.NET DEVELOPER'S GUIDE

murach's C#

(Chapter 12)

Joel Murach
Doug Lowe

Mike Murach & Associates

3484 West Gettysburg, Suite 101
Fresno, CA 93722-7801
(559) 440-9071 • (800) 221-5528

murachbooks@murach.com • www.murach.com

Copyright © 2004 Mike Murach & Associates. All rights reserved.

12

How to create and use classes

This chapter presents the basics of creating and using classes in C# applications. Here, you'll learn how to create classes that include properties, methods, fields, and constructors, as well as classes that contain static members. In addition, you'll see a complete application that uses three user-defined classes.

When you complete this chapter, you'll start to see how creating your own classes can help simplify the development of an application. As a bonus, you'll have a better understanding of how the .NET classes work.

An introduction to classes 330
How classes can be used to structure an application 330
The members you can define within a class 332
The code for the Product class 334
How instantiation works 336
How to create a class 338
How to add a class file to a project 338
How to code fields 340
How to code properties 342
How to code methods 344
How to code constructors 346
How to code static members 348
The Product Maintenance application 350
The operation of the Product Maintenance application 350
The classes used by the Product Maintenance application 352
The code for the Product Maintenance application 354
How to work with classes in Visual Studio 358
How to use the Class View window 358
How to use wizards to create class members 360
Perspective 362

An introduction to classes

The topics that follow introduce you to the concepts you need before you create your own classes. First, you'll learn how classes are typically used in a business application to simplify the overall design of the application. Next, you'll learn about the variety of members you can add to a class. Then, you'll see a complete example of a simple class. Finally, you'll learn how classes are instantiated to create objects.

How classes can be used to structure an application

Figure 12-1 shows how you can use classes to simplify the design of a business application using a *multi-layered architecture*, also called a *multi-tiered architecture.* In a multi-layered application, the classes that perform different functions of the application are separated into two or more layers, or tiers.

A *three-tiered* application architecture like the one shown in this figure consists of a presentation layer, a middle layer, and a database layer. In practice, the middle layer is sometimes eliminated and its functions split between the database and presentation layers. On the other hand, the design of some applications further develops the middle layer into additional layers.

The classes in the *presentation layer* handle the details of the application's user interface. For a Windows application, this consists of the form classes that display the user interface. One class is required for each form displayed by the application.

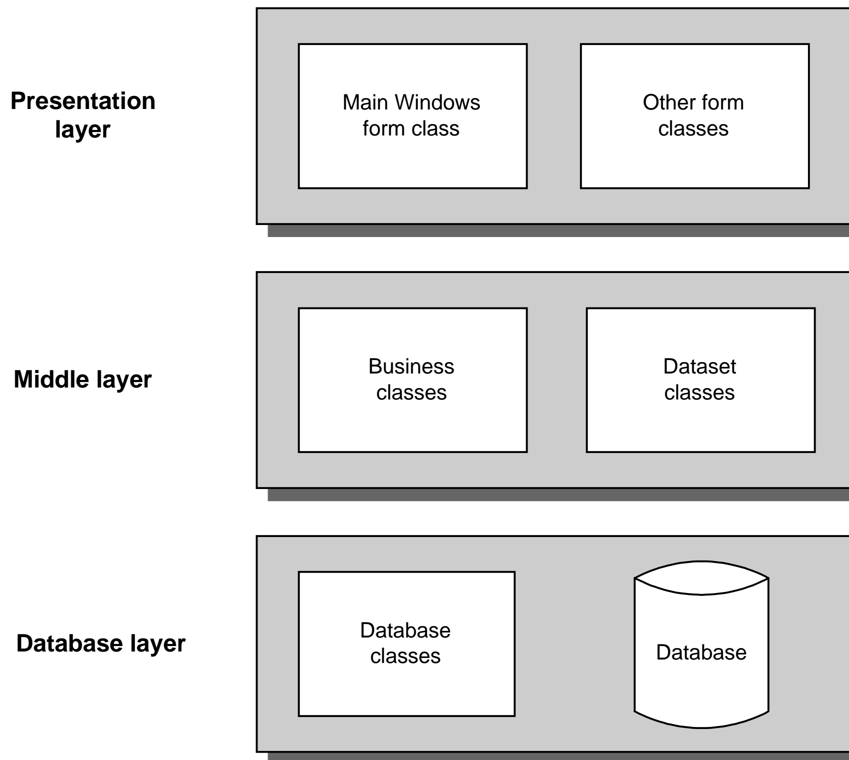
The classes of the *database layer* are responsible for all database access required by the application. These classes typically include methods that connect to the database and retrieve, insert, add, and delete information from the database. Then, the other layers can call these methods to access the database, leaving the details of database access to the database classes. Although we refer to this layer as the database layer, it can also contain classes that work with data that's stored in files.

The *middle layer* provides an interface between the database layer and the presentation layer. This layer often includes classes that correspond to business entities (for example, products and customers). It may also include classes that implement business rules, such as discount or credit policies.

One key advantage of developing applications using a tiered architecture is that it allows application development to be spread among members of a development team. For example, one group of developers might work on the database layer, another group on the middle layer, and still another group on the presentation layer.

