CHAPTER

2

Building Multitier Programs with Classes

at the completion of this chapter, you will be able to . . .

1. Discuss object-oriented terminology.
2. Create your own class and instantiate objects based on the class.
3. Create a new class based on an existing class.
4. Divide an application into multiple tiers.
5. Throw and catch exceptions.
6. Choose the proper scope for variables.
7. Validate user input using the `TryParse` and display messages using an ErrorProvider component.

At this point in your programming career, you should be comfortable with using objects, methods, and properties. You have already learned most of the basics of programming including decisions, loops, and arrays. You must now start writing your programs in styles appropriate for larger production projects.

Most programming tasks are done in teams. Many developers may work on different portions of the code and all of the code must work together. One of the key concepts of object-oriented programming (OOP) is that of using building blocks. You will now break your programs into blocks, or, using the proper term, classes.

This chapter reviews object-oriented programming concepts and techniques for breaking your program into multiple tiers with multiple classes. Depending on how much of your first course was spent on OOP, you may find that much of this chapter is review.

Object-Oriented Programming

Visual Basic is an object-oriented language and all programming uses the OOP approach. You have *used* objects but were likely shielded from most of the nitty-gritty of *creating* objects. In VB you will find that everything you do is based on classes. Each form is a class, which must be instantiated before it can be used. Even variables of the basic data types are objects, with properties and methods.

OOP Terminology

The key features of object-oriented programming are abstraction, encapsulation, inheritance, and polymorphism.

Abstraction

Abstraction means to create a model of an object, for the purpose of determining the characteristics (properties) and the behaviors (methods) of the object. For example, a Customer class is an abstract representation of a real customer, and a Product class is an abstract version of a real product. You need to use abstraction when planning an object-oriented program, to determine the classes that you need and the necessary properties and methods. It is helpful to think of objects generically; that is, what are the characteristics of a typical product, rather than a specific product.

Encapsulation

Encapsulation refers to the combination of characteristics of an object along with its behaviors. You have one "package" that holds the definition of all properties, methods, and events.

Encapsulation makes it possible to accomplish data hiding. Each object keeps its data (properties) and procedures (methods) hidden. Through use of the `Public` and `Private` keywords, an object can "expose" only those data elements and procedures that it wishes to allow the outside world to see.

You can witness encapsulation by looking at any Windows program. The form is actually a class. All of the methods and events that you code are

enclosed within the `Class` and `End Class` statements. The variables that you place in your code are actually properties of that specific form class.

Inheritance

Inheritance is the ability to create a new class from an existing class. You can add enhancements to an existing class without modifying the original. By creating a new class that inherits from an existing class, you can add or change class variables and methods. For example, each of the forms that you create is inherited from, or derived from, the existing Form class. The original class is known as the **base class**, **superclass**, or **parent class**. The inherited class is called a **subclass**, a **derived class**, or a **child class**. Of course, a new class can inherit from a subclass—that subclass becomes a superclass as well as a subclass.

You can see the inheritance for a form, which is declared in the form's designer.vb file. Show all files in the Solution Explorer, expand the files for a form, and open the form's designer.vb file. Look closely at the first line of code:

```
Partial Public Class MainForm
    Inherits System.Windows.Forms.Form
```

Inherited classes should always have an "is a" relationship with the base class. In the form example, the new MainForm "is a" Form (Figure 2.1). You could create a new Customer class that inherits from a Person class; a customer "is a" person. But you should not create a new SalesOrder class that inherits from Person; a sales order is *not* a person.

The real purpose of inheritance is **reusability**. You may need to reuse or obtain the functionality from one class when you have another similar situation. The new MainForm class that you create has all of the characteristics and actions of the base class, System.Windows.Forms.Form. From there you can add the functionality for your own new form.

You can create your own hierarchy of classes. You place the code you want to be common in a base class. You then create other classes, the derived classes or subclasses, which can call the shared functions. This concept is very helpful if you have features that are similar in two classes. Rather than writing two classes that are almost identical, you can create a base class that contains the similar procedures.

Sometimes you create a class specifically to use it as a base for derived classes. You can create a class strictly for inheritance; such a class is called an **abstract class** and is declared with `MustInherit` in the class header. You

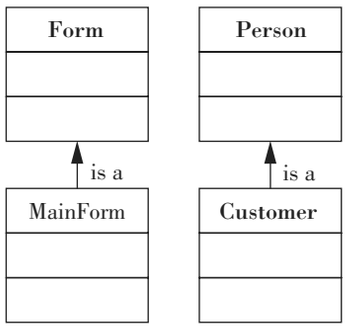


Figure 2.1

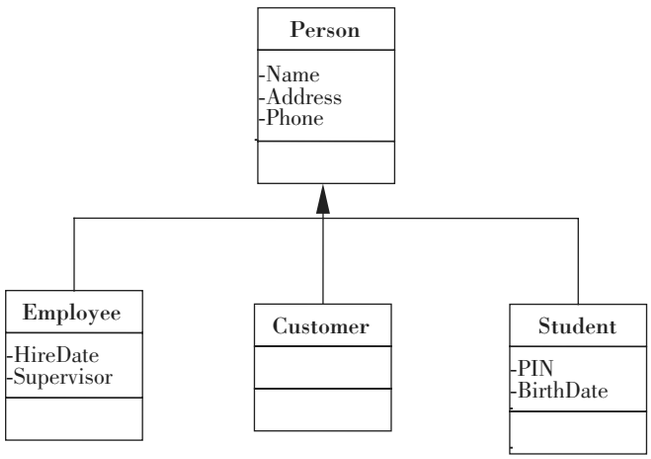
A derived or inherited class has an "is a" relationship with its base class.

cannot instantiate objects from an abstract class, only inherit new classes from it. Some of the methods in the base class may not even contain any code but are there as placeholders, forcing any derived classes to have methods with the defined names. A derived class with a method named the same as a method in the base class is said to **override** the method in the base class. Overriding allows an inherited class to take different actions from the identically named method in the base class.

An example of reusing classes could be a Person class, where you might have properties for name, address, and phone number. The Person class can be a base class, from which you derive an Employee class, a Customer class, or a Student class (Figure 2.2). The derived classes could call procedures from the base class and contain any additional procedures that are unique to the derived class. In inheritance, typically the classes go from the general to the more specific. You can add functionality to an inherited class. You also can change a function by overriding a method from the base class.

Figure 2.2

Multiple subclasses can inherit from a single base class.



Polymorphism

The term ***polymorphism*** actually means the ability to take on many shapes or forms. As applied to OOP, polymorphism refers to method names that have identical names but different implementations, depending on the situation. For example, radio buttons, check boxes, and list boxes each has a `Select` method. In each case, the `Select` method operates appropriately for its class.

When a derived class overrides a method of its base class, both methods have the same signature (name plus parameter list). But in each case, the actions performed are appropriate for the class. For example, a Person class might have a `Print` method that prints an address label with name and address information. But the `Print` method of the Employee class, which overrides the `Print` method of the Person class, might display the employee's information, including hire date and supervisor name, on the screen.

Polymorphism also allows a single class to have more than one method with the same name but a different argument list. The method is said to be *overloaded*. When an overloaded method is called, the argument type determines which version of the method to use. Each of the identically named methods performs its tasks in a slightly different way from the other methods.

Reusable Objects

A big advantage of object-oriented programming over traditional programming is the ability to reuse objects. When you create a new class by writing a class module, you can then use that class in multiple projects. Each object that you create from the class has its own set of properties. This process works just like the built-in VB controls you have been using all along. For example, you can create two PictureBox objects: imageOnePictureBox and imageTwoPictureBox. Each has its own Visible property and Image property, which will probably be set differently for each one.

The building-block concept can streamline programming. Consider a large corporation such as Microsoft, with many different programming teams. Perhaps one team develops the Word product and another team works on Excel. What happened when the Word team decided to incorporate formulas in tables? Do you think they wrote all new code to process the formulas? Likewise, there was a point when the Excel team added spell checking to worksheets. Do you think that they had to rewrite the spell-checking code? Obviously, it makes more sense to have a spell-checking object that can be used by any application and a calculation object that processes formulas in any application where needed.

Developing applications should be like building objects with Lego blocks. The blocks all fit together and can be used to build many different things.

Multitier Applications

A common use of classes is to create applications in multiple "tiers" or layers. Each of the functions of a **multitier application** can be coded in a separate component and the components may be stored and run on different machines.

One of the most popular approaches is a three-tier application. The tiers in this model are the presentation (or user interface) tier, business services tier, and data tier (Figure 2.3). You also may hear the term "*n*-tier" application, which is an expansion of the three-tier model. The middle tier, which contains all of the business logic or **business rules**, may be written in multiple classes that can be stored and run from multiple locations.

In a multitier application, the goal is to create components that can be combined and replaced. If one part of an application needs to change, such as a redesign of the user interface or a new database format, the other components do not need to be replaced. A developer can simply "plug in" a new user interface and continue using the rest of the components of the application.

The **presentation tier** refers to the user interface, which in a Windows application is the form. Consider that, in the future, the user interface could be completely redesigned or even converted to a Web page.

Figure 2.3

The three-tier model for application design.

Presentation Tier	Business Services Tier	Data Tier
User Interface Forms, controls, menus	Business Objects Validation Calculations Business logic Business rules	Data Retrieval Data storage

The **business services tier** is a class or classes that manipulate the data. This layer can include validation to enforce business rules as well as the calculations. If the validation and calculations are built into the form, then modifying the user interface may require a complete rewrite of a working application.

The **data tier** includes retrieving and storing the data in a database or other data store. Occasionally an organization will decide to change database vendors or need to retrieve data from several different sources. The data tier retrieves the data and passes the results to the business services tier, or takes data from the business services tier and writes them in the appropriate location.

Programmers must plan ahead for reusability in today's environment. You may develop the business services tier for a Windows application. Later the company may decide to deliver the application via the Web or a mobile device, such as a cell phone or palm device. The user interface must change, but the processing shouldn't have to change. If you develop your application with classes that perform the business logic, you can develop an application for one interface and easily move it to another platform.

Feedback 2.1

1. Name at least three types of operations that belong in the business services tier.
2. List as many operations that you can think of that belong in the presentation tier.

Creating Classes

You most likely learned to create classes in your introductory course. It's time to review the techniques and to delve deeper into the concepts. If you are comfortable with creating new classes, writing property procedures including read-only properties, and using a parameterized constructor, you may want to skip over the next few sections and begin with "A Basic Business Class."

Designing Your Own Class

To design your own class, you need to analyze the characteristics and behaviors that your object needs. The characteristics or properties are defined as variables, and the behaviors (methods) are sub procedures or function procedures.

Creating Properties in a Class

Inside your class you define private variables, which contain the values for the properties of the class. Theoretically, you could declare all variables as `Public` so that all other classes could set and retrieve their values. However, this approach violates the rules of encapsulation that require each object to be in charge of its own data. Remember that you use encapsulation to implement data hiding. To accomplish encapsulation, you will declare all variables in a class as `Private`. As a private variable, the value is available only to the procedures within the class, the same way that private module-level variables in a form are available only to procedures within the form's class.

When your program creates objects from your class, you will need to assign values to the properties. Because the properties are private variables, you will

use special property procedures to pass the values to the class module and to return values from the class module.

Property Procedures

The way that your class allows its properties to be accessed is through **property procedures**. A property procedure may contain a `Get` to retrieve a property value and/or a `Set` to assign a value to the property. The name that you use for the `Property` procedure becomes the name of the property to the outside world. Create "friendly" property names that describe the property without using a data type, such as LastName or EmployeeNumber.

The Property Procedure—General Form

General Form

```
Private ClassVariable As DataType  ' Declared at the module level.

[Public] Property PropertyName() As DataType
    Get
        PropertyName = ClassVariable
      or
        Return ClassVariable
    End Get

    Set(ByVal Value As DataType)

        [Statements, such as validation]
        ClassVariable = Value
    End Set
End Property
```

The `Set` statement uses the **`Value` keyword** to refer to the incoming value for the property. Property procedures are public by default, so you can omit the optional `Public` keyword. `Get` blocks are similar to function procedures in at least one respect: Somewhere inside the procedure, before the `End Get`, you must assign a return value to the procedure name or use a `Return` statement. The data type of the incoming value for a `Set` must match the type of the return value of the corresponding `Get`.

The Property Procedure—Example

Example

```
Private LastNameString As String  ' Declared at the module level.

Public Property LastName() As String
    Get
        Return LastNameString
        ' Alternate version:
        ' LastName = LastNameString
    End Get

    Set(ByVal Value As String)
        LastNameString = Value
    End Set
End Property
```

Remember, the private module-level variable holds the value of the property. The `Property Get` and `Set` retrieve the current value and assign a new value to the property.

