

Developing Object-Oriented C# Programs

In this chapter you will:

- ② Study object-oriented programming concepts
- ② Define custom classes
- ② Declare class fields
- ② Work with class methods

The C# programs you have written so far have mostly been self-contained. That is, most of the code—including variables, statements, and functions—exists within a script section. For example, you might create a Web page for an online retailer that uses C# to calculate the total for a sales order that includes state sales tax and shipping. However, suppose the retailer sells different types of products on different Web pages, with one page selling baseball uniforms, another page selling jelly beans, and so on. If you want to reuse the C# sales total code on multiple Web pages, you must copy all of the statements or re-create them from scratch for each Web page. Object-oriented programming takes a different approach. Essentially, object-oriented programming allows you to use and create self-contained sections of code—known as objects—that can be reused in your programs. In other words, object-oriented programming allows you to reuse code without having to copy or re-create it.

Introduction to Object-Oriented Programming

As you learned in Chapter 2, object-oriented programming (OOP) refers to the creation of reusable software objects that can be easily incorporated into another program. An object is programming code and data that can be treated as an individual unit or component. Objects are often also called **components**. In object-oriented programming, the code that makes up an object is organized into classes. Objects can range from simple controls, such as a button, to entire programs, such as a database application. In fact, some programs consist entirely of other objects. You'll often encounter objects that have been designed to perform a specific task. For example, in a retail sales program, you could refer to all of the code that calculates the sales total as a single object. You could then reuse that object over and over again in the same program just by typing the object name.

Popular object-oriented programming languages include C++, Java, and Visual Basic. Using any of these or other object-oriented languages, programmers can create objects themselves or use objects created by other programmers or supplied by the manufacturer. For example, if you are creating an accounting program in Visual Basic, you can use an object named `Payroll` that was created in C++. The `Payroll` object might contain one method that calculates the amount of federal and state tax to deduct, another function that calculates the FICA amount to deduct, and so on. Properties of the `Payroll` object might include an employee's number of tax withholding allowances, federal and state tax percentages, and the cost of insurance premiums. You do not need to know how the `Payroll` object was created in C++, nor do you need to re-create it in Visual Basic. You only need to know how to access the methods and properties of the `Payroll` object from the Visual Basic program.

Understanding Encapsulation

Objects are **encapsulated**, which means that all code and required data are contained within the object itself. In most cases, an encapsulated object consists of a single computer file that contains all code and required data. Encapsulation places code inside what programmers like to call a “black box.” When an object is encapsulated, you cannot see “inside” it—all internal workings are hidden. The code (methods and statements) and data (variables and constants) contained in an encapsulated object are accessed through an interface. The term **interface** refers to the methods and properties that are required for a source program to communicate with an object. For example, interface elements required to access a `Payroll` object might be a method named `calcNetPay()`, which calculates an employee’s net pay, and properties containing the employee’s name and pay rate.

When you include encapsulated objects in your programs, users can see only the methods and properties of the object that you allow them to see. By removing the ability to see inside the black box, encapsulation reduces the complexity of the code, allowing programmers who use the code to concentrate on the task of integrating the code into their programs. Encapsulation also prevents other programmers from accidentally introducing a bug into a program, or from possibly even stealing the code and claiming it as their own.

You can compare a programming object and its interface to a handheld calculator. The calculator represents an object, and you represent a program that wants to use the object. You establish an interface with the calculator object by entering numbers (the data required by the object) and then pressing calculation keys (which represent the methods of the object). You do not need to know, nor can you see, the inner workings of the calculator object. As a programmer, you are concerned only with an object’s methods and properties. To continue the analogy, you are only concerned with the result you expect the calculator object to return. Figure 10-1 illustrates the idea of the calculator interface.

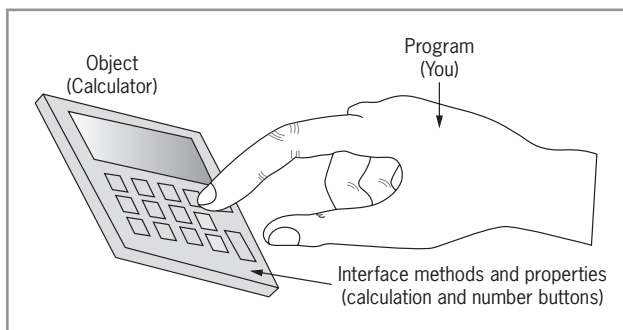


Figure 10-1 Calculator interface

Another example of an object and its interface is Microsoft Word. Word itself is actually an object made up of numerous other objects. The program window (or user interface) is one object. The items you see in the interface, such as the menu and toolbars, are used to execute methods. For example, the Bold button on the toolbar executes a bold method. The text of your document is the data you provide to the program. Word is a helpful tool that you can use without knowing how the various methods work. You only need to know what each method does. To get full satisfaction out of Word, you only need to provide the data (text) and execute the appropriate methods (such as the bold method), when necessary. In the same way, when using objects in your code, you only need to provide the necessary data (such as an employee's gross pay) and execute the appropriate method (such as the `calcNetPay()` method).

Using Objects in C# Programs

Up to this point, all of the C# programs you have written have contained procedural statements that did not rely on objects. This does not mean that the skills you have learned so far are useless in constructing object-oriented programs. However, object-oriented techniques will help you build more extensible code that is easier to reuse, modify, and enhance.

In object-oriented programming, the code, methods, attributes, and other information that make up an object are organized into classes. Essentially, a class is a template, or blueprint, that serves as the basis for new objects. When you use an object in your program, you actually create an instance of the class of the object. In other words, an instance is an object that has been created from a class. When you create an object from a class, you are said to be **instantiating** the object.

Later in this chapter, you will learn how to create, or instantiate, an object from custom classes that you write yourself. However, as a conceptual example, consider an object named `BankAccount` that contains methods and properties that you might use to record transactions associated with a checking or savings account. The `BankAccount` object is created from a `BankAccount` class. To use the `BankAccount` class, you create an instance of the class. A particular instance of an object **inherits** its methods and properties from a class—that is, it takes on the characteristics of the class on which it is based. The `BankAccount` object, for instance, would inherit all of the methods and properties of the `BankAccount` class. To give another example, when you create a new word-processing document, which is a type of object, it usually inherits the properties of a template on which it is based. The template is a type of class. The document inherits characteristics of the template, which might include font size, line spacing, and boilerplate text. In the same manner, programs that include instances of objects inherit the object's functionality.

You use the following constructor syntax to instantiate an object from a class:

```
ClassName objectName = new ClassName();
```

The identifiers you use for an object name must follow the same rules as identifiers for variables: They must begin with an upper-case or lowercase ASCII letter, can include numbers (but not as the first character), cannot include spaces, cannot be keywords, and are case sensitive. The following statement instantiates an object named `checking` from the `BankAccount` class:

```
BankAccount checking = new BankAccount();
```

Class constructors are primarily used to initialize properties when an object is first instantiated. For this reason, you can pass arguments to many constructor methods. For example, the `BankAccount` class might require you to pass three arguments: the checking account number, a check number, and a check amount, as follows:

```
BankAccount checking = new BankAccount(01234587, 1021, 97.58);
```

After you instantiate an object, you use a period to access the methods and properties contained in the object. With methods, you must also include a set of parentheses at the end of the method name, just as you would with functions. Like functions, methods can also accept arguments. The following statements demonstrate how to call two methods, `getBalance()` and `getCheckAmount()`, from the `checking` object. The `getBalance()` method does not require any arguments, whereas the `getCheckAmount()` method requires an argument containing the check number.

```
double balance = checking.getBalance();
checkNumber = 1022;
double amount = checking.getCheckAmount(checkNumber);
```

To access property values in an object, you do not include parentheses at the end of the property name, as you do with functions and methods. The following statements update and display the value in a property named `balance` in the `checking` object:

```
checkAmount = 124.75;
checking.balance = checking.balance + checkAmount;
Response.Write("<p>Your updated checking account balance is "
    + String.Format("{0:C}", checking.balance) + "</p>");
```

In this chapter, you will work on a Web site for an online bakery named Central Valley Bakery. The store includes four shopping categories: cakes, cookies, pies, and breads. The purpose of the Web site is to demonstrate code reuse with classes. As you progress through this chapter, you will develop a class named `ShoppingCart` that handles the functionality of building and updating a shopping cart as a user selects items to purchase. Shopping cart classes are very popular with Web developers because of the many Web sites that offer online shopping.