Another advantage is that it allows classes to be shared among applications. In particular, the classes that make up the database and middle layers can be placed in *class libraries* that can be used by more than one project. You'll learn how to work with class libraries in chapter 15.

The architecture of a three-tiered application



Description

- To simplify development and maintenance, many applications use a *three-tiered architecture* to separate the application's user interface, business rules, and database processing. Classes are used to implement the functions performed at each layer of the architecture.
- The classes in the *presentation layer* control the application's user interface. For a Windows Forms application, the user interface consists of the various forms that make up the application.
- The classes in the *database layer* handle all of the application's data processing.
- The classes in the *middle layer*, sometimes called the *business rules layer*, act as an interface between the classes in the presentation and database layers. In some cases, these classes correspond to business entities, such as customers or products. This layer may also include classes that implement business rules, such as discount or credit policies.
- In some cases, the classes that make up the database layer and the middle layer are implemented in class libraries that can be shared among applications. For more information, see chapter 15.

Figure 12-1 How classes can be used to structure an application

The members you can define within a class

As you already know, the *members* of a class include its *properties*, *methods*, and *events*. Throughout this book, you've seen many examples of applications that work with the .NET Framework classes and their members. You've also seen examples that use *constructors*, which are a special type of method that's executed when an object is created.

The classes you design yourself can also have properties, methods, constructors, and events. For example, figure 12-2 presents the members of a Product class that can be used to work with products. This class has three properties that store the code, description, and price for each product, a method named GetDisplayText that returns a formatted string that contains the code, description and price for a product, and two constructors that create instances of the class.

The second table in this figure lists all the different types of members a class can have. You already know how to code two of these types of members: constants and enumerations. You also know the basic skills for coding methods. By the time you finish the chapters in this section, you'll know how to create the other types of members as well.

Of course, not every class you create will contain all these types of members. In fact, most classes will have just properties, methods, and constructors. But it's important to know about all of the possible member types so you'll be able to decide which types are appropriate for the classes you create.

This figure also reviews the basic concepts of object-oriented programming that were first introduced in chapter 3. In addition, it presents a fundamental concept of object-oriented programming that you may not be familiar with. This is the concept of *encapsulation*.

Encapsulation lets the programmer hide, or encapsulate, some of the data and operations of a class while exposing others. For example, although a property or method of a class can be called from other classes, its implementation is hidden within the class. That way, users of the class can think of it as a black box that provides useful properties and methods. This also means that you can change the code within a class without affecting the other classes that use it. This makes it easier to enhance or change an application because you only need to change the classes that need changing. You'll get a better idea of how encapsulation works when you see the code for the Product class in the next figure.

The members of a Product class

Properties	Description
Code	A string that contains a code that uniquely identifies each product.
Description	A string that contains a description of the product.
Price	A decimal that contains the product's price.
Method	**Description**
GetDisplayText(sep**)**	Returns a string that contains the code, description, and price in a displayable format. The sep parameter is a string that's used to separate the elements. It's typically set to a tab or new line character.
Constructors	**Description**
()	Creates a product object with default values.
(code, description, price**)**	Creates a product object using the specified code, description, and price values.

Types of class members

Class member	Description
Property	Represents a data value associated with an object instance.
Method	An operation that can be performed by an object.
Constructor	A special type of method that's executed when an object is instantiated.
Event	A signal that notifies other objects that something noteworthy has occurred.
Field	A variable that's declared at the class level.
Constant	A constant.
Indexer	A special type of property that allows individual items within the class to be accessed by index values. Used for classes that represent collections of objects.
Operator	A special type of method that's performed for a C# operator such as + or ==.
Enumeration	An enumeration.
Class	A class that's defined within the class.

Class and object concepts

- An *object* is a self-contained unit that has *properties*, *methods*, and other *members*. A *class* contains the code that defines the members of an object.
- An object is an *instance* of a class, and the process of creating an object is called *instantiation.*
- *Encapsulation* is one of the fundamental concepts of object-oriented programming. It lets you control the data and operations within a class that are exposed to other classes.
- The data of a class is typically encapsulated within a class using *data hiding*. In addition, the code that performs operations within the class is encapsulated so it can be changed without changing the way other classes use it.
- Although a class can have many different types of members, most of the classes you create will have just properties, methods, and constructors.

Figure 12-2 The members you can define within a class

The code for the Product class

Figure 12-3 shows the complete code for the Product class whose members were described in figure 12-2. As you can see, it begins with a class statement that declares the Product class with the public access modifier. This access modifier lets other classes access the class.

The code within the class block defines the members of the Product class. In the rest of this chapter, you'll learn how to write code like the code shown here. For now, I'll just present a preview of this code so you have a general idea of how it works.

The first three statements in this class are declarations for three class variables, called *fields*. As you'll see in a minute, these fields are used to store the data for the Code, Description, and Price properties. Because these variables are defined with the private access modifier, they cannot be referred to from outside the class.

After the fields are declarations for the two constructors of the Product class. The first constructor, which accepts no arguments, creates an instance of the Product class and initializes its fields to default values. The second constructor creates an instance of the class and initializes it with values passed via the code, description, and price parameters.

Next are the declarations for the three properties of the Product class. These properties provide access to the values stored in the three fields. Within each of these property declarations are two blocks of code that get and set the value of the property.