Read-Only and Write-Only Properties

In some instances, you may wish to set a value for a property that can only be retrieved by an object but not changed. To create a read-only property, use the **`ReadOnly`** modifier and write only the `Get` portion of the property procedure. Security recommendations are to not include a `Set` procedure unless one is needed for your application or for class inheritance.


```
Private PayDecimal As Decimal    ' Declared at the module level

Public ReadOnly Property Pay() As Decimal    ' Make the property read-only.
    Get
        Return PayDecimal
    End Get
End Property
```

A write-only property is one that can be set but not returned. Use the **`WriteOnly`** modifier and write only the `Set` portion of the property procedure:

```
Private PasswordString As String    ' Declared at the module level.

Public WriteOnly Property Password() As String ' Make it write-only.
    Set(ByVal Valve As String)
        PasswordString = Value
    End Set
End Property
```

Constructors and Destructors

A **constructor** is a method that automatically executes when a class is instantiated. A **destructor** is a method that automatically executes when an object is destroyed. In VB, the constructor must be a procedure named `New`. The destructor must be named `Dispose` and must override the `Dispose` method of the base class. You will generally write constructors for your classes, but usually not destructors. Most of the time the `Dispose` method of the base class handles the class destruction very well.

You create a constructor for your class by writing a `Sub New` procedure. The constructor executes automatically when you instantiate an object of the class. Because the constructor method executes before any other code in the class, the constructor is an ideal location for any initialization tasks that you need to do, such as opening a database connection.

The `Sub New` procedure must be `Public` or `Protected` because the objects that you create must execute this method. Remember that the default is `Public`.

```
Sub New()
    ' Constructor for class.

    ' Initialization statements.
End Sub
```

Overloading the Constructor

Recall that ***overloading*** means that two methods have the same name but a different list of arguments (the signature). You can create overloaded methods in your class by giving the same name to multiple procedures, each with a different argument list. The following example shows an empty constructor (one without arguments) and a constructor that passes arguments to the class.

```
' Constructors in the Payroll class.

Sub New()
    ' Constructor with empty argument list.
End Sub

Sub New(ByVal HoursInDecimal As Decimal, ByVal RateInDecimal As Decimal)
    ' Constructor that passes arguments.

    ' Assign incoming values to private variables.
    HoursDecimal = HoursInDecimal
    RateDecimal = RateInDecimal
End Sub
```

Note: It isn't necessary to include the `ByVal` modifier to arguments since `ByVal` is the default. The editor adds `ByVal` to the arguments if you leave it out.

A Parameterized Constructor

The term ***parameterized constructor*** refers to a constructor that requires arguments. This popular technique allows you to pass arguments/properties as you create the new object. In the preceding example, the Payroll class requires two decimal arguments: the hours and the rate. By instantiating the Payroll object in a `Try / Catch` block, you can catch any missing input value as well as any nonnumeric input.

```
' Code in the Form class to instantiate an object of the Payroll class.

Try
    Dim APayroll As New Payroll( _
      Decimal.Parse(HoursTextBox.Text), Decimal.Parse(RateTextBox.Text))

Catch Err As Exception
    MessageBox.Show("Enter the hours and rate.", "Payroll")
End Try
```

Assigning Arguments to Properties

As a further improvement to the Payroll parameterized constructor, we will use the property procedures to assign initial property values. Within the class module, use the `Me` keyword to refer to the current class. So `Me.Hours` refers to the Hours property of the current class. `HoursInDecimal` refers to the class-level variable. Assigning the passed argument to the property name is preferable to just assigning the passed argument to the module-level property variable since validation is performed in the `Property Set` procedures.

```
' Improved constructor for the Payroll class.
Sub New(ByVal HoursInDecimal As Decimal, ByVal RateInDecimal As Decimal)
    ' Assign arguments to properties.
```

```
    With Me
        .Hours = HoursInDecimal
        .Rate = RateInDecimal
    End With
End Sub
```

When your class has both an empty constructor and a parameterized constructor, the program that creates the object can choose which method to use.

A Basic Business Class

The following example creates a very simplistic payroll application in two tiers (Figure 2.4). The application does not have a data tier since it doesn't have any database element.

Figure 2.4

Create a nondatabase project in two tiers.

Presentation Tier	Business Services Tier
User Interface PayrollForm Controls Menus	Business Objects Validation Calculations Business logic Business rules

This first version of the payroll application inputs hours and rate from the user, validates for numeric data and some business rules, calculates the pay, and displays the pay on the form. We must analyze the tasks that belong in the presentation tier and those that belong in the business services tier (Figure 2.5).

The Presentation Tier

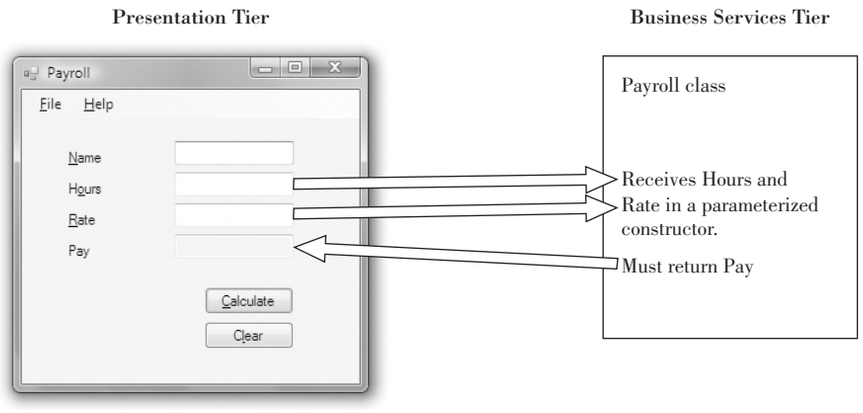
The presentation tier, also called the *user interface*, must handle all communication with the user. The user enters input data and clicks the *Calculate* button. The result of the calculation and any error messages to the user must come from the presentation tier. Generally, validation for numeric input is handled in the form, but validation for business rules is handled in the business services tier.

The Business Services Tier

Looking at Figure 2.5, you can see what should go in the class for the business services tier. The class needs private property variables for Hours, Rate, and

Figure 2.5

The form is the user interface; the validation and calculations are performed in the Payroll class, which is the business services tier.



Pay. It also needs a parameterized constructor to pass the arguments, property procedures to validate and set the Hours and Rate, and a read-only property to allow a Payroll object to retrieve the calculated pay.

The property procedures will include code to validate the input Hours and Rate for business rules. At this point, company policy is that the number of hours must be in the range 0–60 and the pay rate must be at least 6.25 and no more than 50. If the input values for Hours or Rate are outside of the acceptable range, the class will throw an exception that can be caught in the form's code. Remember that all user interaction, including any error messages, should occur in the presentation tier (the form).

Note: Throwing exceptions is covered in the section that follows the class code.

The Payroll Class

```
'Project:      Ch02PayrollApplication
'Module:       Payroll Class
'Programmer:   Bradley/Millspaugh
'Date:         June 2009
'Description:  Business services tier for payroll calculation: validates
               input data and calculates the pay.

Public Class Payroll

    ' Private class variables.
    Private HoursDecimal As Decimal  ' Hold the Hours property.
    Private RateDecimal As Decimal   ' Hold the Rate property.
    Private PayDecimal As Decimal    ' Hold the Pay property.

    ' Constants.
    Private Const MINIMUM_WAGE_Decimal As Decimal = 6.25D
    Private Const MAXIMUM_WAGE_Decimal As Decimal = 50D
    Private Const MINIMUM_HOURS_Decimal As Decimal = 0D
    Private Const MAXIMUM_HOURS_Decimal As Decimal = 60D
    Private Const REGULAR_HOURS_Decimal As Decimal = 40D
    Private Const OVERTIME_RATE_Decimal As Decimal = 1.5D

    ' Constructor.
    Sub New(ByVal HoursInDecimal As Decimal, ByVal RateInDecimal As Decimal)
        ' Assign properties and calculate the pay.

        Hours = HoursInDecimal
        Rate = RateInDecimal
        FindPay()
    End Sub

    Private Sub FindPay()
        ' Calculate the pay.
        Dim OvertimeHoursDecimal As Decimal

        If HoursDecimal <= REGULAR_HOURS_Decimal Then  ' No overtime.
            PayDecimal = HoursDecimal * RateDecimal
            OvertimeHoursDecimal = 0D
        Else                                           ' Overtime.
            OvertimeHoursDecimal = HoursDecimal - REGULAR_HOURS_Decimal
            PayDecimal = (REGULAR_HOURS_Decimal * RateDecimal) + _
              (OvertimeHoursDecimal * OVERTIME_RATE_Decimal * RateDecimal)
        End If
    End Sub
```

```
    ' Property procedures.
    Public Property Hours() As Decimal
        Get
            Return HoursDecimal
        End Get
        Set(ByVal Value As Decimal)
            If Value >= MINIMUM_HOURS_Decimal And _
              Value <= MAXIMUM_HOURS_Decimal Then
                HoursDecimal = Value
            Else
                Dim Ex As New ApplicationException( _
                  "Hours are outside of the acceptable range.")
                Ex.Source = "Hours"
                Throw Ex
            End If
        End Set
    End Property

    Public Property Rate() As Decimal
        Get
            Return RateDecimal
        End Get
        Set(ByVal Value As Decimal)
            If Value >= MINIMUM_WAGE_Decimal And _
              Value <= MAXIMUM_WAGE_Decimal Then
                RateDecimal = Value
            Else
                Throw New ApplicationException( _
                  "Pay rate is outside of the acceptable range.")
            End If
        End Set
    End Property

    Public ReadOnly Property Pay() As Decimal
        Get
            Return PayDecimal
        End Get
    End Property
End Class
```

Throwing and Catching Exceptions

The system throws an exception when an error occurs. Your program can catch the exception and take some action, or even ignore the exception. Your own class also can **throw an exception** to indicate that an error occurred, which generally is the best way to pass an error message back to the user interface. You can enclose any code that could cause an exception in a `Try / Catch` block.

```
' Code in the form's class.
Try
    Dim APayroll As New Payroll( _
      Decimal.Parse(HoursTextBox.Text), Decimal.Parse(RateTextBox.Text))
Catch Err As ApplicationException
    ' Display a message to the user.
    MessageBox.Show(Err.Message)
End Try
```

Note: If you are not familiar with structured exception handling using a `Try / Catch` block, see Appendix B.

What Exception to Throw?

The .NET Framework has several exception classes that you can use, or you can create your own new exception class that inherits from one of the existing classes. However, the system-defined exception classes can handle most every type of exception.

Microsoft recommends that you use the System.ApplicationException class when you throw your own exceptions from application code. System.ApplicationException has the same properties and methods as the System.Exception class, which is the generic system exception. All specific exceptions generated by the CLR inherit from System.Exception.

When you want to throw a generic application exception, use the **`Throw` statement** in this format:

```
Throw New ApplicationException("Error message to display.")
```

The message that you include becomes the Message property of the exception, which you can display when you catch the exception.

Passing Additional Information in an Exception

The constructor for the ApplicationException class takes only the error message as a parameter. But the class has additional properties that you can set and check. For example, you can set the Source property and the Data property, which can hold sets of key/value pairs.

In our Payroll class, we want to be able to indicate which field is in error, so that the code in the form can set the focus and select the text in the field in error. For this, we will use the exception's Source property. We must instantiate a new exception object, set the Source property, and then throw the exception:

```
Public Property Hours() As Decimal
    Get
        Return HoursDecimal
    End Get
    Set(ByVal Value As Decimal)
        If Value >= MINIMUM_HOURS_Decimal And _
          Value <= MAXIMUM_HOURS_Decimal Then
            HoursDecimal = Value
        Else
            Dim Ex As New ApplicationException( _
              "Hours are outside of the acceptable range.")
            Ex.Source = "Hours"
            Throw Ex
        End If
    End Set
End Property

Public Property Rate() As Decimal
    Get
        Return RateDecimal
    End Get
```

```
    Set(ByVal Value As Decimal)
        If Value >= MINIMUM_WAGE_Decimal And _
          Value <= MAXIMUM_WAGE_Decimal Then
            RateDecimal = Value
        Else
            Dim Ex As New ApplicationException( _
              "Pay rate is outside of the acceptable range.")
            Ex.Source = "Rate"
            Throw Ex
        End If
    End Set
End Property
```

Throwing Exceptions Up a Level

You should show messages to the user only in the user interface. At times, you may have several levels of components. For example, the form creates an object that calls code in another class. If an exception occurs in a class that does not have a user interface, you should pass the exception up to the next higher level—the component that called the current code. Use the `Throw` keyword to pass an exception to the form or other component that invoked the class.

```
Try
    ' Code that might cause an exception.
Catch Err As Exception
    Throw Err
End Try
```

Guidelines for Throwing Exceptions

When you throw exceptions, you should always include an error message. The message should be

- Descriptive.
- Grammatically correct, in a complete sentence with punctuation at the end.

Alternatives to Exception Handling

It takes considerable system resources to handle exceptions. You should use exception handling for situations that are errors and truly out of the ordinary. If an error occurs fairly often, you should look for another technique to handle it. However, Microsoft recommends throwing exceptions from components rather than returning an error code.

You can use another tool to help avoid generating parsing exceptions for invalid user input. You can use the **`TryParse` method** of the numeric classes instead of using `Parse`. The `TryParse` method sets the variable to zero and returns Boolean *false* if the parse fails, rather than throwing an exception.

The TryParse Method—General Form

General Form

```
DataType.TryParse(ValueToParse, NumericVariableToHoldResult)
```

The `TryParse` method converts the ValueToParse into an expression of the named data type, returns *true*, and places the result into the numeric variable, which should be declared before this statement. If the conversion fails, the numeric variable is set to zero and the method returns Boolean *false*.