Rather than re-creating shopping cart functionality for each online Web site you develop, you can much more easily develop the Web site by reusing an existing shopping cart class. As you create the `ShoppingCart` class, notice that its functionality has nothing to do with the products sold by Central Valley Bakery. Instead the code is generic enough that it can be used with any Web site that sells products, provided the pages in the site and the associated database conform to the requirements of the class. Your Chapter folder for Chapter 10 contains a folder named `Bakery` where you can find the files that you will need for this project. The Web site and a database named `Bakery` have already been created; you only need to focus on the class development techniques. Figure 10-2 shows the Central Valley Bakery home page.

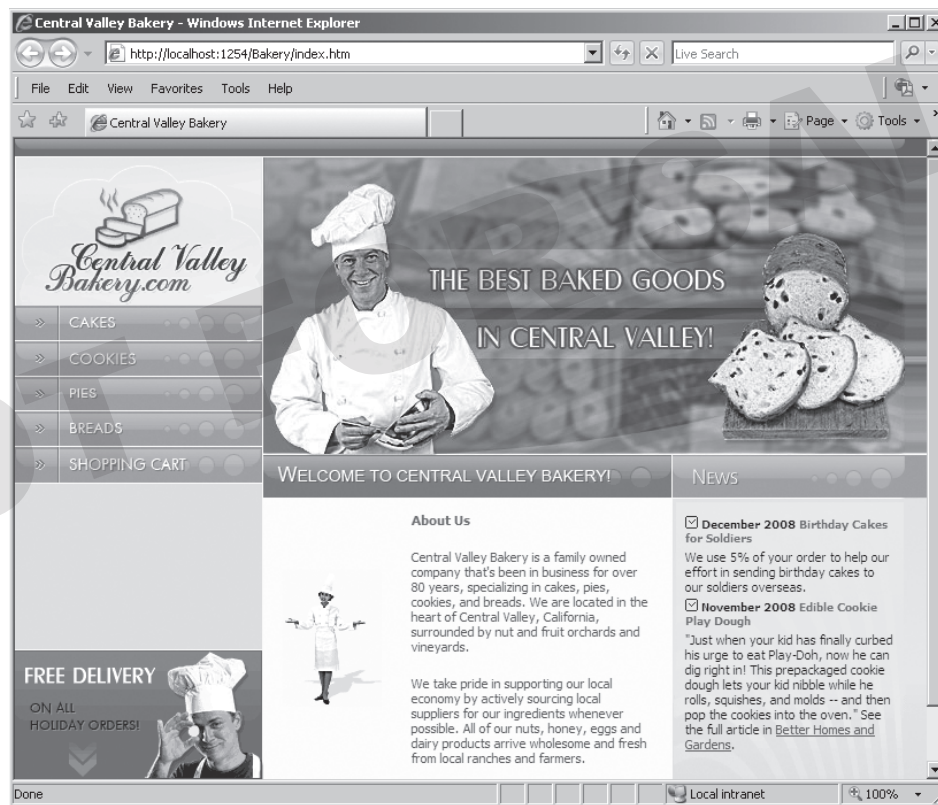


Figure 10-2 Central Valley Bakery home page

First, you will add to each of the product pages a Literal control that displays messages from the `ShoppingCart` class and a GridView control that displays product information. The `ShoppingCart` class requires that product information is stored in tables containing four fields: `productID`, `name`, `description`, and `price`. The `productID` field is the primary key and consists of a unique text field. For example, the primary

key for the first cake product is CAKE001. To keep things simple, the **ShoppingCart** class does not store customer or payment information.

You'll start by adding the Literal control to the `cakes.aspx` page.

To add controls to the `cakes.aspx` page:

1. Start Visual Web Developer, select the **File** menu and then select **Open Web Site**. The Open Web Site dialog box opens. Locate and select the **Bakery** folder, located in your Chapter folder for Chapter 10, and then click **Open**. The Bakery Web site opens in the Visual Studio IDE.
2. Open the **cakes.aspx** file in Design view.
3. Replace **[Add code here]** with a Literal control and change its ID to **ProductPage**.
4. Expand the **App_Data** folder in Solution Explorer and then double-click the **Bakery.mdf** file to open the database in Database Explorer.
5. Add a GridView control with an ID of **ProductGrid** immediately after the Literal control. After you add the control, select **Choose Data Source** from the GridView Tasks menu, and then select **<New data source ...>**. The Choose a Data Source Type page of the Data Source Configuration Wizard opens.
6. On the Choose a Data Source Type page, select **Database**, and enter an ID of **BakeryDataSource**. Click **OK**. The Choose Your Data Connection page opens.
7. On the Choose Your Data Connection page, click the data connection box and then select **Bakery.mdf**. Click **Next**. The Save the Connection String to the Application Configuration File opens.
8. On the Save the Connection String to the Application Configuration File page, change the name of the connection string to **BakeryConnectionString**, and then click **Next**. The Configure the Select Statement page opens.
9. On the Configure the Select Statement page, select Cakes from the Name box and * from the Columns list, and then click **Next**. The Test Query page opens.
10. On the Test Query page, click the **Test Query** button. You should see a list of cake and their product IDs, descriptions, and prices. Click **Finish** to close the Data Source Configuration Wizard.

Next, you will format the GridView control on the `cakes.aspx` page.

To format the GridView control on the `cakes.aspx` page:

1. With the GridView control selected in Design view, expand the **Font** property in the Properties window and select **Smaller** in the Size box.

2. If necessary, change the value assigned to the `DataKeyNames` property to **productID**.
3. Select the **GridView** control in Design view and then click the arrow to the right of the control to display the **GridView Tasks** menu.
4. Select the **Auto Format** command from the GridView Tasks menu. The AutoFormat dialog box displays. Select the **Brown Sugar** scheme and then click **OK**.
5. From the GridView Tasks menu, select **Enable Selection** and then select **Edit Columns**. The Fields dialog box opens.
6. In the Fields dialog box, click the **Select** field in the Selected fields list, and then click the down arrow until the Select field is the last control in the list.
7. In the CommandField properties section, change the Button-Type property to **Button** and the SelectImageUrl property to `~/images/ordernow.gif`. The ordernow.gif file is located in the images folder within the project folder.
8. Click the **productID** field in the Selected fields list and click the **X** button to remove it from the list.
9. Click the **name** field in the Selected fields list and change its HeaderText property to **Name**.
10. Click the **description** field in the Selected fields list and change its HeaderText property to **Description**.
11. Click the **price** field in the Selected fields list and change its HeaderText property to **Price**. Also, change its DataFormatString property to `{0:c}`, which displays price fields as currency.
12. Click **OK** to close the Fields dialog box.

Next, you will copy the controls from the cakes.aspx file to the breads.aspx, cookies.aspx, and pies.aspx files.

To copy the GridView control from the cakes.aspx file to the breads.aspx, cookies.aspx, and pies.aspx files:

1. Click the **Source** button at the bottom of the IDE window to view the cakes.aspx page in the Code Editor window.
2. Locate and copy the `<asp:Literal>`, `<asp:GridView>`, and `<asp:SqlDataSource>` controls.
3. Open the **breads.aspx** file in the Code Editor window.
4. Locate the text **[Add code here]** and replace it with the controls you copied from the cakes.aspx file. Then, locate the

- ConnectionString property in the **<asp:SqlDataSource>** control and replace [Cakes] with **[Breads]**.
- Open the **cookies.aspx** file in the Code Editor window.
 - Locate the text **[Add code here]** and replace it with the controls you copied from the cakes.aspx file. Then, locate the ConnectionString property in the **<asp:SqlDataSource>** control and replace [Cakes] with **[Cookies]**.
 - Open the **pies.aspx** file in the Code Editor window.
 - Locate the text **[Add code here]** and replace it with the controls you copied from the cakes.aspx file. Then, locate the ConnectionString property in the **<asp:SqlDataSource>** control and replace [Cakes] with **[Pies]**.
 - Start the Web site and test the product pages. Figure 10-3 shows the Cakes product page. Be sure not to click any of the Order Now buttons because you still need to add code to give them their functionality.
 - Close your Web browser window.