Last is the declaration for the GetDisplayText method, which accepts a string parameter named *sep*. This method returns a string that concatenates the code, description, and price values, separated by the value passed via the sep parameter.

Notice that you always use the public access modifier to identify the properties and methods that can be accessed from other classes. In contrast, you use the private access modifier to declare fields that you don't want to be accessed from other classes. In this case, for example, the fields can only be accessed through the properties defined by the class. You can also use the private access modifier to code properties and methods that you don't want to be accessed from other classes.

The Product class

```
using System;
namespace ProductMaintenance
{
    public class Product
    {
        private string code;
        private string description;
        private decimal price;

        public Product()
        {
        }

        public Product(string code, string description, decimal price)
        {
            this.Code = code;
            this.Description = description;
            this.Price = price;
        }

        public string Code
        {
            get
            {
                return code;
            }
            set
            {
                code = value;
            }
        }

        public string Description
        {
            get
            {
                return description;
            }
            set
            {
                description = value;
            }
        }

        public decimal Price
        {
            get
            {
                return price;
            }
            set
            {
                price = value;
            }
        }

        public string GetDisplayText(string sep)
        {
            return code + sep + price.ToString("c") + sep + description;
        }
    }
}
```


Figure 12-3 The code for the Product class

How instantiation works

The process of creating an object from a class is called *instantiation.* Figure 12-4 describes how instantiation works. Here, you can see two object *instances* that were created from the Product class. Each instance represents a different Product object. Because both instances were created from the same class, they both have the same properties. However, the instances have distinct values for each property. For example, the value of the Code property for the product1 object is JAVA, but the value of the Code property for the product2 object is VASP.

The code example in this figure shows how you can create these two object instances. Here, the first line of code declares two variables named product1 and product2 that have a type of Product. Then, the next two lines create Product objects. To do that, they use the new keyword, followed by the name of the constructor for the Product class and the values that will be used to initialize the objects.

At this point, it's important to realize that a class defines a *reference type*. That means that the variable that's used to access an object contains the address of the memory location where the object is stored, not the object itself. In other words, the product1 variable holds a *reference* to a Product object, not an actual Product object.

Two Product objects that have been instantiated from the Product class

product1	product2
Code=JAVA Description=Murach's Beginning JAVA 2 Price=49.50	Code=VASP Description=Murach's ASP.NET Web Programming with VB.NET Price=49.50

Code that creates these two object instances

```
Product product1, product2;
product1 = new Product("JAVA", "Murach's Beginning JAVA 2", 49.50m);
product2 = new Product("VASP",
    "Murach's ASP.NET Web Programming with VB.NET", 49.50m);
```

Description

- When an object is instantiated, a *constructor* is executed to initialize the data that makes up the object. If a class doesn't provide a constructor, a default constructor is executed. The default constructor simply initializes all the data to default values.
- The data that makes up an object is sometimes referred to as the object's *state*. Once an object has been instantiated, its state can change.
- The state of an object changes whenever you change the value of one of the object's properties or public fields. The state can also change when you call a method that affects the data stored within an object.
- An application can create two or more instances of the same class. Each instance is a separate entity with its own state. If you change the state of one object, the state of other objects created from the same class is not affected.
- A class defines a *reference type*. That means that the variable that's used to access an object instantiated from a class contains the address of the object, not the actual object.

Figure 12-4 How instantiation works

How to create a class

Now that you've learned about the members that make up a class and you've seen the code for the Product class, you're ready to learn the basic skills for creating and using your own classes. The topics that follow present these skills.

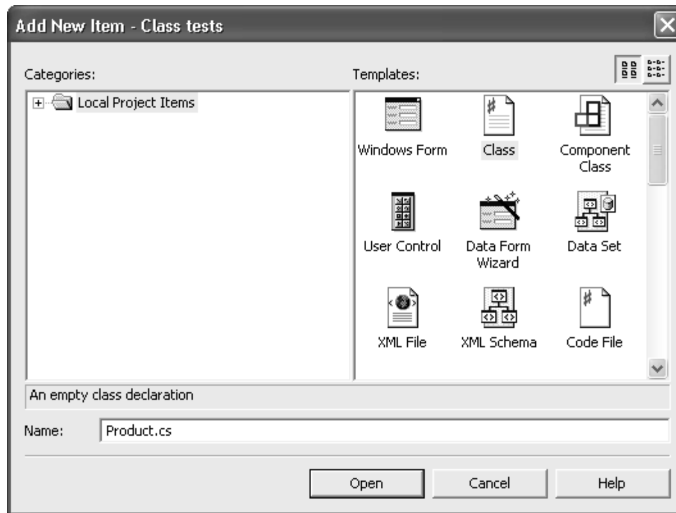
How to add a class file to a project

To create a user-defined class, you start by adding a *class file* to your project. To do that, you use the dialog box shown in figure 12-5. When you complete this dialog box, the class file will appear in the Solution Explorer with the extension *cs*.

When you add a class to a project, Visual Studio automatically generates the class declaration and an empty constructor. Then, you can complete the class by adding fields, properties, methods, and whatever other members the class may require.