The TryParse Method—Example

Example

```
Dim HoursDecimal As Decimal

Decimal.TryParse(HoursTextBox.Text, HoursDecimal)
If HoursDecimal > 0 Then
    ' Passed the conversion; perform calculations.
Else
    MessageBox.Show("Invalid data entered.")
End If

If Decimal.TryParse(HoursTextBox.Text, HoursDecimal Then
    ' HoursDecimal contains the converted value.
Else
    MessageBox.Show("Invalid data entered.")
End If
```

As you can see, this technique is preferable for numeric validation of user input since it does not throw an exception for nonnumeric data. Instead, bad input data are handled by the `Else` clause.

Modifying the User Interface to Validate at the Field Level

You can further improve the user interface in the payroll application by performing field-level validation. This technique displays a message directly on the form, next to the field in error, before the user moves to the next control. You can use an ErrorProvider component for the message, rather than a message box, which is a more up-to-date approach. You perform field-level validation for numeric data in the **Validating event** of each text box.

The Validating Event

As the user enters data in a text box and moves to another control, the events of the text box occur in this order:

Enter
GotFocus
Leave
Validating
Validated
LostFocus

Each control on the form has a CausesValidation property that is set to *true* by default. When the user finishes an entry and presses Tab or clicks on another control, the Validating event occurs for the control just left. That is, the event occurs if the CausesValidation property of the *new* control is *true*. You can leave the CausesValidation property of most controls set to *true* so that validation occurs. Set CausesValidation to *false* on a control such as *Cancel* or *Exit* to give the user a way to bypass the validation when canceling

the transaction. *Note*: The Validating event occurs only for a control that receives the focus; you also may need to perform form-level validation to determine that the user skipped a field entirely.

The Validating event handler is the preferred location for field-level validation. Here is the procedure header for a Validating event handler:

```
Private Sub RateTextBox_Validating(ByVal sender As Object, _
  ByVal e As System.ComponentModel.CancelEventArgs) _
  Handles RateTextBox.Validating
```

Canceling the Validating Event You can use the CancelEventArgs argument of the Validating event handler to cancel the Validating event and return focus to the control that is being validated.

```
e.Cancel = True
```

Canceling the event returns the focus to the text box, making the text box "sticky." The user is not allowed to leave the control until the input passes validation.

One note of caution: If you use the validating event on the field that receives focus when the form is displayed, and the validation requires an entry, the user will be unable to close the form without making a valid entry in the text box. To get around this problem, write an event handler for the form's FormClosing event and set `e.Cancel = False`.

```
Private Sub PayrollForm_FormClosing(ByVal sender As Object, _
  ByVal e As System.Windows.Forms.FormClosingEventArgs) _
  Handles Me.FormClosing
    ' Do not allow validation to cancel the form's closing.

    e.Cancel = False
End Sub
```

Controlling Validating Events You can get into trouble if you generate Validating events when you don't want them. For example, after an input value has passed the numeric checking, it may fail a business rule, such as not falling within an acceptable range of values. To display a message to the user, you will probably execute the `Focus` method of the text box in error. But the `Focus` method triggers a Validating event on the control most recently left, which is likely not the result that you want. You can suppress extra Validating events by temporarily turning off CausesValidation. You will see this technique used in the form's code in the "The Code for the Modified Form" section.

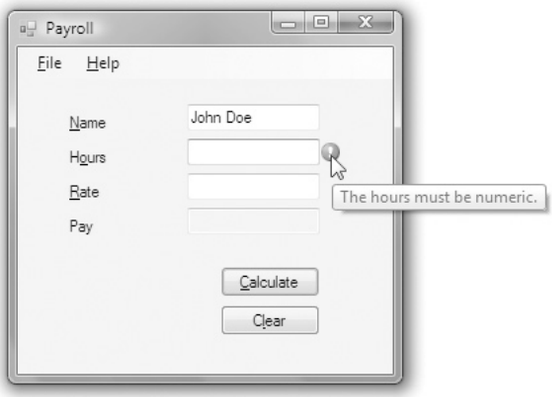
```
With .RateTextBox
    .SelectAll()
    .CausesValidation = False
    .Focus()
    .CausesValidation = True
End With
```

The ErrorProvider Component

Using an **ErrorProvider component**, you can make an error indicator appear next to the field in error, rather than pop up a message box. Generally, you use one ErrorProvider for all controls on a form. You add the ErrorProvider to the form's component tray at design time and set its properties in code. If the input data value is invalid, the ErrorProvider component can display a blinking icon next to the field in error and display a message in a popup, similar to a ToolTip (Figure 2.6).

Figure 2.6

The ErrorProvider displays a blinking icon next to the field in error. When the user points to the icon, the error message appears in a popup.



The ErrorProvider SetError Method—General Form

You turn on the blinking error indicator and error message with the ErrorProvider's `SetError` method.

General Form

```
ErrorProviderObject.SetError(ControlName, MessageString)
```

The ErrorProvider SetError Method—Examples

Examples

```
ErrorProvider1.SetError(QuantityTextBox, "Quantity must be numeric.")
ErrorProvider1.SetError(CreditCardTextBox, "Required field.")
```

You can replace message boxes with ErrorProviders in most any program without changing the logic of the program.

Turning Off the Error Indicator You must clear the ErrorProvider after the error is corrected. Use the ErrorProvider's `Clear` method to turn off the error indicator.

```
ErrorProvider1.Clear()
```

In a button's Click event handler, the best approach is to clear the ErrorProvider at the top of the procedure and turn it on anywhere that a value fails validation.

```
Private Sub CalcuateButton_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles CalculateButton.Click
    ' Validate and perform calculations.
    Dim HoursDecimal As Decimal

    ' Check for valid input data.
    ErrorProvider1.Clear()
    If Decimal.TryParse(HoursTextBox.Text, HoursDecimal) Then
        ' Perform any calculations with good data.
    Else
        ' Hours did not pass validation.
        ErrorProvider1.SetError(HoursTextBox, _
          "The hours must be numeric.")
    End If
End Sub
```

In a Validating event handler, the most common technique is to use an `If` statement and turn the ErrorProvider on or off.

```
Private Sub HoursTextBox_Validating(ByVal sender As Object, _
  ByVal e As System.ComponentModel.CancelEventArgs) _
  Handles HoursTextBox.Validating
    ' Test hours for numeric.
    Dim HoursDecimal As Decimal

    If Decimal.TryParse(HoursTextBox.Text, HoursDecimal) Then
        ErrorProvider1.Clear()
    Else
        ErrorProvider1.SetError(HoursTextBox, _
            "The hours must be numeric.")
        HoursTextBox.SelectAll()
        e.Cancel = True
    End If
End Sub
```

The Code for the Modified Form

Here is the code for the modified form, using the `TryParse` and field-level validation in the Validating event handlers of the text boxes.

```
'Project:      Ch02PayrollApplication
'Module:       Payroll Form
'Programmer:   Bradley/Millspaugh
'Date:         June 2009
'Description:  User interface for payroll application.
'              Provides data entry and validates for nonnumeric data.

Public Class PayrollForm
    Private Sub CalcuateButton_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles CalculateButton.Click
        ' Create a Payroll object to connect to the business services tier.
        Dim HoursDecimal As Decimal
        Dim RateDecimal As Decimal

        ' Check for valid input data.
        ErrorProvider1.Clear()
```

```
        If Decimal.TryParse(HoursTextBox.Text, HoursDecimal) Then
            If Decimal.TryParse(RateTextBox.Text, RateDecimal) Then

                ' Both values converted successfully.
                Try
                    Dim APayroll As New Payroll(HoursDecimal, RateDecimal)
                    PayTextBox.Text = APayroll.Pay.ToString("C")

                Catch Err As ApplicationException
                    ' Catch exceptions from the Payroll class.
                    Select Case Err.Source
                        Case "Hours"
                            ErrorProvider1.SetError(HoursTextBox, _
                              Err.Message)
                            With HoursTextBox
                                .SelectAll()
                                .Focus()
                            End With
                        Case "Rate"
                            ErrorProvider1.SetError(RateTextBox, _
                              Err.Message)
                            With RateTextBox
                                .SelectAll()
                                .Focus()
                            End With
                    End Select
                End Try
            Else
                ' Rate did not pass validation.
                ErrorProvider1.SetError(RateTextBox, _
                  "The rate must be numeric.")
            End If
        Else
            ' Hours did not pass validation.
            ErrorProvider1.SetError(HoursTextBox, _
              "The hours must be numeric.")
        End If
    End Sub

    Private Sub ClearButton_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles ClearButton.Click
        ' Clear the screen fields.

        ErrorProvider1.Clear()
        With NameTextBox
            .Clear()
            .Focus()
        End With
        HoursTextBox.Clear()
        RateTextBox.Clear()
        PayTextBox.Clear()
    End Sub

    Private Sub HoursTextBox_Validating(ByVal sender As Object, _
      ByVal e As System.ComponentModel.CancelEventArgs) _
      Handles HoursTextBox.Validating
        ' Test hours for numeric.
        Dim HoursDecimal As Decimal
```

```
        If Decimal.TryParse(HoursTextBox.Text, HoursDecimal) Then
            ErrorProvider1.Clear()
        Else
            ErrorProvider1.SetError(HoursTextBox, _
              "The hours must be numeric.")
            HoursTextBox.SelectAll()
            e.Cancel = True
        End If
    End Sub

    Private Sub RateTextBox_Validating(ByVal sender As Object, _
      ByVal e As System.ComponentModel.CancelEventArgs) _
      Handles RateTextBox.Validating
        ' Test pay rate for numeric.
        Dim RateDecimal As Decimal

        If Decimal.TryParse(RateTextBox.Text, RateDecimal) Then
            ErrorProvider1.Clear()
        Else
            ErrorProvider1.SetError(RateTextBox, _
              "The hours must be numeric.")
            RateTextBox.SelectAll()
            e.Cancel = True
        End If
    End Sub

    Private Sub ExitToolStripMenuItem_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles ExitToolStripMenuItem.Click
        ' Close the program.

        Me.Close()
    End Sub

    Private Sub AboutToolStripMenuItem_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles AboutToolStripMenuItem.Click
        ' Show the About box.

        Dim AnAboutBox As New AboutBox1
        AnAboutBox.ShowDialog()
    End Sub

    Private Sub PayrollForm_FormClosing(ByVal sender As Object, _
      ByVal e As System.Windows.Forms.FormClosingEventArgs) _
      Handles Me.FormClosing
        ' Do not allow validation to prevent the form closing.

        e.Cancel = False
    End Sub
End Class
```

Modifying the Business Class

As business rules change, you can modify the business class or create a new class that inherits from the original class. You can usually add properties and methods to an existing class without harming any application that uses the class, but you should not change the behavior of existing properties and methods if any applications use the class.

In our Payroll example, we will expand the user interface to display a summary form. The summary form displays the number of employees processed, the total amount of pay, and the number of overtime hours. We must modify the Payroll class to calculate these values and return the values in read-only properties (Figure 2.7).

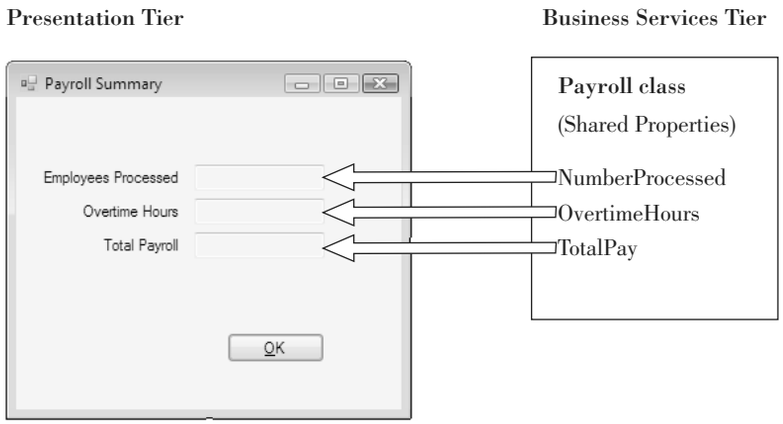


Figure 2.7

The new summary form displays summary information. The Payroll class must accumulate the summary figures in shared properties.

Instance Variables versus Shared Variables

Each new instance of the Payroll object has its own values for the hours, pay rate, and pay. These properties are called ***instance properties***, ***instance variables***, or ***instance members***. But the properties we are adding now, such as the number of employees processed and the total pay amount, must accumulate totals for all instances of the class. These properties are called ***shared properties***, ***shared variables***, or ***shared members***. Recall that properties are just the variables of a class, so the terms *properties* and *variables* can be used interchangeably.

