `Decimal`, `Double`, `Integer`, `Long`, `Object`, `Short`, `Single`, or `String`, or the name of an enumeration, structure, class, or interface.

The `Return` statement, if there is one, sets the return value and exits the function; any number of `Return` statements can appear anywhere in the function, but as soon as one of them is executed, you return from the function to the calling code. You can also use the `Exit Function` statement to exit the function at any time. If you use `Exit Function`, how can you return a value from a function? You just assign that value to the function name itself, like this:

```vbc
Function Summer(ByVal int1 As Integer, ByVal int2 As Integer) As Long
    Summer = int1 + int2
    Exit Function
End Function
```

If you use `Exit Function` without setting a return value, the function returns the default value appropriate to `argtype`. That’s 0 for `Byte`, `Char`, `Decimal`, `Double`, `Integer`, `Long`, `Short`, and `Single`; Nothing for `Object`, `String`, and all arrays; False for `Boolean`; and 1/1/0001 12:00 AM for `Date`.
Using Optional Arguments

You can make some arguments in a procedure call *optional*, which means that if the calling code doesn’t specify a value for them, a default value will be used. To make an argument optional, you use the `Optional` keyword and supply a default value like this, where the `strText` argument of the `ShowMessage` function is optional and the default value is "Hello there!":

```
Module Module1
    Sub Main()
    End Sub

    Sub ShowMessage(Optional ByVal strText As String = "Hello there!")
        Console.WriteLine(strText)
    End Sub
End Module
```

Now if you call `ShowMessage` with no arguments, as shown in Listing 3.3 and the Optional project in the code for this book, the message "Hello there!" will be displayed.

**Listing 3.3  Using Optional Arguments (Optional project, Module1.vb)**

```
Module Module1

    Sub Main()
        ShowMessage()
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
    End Sub

    Sub ShowMessage(Optional ByVal strText As String = "Hello there!")
        Console.WriteLine(strText)
    End Sub

End Module
```

Note that if you declare one argument optional in a procedure’s argument list, all following arguments must be optional too (otherwise, Visual Basic wouldn’t know which argument had been omitted).
Passing a Variable Number of Arguments

Here’s another valuable technique: You can create procedures that can accept a varying number of arguments. You do that with the `ParamArray` keyword in the argument list, which makes all the arguments passed at that point in the list and after it part of an array. If you use a `ParamArray` argument, it must be the last argument in the argument list.

Here’s an example; in this case, the `ShowMessage` Sub procedure is able to handle a variable number of arguments:

```vbnet
Module Module1
    Sub Main()
        End Sub
        Sub ShowMessage(ByVal ParamArray Text() As String)
        .
        .
        .
        End Sub
    End Module

This means that the `Text` argument here is really an array of arguments. In this example, we can loop over all the arguments in this array, displaying them like this:

```vbnet
Module Module1
    Sub Main()
        End Sub
        Sub ShowMessage(ByVal ParamArray Text() As String)
            Dim intLoopIndex As Integer
            For intLoopIndex = 0 To UBound(Text)
                Console.Write(Text(intLoopIndex))
            Next intLoopIndex
            Console.WriteLine(""")    'Skip to the next line
        End Sub
    End Module
```

Now you can call `ShowMessage` with different numbers of arguments, as you see in Listing 3.4.
LISTING 3.4  Using Variable Numbers of Arguments (VariableArgs project, Module1.vb)

Module Module1

Sub Main()
    ShowMessage("Hello there!")
    ShowMessage("Hello", " there!")
    Console.WriteLine("Press Enter to continue...")
    Console.ReadLine()
End Sub

Sub ShowMessage(ByVal ParamArray Text() As String)
    Dim intLoopIndex As Integer
    For intLoopIndex = 0 To UBound(Text)
        Console.Write(Text(intLoopIndex))
    Next intLoopIndex
    Console.WriteLine("")
End Sub

End Module

Here’s what you see when this code runs:

Hello there!
Hello there!
Press Enter to continue...

Preserving Data Between Procedure Calls

Suppose that you want to keep track of the number of times you’ve called a procedure. You might write a function like Tracker in this code, which has a variable named intCount that it increments each time you call the function:

Module Module1

Sub Main()
    For intLoopIndex As Integer = 0 To 5
        Console.WriteLine(Tracker())
    Next intLoopIndex
    Console.WriteLine("Press Enter to continue...")
    Console.ReadLine()
End Sub

Function Tracker() As Integer
    Dim intCount As Integer
    intCount += 1
    Return intCount
End Function
Because the code calls Tracker six times and displays its return value each time, you might expect to see this program display 1, 2, 3, 4, 5, and 6. But you actually get this result:
1
1
1
1
1
1
Press Enter to continue...

The problem here is that the intCount variable in Tracker is re-initialized to 0 each time the procedure is called, so the return value, after intCount is incremented, is always 1. The solution is to declare intCount as static, as shown in Listing 3.5 and the Static project in the code for this book.

**Listing 3.5** Using Static Variables (Static project, Module1.vb)

```vbnet
Module Module1

Sub Main()
    For intLoopIndex As Integer = 0 To 5
        Console.WriteLine(Tracker())
    Next intLoopIndex
    Console.WriteLine("Press Enter to continue...")
    Console.ReadLine()
End Sub

Function Tracker() As Integer
    Static intCount As Integer
    intCount += 1
    Return intCount
End Function

End Module
```

Now the value in intCount is preserved between calls to the Tracker function, and you do indeed see the result 1, 2, 3, 4, 5, and 6.
Besides using the **Static** keyword, you can also make `intCount` a module-level variable
to do the same thing, by taking it out of any procedure:

```vba
Dim intCount As Integer

Function Tracker() As Integer
    intCount += 1
    Return intCount
End Function
```

The result here is the same as using **Static** to declare the variable; because the variable
is outside any procedure, its value isn’t reset when you call a procedure.

To declare module-level variables, you place the declaration outside any procedure in the
module. You can also select the module in the left drop-down list box at the top of the code
designer and the **(Declarations)** item in the right drop-down box, which will place the
cursor at the beginning of the module, outside any procedure.

Making a variable into a module-level variable outside any procedure like this introduces
the idea of **scope**. Here, the scope of this new variable is module-level scope. The scope
of an item in your program is the area in which it’s accessible in your code, and there are
all different types of scope—module-level, procedure-level, block-level, and more. And
we’ll look at this issue next.

### Understanding Scope

Now that we’re dividing our code into procedures, it’s a good idea to look at the issue of
scope because putting code in procedures restricts that code’s scope. Now that Visual
Basic .NET is emphasizing OOP more than ever before, scope has become even more
important because much of the power of classes and objects is all about restricting scope
and hiding implementation details to make things simpler.

The **scope** of a programming element in your code is all the code that can access it. In
other words, an element’s scope is its **accessibility** in your code. In Visual Basic .NET,
where you declare an element determines its scope, and an element can have one of the
following levels of scope:

- **Block scope**—The item is available only within the code block in which it is
  declared.
• **Procedure scope**—The item is available only within the procedure in which it is declared.

• **Module scope**—The item is available to all code within the module, class, or structure in which it is declared.

• **Namespace scope**—The item is available to all code in the namespace.

Let’s look at these various levels of scope.

**Block Level**
A code block is the body of a compound statement. A compound statement is one that can hold other statements, such as an `If` statement. Here’s an `If` statement in which a variable, `strText`, is declared. Note that `strText` is *inaccessible* outside the `If` statement, so code that tries to display its value won’t work:

```vbnet
Module Module1
    Sub Main()
        Console.WriteLine("Enter a letter...")
        Dim strInput = Console.ReadLine()
        If strInput = "q" Then
            End
        Else
            Dim strText As String = "Please type q to quit."
            Console.WriteLine(strText)
        End If
        Console.WriteLine(strText) 'Will not work!'
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
    End Sub
End Module
```

**Procedure Level**
An element declared in a procedure is not available outside that procedure, which means that only the code in the procedure that contains the declaration can access it. Elements at this level are called *local* elements, and you declare them with the `Dim` or `Static` statements. In the following example, the variable `strText` declared in the `ShowMessage` Sub procedure cannot be accessed in the `Main` Sub procedure:
Module Module1

Sub Main()
    ShowMessage()
    Console.WriteLine(strText) 'Will not work!
    Console.WriteLine("Press Enter to continue...")
    Console.ReadLine()
End Sub

Sub ShowMessage()
    Dim strText = "Hi there!"
    Console.WriteLine(strText)
End Sub

End Module

Module Level

Visual Basic .NET uses the term **module level** to apply to three programming elements: modules, classes, and structures. (We’ll see classes later today and structures in Day 9.) You declare elements at this level by placing the declaration outside any procedure or block in the module, class, or structure.

Unlike in blocks or procedures (where you can use only Dim or Static), at the module level you can also use these keywords to restrict or enlarge scope. (Don’t feel you have to memorize these definitions at this stage; we’ll see more on these terms throughout the book.)

- **Public**—The Public statement declares elements to be accessible anywhere. This includes inside the same project, from other projects that reference the current project, assemblies built from the project, and so on.
- **Protected**—The Protected statement declares elements to be accessible only from within the same class or from a class derived from this class. You can use Protected only at class level and only when declaring a member of a class.
- **Friend**—The Friend statement declares elements to be accessible from within the same project, but not from outside the project.
- **Protected Friend**—The Protected statement with the Friend keyword declares elements to be accessible either from derived classes or from within the same project, or both. You can use Protected Friend only at class level.
- **Private**—The Private statement declares elements to be accessible only from within the same module, class, or structure.
Let’s look at module-level scope with some examples. For example, you can create a new code module, Module2, like this:

```vbnet
Module Module1
    Sub Main()
        ShowMessage()
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
    End Sub
End Module

Module Module2
    Sub ShowMessage()
        Console.WriteLine("Hello there!")
    End Sub
End Module
```

Although this example declares two modules in the same file (Module1.vb), you can also add a module in a new file to a Visual Basic project by selecting the Project, Add Module menu item (which will create Module2.vb, Module3.vb, and so on).

And if you declare a new Sub procedure, ShowMessage, in the new module, you can access it from the first module:

```vbnet
Module Module1
    Sub Main()
        ShowMessage() 'Will not work!
    End Sub
End Module

Module Module2
    Sub ShowMessage()
        Console.WriteLine("Hello there!")
    End Sub
End Module
```

However, if you declare the Sub procedure Private to the new module, you cannot access it in the first module:
Console.WriteLine("Press Enter to continue...")
Console.ReadLine()
End Sub
End Module

Module Module2
    Private Sub ShowMessage()
        Console.WriteLine("Hello there!"
    End Sub
End Module

In module scope, you can also make variables—not just procedures—public or private; this example declares strText in the second module using a Dim statement:

Module Module1
    Sub Main()
        Console.WriteLine(strText) 'Will not work!
    End Sub
End Module

Module Module2
    Dim strText = "Hello there!"
End Module

By default, module-level variables are declared Private when you use Dim, so strText cannot be accessed outside its module. However, if you declare this new variable Public, it can be accessed in the first module with no problem:

Module Module1
    Sub Main()
        Console.WriteLine(strText)
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
    End Sub
End Module

Module Module2
    Public strText = "Hello there!"
End Module

**Namespace Scope**

You can also declare elements at namespace level in Visual Basic. A namespace is an OOP feature used to keep elements with the same name from conflicting with each other in larger programs. (If you don’t use a Namespace statement in your code, all your code is in the same namespace.) Declaring a module-level element Friend or Public makes it available to all procedures throughout the namespace.
We now have the background we’ll need on procedures and scope, two very important programming concepts. Next, let’s tackle handling the runtime errors that may crop up because the Visual Basic language puts special emphasis on this topic.

Handling Runtime Errors

You may have taken all the bugs out of your code, but there’s another kind of problem that is often impossible to head off—runtime errors, called exceptions in Visual Basic .NET. A runtime error occurs when your program is asked to do something it can’t do—divide a number by zero, for example, or open a file that doesn’t exist. Tomorrow we’ll start working with Windows forms programming and creating programs ready for public release, so it’s a good idea to know how to handle runtime errors; otherwise, they’ll crash your program.

Visual Basic .NET has good support for handling runtime errors. In fact, there are two ways of handling exceptions in Visual Basic .NET: unstructured (the VB6 way) and structured (the Visual Basic .NET way). We’ll see them both today.

Unstructured exception handling centers on the **On Error GoTo** statement, whereas structured exception handling centers on the **Try/Catch** statement. “Unstructured” exception handling is called that because you can handle it anywhere in your code with the **On Error GoTo** statement, whereas “structured” exception handling with the **Try/Catch** statement restricts exception handling to specific code blocks (as an **If** statement does). We’ll see unstructured exception handling first.

Unstructured Exception Handling

In Visual Basic, unstructured exception handling revolves around the **On Error GoTo** statement. Here’s how the **On Error GoTo** statement works:

```
On Error { GoTo [ line ] 0 | -1 | Resume Next }
```

The parts of this statement are as follows:

- **GoTo line**—Calls the error-handling code that starts at the line specified at **line**. Here, **line** is a line label or a line number. If a runtime error occurs, program execution goes to the given location. The specified line must be in the same procedure as the **On Error** statement.

- **GoTo 0**—Disables the enabled error handler in the current procedure.

- **GoTo -1**—Same as **GoTo 0**.
Resume Next—Specifies that when an exception occurs, execution skips over the statement that caused the problem and goes to the statement immediately following. Execution continues from that point.

The following example shows how to use the On Error GoTo statement that uses division by zero to create an overflow error. In this case, the code redirects execution to the label Handler. You can create this label by placing it on a line of its own followed by a colon:

```vba
Module Module1
    Sub Main()
        On Error GoTo Handler
        .
        .
        Exit Sub
    Handler:
        .
    End Sub
End Module
```

Note that we’ve used an Exit Sub statement here so that in normal execution, the procedure stops before reaching the error-handling code that follows the Handler label. Now we can add the code that causes the overflow error:

```vba
Module Module1
    Sub Main()
        On Error GoTo Handler
        Dim intItem1 As Integer = 0
        Dim intItem2 As Integer = 128
        Dim intResult As Integer
        intResult = intItem2 / intItem1
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
        Exit Sub
    Handler:
        .
    End Sub
End Module
```

When the overflow error occurs, control is transferred to the Handler label, and we can add code there to display an error message to the user, as shown in Listing 3.6 and the OnError project in the code for this book.
**Listing 3.6**  Using Unstructured Exception Handling (OnError project, Module1.vb)

```vbnet
Module Module1
    Sub Main()
        On Error GoTo Handler
        Dim intItem1 As Integer = 0
        Dim intItem2 As Integer = 128
        Dim intResult As Integer
        intResult = intItem2 / intItem1
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
        Exit Sub
    Handler:
        Console.WriteLine("An overflow error occurred.")
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
    End Sub
End Module
```

Now when you run this code, you’ll see this result:

An overflow error occurred.
Press Enter to continue...

In this way, you were able to handle the exception without having Visual Basic display all kinds of error message boxes.

Here’s something else to know: If you call procedure B from procedure A, and B doesn’t have any `On Error` exception-handling code but A does, then if an exception occurs in B, control will transfer back to A, where the exception will be handled.

**Getting an Exception’s Number and Description**

A built-in error object named `Err` has a `Number` property that enables you to determine an error’s number. While testing your program, you can use `Err.Number` to determine the numbers of the errors you might run into and can then use those numbers to handle those errors in different ways. For example, this code handles only overflow errors, which are error number 6:

```vbnet
Module Module1
    Sub Main()
        On Error GoTo Handler
        Dim intItem1 As Integer = 0
        Dim intItem2 As Integer = 128
        Dim intResult As Integer
```
intResult = intItem2 / intItem1
Console.WriteLine("Press Enter to continue...")
Console.ReadLine()
Exit Sub
Handler:
If (Err.Number = 6) Then
    Console.WriteLine("An overflow error occurred.")
End If
Console.WriteLine("Press Enter to continue...")
Console.ReadLine()
End Sub
End Module

As of this writing, the Visual Basic documentation no longer includes a list of errors by error number. To find a runtime error's number, you can always create that error yourself and have your code display the value of Err.Number.

Tip
You can also use the Err object's Description property to get a short description of the error, as in this code:

Module Module1
    Sub Main()
        On Error GoTo Handler
        Dim intItem1 As Integer = 0
        Dim intItem2 As Integer = 128
        Dim intResult As Integer
        intResult = intItem2 / intItem1
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
        Exit Sub
    Handler:
        Console.WriteLine(Err.Description)
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
    End Sub
End Module

This code displays this message:

Arithmetic operation resulted in an overflow.
Press Enter to continue...
You can determine more details about the source of an error by using the Err object's Source property. This property holds the name of the object or application that caused the error. For example, if you connect your program to Microsoft Excel and it generates an error, Err.Source will hold "Excel.Application".

**Using Exception Objects**

You can also use Err.GetException to get an exception object that enables you to determine what kind of runtime error occurred. Visual Basic .NET has many exception objects, and you can see a sampling in Table 3.1.

**Table 3.1 Some Visual Basic Exceptions**

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppDomainUnloadedException</td>
<td>ArgumentException</td>
</tr>
<tr>
<td>ArgumentException</td>
<td>ArgumentNullException</td>
</tr>
<tr>
<td>ArgumentOutOfRangeException</td>
<td>ArgumentException</td>
</tr>
<tr>
<td>ArrayTypeMismatchException</td>
<td>BadImageFormatException</td>
</tr>
<tr>
<td>CannotUnloadAppDomainException</td>
<td>ComponentException</td>
</tr>
<tr>
<td>DivideByZeroException</td>
<td>DllNotFoundException</td>
</tr>
<tr>
<td>DuplicateWaitObjectException</td>
<td>EntryPointNotFoundException</td>
</tr>
<tr>
<td>ExecutionEngineException</td>
<td>ExternalException</td>
</tr>
<tr>
<td>FieldAccessException</td>
<td>FormatException</td>
</tr>
<tr>
<td>IndexOutOfRangeException</td>
<td>InvalidCastException</td>
</tr>
<tr>
<td>InvalidOperationException</td>
<td>InvalidProgramException</td>
</tr>
<tr>
<td>MissingFieldException</td>
<td>MissingMemberException</td>
</tr>
<tr>
<td>MissingMethodException</td>
<td>MulticastNotSupportedException</td>
</tr>
<tr>
<td>NotFiniteNumberException</td>
<td>NotImplementedException</td>
</tr>
<tr>
<td>NotSupportedException</td>
<td>NullReferenceException</td>
</tr>
<tr>
<td>ObjectDisposedException</td>
<td>OperationException</td>
</tr>
<tr>
<td>OutOfMemoryException</td>
<td>OverflowException</td>
</tr>
<tr>
<td>PlatformNotSupportedException</td>
<td>RankException</td>
</tr>
<tr>
<td>SafeArrayTypeMismatchException</td>
<td>StackOverflowException</td>
</tr>
</tbody>
</table>
For example, to test for an overflow error, you can check if the type of the exception is `OverflowException`. To do that, you use the `TypeOf` and `Is` keywords, which are specifically designed for just this purpose to be used in `If` statements to work with objects:

```vbscript
Module Module1
    Sub Main()
        On Error GoTo Handler
        Dim intItem1 As Integer = 0
        Dim intItem2 As Integer = 128
        Dim intResult As Integer
        intResult = intItem2 / intItem1
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
        Exit Sub
    Handler:
        If (TypeOf Err.GetException() Is OverflowException) Then
            Console.WriteLine("An overflow error occurred.")
        End If
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
    End Sub
End Module
```

Even though structured exception handling is newer than unstructured exception handling, you can still do things with unstructured exception handling that you can’t with structured exception handling. The main thing is that you can resume execution of the part of your code that caused the error (after skipping the problematic line) using the `Resume` statement.

**Using the Resume Statement**

After an exception occurs, you can use the `Resume` statement to resume program execution in unstructured exception handling. Here are the possibilities:
• Resume resumes execution with the statement that caused the error.
• Resume Next resumes execution with the statement after the one that caused the error.
• Resume line resumes execution at line, a line number or label that specifies where to resume execution.

Listing 3.7 is an example using Resume Next; this is the Resume project in the code for this book. It enables you to skip over the line that caused the problem and keep going with the following line.

**Listing 3.7** Using the Resume Statement (Resume project, Module1.vb)

```vbnet
Module Module1
    Sub Main()
        On Error GoTo Handler
        Dim intItem1 As Integer = 0
        Dim intItem2 As Integer = 128
        Dim intResult As Integer
        intResult = intItem2 / intItem1
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
        Exit Sub
    Handler:
        Console.WriteLine("An overflow error occurred.")
        Resume Next
        End Sub
End Module
```

Here’s what you see when you run this application:

An overflow error occurred.
Press Enter to continue...

And here’s an example using Resume line to do exactly the same thing. Here, we’re using a line label, LineAfter, which is just a (nonreserved) word followed by a colon that can be used to label a line of code:

```vbnet
Module Module1
    Sub Main()
        On Error GoTo Handler
        Dim intItem1 As Integer = 0
        Dim intItem2 As Integer = 128
        Dim intResult As Integer
        intResult = intItem2 / intItem1
```

You can use the statement `On Error Resume Next` or `On Error Resume` at the beginning of your code when you don't want to add explicit exception-handling code. These statements will make Visual Basic .NET continue execution after an exception has occurred.

### Turning Off Exception Handling

To turn off unstructured error handling, you can use the `On Error GoTo 0` or `On Error GoTo -1` statements (they do the same thing). Here’s an example:

```vbnet
Module Module1
    Sub Main()
        On Error GoTo Handler
        Dim intItem1 As Integer = 0
        Dim intItem2 As Integer = 128
        Dim intResult As Integer
        intResult = intItem2 / intItem1
        On Error GoTo 0
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
        Exit Sub
    Handler:
        Console.WriteLine("An overflow error occurred.")
        Resume LineAfter
        Exit Sub
    End Module
```

Now that we have a good understanding of unstructured exception handling, we’ll look at structured exception handling next.

### Structured Exception Handling

In addition to unstructured exception handling, Visual Basic .NET also supports structured exception handling. It’s important to know about structured exception handling...
because not only does it allow you to handle exceptions, but in some circumstances, Visual Basic will insist that you handle possible exceptions using structured exception handling before running your code.

**NEW TERM**

In Visual Basic .NET, structured exception handling centers on the **Try/Catch** statement. In this statement, you put sensitive, exception-prone code in a **Try** block, and if an exception occurs, the code in the **Try** block will **throw** the exception (actually an exception object), which will then be **caught** by a following **Catch** block. The code in the **Catch** block handles the exception. Unlike unstructured exception handling, where exception handling can be spread throughout your code, structured exception handling all takes place inside a **Try/Catch** statement. Here’s the syntax of that statement:

```
Try
  [ tryStatements ]
  [Catch [ exception1 [ As type1 ] ] [ When expression1 ]
    catchStatements1
    [ Exit Try ]
  [Catch [ exception2 [ As type2 ] ] [ When expression2 ]
    catchStatements2
    [ Exit Try ]
    .
    .
  [Catch [ exceptionn [ As typen ] ] [ When expressionn ]
    catchStatementsn
    [ Exit Try ]
  [ Finally
    [ finallyStatements ] ]
End Try
```

The parts of this statement are as follows:

- **Try**—Starts the **Try** block.
- **tryStatements**—Specifies the sensitive statements where you anticipate possible exceptions.
- **Catch**—Starts the block that catches and handles the exception(s).
- **exception**—Specifies a variable that you give to the exception.
- **type**—Indicates the type of the exception you want to catch in a **Catch** block.
- **When expression**—Specifies a **Catch** block clause that means the **Catch** block will catch exceptions only when **expression** is True. The **expression** is an expression
used to select exceptions to handle; it must be convertible to a Boolean value. It is often used to select errors by number.

- **catchStatements**—Specifies statements that handle exceptions occurring in the Try block.
- **Exit Try**—Exits a Try/Catch statement immediately. Execution moves to the code immediately following the End Try statement.
- **Finally**—Starts a Finally block that is always executed when execution leaves the Try/Catch statement. If a Try statement does not contain any Catch blocks, it must contain a Finally block. Exit Try is not allowed in Finally blocks.
- **finallyStatements**—Specifies statements that are executed after all other exception processing has occurred.

Here’s an example, the Exception project in the code for this book, to get us started. In this case, the exception-prone code in the Try block executes a division by zero, which generates an overflow exception:

```vbnet
Module Module1
    Sub Main()
        Dim intItem1 As Integer = 0
        Dim intItem2 As Integer = 128
        Dim intResult As Integer
        Try
            intResult = intItem2 / intItem1
            Console.WriteLine("The answer is " & intResult)
            Console.WriteLine("Press Enter to continue...")
            Console.ReadLine()
        End Try
    End Sub
End Module
```

When an exception occurs, control leaves the Try block and enters the Catch block, where you can handle the exception something like you see in Listing 3.8.

**LISTING 3.8** Using Structured Exception Handling (Exception project, Module1.vb)

```vbnet
Module Module1
    Sub Main()
        Dim intItem1 As Integer = 0
        Dim intItem2 As Integer = 128
        Dim intResult As Integer
```
When you run this application, you'll see this result:

An overflow exception occurred.
Press Enter to continue...

You can also get Visual Basic's error message for an exception. To do that, you can create a variable, called e here, which will hold the exception:

Try
    intResult = intItem2 / intItem1
    Console.WriteLine("The answer is " & intResult)
    Console.WriteLine("Press Enter to continue...")
    Console.ReadLine()
Catch e As Exception
    Console.WriteLine("An overflow exception occurred.")
    Console.WriteLine("Press Enter to continue...")
    Console.ReadLine()
End Try

Now that you have the actual Visual Basic exception object corresponding to the exception that occurred in the variable named e, you can use e.ToString to display the exception as text like this:

Try
    intResult = intItem2 / intItem1
    .
    .
Catch e As Exception
    Console.WriteLine(e.ToString())
    Console.WriteLine("Press Enter to continue...")
    Console.ReadLine()
End Try
Here’s what you might see when a division by zero takes place:

```
System.OverflowException: Arithmetic operation resulted in an overflow.
  at Filters.Module1.Main() in C:\vbnet 2003\Day3\Filters\Module1.vb:line 7
Press Enter to continue...
```

Although this result is good for programmers because it indicates the location of the problem, this exception description isn’t very helpful to users. Instead, you can use the exception object’s Message property to display a better error message:

```
Try
    intResult = intItem2 / intItem1
Catch e As Exception
    Console.WriteLine(e.Message)
    Console.WriteLine("Press Enter to continue...")
End Try
```

Here’s what you’ll see in this case:

```
Arithmetic operation resulted in an overflow.
Press Enter to continue...
```

The generic `Catch e As Exception` block in this example catches all types of exceptions, but, as with unstructured exception handling, you can narrow down an exception handler to catching only a single type of exception, a process called filtering.

### Exception Filtering in the Catch Block

A `Catch e As Exception` block catches all types of exceptions and stores the exception object in a variable named `e`. However, you can specify that a `Catch` block catches only certain types of exceptions with the `As type` clause. (Table 3.1 contains a list of some of the Visual Basic .NET exception types.) For example, to catch only exceptions of the type `OverflowException`, you can add a `Catch` block as in Listing 3.9.

### Listing 3.9  Filtered Exception Handling (Filters project, Module1.vb)

```
Module Module1
    Sub Main()
        Dim intItem1 As Integer = 0
        Dim intItem2 As Integer = 128
        Dim intResult As Integer
```
If you want to get the name of an exception and you're able to cause that exception while testing your program, call `e.GetType()` to get the type of the exception as a string. When you know the name of the exception, you can provide a `Catch` block for it.

You can also use a `When` clause in a `Catch` block to further filter exception handling. For example, your code may have multiple sections that may cause overflows, and you want to handle the overflows only when `intItem1` is 0 in a particular `Catch` block. You can do that in a `When` clause that uses the logical expression `intItem1 = 0`:

```vbnet
Try
    intResult = intItem2 / intItem1
    .
    .
Catch e As OverflowException When intItem1 = 0
    Console.WriteLine("An overflow exception occurred.")
    Console.WriteLine("Press Enter to continue...")
    Console.ReadLine()
End Try
```

You can also filter on error numbers like this:

```vbnet
Try
    intResult = intItem2 / intItem1
    .
    .
    .
Catch e As OverflowException When intItem1 = 0
    Console.WriteLine("An overflow exception occurred.")
    Console.WriteLine("Press Enter to continue...")
    Console.ReadLine()
End Try
```
Catch e As OverflowException When Err.Number = 6
    Console.WriteLine("An overflow exception occurred.")
    Console.WriteLine("Press Enter to continue...")
    Console.ReadLine()
End Try

But what about handling the other overflow exceptions caused by your code. Can you have multiple catch blocks in the same Try/Catch statement? You certainly can.

**Using Multiple Catch Statements**

If you want to handle a number of different exceptions, you can use multiple catch blocks. In the following example, different catch blocks handle overflow exceptions, out-of-memory exceptions, and array index out-of-range exceptions (which occur when an array index has been set to a negative value or to a value greater than the upper bound of the array):