Before I go on, you should notice the three comment lines before the class declaration. These lines provide information that Visual Studio can use to create web-based documentation for the class. You'll learn more about how to provide documentation in chapter 13. For now, you can ignore these lines.

The dialog box for adding a class



The starting code for the new class

```
using System;

namespace ProductMaintenance
{
    /// <summary>
    /// Summary description for Product.
    /// </summary>
    public class Product
    {
        public Product()
        {
            //
            // TODO: Add constructor logic here
            //
        }
    }
}
```

Description

- To add a new class to a project, use the Project→Add Class command to display the Add New Item dialog box. Then, enter the name you want to use for the new class and click the Open button.
- When you complete the Add New Item dialog box, a *class file* is added to the project. This class file will appear in the Solution Explorer window with the extension *cs*.
- The namespace and class blocks are automatically added to the class along with an empty constructor for the class. Then, you can enter the code for the class within the class block.

Figure 12-5 How to add a class file to a project

How to code fields

Figure 12-6 shows how to code the fields that define the variables used by a class. A class can contain two types of fields: *instance variables* and *static variables*. This figure shows you how to work with instance variables. You'll learn about static variables in figure 12-10.

When you declare a field, you should use an access modifier to control the accessibility of the field. If you specify *private* as the access modifier, the field can be used only within the class that defines it. In contrast, if you specify *public* as the access modifier, the field can be accessed by other classes. You can also use other access modifiers that give you finer control over the accessibility of your fields. You'll learn about those modifiers in chapter 14.

This figure shows three examples of field declarations. The first example declares a private field of type int. The second declares a public field of type decimal.

The third example declares a *read-only field*. As its name implies, a read-only field can be read but not modified. The only time you can assign a value to a read-only field is in the field declaration. In this respect, a read-only field is similar to a constant. The difference is that the value of a constant is set when the class is compiled. In contrast, the value of a read-only field is not set until runtime.

Note that although fields work like regular variables, they must be declared within the class body, not inside properties, methods, or constructors. That way, they're available throughout the entire class. In this book, all of the fields for a class are declared at the beginning of the class. However, when you read through code from other sources, you may find that the fields are declared at the end of the class or at other locations within the class.

This figure also presents another version of the Product class that uses public fields instead of properties. This works because the properties of the Product class that was presented in figure 12-3 didn't do anything except get and set the values of private fields. Because of that, you could just provide direct access to the fields by giving them public access as shown here. In some cases, though, a property will perform additional processing. Then, you'll need to use a property instead of a public field.

By the way, you may notice that the names of public fields in this class and in the second and third examples at the beginning of this figure begin with a capital letter. That's because these fields are used in place of properties, and the names of properties are always capitalized. In contrast, the name of the private field in the first example begins with a lowercase letter because it can only be accessed within the class.

Examples of field declarations

```
private int quantity;            // A private field.
public decimal Price;            // A public field.
public readonly int Limit = 90;  // A public read-only field.
```

A version of the Product class that uses public fields instead of properties

```
public class Product
{
    // Public fields
    public string Code;
    public string Description;
    public decimal Price;

    public Product()
    {
    }

    public Product(string code, string description, decimal price)
    {
        this.Code = code;
        this.Description = description;
        this.Price = price;
    }

    public string GetDisplayText(string sep)
    {
        return Code + sep + Price.ToString("c") + sep + Description;
    }
}
```

Description

- A variable that's defined at the class level within a class is called a *field*.
- A field can be a primitive data type, a class or structure from the .NET Framework, or a user-defined class or structure.
- You can use the *private* access modifier to prevent other classes from accessing a field. Then, the field can be accessed only from within the class.
- An *instance variable* is a field that's defined in a class and is allocated when an object is instantiated. Each object instance has a separate copy of each instance variable.
- If you initialize an instance variable, the initialization will occur before the constructor for the class is executed. As a result, the constructor may assign a new value to the variable.
- You can create a *read-only field* by specifying the *readonly* access modifier in the field's declaration. Then, you can retrieve the field's value, but you can't change it. Note that you provide the value for a read-only field when you declare it.
- A *public field* is a field that's declared with the *public* access modifier. Public fields can be accessed by other classes within the application, much like properties. However, properties have additional features that make them more useful than public fields.

Figure 12-6 How to code fields

How to code properties

Figure 12-7 presents the syntax for coding a property. As you can see, a property declaration specifies both the type and name of the property. In addition, a property is typically declared with the public access modifier so it can be accessed by other classes.

Within the block that defines a property, you can include two blocks called *accessors* because they provide access to the property values. The *get accessor* is executed when a request to retrieve the property value is made, and the *set accessor* is executed when a request to set the property value is made. If both get and set accessors are included, the property is called a *read/write property*. If only a get accessor is included, the property is called a *read-only property*. You can also create a *write-only property*, which has just a set accessor, but that's uncommon.

In most cases, each property has a corresponding private instance variable that holds the property's value. In that case, the property should be declared with the same data type as the instance variable. It's also common to use the same name for the property and the instance variable, but to use Camel notation for the instance variable name (first word starts with a lowercase character, all words after that start with uppercase characters) and to use Pascal notation for the property name (each word starts with an uppercase character). For example, the name of the instance variable for the Code property is code. And a property named UnitPrice would have a corresponding instance variable named unitPrice.