The Payroll class requires three shared variables, one for each of the summary fields. As each instance of the Payroll class is created, the values are accumulated in the shared variables. In this way, the values for employee two are added to the values for employee one, and so on.

```
' Payroll Class.
' Shared properties declared at the module level.

' Hold the NumberProcessed shared property.
Private Shared NumberEmployeesInteger As Integer
' Hold the TotalPay shared property.
Private Shared TotalPayDecimal As Decimal
' Hold the OvertimeHours shared property.
Private Shared TotalOvertimeHoursDecimal As Decimal
```

Since these variables are Private to the class, public `Get` methods are required to make the properties accessible. You retrieve shared properties by using the class name such as Payroll.NumberProcessed or Payroll.OvertimeHours. This is the same concept that you use when converting input values: `Decimal.Parse()` calls the `Parse` method of the Decimal class.

```
Public Shared ReadOnly Property NumberProcessed() As Integer
    Get
        Return NumberEmployeesInteger
    End Get
End Property

Public Shared ReadOnly Property TotalPay() As Decimal
    Get
        Return TotalPayDecimal
    End Get
End Property

Public Shared ReadOnly Property OvertimeHours() As Decimal
    Get
        Return TotalOvertimeHoursDecimal
    End Get
End Property
```

The `FindPay` method must be modified to add to the summary fields:

```
' Payroll class.
' Additional module-level named constants.
Private Const REGULAR_HOURS_Decimal As Decimal = 40D
Private Const OVERTIME_RATE_Decimal As Decimal = 1.5D

Private Sub FindPay()
    ' Calculate the Pay.
    Dim OvertimeHoursDecimal As Decimal

    If HoursDecimal <= REGULAR_HOURS_Decimal Then        ' No overtime.
        PayDecimal = HoursDecimal * RateDecimal
        OvertimeHoursDecimal = 0D
    Else        ' Overtime.
        OvertimeHoursDecimal = HoursDecimal - REGULAR_HOURS_Decimal
        PayDecimal = (REGULAR_HOURS_Decimal * RateDecimal) + _
          (OvertimeHoursDecimal * OVERTIME_RATE_Decimal * RateDecimal)
    End If
    TotalOvertimeHoursDecimal += OvertimeHoursDecimal
    TotalPayDecimal += PayDecimal
    NumberEmployeesInteger += 1
End Sub
```

Following is the completed Payroll class that calculates and returns the shared properties:

```
'Project:      Ch02PayrollWithSummary
'Module:       Payroll Class
'Programmer:   Bradley/Millspaugh
'Date:         June 2009
'Description:  Business services tier for payroll calculation: validates input
'              data and calculates the pay, with overtime, regular, and
'              summary data.

Public Class Payroll

    ' Instance variables.
    Private HoursDecimal As Decimal     ' Hold the Hours property.
    Private RateDecimal As Decimal      ' Hold the Rate property.
    Private PayDecimal As Decimal       ' Hold the Pay property.
```

```
' Shared variables.
' Hold the NumberProcessed shared property.
Private Shared NumberEmployeesInteger As Integer
' Hold the TotalPay shared property.
Private Shared TotalPayDecimal As Decimal
' Hold the OvertimeHours shared property.
Private Shared TotalOvertimeHoursDecimal As Decimal

' Constants.
Private Const MINIMUM_WAGE_Decimal As Decimal = 6.25D
Private Const MAXIMUM_WAGE_Decimal As Decimal = 50D
Private Const MINIMUM_HOURS_Decimal As Decimal = 0D
Private Const MAXIMUM_HOURS_Decimal As Decimal = 60D
Private Const REGULAR_HOURS_Decimal As Decimal = 40D
Private Const OVERTIME_RATE_Decimal As Decimal = 1.5D

' Constructor.
Sub New(ByVal HoursInDecimal As Decimal, ByVal RateInDecimal As Decimal)
    ' Assign properties and calculate the pay.

    Me.Hours = HoursInDecimal
    Me.Rate = RateInDecimal
    FindPay()
End Sub

Private Sub FindPay()
    ' Calculate the pay.
    Dim OvertimeHoursDecimal As Decimal

    If HoursDecimal <= REGULAR_HOURS_Decimal Then   ' No overtime.
        PayDecimal = HoursDecimal * RateDecimal
        OvertimeHoursDecimal = 0D
    Else                                            ' Overtime.
        OvertimeHoursDecimal = HoursDecimal - REGULAR_HOURS_Decimal
        PayDecimal = (REGULAR_HOURS_Decimal * RateDecimal) + _
          (OvertimeHoursDecimal * OVERTIME_RATE_Decimal * RateDecimal)
    End If
    TotalOvertimeHoursDecimal += OvertimeHoursDecimal
    TotalPayDecimal += PayDecimal
    NumberEmployeesInteger += 1
End Sub

' Property procedures.
Public Property Hours() As Decimal
    Get
        Return HoursDecimal
    End Get
    Set(ByVal Value As Decimal)
        If Value >= MINIMUM_HOURS_Decimal And _
          Value <= MAXIMUM_HOURS_Decimal Then
            HoursDecimal = Value
        Else
            Dim Ex As New ApplicationException( _
              "Hours are outside of the acceptable range.")
            Ex.Source = "Hours"
            Throw Ex
        End If
    End Set
End Property
```

```
    Public Property Rate() As Decimal
        Get
            Return RateDecimal
        End Get
        Set(ByVal Value As Decimal)
            If Value >= MINIMUM_WAGE_Decimal And _
              Value <= MAXIMUM_WAGE_Decimal Then
                RateDecimal = Value
            Else
                Dim Ex As New ApplicationException( _
                  "Pay rate is outside of the acceptable range.")
                Ex.Source = "Rate"
                Throw Ex
            End If
        End Set
    End Property

    Public ReadOnly Property Pay() As Decimal
        Get
            Return PayDecimal
        End Get
    End Property

    Public Shared ReadOnly Property NumberProcessed() As Decimal
        Get
            Return NumberEmployeesInteger
        End Get
    End Property

    Public Shared ReadOnly Property TotalPay() As Decimal
        Get
            Return TotalPayDecimal
        End Get
    End Property

    Public Shared ReadOnly Property OvertimeHours() As Decimal
        Get
            Return TotalOvertimeHoursDecimal
        End Get
    End Property
End Class
```

Displaying the Summary Data

To display a second form from the main form, you can declare an instance of the form's class and show the form.

```
Dim ASummaryForm As New SummaryForm()
ASummaryForm.ShowDialog()
```

You also can take advantage of the default instance of a form and just show the default instance:

```
SummaryForm.ShowDialog()
```

You can choose from two techniques for filling the screen fields with the summary data:

1. Set the summary output from the Payroll form using the `Shared` methods of the Payroll class before showing the Summary form:

```
' In PayrollForm:
Private Sub SummaryButton_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles SummaryButton.Click
    ' Show the summary form.

    Dim ASummaryForm As New SummaryForm()

    With ASummaryForm
        .CountLabel.Text = Payroll.NumberProcessed.ToString()
        .OvertimeLabel.Text = Payroll.OvertimeHours.ToString("N1")
        .TotalPayLabel.Text = Payroll.TotalPay.ToString("C")
        .ShowDialog()
    End With
End Sub
```

2. Use the shared properties from the Payroll class in the Form_Load procedure of the Summary form and fill the labels there.

```
' In SummaryForm.
Private Sub SummaryForm_Load(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load
    ' Retrieve the summary values.

    CountLabel.Text = Payroll.NumberProcessed.ToString()
    OvertimeLabel.Text = Payroll.OvertimeHours.ToString("N1")
    TotalPayLabel.Text = Payroll.TotalPay.ToString("C")
End Sub
```

Although both of these techniques work perfectly well, the second method is preferable for encapsulating the forms' data. Each of the forms in the project can access the shared properties, which is preferable to having PayrollForm access the controls on SummaryForm.

Feedback 2.2

1. What is the purpose of property procedures?
2. Why should the variables for the properties of a class be declared as `Private`?
3. You want to create a new class called Student that inherits from Person. Properties required to create an instance of the class are LastName, FirstName, and BirthDate. Write a parameterized constructor for the class.
4. Write the statement(s) to create an instance of the Student class defined in the previous question. Supply the arguments for the parameterized constructor.
5. An error occurs in a class written for the business services tier. Explain how to handle the error condition and how the user should be notified.

Namespaces, Scope, and Accessibility

This section is intended as a review of the topics of scope and visibility of variables, constants, and classes. You may want to skip this section if you feel comfortable with declaring and using namespaces, scope, lifetime, and accessibility domains such as Public, Private, Protected, and Friend.

Namespaces

Namespaces are used for grouping and referring to classes and structures. A class or structure name must be unique in any one namespace. You can think of namespaces like telephone area codes; a given phone number can exist only once in a single area, but that number may appear in many different area codes.

The classes in a namespace do not have to be in a single file. In fact, most of the classes in the .NET Framework are in the System namespace, which is stored in many files.

You can declare namespaces in your VB projects. In fact, by default each project has a namespace that matches the project name. If you display the Project Designer for any project, you will see an entry titled *Root Namespace*. However, if you change the project name in the Solution Explorer, the root namespace does not change automatically. Declare namespaces within your project using the `Namespace / End Namespace` construct:

```
Namespace RnRApplications
    ' Classes and structures in the namespace can appear here.
End Namespace
```

You can place the same `Namespace` statement in more than one project.

For most projects, there is no advantage in declaring a namespace. A company might choose to group applications by using namespaces.

Scope

The **scope** of a variable or constant refers to the area of the program that can "see" and reference it. For simplicity and clarity, we use the term *variable*, but each of the following examples applies to named constants as well as variables.

You determine the scope of a variable by the location of the declaration and the accessibility modifier (`Public` or `Private`). The choices for scope, from the widest to the narrowest, are namespace, module level, procedure level, and block level.

Namespace

Any variable, constant, class, or structure declared with the `Public` modifier has **namespace scope**. You can refer to the identifier anywhere within the namespace. Because each project is in its own namespace by default, generally *namespace scope* also means *project scope*. However, as you know, you can structure your own namespaces to contain multiple projects.

You usually need to declare classes and structures as Public, but not variables and constants. It is considered poor OOP programming to declare variables with namespace scope because it violates the rules of encapsulation.

Each class should be in charge of its own data and share variables only by using `Property Set` and `Get` procedures.

Note: Earlier versions of VB, as well as many other programming languages, refer to variables that can be referenced from any location in a project as *global variables*. VB has dropped this terminology.

Module Level

Module-level scope is sometimes also called *class-level* scope. A module-level variable is a Private variable that is declared inside any class, structure, or module but outside of any sub procedure or function. By convention, you should declare module-level variables at the top of the class, but the variables can actually be declared anywhere inside the class that is outside of a procedure or function.

```
Private TotalDecimal As Decimal
```

Note: If you leave off the accessibility modifier (`Public` or `Private`), the variable is Private by default.

In some previous versions of Visual Basic, each file was called a module, so any variable declared as Private at the top of the file (not inside a sub procedure or function) was a module-level variable. The terminology carries through to the current version of VB, even though the language now has a `Module / End Module` construct, which can contain miscellaneous procedures and functions that are not included in a class.

Procedure Level

Any variable that you declare inside a procedure or function, but not within a block, has **procedure-level scope**, also called *local scope*. You can reference the variable anywhere inside the procedure but not in other procedures. Note that the `Public` keyword is not legal inside a procedure; all procedure-level variables are private and are declared with the `Dim` keyword.

Block Level

If you declare a variable inside a code block, the variable has **block-level scope**. That is, the variable can be referenced only inside that block. Code blocks include

```
If / End If
Do / Loop
For / Next
Select Case / End Select
Try / Catch / Finally / End Try
```

The blocks that are likely to cause confusion are the `Try / Catch / Finally / End Try`. The `Try` is one block; each `Catch` is a separate block; and the `Finally` is a separate block. This means that you cannot declare a variable in the `Try` and reference it in the `Catch` or the `Finally` blocks. It also means that you can declare the same variable name for each `Catch` since the scope of each is only that `Catch` block.

```
Try
    ' Declare a block-level variable.
    ' Bad idea, since it cannot be referenced outside of this Try block.
    Dim AmountDecimal As Decimal = Decimal.Parse(AmountTextBox.Text)
Catch Err As InvalidCastException
    ' Err is a block-level variable valid only inside this Catch block.
    MessageBox.Show(Err.Message, "Invalid Input Data.")
Catch Err As Exception
    ' Err is a block-level variable valid only inside this Catch block.
    MessageBox.Show(Err.Message, "Unknown Error.")
Finally
    ' Any variable declared here is valid only inside this Finally block.
End Try
```

When you instantiate objects, if there is any chance the creation will fail, you should create the new object inside a `Try/Catch` block. But if you declare the variable inside the `Try` block, the variable goes out of scope when the `Try` block completes. Therefore, most of the time you will declare the object variable at the module level or procedure level and instantiate the object inside the `Try` block.

```
' Declare the object variable at the module level.
Private APayroll As Payroll

Private Sub CalculateButton_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles CalculateButton.Click
    ' Create a Payroll object to connect to the business services tier.

    Try
        ' Instantiate the object in the Try block.
        APayroll = New Payroll(Decimal.Parse(Me.HoursTextBox.Text), _
          Decimal.Parse(Me.RateTextBox.Text))
    Catch . . .
```

Lifetime

The **lifetime** of a variable, including object variables, is as long as the variable remains in scope. The lifetime of a namespace-level variable is as long as the program is running. The lifetime of a module-level variable is as long as any reference to the class remains, which is generally as long as the program runs.

The lifetime of a procedure-level variable is one execution of the procedure. Each time the procedure is executed, a new variable is established and initialized. For this reason, you cannot use procedure-level variables to maintain running totals or counts unless you declare them with the `Static` keyword, which changes the lifetime of a procedure-level variable to the life of the class or module.