Figure 10-3 Cakes product page after adding Web server controls

Short Quiz 1

1. Why do programmers refer to encapsulation as a black box?
2. What is instantiation as it relates to classes, objects, and object-oriented programming?
3. Explain how to instantiate an object from a class.
4. What are class constructor statements primarily used for?
5. How do you access an object's methods and properties?

Defining Custom C# Classes

Classes were defined earlier in this chapter as the code, methods, attributes, and other information that make up an object. In C#, classes more specifically refer to data structures that contain fields along with methods for manipulating the fields. The term **data structure** refers to a system for organizing data, whereas the term **field** refers to variables that are defined within a class. Some of the data structures you have already used include arrays and lists. The methods and fields defined in a class are called **class members** or simply **members**. Class variables are referred to as **data members** or **member variables**, whereas methods are referred to as **function members** or **member functions**. To use the fields and methods in a class, you instantiate an object from that class. After you instantiate a class object, class data members (or fields) are referred to as properties of the object and class function members (or methods) are referred to as methods of the object.

Classes themselves are also referred to as *user-defined data types* or *programmer-defined data types*. These terms can be somewhat misleading, however, because they do not accurately reflect the fact that classes can contain function members. In addition, classes usually contain multiple fields of different data types, so calling a class a data type becomes even more confusing. One reason classes are referred to as user-defined data types or programmer-defined data types is that you can work with a class as a single unit, or object, in the same

way you work with a variable. In fact, the terms variable and object are often used interchangeably in object-oriented programming. The term object-oriented programming comes from the fact that you can bundle variables and functions together and use the result as a single unit (a variable or object).

What this means will become clearer to you as you progress through this chapter. For now, think of the handheld calculator example. A calculator could be considered an object of a `Calculation` class. You access all of the `Calculation` methods (such as addition and subtraction) and its fields (operands that represent the numbers you are calculating) through your calculator object. You never actually work with the `Calculation` class yourself, only with an object of the class (your calculator).

But why do you need to work with a collection of related fields and methods as a single object? Why not simply call each individual field and method as necessary, without bothering with all this class business? The truth is: You are not required to work with classes; you can create much of the same functionality without classes as you can by using classes. In fact, many of the scripts that you create—and that you find in use today—do not require object-oriented techniques to be effective. Classes help make complex programs easier to manage, however, by logically grouping related methods and fields and by allowing you to refer to that grouping as a single object. Another reason for using classes is to hide information that users of a class do not need to access or know about. Information hiding, which is explained in more detail later in this chapter, helps minimize the amount of information that needs to pass in and out of an object, which helps increase program speed and efficiency. Classes also make it much easier to reuse code or distribute your code to others for use in their programs. Without a way to package fields and methods in classes and include those classes in a new program, you would need to copy and paste each segment of code you wanted to reuse (methods, fields, and so on) into any new program.



You will learn how to create your own classes and include them in your scripts shortly.



An additional reason to use classes is that instances of objects inherit


their characteristics, such as class members, from the class upon which they are based. This inheritance allows you to build new classes based on existing classes without having to rewrite the code contained in the existing classes.


Working with Access Modifiers

The first thing you need to understand about classes is **access modifiers**, which control a client's access to classes, individual fields, and methods and their members. Table 10-1 lists the access modifiers you can use with C# classes:

Access modifier	Descriptions
<code>public</code>	Allows anyone to access a class or class member.
<code>private</code>	Prevents clients from accessing a class or class member and is one of the key elements in information hiding. Private access does not restrict a class's internal access to its own members; a class method can modify any private class member.
<code>protected</code>	Allows only the class or a derived class to access the class or class member.
<code>internal</code>	Allows a class or class member to be accessed from anywhere in the application, but not from external applications.
<code>protected internal</code>	Allows only code in the same structure, or from a derived class, to access the class or class member.

Table 10-1 C# access modifiers

 If you do not specify an access modifier, C# automatically assigns the `internal` access modifier to the class or class member.

 Derived classes are used with a more advanced object-oriented programming technique called inheritance.

Next, you will learn how to use access modifiers with classes you define.

Creating a Class Definition

A **class definition** contains the class members that make up the class. To create a class definition in C#, you use the `class` keyword and precede it with an access modifier, as follows:

```
accessModifier class ClassName
{
    // Class member definitions
}
```

The `ClassName` portion of the class definition is the name of the new class. You can use any name you want for a structure, as long as you follow the same naming conventions that you use when declaring other identifiers, such as variables and functions. Also, keep in mind that class names usually begin with an uppercase letter to distinguish them from other identifiers. Within the class's curly braces, you declare the fields and methods that make up the class.

You have already seen examples of class definitions with the event handlers that you have worked with since Chapter 4. For example, the following code shows the default code-behind page for an ASP.NET Web site. As you can see, the `_Default` class uses a public access modifier, whereas the `Page_Load()` event handler method uses a protected access modifier.

```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        // Class member definitions
    }
}
```

The `partial` keyword in the preceding definition indicates that the class can be split across multiple files, which allows multiple programmers to work on the same code simultaneously. The colon and `System.Web.UI.Page` at the end of the class definition header indicates that the `_Default` class derives from the `System.Web.UI.Page` class, which means that the `_Default` class inherits the members of the `System.Web.UI.Page` class.



Class names in a class definition are not followed by parentheses, as are function names in a function definition.

537

The following code demonstrates how to declare a public class named `BankAccount` that inherits the members of the `System.Web.UI.Page` class. The statements following the class definition instantiate an object of the class named `checking` and print the object's type (`BankAccount`):

```
public class BankAccount : System.Web.UI.Page
{
    // Class member definitions
}
BankAccount checking = new BankAccount();
Response.Write(checking.GetType());
```

Next, you will start creating the `ShoppingCart` class.

To start creating the `ShoppingCart` class:

1. Select the **Website** menu and then select **Add New Item**. The Add New Item dialog box opens.
2. In the Add New Item dialog box, select **Class** from the Templates list and then change the name of the class file to **ShoppingCart.cs**. Click **Add** and then click **Yes** when prompted to save the file in the `App_Code` folder. The `ShoppingCart` class file opens in the Code Editor window and contains the following statements:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
/// <summary>
/// Summary description for ShoppingCart
/// </summary>
public class ShoppingCart
{
    public ShoppingCart()
    {
        //
        // TODO: Add constructor logic here
        //
    }
}
```

3. Replace the "Summary description for `ShoppingCart`" comment with **Generic shopping cart class**.

4. Open the **shopping_cart.aspx** file in the Code Editor window. Locate the text **[Add code here]** and replace it with a Literal control. Change the ID of the Literal control to **CartBody**.
5. Open the **shopping_cart.aspx.cs** file in the Code Editor window. Add the following statements to the **Page_Load()** event handler to instantiate a **ShoppingCart** object. The statements write success or failure messages to a Literal control with an ID of **CartBody** in the **shopping_cart.aspx** file.