Because a get accessor returns the value of the property, it typically ends with a return statement that provides that value. In the first property in this figure, for example, the return statement simply returns the value of the instance variable that holds the property value. The second property, however, illustrates that the get accessor can be more complicated than that. Here, the get accessor performs a calculation to determine the value that's returned.

The set accessor uses an implicit parameter named value to access the value to be assigned to the property. Typically, the set accessor simply assigns this value to the instance variable that stores the property's value. However, the set accessor can perform more complicated processing if necessary. For example, it could perform data validation.

The syntax for coding a public property

```
public type PropertyName
{
    [get { get accessor code }]
    [set { set accessor code }]
}
```

A read/write property

```
public string Code
{
    get
    {
        return code;
    }
    set
    {
        code = value;
    }
}
```

A read-only property

```
public decimal DiscountAmount
{
    get
    {
        discountAmount = subtotal * discountPercent;
        return discountAmount;
    }
}
```

A statement that sets a property value

```
product.Code = txtProductCode.Text;
```

A statement that gets a property value

```
string code = product.Code;
```

Description

- You use a *property* to get and set data values associated with an object. Typically, each property has a corresponding private instance variable that stores the property's value.
- It's common to use the same name for the property and the related instance variable, but to begin the property name with an uppercase letter and the instance variable name with a lowercase letter.
- You can code a *get accessor* to retrieve the value of the property. Often, the get accessor simply returns the value of the instance variable that stores the property's value.
- You can code a *set accessor* to set the value of the property. Often, the set accessor simply assigns the value passed to it via the *value* keyword to the instance variable that stores the property's value.
- A property that has both a get and a set accessor is called a *read/write property*. A property that has just a get accessor is called a *read-only property*. And a property that has just a set accessor is called a *write-only property*.

Figure 12-7 How to code properties

How to code methods

Figure 12-8 shows you how to code the methods for a class. Because the basics of coding methods were presented in chapter 6, most of the information in this figure should review for you. In fact, this figure only introduces two new techniques. The first is the use of the public access modifier on a method declaration, which makes the method available to other classes.

The second is the concept of overloading. When you *overload* a method, you code two or more methods with the same name, but with unique combinations of parameters. In other words, you code methods with unique *signatures*.

For a method signature to be unique, the method must have a different number of parameters than the other methods with the same name, or at least one of the parameters must have a different data type. Note that the names of the parameters aren't part of the signature. So using different names isn't enough to make the signatures unique. Also, the return type isn't part of the signature. As a result, you can't create two methods with the same name and parameters but different return types.

The purpose of overloading is to provide more than one way to invoke a given method. For example, this figure shows two versions of the GetDisplayText method. The first one is the one you saw in figure 12-3 that accepts a parameter named sep. The second one doesn't accept this parameter. Instead, it uses a comma and a space to separate the code, price, and description.

When you refer to an overloaded method, the number of arguments you specify and their types determine which version of the method is executed. The two statements in this figure that call the GetDisplayText method illustrate how this works. Because the first statement specifies an argument, it will cause the version of the GetDisplayText method that accepts a parameter to be executed. In contrast, the second statement doesn't specify an argument, so it will cause the version of the GetDisplayText method that doesn't accept a parameter to be executed.

In chapter 3, you learned that if you type the name of a method followed by a left parenthesis into the Code Editor, Visual Studio's IntelliSense feature displays a list of the method's parameters. You may not have realized, though, that if up and down arrows appear to the left of the argument list, it indicates that the method is overloaded. Then, you can click the up and down arrows or press the up and down arrow keys to move from one overloaded method to another.

This works with overloaded methods in user-defined classes as well. For example, the illustration in this figure shows how the Intellisense feature displays the overloaded GetDisplayText methods. In this case, the first of the two methods is displayed.

The syntax for coding a public method

```
public returnType MethodName([parameterList])
{
   statements
}
```

A method that accepts parameters

```
public string GetDisplayText(string sep)
{
    return code + sep + price.ToString("c") + sep + description;
}
```

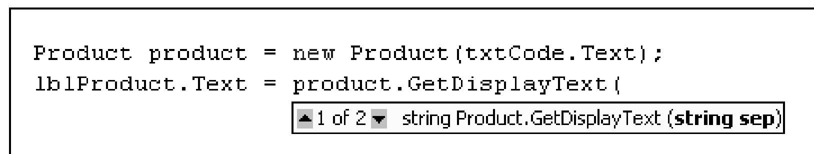
An overloaded version of the GetDisplayText method

```
public string GetDisplayText()
{
    return code + ", " + price.ToString("c") + ", " + description;
}
```

Two statements that call the GetDisplayText method

```
lblProduct.Text = product.GetDisplayText("\t");
lblProduct.Text = product.GetDisplayText();
```

How the IntelliSense feature lists overloaded methods



Description

- To provide other classes with access to a method, you declare it using the public access modifier. To prevent other classes from accessing a method, you declare it using the private access modifier.
- The name of a method combined with its parameters form the method's *signature*. Although you can use the same name for more than one method, each method must have a unique signature.
- When you create two or more methods with the same name but with different parameter lists, the methods are *overloaded*. It's common to use overloaded methods to provide two or more versions of a method that work with different data types or that supply default values for omitted parameters.
- When you type a method name followed by a left parenthesis, the IntelliSense feature of Visual Studio displays the parameters expected by the method. If up and down arrows are displayed as shown above, you can click these arrows or press the up and down arrow keys to display each of the method's overloaded parameter lists.