Accessibility Domains

You have already declared variables and classes with the `Public` and `Private` keywords. You also can use `Protected`, `Friend`, and `Protected Friend` (Table 2.1). Each of these keywords defines the **accessibility** of the variable or class.

Keywords to Declare Accessibility Domains Table 2.1

Keyword	Description
`Public`	Accessible from anywhere in the project or from any other project that references this one.
`Private`	Accessible from anywhere inside this class.
`Protected`	Accessible from anywhere inside this class or in any class that inherits from this class.
`Friend`	Accessible from anywhere inside this project/assembly.
`Protected Friend`	A combination of `Protected` and `Friend`. Accessible from anywhere inside this project/assembly and in any class that inherits from this class, even though the derived class is in a different project/assembly.

Creating Classes That Inherit

To create a class that inherits, you should first add a new class to the project. Although a single file can hold multiple class definitions, the recommended approach is to create a new file for each Public class and make the name of the file match the class name. The only exceptions are small "helper classes" that would never be used by any other application. These helper classes should be declared with the `Friend` keyword because they are used only in the current project.

Adding a New Class File

You can add a new file for a class by selecting *Project / Add Class* or *Add Component*. The difference between the two is that a component has a visual designer and a class file does not. In Chapter 3 you will use the component to add database elements to a new class. In this chapter, select *Add Class*. Both options create a new file with the extension .vb. Make sure to give the class the name that you want to use; the file will be named correctly and the solution and project will be set up with the correct name.

The newly added class will have the first and last lines of code:

```
Public Class ClassName
End Class
```

Add the `Inherits` clause on the first line following the `Class` declaration and add comments above the `Class` statement.

```
'Project:      Ch02PayrollInheritance
'Module:       PayrollSalaried Class
'Programmer:   Bradley/Millspaugh
'Date:         June 2009
'Description:  A class in the business services tier for payroll calculation:
'              validates input data and calculates the pay for
'              salaried employees.

Public Class PayrollSalaried
    Inherits Payroll
End Class
```

Creating a Constructor

A subclass must have its own constructor because constructors are not inherited. However, if you do not create a constructor (a Sub New), VS creates an implicit empty constructor.

The first statement in a constructor of an inherited class should call the constructor of the base class using the `MyBase` keyword:

```
MyBase.New()
```

If the base class has only a parameterized constructor, you must pass arguments to the constructor.

```
MyBase.New(HoursDecimal, RateDecimal)
```

And just like the base class, you can have several overloaded New constructors, one for each signature that the base class has.

Inheriting Variables and Methods

As you know, when you derive a new class from an existing class, all Public and Protected variables and methods are inherited, with the exception of the base class's constructors.

Shadowing and Overriding Methods

An inherited class can have a method with the same name as a method in its base class. Depending on how it is declared, the new method may shadow or override the base class method.

Overriding To override a method in the base class, the method must be declared as **overridable**:

```
' Base Class.
Public|Protected Overridable Sub DoSomething()
```

In the derived class, you must use the `Overrides` keyword and have the same accessibility (`Public|Private`) the base class has:

```
' Derived Class.
Public|Protected Overrides Sub DoSomething()
```

If the base-class method has more than one signature (overloaded methods), the override applies only to the base-class method with the identical signature. You must write separate methods to override each version (signature) of the base-class method.

Shadowing A method in a derived class can **shadow** a method in the base class. The new (shadowing) method replaces the base-class method in the derived class but not in any new classes derived from that class. The shadowing method "hides" all signatures (overloaded methods) with the same name in the base class.

```
' Base Class.
Public|Protected [Overridable] Sub DoSomething()
```

In the derived class, you can use the `Shadows` keyword:

```
' Derived Class.
Public|Protected Shadows Sub DoSomething()
```

If you do not use either the `Overrides` or `Shadows` keyword, `Shadows` is assumed. And if you use the `Overrides` or `Shadows` keyword for one method of a group, you must include the keyword for all overridden or shadowed methods.

Using Properties and Methods of the Base Class

You can reference any Public property or method of the base class from the subclass. If the base-class method has not been overridden or shadowed in the subclass, you can call the method directly:

```
' Base class.
Public Function FindPay()
    ' Code to calculate the pay.
End Function

' Sub class.
' Call the FindPay function from the base class.
FindPay()
```

If the subclass also has a `FindPay` function, you can call the function in the base class by including the `MyBase` keyword:

```
MyBase.FindPay()
```

It is legal to use the `MyBase` keyword even when it isn't required, which can make your program more understandable. For example, assuming that the subclass does not have a `FindPay` function, you can still call the base-class function with

```
MyBase.FindPay()
```

You can use the same rules for accessing Public properties of the base class. You can reference the property directly or add the `MyBase` keyword, which aids in readability.

```
' Assign a value to a read/write Public property of the base class.
Hours = HoursDecimal
```

or

```
MyBase.Hours = HoursDecimal
```

You can use the `Me` keyword to refer to a property or method of the current class to clarify the code.

```
' Sub class.

Sub New(ByVal LevelInteger As Integer)
    ' Constructor of the sub class.

    MyBase.New()
    Me.SalaryLevel = LevelInteger
    Me.FindPay()
    MyBase.AddEmployee()
End Sub
```

Note: You can find the complete inheritance example on the text Web site (www.mhhe.com/AdvVB2008/) as Ch02PayrollWithInheritance.

Passing Control Properties to a Component

So far in this chapter, all examples pass the Text property of text boxes to the business services tier component. But often you need to pass data from check boxes, radio buttons, or list boxes. How you pass the data depends on how the properties are declared in the business class.

The examples in this section are based on a two-tier application to calculate prices for theater tickets (Figure 2.8). Seat prices vary by the section: General, Balcony, or Box Seats. Seniors and students receive a $5.00 discount from the ticket price.

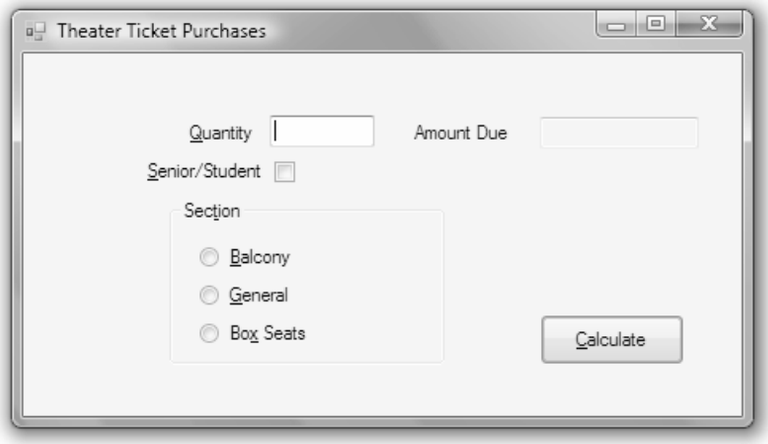


Figure 2.8

In the user interface, the user makes selections in radio buttons and a check box, which must be used to set properties in the business-services-tier component.

The business services tier needs to know the section, the number of tickets, and whether a discount is to be given. Therefore, the constructor will receive three values:

```
Sub New(ByVal QuantityInteger As Integer, ByVal SectionInteger As Integer, _
  ByVal DiscountBoolean As Boolean)
```

Notice that there are three values: the quantity, the section, and a Boolean value for the discount. Passing the quantity is straightforward; you can convert the text box value to integer: `Integer.Parse(QuantityTextBox.Text)`. And you can easily pass the Checked property of a check box to a Boolean property:

```
Dim ATicketPrice As New TicketPrice(Integer.Parse(Me.QuantityTextBox.Text), _
  SectionInteger, DiscountCheckBox.Checked)
```

Setting a property based on a selection in radio buttons or a list box presents an additional challenge, both in determining the best way to set up the property in the business-services-tier component and in setting the correct value in the user interface. Notice that the Section property is declared as integer. Although you could set up the property as string, there is a real advantage in using integer—you can create an enumeration for the available choices.

Creating an Enumeration

Whenever you have a list of choices for a property, it's because someone set up an **enumeration** that lists the choices. For example, selecting `Color.Red`, `Color.Blue`, or `Color.Yellow` is choosing one of the elements from the Color enumeration. When you choose one of the elements of the Color enumeration, the VB compiler actually substitutes the numeric value of the element. This saves you, the developer, from having to remember either the color names or the color numbers. You just type the name of the enumeration and a period, and the possible choices pop up in IntelliSense.

You can create your own enumeration, which is called an ***enum*** ("E-noom"). An enum is a list of named constants. The data type of the constants must be one of the integer types (integer, short, long, or byte). Whenever you create a reusable component class that has a list of possible choices for a property, consider setting up an enum.

The Enum Statement—General Form

General Form

```
Enum EnumName
    ConstantName1 [ConstantValue]
    ConstantName2 [ConstantValue]
    . . .
End Enum
```

The `Enum` statement belongs at the namespace level or class level, which means that it cannot appear inside a procedure. By default, an `Enum` is public, but you can declare it to be private, friend, or protected, if you wish.

The Enum Statement—Examples

Examples

```
Public Enum SectionType
    General
    Balcony
    Box
End Enum

Enum ReportType
    BooksBySubject  10
    BooksByAuthor
End Enum

Enum EvenNumbers
    Two    2
    Four   4
    Six    6
    Eight  8
End Enum
```

When you don't assign a constant value to the element, VB automatically assigns the first element a value of zero, and each following element one greater than the last. So, in the first of the examples above, *General* has a constant value of 0, *Balcony* has a value of 1, and *Box* has a value of 2. If you assign one element, as in the second example above for ReportType, each following element is assigned one greater than the last. So, in the ReportType example, BooksBySubject has a constant value of 10, which you assigned, and BooksByAuthor has a value of 11.

In the business-services-tier component for the program example, which you can see in Ch02EnumRadioButtons, the Section property is set up as an integer with an enum. In the CalculatePrice procedure, use the enum values in a `Select Case` to determine the correct constant to use for the price.

```
' Enum declared at the namespace level, above the class declaration.
Public Enum SectionType
    General
    Balcony
    Box
End Enum

Public Class TicketPrice

' Private variable for Section property.
Private SectionInteger As Integer
' Alternate declaration:
' Private SectionInteger As SectionType

' . . .Omitted code for class.

    Private Sub CalculatePrice()
        ' Determine the amount due.
        Dim PriceDecimal As Decimal
        Select Case SectionInteger
            Case SectionType.General
                PriceDecimal = GENERAL_Decimal
```

```
            Case SectionType.Balcony
                PriceDecimal = BALCONY_Decimal
            Case SectionType.Box
                PriceDecimal = BOX_Decimal
        End Select
        If DiscountBoolean Then
            PriceDecimal -= DISCOUNT_Decimal
        End If
        AmountDueDecimal = PriceDecimal * QuantityInteger
    End Sub
End Class
```

Use the following code in the form's CalculateButton_Click event handler to use the enum. Note that if you declare the enum inside the class in the business-services-tier component, you also must specify the class name when using the enum (TicketPrice.SectionType.General).

```
Private Sub CalculateButton_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles CalculateButton.Click
    ' Find price by passing data input in the presentation tier
    ' to the business services tier using a TicketPrice object.
    Dim SectionInteger As Integer

    ' Determine the section from radio buttons.
    If BalconyRadioButton.Checked Then
        SectionInteger = SectionType.Balcony
    ElseIf BoxRadioButton.Checked Then
        SectionInteger = SectionType.Box
    Else
        SectionInteger = SectionType.General    ' Default to General.
    End If

    Try
        Dim ATicketPrice As New TicketPrice( _
          Integer.Parse(QuantityTextBox.Text), SectionInteger, _
          DiscountCheckBox.Checked)
        AmountTextBox.Text = ATicketPrice.AmountDue.ToString("C")
        ErrorProvider1.Clear()
    Catch
        ErrorProvider1.SetError(QuantityTextBox, _
          "Quantity must be numeric.")
    End Try
End Sub
```

This example comes from Ch02EnumRadioButtons. To see an example of selecting from a combo box rather than radio buttons, see Ch02EnumComboBox.

Garbage Collection

The .NET Framework destroys unused objects and reclaims memory in a process called ***garbage collection***. The garbage collector runs periodically and destroys any objects and variables that no longer have any active reference. You have no way of knowing when the garbage collection will occur. In earlier versions of VB, you were advised to set object variables to Nothing and to write

Finalize procedures for your classes. For current versions of VB, Microsoft recommends that you just allow object variables to go out of scope when you are finished with them.

Feedback 2.3

Use this declaration to answer questions 1–4.

```
Private VariableInteger As Integer
```

1. What is the scope of VariableInteger if it is declared inside a class but not inside a procedure?
2. What is its lifetime?
3. What is its accessibility?
4. If the class in which VariableInteger is declared is used as a base class for inheritance, will the derived class have access to the variable?

Your Hands-On Programming Example

R 'n R—For Reading and Refreshment needs an application to calculate payroll. Create a multiple-form project that includes an MDI parent form, a Payroll form, a Summary form, an About form, and a Splash form. The Payroll form, Summary form, and About form should be child forms of the parent form. If you completed the hands-on project for Chapter 1, you will now complete the Payroll and Summary forms.