```
try
{
    ShoppingCart cart = new ShoppingCart();
    CartBody.Text = "<p>Successfully instantiated an
                    object of the ShoppingCart class.</p>";
}
catch
{
    CartBody.Text = "<p>The ShoppingCart class
                    is not available!</p>";
}
```

6. Start the Web site and open the Shopping Cart page. You should see the message shown in Figure 10-4.
7. Close your Web browser window.



Figure 10-4 Shopping Cart page after instantiating a **ShoppingCart** object

Collecting Garbage

If you have worked with other object-oriented programming languages, you might be familiar with the term **garbage collection**, which refers to cleaning up, or reclaiming, memory that is reserved by a program. When you declare a variable or instantiate a new object, you are actually reserving computer memory for the variable or object. With some programming languages, you must write code that deletes a variable or object after you are through with it to free the memory for use by other parts of your program, or by other programs running on your computer. With C#, you do not need to worry about reclaiming memory that is reserved for your variables or objects because C# knows when your program no longer needs a variable or object and automatically cleans up the memory for you. The one exception has to do with open database connections. As you learned in Chapter 8, because database connections can take up a lot of memory, you should explicitly close a database connection when you are through with it by calling the `Close()` method. This ensures that the connection doesn't keep taking up space in your computer's memory while the script finishes processing.

Short Quiz 2

1. What is a data structure and what are some of the types of data structures you have worked with in this book?
2. Why are classes referred to as user-defined data types or programmer-defined data types, and why are these terms somewhat misleading?
3. What are some of the benefits to working with classes and objects?
4. Explain the level of protection provided by each of the C# access modifiers.
5. How do you specify in a class definition a class from which another class should inherit its class members?

Declaring Class Fields

In this section, you will learn how to declare fields within a class. Declaring and initializing fields is a little more involved than declaring and instantiating standard C# variables. Before you can declare fields, you must first understand the principle of information hiding, which you will study first.

What Is Information Hiding?

One of the fundamental principles in object-oriented programming is the concept of information hiding. Information hiding gives an encapsulated object its black box capabilities so that users of a class can see only the members of the class that you allow them to see. Essentially, the principle of **information hiding** states that any class members that other programs, sometimes called clients, do not need to access or know about, should be hidden. Information hiding helps minimize the amount of information that needs to pass in and out of an object; this in turn helps increase program speed and efficiency. Information hiding also reduces the complexity of the code that clients see, allowing them to concentrate on the task of integrating an object into their programs. For example, if a client wants to add to her accounting program a `Payroll` object, she does not need to know the underlying details of the `Payroll` object's methods, nor does she need to modify any local fields that are used by those methods. The client only needs to know which of the object's methods to call and what data (if any) needs to be passed to those methods.

Now consider information hiding on a larger scale. Professionally developed software packages are distributed in an encapsulated format, which means that the casual user—or even an advanced programmer—cannot see the underlying details of how the software is developed. Imagine what would happen if Microsoft distributed Excel without hiding the underlying programming details. Most users of the program would be bewildered if they accidentally opened the source files. There is no need for Microsoft to allow users to see the underlying details of Excel because users do not need to understand how the underlying code performs the various types of spreadsheet calculations. Microsoft also has a critical interest in protecting proprietary information, as do you. The design and sale of software components is big business. You certainly do not want to spend a significant amount of time designing an outstanding software component, only to have an unscrupulous programmer steal the code and claim it as his own.

This same principle of information hiding needs to be applied in object-oriented programming. There are few reasons why clients of your classes need to know the underlying details of your code. Of course, you cannot hide all of the underlying code, or other programmers will never be able to integrate your class with their applications. But you need to hide most of it.

Information hiding on any scale also prevents other programmers from accidentally introducing a bug into a program by modifying a class's internal workings. Programmers are curious creatures and will often attempt to “improve” your code, no matter how well it is written.

Before you distribute your classes to other programmers, your classes should be thoroughly tested and bug-free. With tested and bug-free classes, other programmers can focus on the more important task of integrating your code into their programs using the fields and methods you designate.

The opposite of software that adheres to the principles of information hiding is open source software, for which the source code can be freely used and modified. Instead of intentionally hiding the internal workings of a software application for proprietary purposes, open source software encourages programmers to use, change, and improve the software. Open source software can be freely distributed or sold, provided it adheres to the software's copyright license.

To enable information hiding in your classes, you must designate access modifiers for each of your class members, similar to the way you must designate an access modifier for a class definition.

Using Access Modifiers with Fields

You declare a field in the same way that you declare a standard variable, except that you must include an access modifier at the beginning of a field declaration statement. For example, the following statement declares a public field named `balance` in the `BankAccount` class and initializes it with a value of 0:

```
public class BankAccount : System.Web.UI.Page
{
    public double balance = 0;
}
```

As with standard C# variables, it is considered good programming practice to assign an initial value to a field when you first declare it. The best way to initialize a field is with a constructor method (discussed later in this chapter), although you can also assign values to fields when you first declare them.

Recall that to access a field as an object property, you append the property name to the object with a period. The following statements assign a new value to the `balance` field and then print its value:

```
BankAccount checking = new BankAccount();
checking.balance = 743.26;
Response.Write("<p>Your updated checking account balance is "
    + String.Format("{0:C}", checking.balance) + "</p>");
```

Next, you will declare four data members, `dbConnection`, `sqlString`, `tableName`, and `orders[]`, in the `ShoppingCart` class. The `dbConnection`, `sqlString`, and `tableName` fields store the database connection details. The `orders[]` array is an array list that keeps track of the



It is common practice to list public class members first to clearly identify the parts of the class that can be accessed by clients.



Refer to Appendix A for information on how to use array lists.

products in a customer's shopping cart. The array will consist of three elements, each of which contains another array list. The first dimension stores product IDs, the second dimension stores the quantity of each product purchased, and the third dimension stores the table in the Bakery database that contains the product information. To adhere to the principles of information hiding, you must declare all of the data members as private. Later in this chapter, you will write member functions that access and manipulate the values in each array.

To declare four data members in the `ShoppingCart` class:

1. Return to the **ShoppingCart.cs** file in the Code Editor window.
2. Add the following using directives to the end of the using directives list.
3. Add the following declaration statements to the class. Be sure to replace the value assigned to the Data Source property with the name of the SQL Server instance to which you want to connect. Note that you must include the entire path to the Bakery.mdf file.

```
using System.Collections;
using System.Data.SqlClient;

public class ShoppingCart
{
    private SqlConnection dbConnection
        = new SqlConnection("Data Source=
        .\\SQLEXPRESS;AttachDbFilename
        ='path\\Bakery.mdf'; Integrated Security=True;
        User Instance=True");
    private ArrayList productID = new ArrayList();
    private ArrayList productQuantity = new ArrayList();
    private ArrayList productTable = new ArrayList();
    public ShoppingCart()
    {
        //
        // TODO: Add constructor logic here
        //
    }
}
```

Serializing Objects

In Chapter 9, you learned about C#'s various state preservation techniques, including how to use sessions. In addition to keeping track of current Web site visitors, session variables can store information that can be shared among multiple scripts that are called as part of the same session. But how do you share objects within the same session or across multiple sessions and applications? You could assign

the value of an object's fields to session variables, but you would need to instantiate a new object and reassign the session variable values to the fields each time you execute a program. However, this approach would be difficult if you have an object with dozens of fields. A better choice is to serialize the object.

Serialization refers to the process of converting an object's fields into a string that you can store for reuse. The .NET Framework supports two types of serialization technologies: binary serialization and XML serialization. **Binary serialization** converts object properties to a binary format, whereas **XML serialization** converts object properties to XML. Binary serialization is more efficient in terms of speed and memory usage, and it converts all of an object's fields to binary format while maintaining their data types. XML serialization converts only an object's public fields and properties to XML and does not maintain their data types. Because only the .NET Framework can read binary serialized objects created with C#, you would use XML serialization if you need to share the serialized data with another application. This chapter discusses binary serialization.

Before you can serialize a class object, you must mark the class as serializable by adding the `Serializable` attribute immediately above the class definition, surrounded by brackets (`[]`), as follows:

```
[Serializable]
public class BankAccount : System.Web.UI.Page
{
    // Class member definitions
}
```

Binary serialized objects are most commonly stored in binary files on a local computer. For this reason, you need to understand how to create a **file stream**, which is used for accessing a resource, such as a file, that you can read from and write to. An **input stream** reads data from a resource (such as a file), whereas an **output stream** writes data to a resource (again, such as a file). You have already used an output stream frequently with `Response.Write()` statements, which send data to an output stream (the Web browser window). Using a file stream involves the following steps:

1. Create an object of the `FileStream` class, passing to the class constructor the name and path of the file and a parameter that specifies what to do with the file. To create a file, you pass `FileMode.Create` as the second parameter of the class constructor. To open a file, you pass `FileMode.Open` as the second parameter of the class constructor.
2. Write data to or read data from the file stream.
3. Close the file stream with the `Close()` method.



You must include the `System.IO` namespace before you

can use the `FileStream` class.



This section only covers the most basic methods for working with the `FileStream` class. For more information, refer to the “`FileStream` Class” page on MSDN at <http://msdn.microsoft.com/en-us/library/system.io.filestream.aspx>.



You must include the `System.Runtime.Serialization.Formatters`.

Binary namespace before you can use the `BinaryFormatter` class.

The following statements demonstrate how to create, work with, and then close a `FileStream` object named `accountFile`. The class constructor creates a new file named `accountInfo.dat` in a folder named `accountData` on the C drive.