Figure 12-8 How to code methods

How to code constructors

By default, when you use the new keyword to create an instance of a user-defined class, C# assigns default values to all of the instance variables in the new object. If that's not what you want, you can code a special method called a *constructor* that's executed when an object is created from the class. Figure 12-9 shows you how to do that.

To create a constructor, you declare a public method with the same name as the class name. For example, a constructor for the Product class must be named Product. Within the constructor, you initialize the instance variables, and you include any additional statements you want to be executed when an object is created from the class. Note that a constructor must not be declared with a return type.

The first example in this figure is the constructor that's generated automatically when you create a class using the Project➔Add Class command. This constructor doesn't provide for any parameters, so it's called when you create an instance of the class without specifying any arguments. Because this constructor has no executable statements, it simply initializes all the instance variables to their default values (excluding read-only variables). The default values for the various data types are listed in this figure.

In some cases, a class might not be defined with any constructors. For example, you could delete the constructor that's generated automatically when you create a class. Or, you could code a class within another class (see chapter 13) and not declare a constructor for that class. In that case, C# provides a *default constructor* that's equivalent to the constructor shown in the first example.

The second constructor in this figure shows that you can overload constructors just like you can overload methods. Here, a constructor that accepts three parameters is defined. This constructor uses the values passed to the parameters to initialize the instance variables. This technique is often used to set initial property values when an object is created.

The third constructor shows how you might provide a constructor for the Product class that accepts just a product code as a parameter. Then, it calls a method named GetProduct in a database class named ProductDB to retrieve the data for the specified product. After the data is retrieved, the constructor uses it to initialize the instance variables.

Notice that both the second and third constructors use the this keyword to refer to the properties whose values are being initialized. Although this isn't required, it makes it clear that the property that's being referred to is defined in the current class and not in another class.

This figure also presents statements that execute the three constructors shown here. The first statement executes the constructor with no parameters. The second statement executes the constructor with three parameters. And the third statement executes the constructor with one parameter. Although you've seen statements like these before, you should now have a better understanding of how they work.

A constructor with no parameters

```
public Product()
{
    //
    // TODO: Add constructor logic here
    //
}
```

A constructor with three parameters

```
public Product(string code, string description, decimal price)
{
    this.Code = code;
    this.Description = description;
    this.Price = price;
}
```

A constructor with one parameter

```
public Product(string code)
{
    Product p = ProductDB.GetProduct(code);
    this.Code = p.Code;
    this.Description = p.Description;
    this.Price = p.Price;
}
```

Statements that call these constructors

```
Product product1 = new Product();
Product product2 = new Product("JAVA", "Murach's Beginning Java 2", 49.50m);
Product product3 = new Product(txtCode.Text);
```

Default values for instance variables

Data type	Default value
All numeric types	zero (0)
Boolean	false
Char	binary 0 (null)
Object	null (no value)
Date	12:00 a.m. on January 1, 0001

Description

- The name of a constructor must be the same as the name of the class. In addition, a constructor must always be declared with the public access modifier, and it can't specify a return type.
- To code a constructor that has parameters, code a data type and name for each parameter within the parentheses that follow the class name.
- The name of a class combined with its parameter list form the signature of the constructor. Each constructor must have a unique signature.
- If a constructor doesn't assign a value to an instance variable and the variable hasn't been initialized, the variable will be assigned a default value as shown above.

Figure 12-9 How to code constructors

How to code static members

As figure 12-10 shows, *static members* are members that can be accessed without creating an instance of a class. The idea of static members shouldn't be new to you because you've seen them used in several chapters in this book. In chapter 4, for example, you learned how to use static methods of the Math and Convert classes. And in chapter 9, you learned how to use static members of the DateTime structure and the String class. This figure shows how to create static members in your own classes.

To create a static member, you simply include the *static* keyword on the declaration for the member. The class shown in this figure, for example, provides static members that can be used to perform data validation. This class has a static field named title, a static property named Title, and a static method named IsPresent.

The static IsPresent method validates the Text property of a text box control to make sure the user entered a value. If the Text property is an empty string, the IsPresent method displays an error message, sets the focus to the text box, and returns false. Otherwise, it returns true. Note that this method uses the Tag property of the text box to get the name that's displayed in the dialog box.

The second example in this figure shows how you might call the static IsPresent method to validate three text boxes. Here, the return values from three calls to the IsPresent method are tested. If all three calls return true, a Boolean variable named isValidData is set to true. Otherwise, this variable is set to false.

Although the class shown in this figure contains only static members, a class that has static members can also include non-static members. Then, when you create an instance of that class, all of the instances share the static members. Because of that, you can't access a static member from an instance of the class. Instead, you can only access it only from the class itself.

Keep in mind, too, that a static property or method can only refer to other static members. For example, because the Title property shown in this figure is declared as static, the title field it refers to must be declared as static. Similarly, if a property or method refers to another method, that method must be declared as static.