The parent form should have the following menu:

```
File       View          Window               Help
 Exit       Payroll       Tile Vertical        About
            Summary       Tile Horizontal
                          Cascade
```

This should be a multitier project, with the business rules and calculations in a class separate from the user interface.

Use attributes to display the company name and copyright information on the About form.

Make sure to validate the input data. Display a meaningful message to the user and select the field in error when the user enters bad data.

Include a toolbar and a status bar on the main form.

Planning the Project

Sketch the five forms for the application (Figure 2.9). Your users must sign off the sketches as meeting their needs before you begin programming.

Plan the Objects, Properties, and Methods Plan the classes for the two tiers. Determine the objects and property settings for the forms and controls and for the business services tier. Figure 2.10 shows the diagram of the program classes.

Figure 2.9

Sketch the forms for the R 'n R Payroll project; a. Main form (parent), b. Payroll form; c. Summary form; d. About form; and e. Splash form.

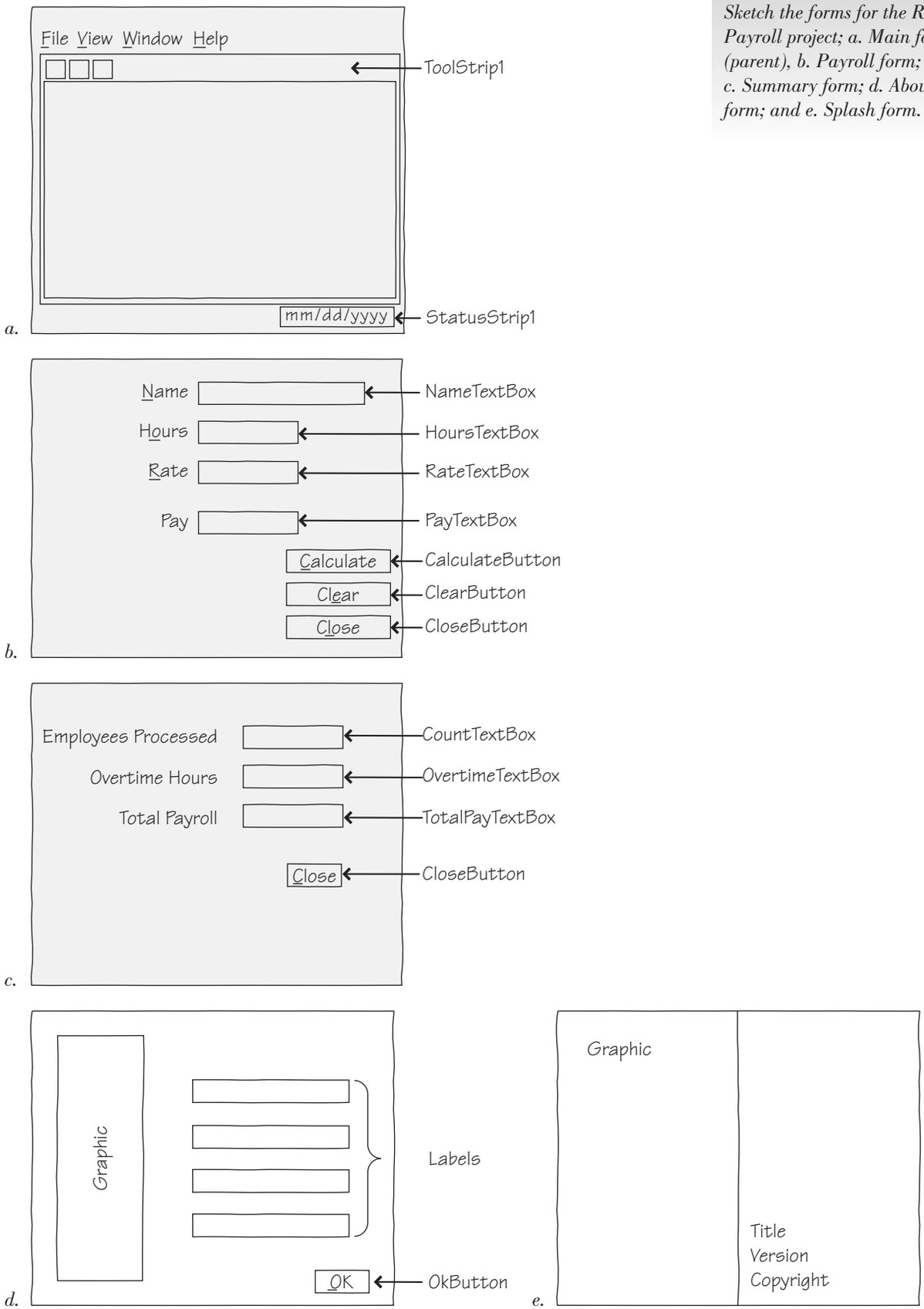


Figure 2.10

The class diagram for the hands-on programming example.

Presentation Tier

MainForm

Object	Property	Setting
MainForm	Text	R 'n R For Reading and Refreshment
	IsMdiContainer	True
MenuStrip1	Items Collection	(drop-down items)
	FileToolStripMenuItem	ExitToolStripMenuItem
	ViewToolStripMenuItem	PayrollFormToolStripMenuItem
		SummaryToolStripMenuItem
	WindowToolStripMenuItem	TileHorizontalToolStripMenuItem
		TileVerticalToolStripMenuItem
		CascadeToolStripMenuItem
	HelpToolStripMenuItem	AboutToolStripMenuItem

Object	Property	Setting
ContextMenuStrip1	ItemsCollection	PayrollToolStripMenuItem SummaryToolStripMenuItem1
ToolStrip1	Items collection	PayrollToolStripButton SummaryToolStripButton AboutToolStripButton
StatusStrip1	Items collection	Add labels for the date and time.

Procedure	Actions—Pseudocode
MainForm_Load	Retrieve the date and time for the status bar.
ExitToolStripMenuItem_Click	Close the form.
AboutToolStripMenuItem_Click AboutToolStripButton_Click	Create an instance of the About form. Set the MdiParent property. Show the form.
PayrollFormToolStripMenuItem_Click PayrollToolStripButton_Click PayrollToolStripMenuItem_Click	Create an instance of the Payroll form. Set the MdiParent property. Show the form. Set the focus on the form.
SummaryToolStripMenuItem_Click SummaryToolStripButton_Click SummaryToolStripMenuItem1_Click	Create an instance of the Summary form. Set the MdiParent property. Show the form. Set the focus on the form.
CascadeToolStripMenuItem_Click	Set MDI layout to Cascade.
TileHorizontalToolStripMenuItem_Click	Set MDI layout to Tile Horizontal.
TileVerticalToolStripMenuItem_Click	Set MDI layout to Tile Vertical.
ClockTimer_Tick	Update the date and time.

PayrollForm

Object	Property	Setting
PayrollForm	AcceptButton CancelButton Text WindowState	CalculateButton ClearButton Payroll Maximized
Label1	Text	&Name
NameTextBox	Text	(blank)
Label2	Text	H&ours
HoursTextBox	Text	(blank)
Label3	Text	&Rate
RateTextBox	Text	(blank)

Object	Property	Setting
Label4	Text	Pay
PayTextBox	Text	(blank)
	ReadOnly	True
CalculateButton	Text	&Calculate
ClearButton	Text	Cl&ear
CloseButton	Text	C&lose

Procedure	Actions—Pseudocode
Instance property Get	If an instance doesn't exist Declare a new instance.

CalculateButton_Click	Clear the error provider.
	Convert the hours to decimal.
	If hours convert successfully
	  Convert the rate to decimal.
	  If rate converts successfully
	  Try
	  Instantiate a Payroll object, passing the input values.
	  Display the pay formatted in a label.
	  Catch
	  Display the error message.
	  Select the control in error.
	  Else
	  Display error for rate.
	  Select the control in error.
	Else
	  Display error for hours.
	  Select the control in error.

ClearButton_Click	Clear all input fields on the screen. Set the focus in NameTextBox.
CloseButton_Click	Close the form.

HoursTextBox_Validating	If not valid
	  Display the error message.
	  Cancel the Validating event handler.
	  Select the control in error.
	Else
	  Clear the error message.

RateTextBox_Validating	If not valid
	  Display the error message.
	  Cancel the Validating event handler.
	  Select the control in error.
	Else
	  Clear the error message.

PayrollForm_FormClosing	Set e.Cancel = False. Set AnInstance = Nothing.
SelectControlInError(ControlName)	Select text. Set the focus.

SummaryForm

Object	Property	Setting
SummaryForm	AcceptButton	CloseButton
	WindowState	Maximized
	Text	Payroll Summary
Label1	Text	Employees Processed
EmployeeCountTextBox	Text	(blank)
	ReadOnly	True
Label2	Text	Overtime Hours
OvertimeHoursTextBox	Text	(blank)
	ReadOnly	True
Label3	Text	Total Payroll
TotalPayrollTextBox	Text	(blank)
	ReadOnly	True
CloseButton	Text	&Close

Procedure	Actions—Pseudocode
Instance propertyGet	If an instance doesn't exist Declare a new instance.
SummaryForm_Activated	Format and display the 3 summary properties in labels.
CloseButton_Click	Close the form.
SummaryForm_FormClosing	Set AnInstance = Nothing.

AboutBox

Object	Property	Setting
AboutBox1	FormBorderStyle	FixedDialog
	StartPosition	CenterParent
	Text	About This Application (Changes at run time.)
	AcceptButton	OkButton
OkButton	Text	&OK

Procedure	Actions—Pseudocode
AboutBox1_Load	Retrieve the attributes and set up the labels. (Code already in template file.)

SplashScreen Include a graphic and labels identifying the company and application. You can use the Splash Screen template and replace the graphic. Add code to hold the form on the screen for a few seconds.

The Business Services Tier

Payroll Class

Properties	Data Type	Property Type	Accessibility
Hours	Decimal	Instance	Read / Write
Rate	Decimal	Instance	Read / Write
Pay	Decimal	Instance	Read Only
NumberProcessed	Decimal	Shared	Read Only
TotalPay	Decimal	Shared	Read Only
OvertimeHours	Decimal	Shared	Read Only

Constants	Data Type	Initial Value
MINIMUM_WAGE_Decimal	Decimal	6.25D
MAXIMUM_WAGE_Decimal	Decimal	50D
MINIMUM_HOURS_Decimal	Decimal	0D
MAXIMUM_HOURS_Decimal	Decimal	60D
REGULAR_HOURS_Decimal	Decimal	40D
OVERTIME_RATE_Decimal	Decimal	1.5D

Methods

```
New(ByVal HoursInDecimal As Decimal, ByVal RateInDecimal As Decimal) (Parameterized constructor)
      Assign parameters to properties.
      Call FindPay.

FindPay
   If hours <= regular hours
      pay = hours * rate
         overtime hours = 0
   Else
      overtime hours = hours – regular hours
      pay = (hours * rate) + (overtime hours * overtime rate)
   Add overtime hours to total.
   Add pay to total.
   Add 1 to number processed.
```

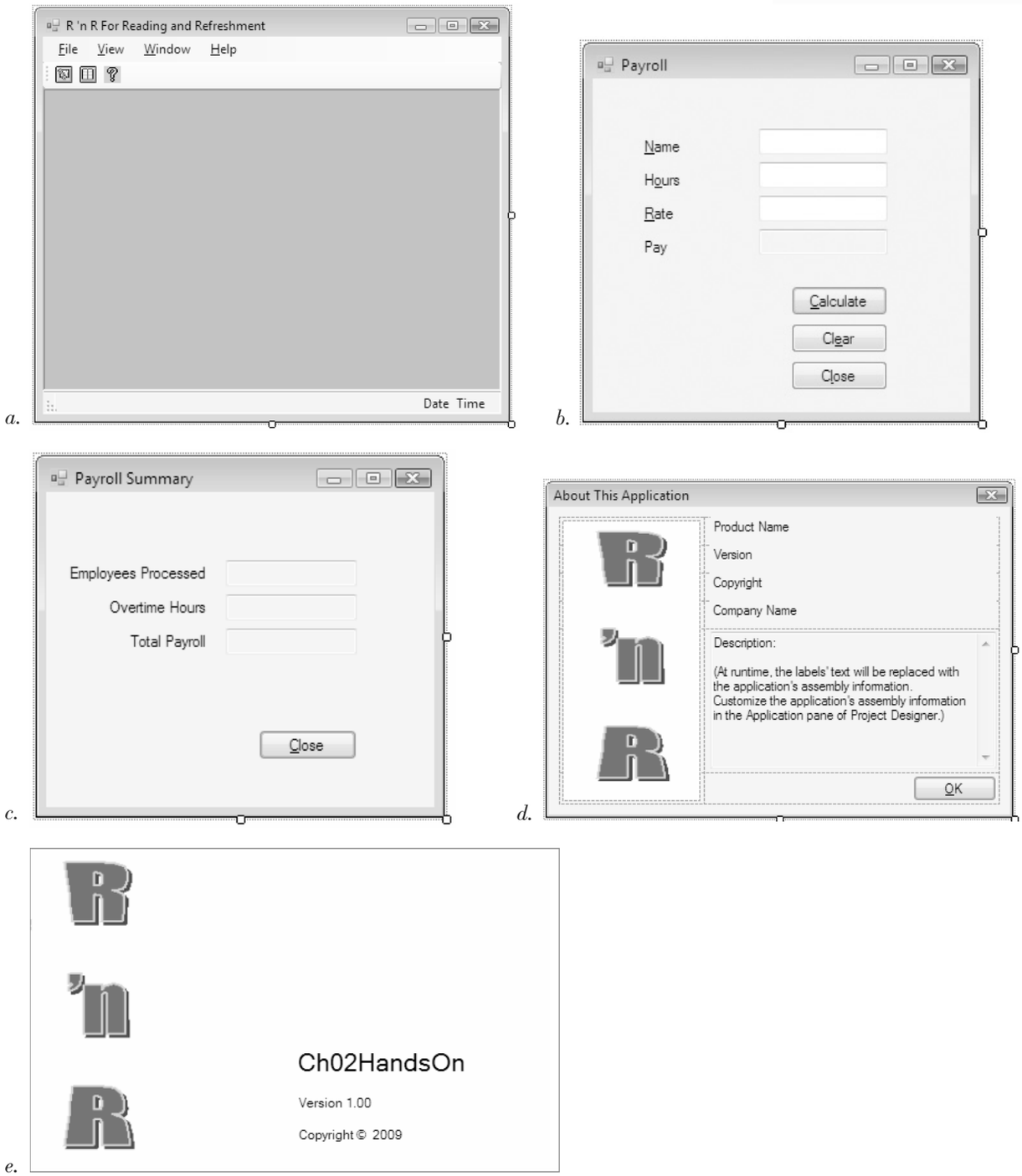
Write the Project Following the sketches in Figure 2.9, create the forms. Figure 2.11 shows the completed forms.