```
Module Module1
    Sub Main()
        Dim intItem1 As Integer = 0
        Dim intItem2 As Integer = 128
        Dim intResult As Integer
        Try
            intResult = intItem2 / intItem1
            Console.WriteLine("The answer is " & intResult)
            Console.WriteLine("Press Enter to continue...")
            Console.ReadLine()
        Catch e As OverflowException
            Console.WriteLine("An overflow exception occurred.")
            Console.WriteLine("Press Enter to continue...")
            Console.ReadLine()
        Catch e As OutOfMemoryException
            Console.WriteLine("Out of memory.")
            Console.WriteLine("Press Enter to continue...")
            Console.ReadLine()
        Catch e As IndexOutOfRange
            Console.WriteLine("Array index out of range.")
            Console.WriteLine("Press Enter to continue...")
            Console.ReadLine()
        End Try
    End Sub
End Module
```

Using multiple catch blocks like this gives you a good handle on handling exceptions of various types. Each catch block serves as a different exception handler, and you can place exception-specific code in each such block.
Using Finally

Another part of the Try/Catch statement that you should know about is the Finally block. The code in the Finally block, if there is one, is always executed in a Try/Catch statement, even if there was no exception, and even if you execute an Exit Try statement. This allows you to make sure that even if there was an exception you’ll be sure of running this code (as long as the whole program is still running). Listing 3.10 shows an example with a Finally block, the Finally example in the code for the book.

**Listing 3.10** Using the Finally Statement (Finally project, Module1.vb)

```vbnet
Module Module1
    Sub Main()
        Dim intItem1 As Integer = 0
        Dim intItem2 As Integer = 128
        Dim intResult As Integer
        Try
            intResult = intItem2 / intItem1
            Console.WriteLine("The answer is " & intResult)
        Catch e As OverflowException
            Console.WriteLine("An overflow exception occurred.")
        Catch e As OutOfMemoryException
            Console.WriteLine("Out of memory.")
        Catch e As IndexOutOfRangeException
            Console.WriteLine("Array index out of range.")
        Finally
            Console.WriteLine("Press Enter to continue...")
        End Try
    End Sub
End Module
```

In this case, you’ll always see the "Press Enter to continue..." prompt, and the code will wait for you to press Enter whether or not an exception was thrown. Here’s what you might see:

An overflow exception occurred.
Press Enter to continue...

Throwing an Exception Yourself

You can throw an exception yourself in your code by using the Throw statement. In this example the code explicitly throws an overflow exception:

```vbnet
Module Module1
    Sub Main()
```
Throwing your own exceptions like this gives you a great deal of control over the exception-handling process. And you’re not limited to just the exceptions that Visual Basic has already defined either, as we’ll see next.

**Creating and Throwing Custom Exceptions**

You can, in fact, customize and create your own exceptions. To do that, you throw an exception using the `ApplicationException` object. Listing 3.11 shows an example, the `CustomException` example in the code for this book, where we’re creating a custom exception and giving it the text “This is a new exception”.

**Listing 3.11  Creating Custom Exceptions (CustomException project, Module1.vb)**

```vbnet
Module Module1
    Sub Main()
        Try
            Throw New ApplicationException("You threw this custom exception.")
            Console.WriteLine("Press Enter to continue...")
            Console.ReadLine()
        Catch e As Exception
            Console.WriteLine(e.Message)
            Console.WriteLine("Press Enter to continue...")
            Console.ReadLine()
        Finally
            Console.WriteLine("Press Enter to continue...")
            Console.ReadLine()
        End Try
    End Sub
End Module
```

Here’s what you see when you run this application:

You threw this custom exception.
Press Enter to continue...

Note the `New` keyword in the statement `Throw New ApplicationException("You threw this custom exception.").` You use this keyword when you create new objects.
As we head into tomorrow’s work with Windows forms, which will take us through five days, we’ll need to know something about classes and objects. For that reason, we’ll get a good introduction to them here.

**Introducing Classes and Objects**

Object-oriented programming (OOP) is a topic that is central to Visual Basic, and we’re going to take it step by step, getting an introduction to classes and objects today and continuing our OOP work throughout the book.

Yesterday, we worked with simple variables, like this integer variable:

```vbnet
Dim intItem1 As Integer
```

Here, `intItem1` is the variable and `Integer` is the variable’s *type*. In the same way, you can declare *objects*, using a *class* as the object’s type:

```vbnet
Dim TheObject As TheClass
```

Here, `TheObject` is an object of `TheClass` class. In this way, you can think of a class as an object’s *type*. What’s different about a class from a simple type like an `Integer`? The difference is that classes and objects can contain *members*. As we’ve already seen, one type of member is a *property*, which holds data, and you access an object’s properties with a dot followed by the property name like this:

```vbnet
TheObject.TheProperty = "Hello!"
```

Classes and objects can also contain built-in procedures, called *methods*. A method can either be a Sub procedure or a function, and you call it as you would any other procedure, except that you need to preface the method name with the class or object name and a dot like this, where `TheMethod` returns a message as a text string:

```vbnet
Dim Message As String = TheObject.TheMethod()
```

In practice, you usually use the properties and methods of objects only. Classes can support properties and methods directly, without needing to create an object, but you have to make special provisions in the class, as we’ll see in Day 9. Those members of a class that you can use with the class directly without needing to create an object first are called *class members*; for example, if `TheMethod` was a class method, you could use it with the class name, no object needed:

```vbnet
Dim Message As String = TheClass.TheMethod()
```
Encapsulation

Encapsulation is the process of packaging a great deal of code into objects that allow programmers to deal with code as a whole, rather than having to deal with the code on an individual basis. The advantage of this is that all the complexities of the code are hidden from the programmer. The programmer is only concerned with the object, which has a well-defined interface that allows access to the object's properties and methods.

An Example in Code

The Windows forms that we’ll be creating in the next five days are all classes, created with the Class statement, as we’ll see tomorrow. For that reason, let’s look at an example that creates a class and an object in code. This example will be much simpler than the
Visual Basic **TextBox** class, but it’s similar in many ways: Both the **TextBox** class and our new class are created with a **Class** statement, and both will support properties and methods. This example will start giving us the edge we need when dealing with OOP.

**Creating a Class**

How do you actually create a class? You use the **Class** statement:

```
[<attrlist>] [Public | Private | Protected | Friend | Protected Friend]
[Shadows] [MustInherit | NotInheritable] Class name
[Implements interfacename]
[statements]
End Class
```

We saw most of these terms described with the **Sub** statement earlier today. Here are what the new items mean:

- **MustInherit**—Indicates that the class contains methods that must be implemented by a deriving class, as we’ll see in Day 9.
- **NotInheritable**—Indicates that the class is one from which no further inheritance is allowed, as we’ll see in Day 9.
- **name**—Specifies the name of the class.
- **statements**—Specifies the statements that make up the code for the class.

Here’s how we might create a new class named **TheClass** in a project named **Classes** (which is in the code for this book):

```
Module Module1
    Sub Main()
    End Sub
End Module

Class TheClass
    ..
End Class
```

**Creating a Data Member**

We can now add members to this class, making them **Public**, **Private**, or **Protected**. **Public** makes them accessible from code outside the class, **Private** restricts their scope to the current class, and **Protected** (which we’ll see in Day 9) restricts their scope to the
present class and any classes derived from the present class. For example, we can add a
data member to our new class just by declaring a variable Public like this:

```vbnet
Module Module1
    Sub Main()
        End Sub
    End Module
End Module

Class TheClass
    Public ThePublicData = "Hello there!"
    .
End Class
```

This is not a property of the new class. You need special code to create a property, as
we'll see. However, this new data member is accessible from objects of this class like
this: TheObject.ThePublicDataMember. By default, data members are private to a class,
but you can make them public—accessible outside the object—with the Public keyword
as done here.

**Creating an Object**

Let's see how to access our new data member by creating an object, TheObject,
from TheClass. Creating an object is also called `instantiating` an object, and an
object is also called an `instance` of a class. Unlike the declaration of a simple variable
like an Integer, this line of code doesn't create a new object; it only declares the object:

```vbnet
Dim TheObject As TheClass
```

To create a new object in Visual Basic, you need to use the `New` keyword:

```vbnet
Dim TheObject As New TheClass
```

This line creates the new object TheObject from TheClass. You can also create the new
object this way:

```vbnet
Dim TheObject As TheClass
TheObject = New TheClass
```

Some classes are written so that you can pass data to them when you create
objects from them. What you're actually doing is passing data to a special
method of the class called a `constructor`. For example, as we'll see tomorrow, you can
create a `Size` object to indicate a Windows form’s size, and to make that form 200 by 200 pixels, you can pass those values to the `Size` class’s constructor this way:

```csharp
Size = New Size(200, 200)
```

We’ll often use constructors when creating objects, but our current example doesn’t use one. Here’s how we can create a new object—`TheObject`—using the class `TheClass`, and display the text in the public data member `TheObject.ThePublicDataMember`:

```csharp
Module Module1
    Sub Main()
        Dim TheObject As New TheClass
        Console.WriteLine("ThePublicData holds ", TheObject.ThePublicData)"
    End Sub
End Module
```

Class `TheClass`

```csharp
Public ThePublicData = "Hello there!"
```

You can use public data members like this instead of properties if you like, but properties were invented to give you some control over what values can be stored. You can assign any value to a public data member like this, but you can use internal code to restrict possible values that may be assigned to a property (for example, you might want to restrict a property named `Color` to the values "Red", "Green", and "Blue"). Properties are actually implemented using code in methods, not just as data members. And we’ll look at methods next.

**Creating a Method**

Let’s add a method to our class now. This method, `TheMethod`, will simply be a function that returns the text "Hello there!", and we can call that method as `TheObject.TheMethod()` to display that text:

```csharp
Module Module1
    Sub Main()
        Dim TheObject As New TheClass
```
That’s all it takes! Just adding a Sub procedure or function to a class like this adds a new method to the class. Although such methods are public by default, you can make methods private (declaring them like this: Private Function TheMethod() As String), which restricts their scope to the present class. You do that for methods that are used only internally in a class. (To continue the car analogy, such internal methods might be the ones entirely internal to the car, such as the ones that regulate oil pressure or battery voltage.)

**Creating a Property**

Let’s see how to create a property now. You declare properties using Get and Set methods in a Property statement:

```vbnet
[ <attrlist> ] [ Default ] [ Public | Private | Protected | Friend | Protected Friend ] [ ReadOnly | WriteOnly ] [ Overloads | Overrides ] [ Overridable | NotOverridable ] [ MustOverride | Shadows | Shared ] Property varname([ parameter list ]) [ As typename ] [ Implements interfacemember ]
[ <attrlist> ] Get
[ block ]
End Get
[ <attrlist> ] Set(ByVal Value As typename )
[ block ]
End Set
End Property
```
Here are the parts of this statement that are different from the keywords we’ve already seen in the Sub statement:

- **Default**—Makes this a default property. Default properties can be set and retrieved without specifying the property name.

- **ReadOnly**—Specifies that a property’s value can be retrieved, but it cannot be modified. **ReadOnly** properties contain **Get** blocks but no **Set** blocks.

- **WriteOnly**—Specifies that a property can be set, but its value cannot be retrieved. **WriteOnly** properties contain **Set** blocks but no **Get** blocks.

- **varname**—Specifies a name that identifies the property.

- **parameter list**—Specifies the parameters you use with the property.

- **typename**—Specifies the type of the property. If you don’t specify a data type, the default type is **Object**.

- **interfacemember**—When a property is part of a class that implements an interface (covered in Day 9), this is the name of the property being implemented.

- **Get**—Starts a **Get** property procedure used to return the value of a property. **Get** blocks are optional unless the property is **ReadOnly**.

- **End Get**—Ends a **Get** property procedure.

- **Set**—Starts a **Set** property procedure used to set the value of a property. **Set** blocks are optional unless the property is **WriteOnly**. Note that the new value of the property is passed to the **Set** property procedure in a parameter named **Value** when the value of the property changes.

- **End Set**—Ends a **Set** property procedure.

The **Set** block enables you to set the value of a property, and the **Get** block enables you to return the value of the property. When you assign a value to a property (as in `TheObject.Property = "Hello there!"`), the **Set** block is called, and when you read a property’s value (as in `Dim strText As String = TheObject.Property`), the **Get** block is called.

You can add code to both blocks to restrict the data stored in a property. Visual Basic passes a parameter named **Value** to the **Set** block during property assignments, and the **Value** parameter contains the value that was assigned to the property when the **Set** block was called. You usually store the actual value of the property in a private data member in the class.
Here’s how this looks in the example. First, you create the new property procedure to handle the property `TheProperty`, which will hold string data:

```vbnet
Module Module1

Sub Main()
    Dim TheObject As New TheClass
    Console.WriteLine("ThePublicData holds " & TheObject.ThePublicData & "")
    Console.WriteLine("TheMethod returns " & TheObject.TheMethod() & "")
    Console.WriteLine("TheProperty holds " & TheObject.TheProperty & "")
    Console.WriteLine("Press Enter to continue...")
    Console.ReadLine()
End Sub

End Module

Class TheClass
    Public ThePublicData = "Hello there!"

    Function TheMethod() As String
        Return "Hello there!"
    End Function

    Public Property TheProperty() As String
        Get
        End Get
        Set(ByVal Value As String)
        End Set
    End Property

End Class
```

**Tip**

When you type the first line of a property procedure—that’s `Public Property TheProperty() As String`—Visual Basic .NET will add a skeleton for the `Get` and `Set` procedures automatically.

The next step is to add the code for this property. In this case, we can store the property’s value in a private data member named `TheInternalData`. All we have to do is add code to the `Set` block to store values in this data member and the `Get` block to return this data member’s value when needed, as shown in Listing 3.12.
**Listing 3.12**  Classes and Objects (Classes project, Module1.vb)

```vbnet
Module Module1

    Sub Main()
        Dim TheObject As New TheClass
        Console.WriteLine("ThePublicData holds ", TheObject.ThePublicData & ","
        Console.WriteLine("TheMethod returns ", TheObject.TheMethod() & ","
        Console.WriteLine("TheProperty holds ", TheObject.TheProperty & ",")
        Console.WriteLine("Press Enter to continue...")
        Console.ReadLine()
    End Sub

End Module

Class TheClass

    Public ThePublicData = "Hello there!"
    Private TheInternalData As String = "Hello there!"

    Function TheMethod() As String
        Return "Hello there!"
    End Function

    Public Property TheProperty() As String
        Get
            Return TheInternalData
        End Get
        Set(ByVal Value As String)
            TheInternalData = Value
        End Set
    End Property

End Class

And that's all we need! Now we've created a new class, given that class a public data
member, method, and property, and displayed the data from each of these elements.
Here's what you see when you run this example:

ThePublicData holds "Hello there!"
TheMethod returns "Hello there!"
TheProperty holds "Hello there!"
Press Enter to continue...

And that gives us the foundation we'll need when we start working with classes and
objects tomorrow.
```
Summary

Today, we saw a great deal about the Visual Basic language. We started by looking at how to create procedures in Visual Basic .NET.

There are two types of procedures: Sub procedures and functions. They can both contain sets of Visual Basic statements that may be called, but only functions can return values. We saw that we can pass arguments to procedures, including optional arguments. We also saw how to support variable-length argument lists. And we also saw how to preserve data values between calls to procedures—both with static variables and module-level variables.

That introduced us to the idea of scope. An element’s scope is the area in a program where that element is accessible. We saw that there are four levels of scope: block level, procedure level, module level, and namespace level.

We also looked at runtime error handling today. Runtime errors are called exceptions, and the two types of exception handling in Visual Basic are structured and unstructured exception handling. Structured exception handling centers around the Try/Catch statement, and unstructured exception handling uses the On Error statement. We saw how to catch exceptions, throw our own exceptions, filter exceptions, and more.

We also got an introduction to classes and objects in OOP today. We saw how to create a class, create an object of that class, and create public data members, properties, and methods in a class. That information will come in handy throughout the book.

And that’s it! We now have the foundation in the Visual Basic language we’ll need for the coming days. Tomorrow, we’re going to begin putting to work what we’ve seen so far as we start working with Windows applications.

Q&A

Q How can I pass an array to a procedure?

A Say you create an array like this: Dim Array(4) As Integer. Then you assign a value to one element like this: Array(1) = 1. Now you can pass the array to a function named TheFunction as follows: TheFunction(Array). The function is declared like this: Function TheFunction(ByVal PassedArray() As Integer) As Integer. And finally you can refer to the passed array using the name PassedArray as follows: Return PassedArray(1).
Q Can I mix structured and unstructured exception handling?
A Only up to a point. You can't mix the On Error and Try/Catch statements, but you can, for example, access the Err.Number property in Catch blocks.

Workshop
This workshop tests whether you understand the concepts you saw today. It's a good idea to make sure you can answer these questions before pressing on to tomorrow's work.

Quiz
1. What's the difference between a function and a Sub procedure?
2. How can a Sub procedure pass data back to the calling code?
3. List three statements that will cause an immediate exit from a procedure.
4. What level of scope does a variable defined in a For loop have?
5. By default, what type of access do data members and methods have in classes?

Quiz Answers
1. A function can have a formal return value, whereas a Sub procedure cannot.
2. You can pass data back to the calling code if you pass a variable by reference. In that case, the code in the Sub procedure can assign a new value to that variable.
4. Block-level scope.
5. Data members are private by default (that is, if declared with the Dim statement), and methods are public by default.

Exercises
1. Modify the Summer function discussed today to use the ParamArray keyword so that it will add as many integers as you pass to this function and return the total sum (not just the sum of two integers, as the version we've already seen today does).
2. As discussed in the text, add a new property, Color, to TheClass in the Classes example that will accept only the values "Red", "Green", and "Blue". If any other value is assigned to the Color property, store the word "Undefined" in that property.