A class that contains static members

```
public class Validator
{
    private static string title = "Entry Error";

    public static string Title
    {
        get
        {
            return title;
        }
        set
        {
            title = value;
        }
    }

    public static bool IsPresent(TextBox textBox)
    {
        if (textBox.Text == "")
        {
            MessageBox.Show(textBox.Tag + " is a required field.", Title);
            textBox.Focus();
            return false;
        }
        return true;
    }
```

Code that uses static members

```
if (  Validator.IsPresent(txtCode.Text) &&
      Validator.IsPresent(txtDescription.Text) &&
      Validator.IsPresent(txtPrice.Text) )
    isValidData = true;
else
    isValidData = false;
```

Description

- A *static member* is a field, property, or method that can be accessed without creating an instance of the class. To define a static member, you use the *static* keyword.
- If you create instances of a class that contains static members, all of those instances share the same copy of the class's static members. Because of that, you can't refer to a static member through an instance of the class. You can refer to a static member only through the class that defines it.
- Static properties and methods can refer only to other static members or to variables declared within the property or method.
- A constant that's declared with the public keyword is implicitly static. You can't code the static keyword on a constant declaration.

Figure 12-10 How to code static members

The Product Maintenance application

Now that you've learned the basic skills for creating classes, the topics that follow present a Product Maintenance application that maintains a simple file of products. As you'll see, this application uses three classes in addition to the two form classes it uses.

The operation of the Product Maintenance application

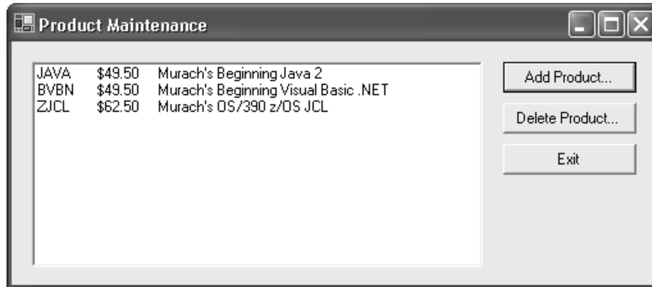
Figure 12-11 describes the operation of the Product Maintenance application. As you can see, this application uses two forms. The main form displays a list of the products that are stored in a file in a list box. The user can use this form to add or delete a product.

If the user clicks the Add Product button, the New Product form is displayed as a dialog box. Then, the user can enter the data for a new product and click the Save button to add the product to the file. After the product is saved, the list box in the Product Maintenance form is refreshed so it includes the new product. The user can also click the Cancel button on the New Product form to cancel the add operation.

To delete a product, the user selects the product in the list and clicks the Delete product button. Then, a dialog box is displayed to confirm the operation. If the operation is confirmed, the product is deleted and the list box is refreshed so it no longer includes the deleted product.

This figure also shows how the Tag properties of the three text boxes on the New Product form are set. As you'll see in a minute, the methods of the data validation class use the Tag property of these text boxes to display meaningful error messages if the user enters incorrect data.

The Product Maintenance form



The New Product form

The Tag property settings for the text boxes on the New Product form

Control	Tag property setting
txtCode	Code
txtDescription	Description
txtPrice	Price

Description

- The Product Maintenance application retrieves product information from a file, displays it in a list box, and lets the user add or delete products.
- To add a product, the user clicks the Add Product button to display the New Product form. Then, the user can enter the data for the new product and click the Save button. Alternatively, the user can click the Cancel button to cancel the add operation. In either case, the user is returned to the Product Maintenance form.
- To delete a product, the user selects the product to be deleted and then clicks the Delete Product button. Before the product is deleted, the delete operation is confirmed.
- The Tag properties of the three text boxes on the New Product form are set so that the Validator class can display meaningful error messages if the user enters invalid data.

Figure 12-11 The operation of the Product Maintenance application

The classes used by the Product Maintenance application

Figure 12-12 summarizes the properties, methods, and constructors for the classes used by the Product Maintenance application. As you can see, this application uses three classes. The Product class represents a single product. The ProductDB class handles the I/O processing for the Products file. And the Validator class handles the data validation for the user entries.

The Product class is the same as the Product class you saw in figure 12-3. It has three properties named Code, Description, and Price that define the values for a Product object. It has a single method named GetDisplayText that returns a string that contains the code, description, and price in a format that can be displayed in the list box of the Product Maintenance form. And it has two constructors: one that initializes the Code, Description, and Price properties to their default values, and one that initializes these properties to the specified values.

The ProductDB class contains two methods. The first one, GetProducts, retrieves all of the products from the Products file and returns them in an array list. The second one, SaveProducts, accepts an array list of Product objects and writes the products in the array list to the Products file, overwriting the previous contents of the file.

Note that the specifications for these methods don't indicate the format of the Products file. That's because the details of how this class saves and retrieves product information are of no concern to the Product Maintenance application. That's one of the benefits of encapsulation: You don't have to know how the class works. You just have to know what members it contains and how you refer to them.

The Validator class contains four static members that provide for different types of data validation. For example, the IsPresent method checks if the user entered data into a text box, and the IsDecimal method checks if the data the user entered is a valid decimal value. If one of these methods determines that the data is invalid, it displays an error message using the Title property as the title for the dialog box, it moves the focus to the text box that's being validated, and it returns false. Otherwise, it returns true.