- Set the properties of each of the objects, as you have planned.
- Write the code for the business services tier class, referring to your planning document.

- Write the code for the forms. Working from the pseudocode, write each procedure.
- When you complete the code, use a variety of test data to thoroughly test the project.

Figure 2.11

The forms for the R 'n R Payroll project; a. Main form (parent), b. Payroll form; c. Summary form; d. About form; and e. Splash form.



The Project Coding Solution

MainForm

```
'Program:      Ch02HandsOn
'Programmer:   Bradley/Millspaugh
'Form:         MainForm
'Date:         June 2009
'Description:  MDI parent form; contains the menu and displays
'              the various forms for the R 'n R Payroll application.

Public Class MainForm

    Private Sub MainForm_Load(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles MyBase.Load
        ' Display the date and time in the status bar.

        DateToolStripStatusLabel.Text = Now.ToShortDateString()
        TimeToolStripStatusLabel.Text = Now.ToLongTimeString()
    End Sub

    Private Sub TileVerticalToolStripMenuItem_Click( _
      ByVal sender As System.Object, ByVal e As System.EventArgs) _
      Handles TileVerticalToolStripMenuItem.Click
        ' Display the open windows tiled vertically.

        Me.LayoutMdi(MdiLayout.TileVertical)
    End Sub

    Private Sub TileHorizontalToolStripMenuItem_Click( _
      ByVal sender As Object, ByVal e As System.EventArgs) _
      Handles TileHorizontalToolStripMenuItem.Click
        ' Display the open windows tiled horizontally.

        Me.LayoutMdi(MdiLayout.TileHorizontal)
    End Sub

    Private Sub CascadeToolStripMenuItem_Click(ByVal sender As Object, _
      ByVal e As System.EventArgs) Handles CascadeToolStripMenuItem.Click
        ' Cascade the open windows.

        Me.LayoutMdi(MdiLayout.Cascade)
    End Sub

    Private Sub PayrollFormToolStripMenuItem_Click( _
      ByVal sender As System.Object, ByVal e As System.EventArgs) _
      Handles PayrollFormToolStripMenuItem.Click, _
      PayrollToolStripButton.Click, PayrollToolStripMenuItem.Click
        ' Create an instance of the payroll form.
        Dim APayrollForm As PayrollForm = PayrollForm.Instance

        With APayrollForm
            .MdiParent = Me
            .Show()
            .Focus()
        End With
    End Sub

    Private Sub SummaryToolStripMenuItem_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) _
      Handles SummaryToolStripMenuItem.Click, SummaryToolStripButton.Click, _
      SummaryFormToolStripMenuItem.Click
```

```
        ' Create an instance of the summary form.
        Dim ASummaryForm As SummaryForm = SummaryForm.Instance

        With ASummaryForm
            .MdiParent = Me
            .Show()
            .Focus()
        End With
    End Sub

    Private Sub ExitToolStripMenuItem_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles ExitToolStripMenuItem.Click
        ' Terminate the program.
        ' Closing the startup form ends the program.

        Me.Close()
    End Sub

    Private Sub AboutToolStripMenuItem_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) _
      Handles AboutToolStripMenuItem.Click, AboutToolStripButton.Click
        ' Display the About Box form with attribute information.

        Dim AnAboutBox As New AboutBox1
        AnAboutBox.ShowDialog()
    End Sub

    Private Sub ClockTimer_Tick(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles Timer1.Tick
        ' Update the date and time in the status bar.
        ' Interval = 1000 milliseconds (one second).

        DateToolStripStatusLabel.Text = Now.ToShortDateString
        TimeToolStripStatusLabel.Text = Now.ToLongTimeString()
    End Sub
End Class
```

PayrollForm

```
'Project:      Ch02HandsOn
'Module:       PayrollForm
'Programmer:   Bradley/Millspaugh
'Date:         June 2009
'Description:  User interface for payroll application.
'              Provides data entry and validates for nonnumeric data.
'              Uses the singleton design pattern to ensure that only one
'              instance of the form can be created.

Public Class PayrollForm
    Private Shared AnInstance As PayrollForm

    Public Shared ReadOnly Property Instance() As PayrollForm
        Get
            If AnInstance Is Nothing Then
                AnInstance = New PayrollForm
            End If
            Return AnInstance
        End Get
    End Property
```

```
Private Sub CalculateButton_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles CalculateButton.Click
    ' Create a Payroll object to connect to the business services tier.
    Dim HoursDecimal As Decimal
    Dim RateDecimal As Decimal

    ' Check for valid input data.
    ErrorProvider1.Clear()
    If Decimal.TryParse(HoursTextBox.Text, HoursDecimal) Then
        If Decimal.TryParse(RateTextBox.Text, RateDecimal) Then
            ' Both values converted successfully.
            Try
                Dim APayroll As New Payroll(HoursDecimal, RateDecimal)
                PayTextBox.Text = APayroll.Pay.ToString("C")

            Catch Err As ApplicationException
                ' Catch exceptions from the Payroll class.
                Select Case Err.Source
                    Case "Hours"
                        ErrorProvider1.SetError(HoursTextBox, _
                          Err.Message)
                        SelectControlInError(HoursTextBox)
                    Case "Rate"
                        ErrorProvider1.SetError(RateTextBox, _
                          Err.Message)
                        SelectControlInError(RateTextBox)
                End Select
            End Try
        Else
            ' Rate did not pass validation.
            ErrorProvider1.SetError(RateTextBox, _
              "The rate must be numeric.")
            SelectControlInError(RateTextBox)
        End If
    Else
        ' Hours did not pass validation.
        ErrorProvider1.SetError(HoursTextBox, _
          "The hours must be numeric.")
        SelectControlInError(HoursTextBox)
    End If
End Sub

Private Sub ClearButton_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles ClearButton.Click
    ' Clear the screen fields.

    ErrorProvider1.Clear()
    With NameTextBox
        .Clear()
        .Focus()
    End With
    HoursTextBox.Clear()
    RateTextBox.Clear()
    PayTextBox.Clear()
End Sub
```

```
    Private Sub CloseButton_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles CloseButton.Click
        ' Close this form.

        Me.Close()
    End Sub

    Private Sub HoursTextBox_Validating(ByVal sender As Object, _
      ByVal e As System.ComponentModel.CancelEventArgs) _
      Handles HoursTextBox.Validating
        ' Test the hours for numeric.
        Dim HoursDecimal As Decimal

        If Decimal.TryParse(HoursTextBox.Text, HoursDecimal) Then
            ErrorProvider1.Clear()
        Else
            ErrorProvider1.SetError(HoursTextBox, _
              "The hours must be numeric.")
            HoursTextBox.SelectAll()
            e.Cancel = True
        End If
    End Sub

    Private Sub RateTextBox_Validating(ByVal sender As Object, _
      ByVal e As System.ComponentModel.CancelEventArgs) _
      Handles RateTextBox.Validating
        ' Test pay rate for numeric.
        Dim RateDecimal As Decimal

        If Decimal.TryParse(RateTextBox.Text, RateDecimal) Then
            ErrorProvider1.Clear()
        Else
            ErrorProvider1.SetError(RateTextBox, _
              "The hours must be numeric.")
            RateTextBox.SelectAll()
            e.Cancel = True
        End If
    End Sub

    Private Sub PayrollForm_FormClosing(ByVal sender As Object, _
      ByVal e As System.Windows.Forms.FormClosingEventArgs) _
      Handles Me.FormClosing
        ' Do not allow validation to cancel the form's closing.

        e.Cancel = False
        ' Release the instance of this form.
        AnInstance = Nothing
    End Sub

    Private Sub SelectControlInError(ByVal ErrorTextBox As TextBox)
        ' Select the control in error.

        With ErrorTextBox
            .SelectAll()
            .Focus()
        End With
    End Sub
End Class
```

SummaryForm

```
'Program:      Ch02HandsOn
'Programmer:   Bradley/Millspaugh
'Form:         SummaryForm
'Date:         June 2009
'Description:  Summary form for the chapter hands-on MDI application.
'              Displays summary information for multiple transactions.
'              Uses the singleton design pattern to ensure that only one
'              instance of the form can be created.

Public Class SummaryForm
    Private Shared AnInstance As SummaryForm

    Public Shared ReadOnly Property Instance() As SummaryForm
        Get
            If AnInstance Is Nothing Then
                AnInstance = New SummaryForm
            End If
            Return AnInstance
        End Get
    End Property

    Private Sub SummaryForm_Activated(ByVal sender As Object, _
      ByVal e As System.EventArgs) Handles Me.Activated
        ' Retrieve and display the summary values.

        EmployeeCountTextBox.Text = Payroll.NumberProcessed.ToString()
        OvertimeHoursTextBox.Text = Payroll.OvertimeHours.ToString("N1")
        TotalPayrollTextBox.Text = Payroll.TotalPay.ToString("C")
    End Sub

    Private Sub CloseButton_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles CloseButton.Click
        ' Close this form.

        Me.Close()
    End Sub

    Private Sub SummaryForm_FormClosing(ByVal sender As Object, _
      ByVal e As System.Windows.Forms.FormClosingEventArgs) _
      Handles Me.FormClosing
        ' Release the form's instance.

        AnInstance = Nothing
    End Sub
End Class
```

Payroll Class

```
'Project:      Ch02HandsOn
'Module:       Payroll Class
'Programmer:   Bradley/Millspaugh
'Date:         June 2009
'Description:  Business services tier for payroll calculation: validates input
'              data and calculates the pay, with overtime, regular, and
'              summary data.
```

```
Public Class Payroll

    ' Instance variables.
    Private HoursDecimal As Decimal     ' Hold the Hours property.
    Private RateDecimal As Decimal      ' Hold the Rate property.
    Private PayDecimal As Decimal       ' Hold the Pay property.

    ' Shared variables.
    ' Hold the NumberProcessed shared property.
    Private Shared NumberEmployeesInteger As Integer
    ' Hold the TotalPay shared property.
    Private Shared TotalPayDecimal As Decimal
    ' Hold the OvertimeHours shared property.
    Private Shared TotalOvertimeHoursDecimal As Decimal

    ' Constants.
    Private Const MINIMUM_WAGE_Decimal As Decimal = 6.25D
    Private Const MAXIMUM_WAGE_Decimal As Decimal = 50D
    Private Const MINIMUM_HOURS_Decimal As Decimal = 0D
    Private Const MAXIMUM_HOURS_Decimal As Decimal = 60D
    Private Const REGULAR_HOURS_Decimal As Decimal = 40D
    Private Const OVERTIME_RATE_Decimal As Decimal = 1.5D

    ' Constructor.
    Sub New(ByVal HoursDecimal As Decimal, ByVal RateDecimal As Decimal)
        ' Assign properties and calculate the pay.

        Me.Hours = HoursDecimal
        Me.Rate = RateDecimal
        FindPay()
    End Sub

    Private Sub FindPay()
        ' Calculate the pay.
        Dim OvertimeHoursDecimal As Decimal

        If HoursDecimal <= REGULAR_HOURS_Decimal Then   ' No overtime.
            PayDecimal = HoursDecimal * RateDecimal
            OvertimeHoursDecimal = 0D
        Else                                            ' Overtime.
            OvertimeHoursDecimal = HoursDecimal - REGULAR_HOURS_Decimal
            PayDecimal = (REGULAR_HOURS_Decimal * RateDecimal) + _
              (OvertimeHoursDecimal * OVERTIME_RATE_Decimal * RateDecimal)
        End If
        TotalOvertimeHoursDecimal += OvertimeHoursDecimal
        TotalPayDecimal += PayDecimal
        NumberEmployeesInteger += 1
    End Sub

    ' Property procedures.
    Public Property Hours() As Decimal
        Get
            Return HoursDecimal
        End Get
        Set(ByVal Value As Decimal)
            If Value >= MINIMUM_HOURS_Decimal And _
              Value <= MAXIMUM_HOURS_Decimal Then
                HoursDecimal = Value
```

```
            Else
                Dim Ex As New ApplicationException( _
                  "Hours are outside of the acceptable range.")
                Ex.Source = "Hours"
                Throw Ex
            End If
        End Set
    End Property

    Public Property Rate() As Decimal
        Get
            Return RateDecimal
        End Get
        Set(ByVal Value As Decimal)
            If Value >= MINIMUM_WAGE_Decimal And _
              Value <= MAXIMUM_WAGE_Decimal Then
                RateDecimal = Value
            Else
                Dim Ex As New ApplicationException( _
                  "Pay rate is outside of the acceptable range.")
                Ex.Source = "Rate"
                Throw Ex
            End If
        End Set
    End Property

    Public ReadOnly Property Pay() As Decimal
        Get
            Return PayDecimal
        End Get
    End Property

    Public Shared ReadOnly Property NumberProcessed() As Decimal
        Get
            Return NumberEmployeesInteger
        End Get
    End Property

    Public Shared ReadOnly Property TotalPay() As Decimal
        Get
            Return TotalPayDecimal
        End Get
    End Property

    Public Shared ReadOnly Property OvertimeHours() As Decimal
        Get
            Return TotalOvertimeHoursDecimal
        End Get
    End Property
End Class
```

Summary

1. In VB, all programming is based on classes, which consist of properties, methods, and events.
2. You can create a new class and use the class to create new objects.
3. Creating a new object is called *instantiating* the class; the object is called an *instance* of the class.