```
FileStream accountFile =
    new FileStream(
        @"C:\\accountData\\accountInfo.dat",
        FileMode.Create);
// Statements that write data to or read data from
// the file stream
accountFile.Close();
```

To serialize an object, you must first create an object of the `BinaryFormatter` class, which serializes and deserializes objects in binary format. The following statements demonstrate how to create the checking object and a `BinaryFormatter` object named `savedAccount`:

```
BankAccount checking = new BankAccount();
BinaryFormatter savedAccount = new BinaryFormatter();
```

Once you have created an object of the class you want to serialize along with a `FileStream` object and a `BinaryFormatter` object, you call the `Serialize()` method of the `BinaryFormatter` object, passing to it the `FileStream` object and then the class object you want to serialize. The following statements show a complete example of how to serialize a `BankAccount` object and save its data to a file named `accountInfo.dat`:

```
BankAccount checking = new BankAccount();
// Statements that modify the fields in the BankAccount object
BinaryFormatter savedAccount = new BinaryFormatter();
FileStream accountFile =
    new FileStream(
        @"C:\\accountData\\accountInfo.dat",
        FileMode.Create);
savedAccount.Serialize(accountFile, checking);
accountFile.Close();
```

To convert serialized data back into an object, you must first open a file stream, passing to it a value of `FileMode.Open` as the second parameter of the `FileStream` class constructor. The remainder of the steps are the same as the preceding serialization steps, except that you call the `Deserialize()` method instead of the `Serialize()` method of the `BinaryFormatter` object. The following statements demonstrate how to serialize a binary file and deserialize its data back into a `BankAccount` object. Notice that the deserialization statement instantiates an object of the `BankAccount` class and then casts the result returned from the `Deserialize()` method into a `BankAccount` class object.


```

BinaryFormatter savedAccount = new BinaryFormatter();
FileStream accountFile =
    new FileStream(@"C:\accountData\accountInfo.dat",
        FileMode.Open);
BankAccount checking =
    (BankAccount)savedAccount.Deserialize(accountFile);
accountFile.Close();

```

Recall that binary serialization converts all of an object's fields to binary format. However, you don't necessarily have to serialize each and every field in a class, particularly for large objects that contain numerous fields. For fields that you do not need to serialize, add the `NonSerialized` attribute before the declaration statement, surrounded by brackets ([]). The following statement demonstrates how to prevent a field named `interestRate` from being serialized:

```

[NonSerialized]
public double interestRate;

```

When working with a shopping cart, you do not normally need to serialize and store order information in a file. Instead, you just use session state to maintain the shopping cart for the duration of the current session. The following statement demonstrates how to assign the checking object to a session variable named `myAccount`:

```

Session["myAccount"] = checking;

```

To restore a serialized object from a session variable, you must cast the session variable to the `BankAccount` class, and then assign the session variable to a `BankAccount` object, as follows:

```

curAccount = (BankAccount)Session["myAccount"];

```

Event handlers that are called when a user clicks a product's Order Now button will handle the creation and storage of `ShoppingCart` objects. When a user clicks an Order Now button, the button's event handler will check if a `ShoppingCart` object exists in a session variable named `savedCart`. If the object does exist, the event handler calls a method that adds the selected product to the `ShoppingCart` object. If the object does not exist, the event handler creates it before attempting to add the selected product. The Shopping Cart page will use the `Page_Load()` event handler to check if the `savedCart` session variable exists. If so, the selected products are printed to a table. If not, a message prints to a Literal control named `CartBody` that informs the user that the shopping cart is empty.

Next, you will add code to the Shopping Cart page's `Page_Load()` event handler that checks if the `savedCart` session variable exists.

To add code to the Shopping Cart page's `Page_Load()` event handler that checks if the `savedCart` session variable exists:

1. Return to the **shopping_cart.aspx.cs** file in the Code Editor window.

2. Add the following statement to the beginning of the `Page_Load()` event handler to instantiate a `ShoppingCart` object:

```
ShoppingCart curCart;
```

3. Replace the `try...catch` block with the following `if` statement that checks if the `savedCart` session variable exists. Later in the chapter, you will learn how to create methods that access the fields stored in the `ShoppingCart` object.

```
if (Session["savedCart"] != null)
{
    curCart =
        (ShoppingCart)Session["savedCart"];
}
```

4. Add the following `else` statement to the end of the `Page_Load()` event handler.

```
else
{
    CardBody.Text = "<p>Your shopping cart is empty.</p>";
}
```

5. Start the Web site and open the Shopping Cart page. You should see the message indicating that the shopping cart is empty.
6. Close your Web browser window.

Short Quiz 3

1. Why should you hide any class members that other programmers do not need to access or know about?
2. How do the principles of information hiding compare with open source software?
3. What are the two ways in which you can assign an initial value to a field?
4. What are the differences between binary and XML serialization? When would you use each type of serialization method?
5. What class do you use to serialize and deserialize objects in binary format, and what class do you use to create a file stream object? How do these two classes work together in the serialization/deserialization processes?

Working with Class Methods

Because methods perform most of the work in a class, you now learn about the various techniques associated with them. Methods are usually declared as public or private, but they can also be declared with any of the other types of access modifiers. Public methods can be called by anyone, whereas private methods can be called only by other methods in the same class.

You might wonder about the usefulness of a private method, which cannot be accessed by a client of the program. Suppose your program needs some sort of utility method that clients have no need to access. For example, the `BankAccount` class might need to calculate interest by calling a method named `calcInterest()`. Because the `calcInterest()` method can be called automatically from within the `BankAccount` class, the client does not need to access the `calcInterest()` method directly. By making the `calcInterest()` method private, you protect your program and add another level of information hiding. A general rule of thumb is to create as public any methods that clients need to access and to create as private any methods that clients do not need to access. The protected and protected internal access modifiers are more flexible than the private access modifier because they also allow derived classes to access a method. The internal access modifier allows a method to be accessed from anywhere in the application, but not from external applications.

You declare a method within the body of a class definition and include an access modifier before the method's return type. Other than including an access modifier, there is little difference between standard functions and methods. You are not required to define a method with an access modifier. If you do exclude the access modifier, the method's default access is internal. However, it's good programming practice to include an access modifier with any method definition to clearly identify the scope of the method. The following statement demonstrates how to declare a method named `withdrawal()` in the `BankAccount` class:

```
public class BankAccount : System.Web.UI.Page
{
    public double balance = 958.20;
    public void withdrawal(double amount) {
        balance -= amount;
    }
}
BankAccount checking = new BankAccount();
checking.withdrawal(200);
Response.Write("<p>Your updated checking account balance is "
    + String.Format("{0:C}", checking.balance) + "</p>");
```

Initializing with Constructor Methods

When you first instantiate an object from a class, you will often want to assign initial values to fields or perform other types of initialization tasks, such as calling a method that might calculate and assign values to fields. Although you can assign simple values to fields when you declare them, a better choice is to use a constructor method. A **constructor method** is a special method that is called automatically when an object from a class is instantiated. You define and declare constructor methods the same way you define other methods, although you do not include a return type because constructor methods do not return values. Each class definition can contain one or more constructor methods whose names are the same as the class. You must specify the public access modifier with a constructor method. The following code demonstrates how to use the `BankAccount()` constructor method to initialize the fields in the `BankAccount` class:

```
public class BankAccount : System.Web.UI.Page
{
    private string accountNumber;
    private string customerName;
    private double balance;
    public BankAccount() {
        accountNumber = "012345678";
        balance = 0;
        customerName = "";
    }
}
```

Constructor methods are commonly used to handle database connection tasks. Next, you will add to the `ShoppingCart` class's constructor method that contains statements that instantiate a new database object.

To add to the `ShoppingCart` class's constructor method statements that instantiate a new database object:

1. Return to the **ShoppingCart.cs** file in the Code Editor window.
2. Replace the comments in the constructor method with the following statement, which opens the database connection:

```
dbConnection.Open();
```

Cleaning Up with Destructor Methods

Just as a default constructor method is called when a class object is first instantiated, a destructor method is called when the object is destroyed. A **destructor method** cleans up any resources allocated to an object after the object is destroyed. You cannot explicitly call a destructor method. Instead, it is called automatically by the C#

garbage collection. You generally do not need to use a destructor method, although many programmers use one to close file database connections. To add a destructor method to a C# class, create a method with the same name as the class, but preceded by a tilde symbol (~). Note that you do not specify an access modifier or data type for a destructor method. The following code contains a destructor method that closes an open database connection:

```
public class BankAccount : System.Web.UI.Page
{
    private SqlConnection dbConnection;
    public BankAccount()
    {
        dbConnection = new SqlConnection(
            "Data Source=DBSERVER\\SQLEXPRESS;
            Integrated Security=true");
        dbConnection.Open();
        dbConnection.ChangeDatabase("accountDB");
    }
    ~BankAccount()
    {
        dbConnection.Close();
    }
}
```

Next, you will add to the `ShoppingCart` class a destructor method that closes the database object that you instantiated with the constructor method.

To add a destructor method:

1. Return to the **ShoppingCart.cs** file in the Code Editor window.
2. Add the following destructor method definition to the end of the class:

```
~ShoppingCart()
{
}
```

3. Add to the destructor method the following statement that closes the database object:

```
dbConnection.Close();
```

Writing Accessors

Even if you make all fields in a class private, you can still allow your program's clients to retrieve or modify the value of fields via accessor methods. **Accessor methods** are public methods that a client can call to retrieve or modify the value of a field. Because accessor methods often begin with the words "set" or "get," they are also referred to as

set or get methods. Set methods modify field values; get methods retrieve field values. To allow a client to pass a value to your program that will be assigned to a private field, you include parameters in a set method's definition. You can then write code in the body of the set method that validates the data passed from the client, prior to assigning values to private fields. For example, if you write a class named `Payroll` that includes a private field containing the current state income-tax rate, you could write a public accessor method named `getStateTaxRate()` that allows clients to retrieve the variable's value. Similarly, you could write a `setStateTaxRate()` method that performs various types of validation on the data passed from the client (such as making sure the value is not null, is not greater than 100%, and so on) prior to assigning a value to the private state tax rate field.

The following code demonstrates how to use set and get methods with the `balance` field in the `BankAccount` class. The `setBalance()` method is declared with an access modifier of `public` and accepts a single parameter containing the value to assign to the `balance` field. The `getBalance()` method is also declared as `public` and contains a single statement that returns the value assigned to the `balance` field. Statements at the end of the example call the methods to set and get the `balance` field.

```
public class BankAccount : System.Web.UI.Page
{
    private double balance = 0;
    public void setBalance(double newBalance)
    {
        balance = newBalance;
    }
    public double getBalance()
    {
        return balance;
    }
}

BankAccount checking = new BankAccount();
checking.setBalance(457.63);
Response.Write("<p>Your updated checking account balance is "
    + String.Format("{0:C}", checking.getBalance()) + "</p>");
```

Next, you will add two accessor methods to the `Bakery` class: `addItem()` and `showCart()`. When the user clicks one of the Order Now buttons from a product page, the `addItem()` method will add new elements to the `productID`, `productQuantity`, and `productTable` fields for the selected item. The `showCart()` method displays the shopping cart when the user selects a new item or opens the Shopping Cart page.

To add `addItem()` and `showCart()` accessor methods to the `Bakery` class:

1. Return to the **ShoppingCart.cs** file in the Code Editor window.
2. Add the following `addItem()` method definition above the class constructor method. This method returns a Boolean value of true if the item was added successfully or false if it already exists in the shopping cart.

```
public bool addItem(string prodID, string table)
{
}
```

3. Add to the `addItem()` method the following `foreach` statement, which loops through the elements in the `productID[]` array list. If the product ID is found, the method returns a value of false and ends because the item already exists in the shopping cart.

```
foreach (string item in productID)
{
    if (item == prodID)
        return false;
}
```

4. Add to the end of the `addItem()` method the following statements, which assign values to the `productID`, `productQuantity`, and `productTable` fields for the selected item. The last statement returns a value of false, indicating that the item was successfully added to the shopping cart.

```
productID.Add(prodID);
productQuantity.Add(1);
productTable.Add(table);
return true;
```

5. Add the following `showCart()` method definition above the class constructor method. This method works by building a table containing the products in the shopping cart. The table is then returned to the calling function.

```
public string showCart()
{
}
```

6. Add the following statements to the `showCart()` method to begin building the table string. The `total` variable will store a running total of the items in the shopping cart.

```
string retValue = "<table width='100%'
    cellpadding='3' rules='all'
    border='1' id='ProductGrid'
    style='background-color:#DEBA84;
    border-color:#DEBA84;border-width:1px;
    border-style:None;font-size:Smaller;'>";
```

```
retValue += "<tr style='color:White;
background-color:#A55129;
font-weight:bold;'><th align='center'>
Product</th><th align='center'>
Quantity</th><th align='center'>
Price Each</th></tr>";
double total = 0;
```

7. Add the following for loop to the end of the showCart() method. These statements run an ExecuteReader() method to open a recordset for each item in the shopping cart. The code reads the price for each item and builds the body of the table.

```
for (int i = 0; i < productID.Count; ++i)
{
    string sqlString = "SELECT * FROM "
        + productTable[i] + " WHERE productID='"
        + productID[i] + "'";
    SqlCommand prodCommand = new
        SqlCommand(sqlString, dbConnection);
    SqlDataReader prodRecords =
        prodCommand.ExecuteReader();
    if (prodRecords.Read())
    {
        retValue += "<tr style='color:#8C4510;
background-color:#FFF7E7;'>"
            + "<td>" + prodRecords["name"] + "</td>"
            + "<td align='center'>"
            + productQuantity[i] + "</td>"
            + "<td align='center'>"
            + String.Format("{0:C}",
                prodRecords["price"])
            + "</td></tr>";
        double price = Convert.ToDouble(
            prodRecords["price"]);
        int quantity = Convert.ToInt16(
            productQuantity[i]);
        total += price * quantity;
    }
    prodRecords.Close();
}
```

8. Add to the end of the showCart() method the following statements, which complete the table and return the string to the calling function:

```
retValue += "<td align='center' colspan='2'>
<strong>Your shopping cart contains "
+ productQuantity.Count
+ " product(s).</strong></td>";
retValue += "<td align='center'><strong>Total: "
+ String.Format("{0:C}", total)
+ "</strong></td></tr>";
retValue += "<asp:Button runat='server'
Text='Button' /></table>";
return retValue;
```

9. Return to the shopping_cart.aspx.cs file in the Code Editor window.
10. Add the following statements to the end of the if statement. These statements call the showCart() function and assign the returned results to the CartBody literal.

```
string retString = curCart.showCart();
CartBody.Text = retString;
```

Next, you will add code to the product pages and Shopping Cart page that calls the new ShoppingCart class methods.

To add code to the product pages and Shopping Cart page that calls the new ShoppingCart class methods:

1. Open **bread.aspx** in the Code Editor window and add the following definition for an event handler named ProductGrid_SelectedIndexChanged().
2. Add to the ProductGrid_SelectedIndexChanged() event handler the following statements. These statements create a new ShoppingCart object or open the existing ShoppingCart object from the session variable, then execute the addItem() method. Depending on the result returned from the addItem() method, the remaining statements either print "You already selected that item" in the Literal control or redirect the browser to the Shopping Cart page.

```
protected void ProductGrid_SelectedIndexChanged(
    object sender, EventArgs e)
{
}
```

```
ShoppingCart curCart;
if (Session["savedCart"] == null)
    curCart = new ShoppingCart();
else
{
    curCart = (ShoppingCart)Session["savedCart"];
}
bool addResult = curCart.addItem(
    ProductGrid.SelectedValue.ToString(), "Breads");
if (addResult == false)
    ProductPage.Text = "<p>You already selected
    that item!</p>";
else
{
    Session["savedCart"] = curCart;
    Response.Redirect("shopping_cart.aspx");
}
```

- Return to **bread.aspx** in the Code Editor window and add the following event handler just before the closing bracket (>) for the <asp:GridView> control:

```
onSelectedIndexChanged="ProductGrid_  
SelectedIndexChanged"
```

- Repeat Steps 1 and 3 for the **cakes.aspx**, **cookies.aspx**, and the **pies.aspx** files. Replace the second argument that is passed to the **addItem()** function with the appropriate product type. For example, you should change the argument in the **cakes.aspx** file from "Breads" to "Cakes".
- Return to the **shopping_cart.aspx.cs** file in the Code Editor window.
- Start the Web site and test the product pages and Shopping Cart page. Figure 10-5 shows the Shopping Cart page after adding several items.
- Close your Web browser window.



Figure 10-5 Shopping Cart page after adding several items

The preceding techniques are the traditional ways of creating accessors in object-oriented programming languages. C# allows you to create accessors using **properties**, which are special methods that you can use as public data members to set and get field values. To create a property, you create a constructor that is similar to a method definition and includes an accessor level and data type, but does not include parentheses at the end of the method name. To create a property's set and get methods, you include the set and get keywords within the property definition. Following the set and get keywords, you place the necessary statements for each method within a set of braces. For the get method, you can perform any type of computation and then return the value using a return statement. The set method includes an implicit parameter named `value` that represents the value being assigned to the field. The following example demonstrates how to set and get the balance in the `BankAccount` program as a property:

```
public class BankAccount : System.Web.UI.Page
{
    private double balance = 0;
    public double Balance {
        get { return balance; }
        set { balance = value; }
    }
}
BankAccount checking = new BankAccount();
checking.Balance = 457.63;
Response.Write("<p>Your updated checking account balance is "
    + String.Format("{0:C}", checking.Balance) + "</p>");
```

Short Quiz 4

1. Why would you use a private class method?
2. If you exclude an access modifier when declaring a class method, what access level does C# use by default?
3. What is the required syntax for declaring constructor and destructor methods?
4. Why would you use accessor methods? Why do they often begin with the words "set" or "get"?
5. How are accessor methods related to C# properties? How do you create C# properties and access them through an instantiated class object?

Summing Up

- An object is programming code and data that can be treated as an individual unit or component.
- Objects are encapsulated, which means that all code and required data are contained within the object itself. Encapsulation places code inside what programmers like to call a “black box.” When an object is encapsulated, you cannot see “inside” it—all internal workings are hidden.
- The term *interface* refers to the methods and properties that are required for a source program to communicate with an object.
- In object-oriented programming, the code, methods, attributes, and other information that make up an object are organized into classes, which is essentially a template, or blueprint, that serves as the basis for new objects.
- When you create an object from an existing class, you are said to be instantiating the object. A particular instance of an object inherits its methods and properties from a class—that is, it takes on the characteristics of the class on which it is based.
- In C#, the term *class* more specifically refers to data structures that contain fields along with methods for manipulating the fields. The term *data structure* refers to a system for organizing data, whereas the term *field* refers to variables that are defined within a class.
- The methods and fields defined in a class are called class members or simply members. Class variables are referred to as data members or member variables, whereas class methods are referred to as function members or member functions.
- Classes help make complex programs easier to manage by logically grouping related methods and fields and by allowing you to refer to that grouping as a single object.
- Access modifiers control a client’s access to classes, individual data members, and function members.
- To create a class in C#, you use the `class` keyword and an access modifier to write a class definition, which contains the class members that make up the class.
- The term *garbage collection* refers to cleaning up, or reclaiming, memory that is reserved by a program. With C#, you do

not need to worry about reclaiming memory that is reserved for your variables or objects because C# knows when your program no longer needs a variable or object and automatically cleans up the memory for you. The one exception has to do with open database connections, which you do need to close manually.

- The principle of information hiding states that any class members that other programmers, sometimes called clients, do not need to access or know about should be hidden.
- You declare a field in the same way that you declare a standard variable, except that you must include an access modifier at the beginning of a field declaration statement.
- Serialization is the process of converting an object's fields into a string that you can store for reuse. Binary serialization converts object properties to a binary format, whereas XML serialization converts object properties to XML.
- A file stream is used for accessing a resource, such as a file, that you can read from and write to. An input stream reads data from a resource (such as a file), whereas an output stream writes data to a resource (again, such as a file).
- Methods are usually declared as public or private, but they can also be declared with any of the other types of access modifiers. Public methods can be called by anyone, whereas private methods can be called only by other methods in the same class.
- A general rule of thumb is to create as public any methods that clients need to access and to create as private any methods that clients do not need to access.
- A constructor method is a special method that is called automatically when an object from a class is instantiated.
- A destructor method cleans up any resources allocated to an object after the object is destroyed.
- Accessor methods are public methods that a client can call to retrieve or modify the value of a field. Because accessor methods often begin with the words "set" or "get," they are also referred to as set or get methods.
- C# allows you to create accessors using properties, which are special methods that you can use as public data members to set and get field values.

Comprehension Check

1. Reusable software objects are often referred to as _____.
 - a. methods
 - b. components
 - c. widgets
 - d. functions
2. Explain the benefits of object-oriented programming.
3. The term *black box* refers to _____.
 - a. a property
 - b. debugging
 - c. encapsulation
 - d. an interface
4. Users can see all of the methods and properties within an encapsulated object. True or False?
5. A(n) _____ is an object that has been created from an existing class.
 - a. pattern
 - b. structure
 - c. replica
 - d. instance
6. What is inheritance? How is it used with classes?
7. The functions associated with an object are called _____. (Choose all that apply.)
 - a. properties
 - b. function members
 - c. methods
 - d. attributes
8. The terms *variable* and *object* are often used interchangeably in object-oriented programming. True or False?

9. Class names usually begin with a(n) _____ to distinguish them from other identifiers.
- a. number
 - b. exclamation mark (!)
 - c. ampersand (&)
 - d. uppercase letter
10. Which of the following access specifiers prevents clients from calling methods or accessing fields? (Choose all that apply.)
- a. `public`
 - b. `private`
 - c. `protected`
 - d. `internal`
11. Which access modifier does C# use by default if you do not include one in a class or class member definition?
- a. `public`
 - b. `private`
 - c. `protected`
 - d. `internal`
12. Explain how to create a class definition.
13. Class names in a class definition are followed by parentheses, the same as with a function definition. True or False?
14. For which of the following programmatic constructs do you need to perform garbage collection?
- a. variables
 - b. objects
 - c. database connections
 - d. file streams
15. Explain the principle of information hiding.

16. What types of serialization does the .NET Framework support? (Choose all that apply.)
 - a. binary
 - b. unary
 - c. XML
 - d. database
17. Explain how to serialize a class object to a file.
18. When is a destructor called?
 - a. when the object is destroyed
 - b. when the constructor method ends
 - c. when you delete a class object with the `unset()` method
 - d. when you call the `serialize()` method
19. Explain the use of accessor functions. How are they often named?
20. What is a property in the context of a C# class? How do you create a property?

Reinforcement Exercises

Exercise 10-1

In this exercise, you will add two member functions, `removeItem()` and `emptyCart()`, to the `ShoppingCart` class. These functions allow you to remove individual items or empty all items from the shopping cart.

To add the `removeItem()` and `emptyCart()` member functions to the `ShoppingCart` class:

1. Return to the **ShoppingCart.cs** file in the Code Editor window.
2. Add the following `removeItem()` method definition above the class constructor:

```
public void removeItem(string prodID)
{
}
```

3. Add to the `removeItem()` method the following statements, which loop through the `productID[]` array list until the element that matches the `prodID` parameter is found. Then, the `if` statement deletes the associated elements in the `productID[]`, `productQuantity[]`, and `productTable[]` methods.

```
for (int i = 0; i < productID.Count; ++i)
{
    if (productID[i].ToString() == prodID)
    {
        productID.RemoveAt(i);
        productQuantity.RemoveAt(i);
        productTable.RemoveAt(i);
        break;
    }
}
```

4. Add the following `emptyCart()` method definition above the class constructor:

```
public void emptyCart(string prodID)
{
}
```

5. Add the following statements to the `emptyCart()` method definition. The statements empty the cart by calling the `Clear()` method for the `productID[]`, `productQuantity[]`, and `productTable[]` methods.

```
productID.Clear();
productQuantity.Clear();
productTable.Clear();
```

6. Next, you need to modify the `showCart()` method so it displays links that call the `removeItem()` and `emptyCart()` functions. First, modify the second statement that creates the table header (`<th>`) elements so it includes another column for the remove item links, as follows:

```
retValue += "<tr style='color:White;
background-color:#A55129;
font-weight:bold;'><th align='center'>
Remove</th><th align='center'>
Product</th><th align='center'>
Quantity</th><th align='center'>
Price Each</th></tr>";
```

7. Modify the value assigned to the `retValue` variable in the `if` statement in the `showCart()` function as follows. These statements create new table cells containing Remove Item links. Notice that a query string named `operation` is appended to

the `shopping_cart.aspx` URL. This query string notifies the class which method to call. In this case, the `removeItem()` method is being called.