4. In OOP terminology, abstraction means to create a model of an object.
5. Encapsulation refers to the combination of the characteristics and behaviors of an item into a single class definition.
6. Inheritance provides a means to derive a new class based on an existing class. The existing class is called a *base class*, *superclass*, or *parent class*. The inherited class is called a *subclass*, *derived class*, or *child class*.
7. An abstract class is a class designed strictly for inheritance; you cannot instantiate an object of the class but must derive new classes from the class.
8. Polymorphism allows classes that inherit to have methods that behave differently than the identically named methods in the base class.
9. One of biggest advantages of object-oriented programming is that classes that you create for one application may be reused in other applications.
10. Multitier applications separate program functions into the presentation tier (the user interface), the business services tier (the logic of calculations and validation), and the data tier (access to stored data).
11. One advantage of using multitier development is that the business rules can be changed without changing the interface or the interface can be changed without changing the business services tier.
12. The variables inside a class used to store the properties should be declared as Private so that data values are accessible only by procedures within the class.
13. The way to make the properties of a class available to code outside the class is to use property procedures. The `Get` portion returns the value of the property and the `Set` portion assigns a value to the property. Validation is often performed in the `Set` portion.
14. You can create read-only and write-only properties.
15. A constructor is a method that executes automatically when an object is created. In VB, the constructor must be named New and must be Public or Protected.
16. You can overload the `New` sub procedure to have more than one signature. A `New` sub procedure that requires arguments is called a *parameterized constructor*.
17. The public functions and sub procedures of a class module are its methods.
18. To instantiate an object of a class, you must use the `New` keyword on either the declaration statement or an assignment statement. The location of the `New` keyword determines when the object is created.
19. Your classes can throw an ApplicationException to indicate an error condition.
20. A class can pass an exception up to the calling code by using the `Throw` keyword.
21. Exceptions require substantial system resources and should be avoided for situations that occur frequently, such as invalid user input.
22. The `TryParse` method of the numeric classes can convert strings to numeric without throwing an exception for invalid data. Instead, the numeric variable is set to zero for an invalid conversion.
23. The Validating event of a text box occurs as the user attempts to move to another control that has its CausesValidation property set to *true*. The Validating event handler is the preferred location to perform field-level validation. The Validating event can be canceled for invalid data, which holds the focus in the field in error.
24. You can use an ErrorProvider component to display an error indicator and message on a form, rather than use a message box.

25. Shared members (properties and methods) have one copy that can be used by all objects of the class, generally used for totals and counts. Instance members have one copy for each instance of the object. Declare shared members with the `Shared` keyword. You can reference Public shared members of a class without creating an instance of the class.
26. A namespace is an area used for grouping and referring to classes and structures.
27. The scope of variables, constants, and objects, from the greatest to the smallest: namespace, module level, procedure level, and block level.
28. The lifetime of a variable, constant, or object corresponds to its scope.
29. You can declare the accessibility of entities using the keywords `Public`, `Private`, `Protected`, `Friend`, and `Protected Friend`.
30. A subclass inherits all public and protected properties and methods of its base class, except for the constructor. An identically named method in a subclass will override or shadow the base-class method. Shadow is the default.
31. To override a method from a base class, the original method must be declared as overridable, and the new method must use the `Overrides` keyword.
32. A class that has a predefined set of possible values for a property should define the values in an enum. The enum structure can appear at the namespace or class level and must define integer values.
33. The garbage collection feature periodically checks for unreferenced objects, destroys the object references, and releases resources.

Key Terms

abstract class *51*
abstraction *50*
accessibility *76*
base class *51*
block-level scope *75*
business rules *53*
business services tier *54*
child class *51*
constructor *56*
data tier *54*
derived class *51*
destructor *56*
encapsulation *50*
enum *81*
enumeration *81*
ErrorProvider component *64*
garbage collection *83*
inheritance *51*
instance member *69*
instance property *69*
instance variable *69*
lifetime *76*
module-level scope *75*
multitier application *53*
namespace *74*
namespace scope *74*
overloading *57*
overridable *78*
override *52*
parameterized constructor *57*
parent class *51*
polymorphism *52*
presentation tier *53*
procedure-level scope *75*
property procedure *55*
`ReadOnly` *56*
reusability *51*
scope *74*
shadow *78*
shared member *69*
shared property *69*
shared variable *69*
subclass *51*
superclass *51*
throw an exception *60*
`Throw` statement *61*
`TryParse` method *62*
Validating event *63*
`Value` keyword *55*
`WriteOnly` *56*

Review Questions

1. Define abstraction, encapsulation, inheritance, and polymorphism.
2. What is an abstract class and how is it used?
3. Why should property variables in a class be declared as private?
4. What are property procedures and what is their purpose?
5. Explain how to create a new class and instantiate an object from that class.
6. What is a constructor, how is it created, and when is it triggered?
7. What is a parameterized constructor?
8. How can you write methods for a new class?
9. What is a shared member? What is its purpose? How is it created?
10. Explain the steps necessary to inherit a class from another class.
11. Differentiate between overriding and overloading.
12. What are the advantages of developing applications using multiple tiers?
13. Describe the steps necessary to perform validation in the business services tier but display the message to the user in the presentation tier.
14. Explain the differences between a namespace-level variable and a module-level variable. How is each created and how is it used?
15. Explain the differences between a procedure-level variable and a block-level variable. How is each created and how is it used?
16. What is the lifetime of a procedure-level variable? a block-level variable? a module-level variable?
17. Explain the difference between overriding and shadowing methods.
18. What is the effect of using the `Protected` accessibility modifier? the `Friend` modifier?
19. What is an advantage of using the `TryParse` methods rather than `Parse`?
20. What is an advantage of using an ErrorProvider component rather than a message box?
21. What is the purpose of an enum? How is one created?
22. What is garbage collection? What does it do and when does it run?

Programming Exercises

2.1 Tricia's Travels: You can add to your Exercise 1.2 or just create the main form.

Presentation Tier

Main Form

Include text boxes for the customer name, phone number, number traveling, departure date, and credit card number. Include a list box for the destinations: Caribbean, Mediterranean, and Alaska. Include radio buttons for 7-day or 14-day packages and a check box for first class. Validate that the user has made an entry for all fields.

Summary Form

Display the total billing amount, the total number traveling, the number for each destination, and the number of first-class fares.

Business Services Tier

Calculate the amount due based on the following schedule:

Days	Destination	Standard price	First-class price
7	Caribbean	3250	5000
14	Caribbean	6000	9000
7	Mediterranean	4250	7999
14	Mediterranean	7999	11999
7	Alaska	3300	5250
14	Alaska	7200	10500

2.2 Kenna's Kandles offers candles in various shapes, scents, and colors. Write an MDI project that contains a Main form, an About form, and a Summary form using a separate tier for the business rules.

Presentation Tier

Main Form

- Text boxes for customer information (name and credit card number).
- Text box for quantity.
- Radio buttons or list box for candle style (tea light, votive, or pillar).
- Radio buttons or list box for color (Federal Blue, Sunflower Yellow, Christmas Red, and Lily White).
- Check box for Scented.
- Label for the price of the item.

Summary Form

Display the subtotal for all candles, the tax of 8 percent, a shipping fee of 3 percent, and the total due.

Business Services Tier

Calculate the price for each candle based on the options selected. The business services tier also should accumulate the information for the total.

Style	Base price	Scented price (additional)
Tea lights	5.75	0.75
Votives	7.50	1.25
Pillar	12.25	1.75

2.3 Create a project for maintaining a checkbook using multiple tiers.

Presentation Tier

Main Form

Use radio buttons or a drop-down list to indicate the transaction type: check, deposit, interest, or service charge. Allow the user to enter the amount in a text box for the amount and display the account balance in a label or read-only text box. Display a message box for insufficient funds, based on an appropriate exception generated by the business services tier.

Summary Form

Display the total number and the total dollar amounts for deposits, checks, interest, and service charges.

Business Services Tier

Validate that the balance can cover a check. If not, throw an exception and deduct a service charge of $10; do not process the check. Process interest and deposits by adding to the balance and checks and service charges by reducing the balance.

Optional Extra

Create an MDI application that includes an About form, a toolbar, and a status bar.

2.4 Piecework workers are paid by the piece. Workers who produce a greater quantity of output are often paid at a higher rate.

Presentation Tier

The program should input the name and number of pieces and calculate the pay. Include a *Calculate* button and a *Clear* button. You can include either a *Summary* button or menu item. The *Summary* option displays the total number of pieces, the total pay, and the average pay per person on a Summary form.

The name and number of pieces are required fields.

Business Services Tier

The number of pieces must be a positive number; throw an exception for negative numbers. Calculate the pay using this schedule:

Pieces completed	Price paid per piece for all pieces
1–199	.50
200–399	.55
400–599	.60
600 or more	.65

Accumulate and return the summary totals for number of pieces, pay, and average pay per person. Notice that you also must accumulate the number of persons to calculate the average.

2.5 Add an inherited class to Exercise 2.4. This class calculates pay for senior workers, who are paid on a different scale. You must add a check box to the form for senior workers and use the inherited class for those workers.

Senior workers receive a base pay of $300 plus a per-piece pay using this schedule:

Pieces completed	Price paid per piece for all pieces
1–199	.20
200–399	.25
400–599	.30
600–799	.35
800 or more	.40

Case Studies

Claytor's Cottages

Modify your Claytor's Cottages case study project from Chapter 1. Complete the Reservations option using a presentation tier and a business services tier.

Presentation Tier

The form should have radio buttons for King, Queen, or Double. Include text boxes for entering the customer's name, address, and phone number; the number of nights stayed; credit card type (use a list box or combo box for Visa, Mastercard, and American Express); and credit card number. Name, nights stayed, and credit card number are required fields. Use a check box for weekend or weekday rate and a check box for AARP or AAA members. Display the price in a label or Read-Only text box.

Business Services Tier

Throw an exception if the number of days is not greater than 0. Calculate the price using this table. Add a room tax of 7 percent. AAA and AARP customers receive a 10 percent discount rate, which is calculated before the tax.

Beds	Sunday through Thursday rate	Weekend rate (Friday and Saturday)
King	95.00	105.00
Queen	85.00	95.00
Double	69.95	79.95

Optional extra: Enter the date of arrival and date of departure instead of the check boxes. You can use a calendar object or text boxes to obtain the dates. Use the methods of the DateTime structure to determine if the check-in dates are weekdays or weekend. Increase the rates by 25 percent in May through September.

Hint: Determine the number of days by using the `Subtract` method of the DateTime structure:

```
NumberDaysInteger = _
  EndDate.Subtract(StartDate).Days
```

Christian's Car Rentals

Modify your Christian's Car Rentals project from Chapter 1. Code the Rentals form using a presentation tier and a business services tier.

Presentation Tier

The presentation tier should include data entry for the size of car: Compact, Mid size, or Luxury. Include text boxes for entering the renter's name, address, phone number, license, credit card type, and credit card number. A group box should include the number of days rented, the beginning odometer reading, and the ending odometer reading.

Validate that the ending odometer reading is greater than the beginning odometer reading before allowing the data to be sent to the business services tier. Make sure that an entry has been made for license and number of days rented.

Business Services Tier

Validate that the number of days rented is greater than 0. There is no mileage charge if the number of miles does not exceed an average of 100 miles per day rented. Use the following rates:

Car size	Daily rate	Mileage rate
Compact	26.95	.12
Mid size	32.95	.15
Luxury	50.95	.20

Corporate and Insurance Accounts (Inheritance)

Corporate accounts waive the mileage rate and have a 5 percent discount; insurance accounts have a 10 percent discount on the daily rate.