```
retValue += "<tr style='color:#8C4510;
background-color:#FFF7E7;'>"
+ "<td align='center'>"
+ "<a href='shopping_cart.aspx?operation"
+ "=removeItem&productID=" + productID[i]
+ "'>Remove</a></td>" + "<td>"
+ prodRecords["name"] + "</td>"
+ "<td align='center'>" + productQuantity[i]
+ "</td>" + "<td align='center'>"
+ String.Format("{0:C}", prodRecords["price"])
+ "</td></tr>";
```

8. Add the following statement after the for loop's closing brace. This statement adds an Empty Cart link to the end of the shopping cart table.

```
retValue += "<td align='center'>"
+ "<a href='shopping_cart.aspx?operation=emptyCart'"
+ "Empty Cart</a></td>";
```

9. Return to the `shopping_cart.aspx.cs` script in the Code Editor window and modify the if statement that checks if the `savedCart` session variable exists so it includes nested if statements that call the `removeItem()` and `emptyCart()` methods, as follows. Also, enclose within an else construct the statements that call the `showCart()` method and assign the return value to the Literal control. Your statements should appear as follows:

```
if (Session["savedCart"] != null)
{
    curCart = (ShoppingCart)
        Session["savedCart"];
    if (Request.QueryString["operation"]
        == "removeItem")
    {
        curCart.removeItem(
            Request.QueryString["productID"]);
        Response.Redirect("shopping_cart.aspx");
    }
    else if (Request.QueryString["operation"]
        == "emptyCart")
    {
        curCart.emptyCart(Request.QueryString
            ["productID"]);
        Response.Redirect("shopping_cart.aspx");
    }
}
```



```

    }
    else
    {
        string retString = curCart.showCart();
        CartBody.Text = retString;
    }
}
else
{
    CartBody.Text = "<p>Your shopping cart
    is empty.</p>";
}

```

10. Start the Web site and test the Remove and Empty Cart links on the Shopping Cart page. Figure 10-6 shows the Shopping Cart page after adding the remove item and empty cart functionality.
11. Close your Web browser window.



Figure 10-6 Shopping Cart Web page after adding the remove item and empty cart functionality

Exercise 10-2

In this project, you will add two member functions, `addOne()` and `removeOne()`, to the `ShoppingCart` class. These functions allow you to change the quantities of products in the shopping cart.

To add the `addOne()` and `removeOne()` member functions to the `ShoppingCart` class:

1. Return to the **ShoppingCart.cs** file in the Code Editor window.
2. Add the following `addOne()` method definition above the class constructor:

```
public void addOne(string prodID)
{
}
```

3. Add to the `addOne()` method the following statements, which increment a product's quantity:

```
for (int i = 0; i < productID.Count; ++i)
{
    if (productID[i].ToString() == prodID)
    {
        productQuantity[i] = Convert.ToInt16(
            productQuantity[i]) + 1;
        break;
    }
}
```

4. Add the following `removeOne()` method definition above the class constructor:

```
public void removeOne(string prodID)
{
}
```

5. Add to the `removeOne()` method the following statements, which decrement a product's quantity:

```
for (int i = 0; i < productID.Count; ++i)
{
    if (productID[i].ToString() == prodID)
    {
        productQuantity[i] = Convert.ToInt16(
            productQuantity[i]) - 1;
        if (Convert.ToInt16(productQuantity[i]) == 0)
        {

```

```

        productID.RemoveAt(i);
        productQuantity.RemoveAt(i);
        productTable.RemoveAt(i);
    }
    break;
}
}

```

6. Modify the value assigned to the `retValue` variable in the `if` statement in the `showCart()` function, as follows. These statements create Add and Remove links within the cell containing the product quantity.

```

retValue += "<tr style='color:#8C4510;
background-color:#FFF7E7;'>"
+ "<td align='center'"
+ "<a href='shopping_cart.aspx?operation=removeItem&productID=" + productID[i]
+ "'>Remove</a></td>" + "<td>"
+ prodRecords["name"] + "</td>"
+ "<td align='center'" + productQuantity[i]
+ "<br /><a href='shopping_cart.aspx?operation=addOne&productID="
+ productID[i] + "'>Add</a>&nbsp;"
+ "<a href='shopping_cart.aspx?operation=removeOne&productID="
+ productID[i] + "'>Remove</a>"
+ "<td align='center'" + String.Format("{0:C}",
prodRecords["price"]) + "</td></tr>";

```

7. Return to the `shopping_cart.aspx.cs` script in the Code Editor window and modify the `Page_Load()` event handler so it includes nested `if` statements that call the `addOne()` and `removeOne()` methods, as follows:

```

if (Request.QueryString["operation"] == "removeItem")
{
    curCart.removeItem(
        Request.QueryString["productID"]);
    Response.Redirect("shopping_cart.aspx");
}
else if (Request.QueryString["operation"] == "emptyCart")
{
    curCart.emptyCart(Request.QueryString["productID"]);
    Response.Redirect("shopping_cart.aspx");
}
else if (Request.QueryString["operation"] == "addOne")
{
    curCart.addOne(Request.QueryString["productID"]);
    Response.Redirect("shopping_cart.aspx");
}

```

```

else if (Request.QueryString["operation"] == "removeOne")
{
    curCart.removeOne(Request.QueryString["productID"]);
    Response.Redirect("shopping_cart.aspx");
}
else
{
    string retString = curCart.showCart();
    CartBody.Text = retString;
}

```

8. Start the Web site and test the Add and Remove links on the Shopping Cart page. Figure 10-7 shows the Shopping Cart page after adding functionality to change the product quantity.
9. Close your Web browser window.



Figure 10-7 Shopping Cart Web page after adding functionality to change the product quantity

Discovery Projects

Save the Web sites you create for the following projects in your Projects folder for Chapter 10.

Project 10-1

567

Create a `HitCounter` class that counts the number of hits to a Web page and stores the results in a database. Use a private data member to store the number of hits and include public set and get member functions to access the private counter member variable. Save the project in a folder named **HitCounter** in your Projects folder for Chapter 10.

Project 10-2

Create a `GuestBook` class that stores Web site visitor names in a database. Use a private data member to store visitor names and include public set and get member functions to access the private visitor name member variable. Save the project in a folder named **GuestBook** in your Projects folder for Chapter 10.

Project 10-3

Create a `Movies` class that determines the cost of a ticket to a cinema, based on the moviegoer's age. Assume that the cost of a full-price ticket is \$10. Assign the age to a private data member. Use a public member function to determine the ticket price, based on the following schedule:

Age	Price
Under 5	Free
5 to 17	Half price
18 to 55	Full price
Over 55	\$2 off

Save the project in a folder named **Movies** in your Projects folder for Chapter 10.

Project 10-4

Create a program that calculates how long it takes to travel a specified number of miles, based on speed, number of stops, and weather conditions for a passenger train that averages a speed of 50 mph. Each stop of the train adds an additional five minutes to the train's schedule. In addition, during bad weather the train can only average a speed of 40 mph. Write a class-based version of this script with a

class named **Train**. Save each piece of information you gather from the user in a private data member, and write the appropriate get and set functions for setting and retrieving each data member. Save the project in a folder named **Train** in your Projects folder for Chapter 10.

568

Project 10-5

In Chapter 5, you wrote a script that calculates the correct amount of change to return when performing a cash transaction. Write a class-based version of this script with a class named **Change**. Allow the user (a cashier) to enter the cost of a transaction and the exact amount of money that the customer hands over to pay for the transaction. Use set and get functions to store and retrieve both amounts to and from private data members. Then use member functions to determine the largest amount of each denomination to return to the customer. Assume that the largest denomination a customer will use is a \$100 bill. Therefore, you will need to calculate the correct amount of change to return for \$50, \$20, \$10, \$5, and \$1 bills, along with quarters, dimes, nickels, and pennies. For example, if the price of a transaction is \$5.65 and the customer hands the cashier \$10, the cashier should return \$4.35 to the customer. Include code that requires the user to enter a numeric value for the cash transaction. Save the project in a folder named **Change** in your Projects folder for Chapter 10.

Project 10-6

Create a **BankAccount** class that allows users to calculate the balance in a bank account. The user should be able to enter a starting balance, and then calculate how that balance changes when she makes a deposit, withdraws money, or enters any accumulated interest. Add the appropriate data members and member functions to the **BankAccount** class that will enable this functionality. Also, add code to the class that ensures that the user does not overdraw her account. Be sure that the program adheres to the information-hiding techniques presented in this chapter. Save the project in a folder named **BankAccount** in your Projects folder for Chapter 10.