The Product class

Property	Description
Code	A string that contains a code that uniquely identifies the product.
Description	A string that contains a description of the product.
Price	A decimal that contains the product's price.
Method	**Description**
GetDisplayText(sep**)**	Returns a string that contains the code, description, and price separated by the sep string.
Constructor	**Description**
()	Creates a Product object with default values.
(code, description, price)	Creates a Product object using the specified values.

The ProductDB class

Method	Description
GetProducts()	A static method that returns an array list that contains a Product object for each product in the Products file.
SaveProducts(arrayList**)**	A static method that writes the products in the specified array list to the Products file.

The Validator class

Property	Description
Title	A static string that contains the text that's displayed in the title bar of a dialog box that's displayed for an error message.
Method	**Description**
IsPresent(textBox**)**	A static method that returns a Boolean value that indicates if data was entered into the specified text box.
IsInt32(textBox**)**	A static method that returns a Boolean value that indicates if an integer was entered into the specified text box.
IsDecimal(textBox**)**	A static method that returns a Boolean value that indicates if a decimal was entered into the specified text box.
IsWithinRange(textBox, min, max**)**	A static method that returns a Boolean value that indicates if the value entered into the specified text box is within the specified range.

Note: Each of these methods displays an error message in a dialog box and moves the focus to the text box if the data is invalid.

Note

- Because you don't need to know how the ProductDB class works, its code isn't shown in this chapter. Please refer to chapters 21 and 22 for three different versions of this class.

Figure 12-12 The classes used by the Product Maintenance application

The code for the Product Maintenance application

Figures 12-13 through 12-15 show the code for the Product Maintenance form, the New Product form, and the Validator class. You saw the code for the Product class in figure 12-3, so I won't repeat it here. And, because you don't need to know how the ProductDB class is implemented to understand how this application works, I won't present the code for that class here either. If you're interested, however, you'll find three different implementations of this class in chapters 21 and 22.

The code for the Product Maintenance form, shown in figure 12-13, begins by declaring a class variable named products of type ArrayList. Then, in the Load event handler for the form, the GetProducts method of the ProductsDB class is called to fill this array list with Product objects created from the data in the Products file. Then, the FillProductListBox method is called. This method uses a foreach loop to add the string returned by each product's GetDisplayText method to the list box. Notice that a tab character is passed to this method so that the products appear as shown in figure 12-11.

If the user clicks the Add Product button, an instance of the New Product form is created, and the GetNewProduct method of that form is called. If the Product object returned by this method isn't null, the product is added to the array list of products. Then, the SaveProducts method of the ProductDB class is called to update the Products file, and the FillProductListBox method is called to refresh the list box so the new product is included.

If the user selects a product in the list and clicks the Delete Product button, a confirmation dialog box is displayed. Then, if the user confirms the deletion, the product is removed from the array list, the Products file is updated, and the list box is refreshed.

The code for the New Product form is shown in figure 12-14. It declares a private Product object named product. Then, the GetNewProduct method that's called from the Product Maintenance form starts by displaying the New Product form as a dialog box. If the user clicks the Save button in this dialog box, the IsValidData method is called to validate the data. This method calls the IsPresent method of the Validator class for each text box on the form. In addition, it calls the IsDecimal method for the Price text box.

If all of the values are valid, a new Product object is created with the values entered by the user, the dialog box is closed, and the Product object is returned to the Product Maintenance form. In contrast, if the user clicks the Cancel button, the dialog box is closed and the product variable, which is initialized to null, is returned to the Product Maintenance form.

The code for the Validator class, shown in figure 12-15, should present no surprises. In fact, you saw code similar to this code back in figure 12-10. The only difference is that this version of the Validator class includes an IsDecimal method as well as an IsPresent method. Note that because the Product Maintenance application doesn't use the IsInt32 or IsWithinRange methods, I omitted those methods from this figure.

The code for the Product Maintenance form

```
public class frmProductMaintenance : System.Windows.Forms.Form
{
    private ArrayList products = null;

    private void frmProductMain_Load(object sender, System.EventArgs e)
    {
        products = ProductDB.GetProducts();
        FillProductListBox();
    }

    private void FillProductListBox()
    {
        lstProducts.Items.Clear();
        foreach (Product p in products)
        {
            lstProducts.Items.Add(p.GetDisplayText("\t"));
        }
    }

    private void btnAdd_Click(object sender, System.EventArgs e)
    {
        frmNewProduct newProductForm = new frmNewProduct();
        Product product = newProductForm.GetNewProduct();
        if (product != null)
        {
            products.Add(product);
            ProductDB.SaveProducts(products);
            FillProductListBox();
        }
    }

    private void btnDelete_Click(object sender, System.EventArgs e)
    {
        int i = lstProducts.SelectedIndex;
        if (i != -1)
        {
            Product product = (Product) products[i];
            string message = "Are you sure you want to delete "
                + product.Description + "?";
            DialogResult button =
                MessageBox.Show(message, "Confirm Delete",
                MessageBoxButtons.YesNo);
            if (button == DialogResult.Yes)
            {
                products.Remove(product);
                ProductDB.SaveProducts(products);
                FillProductListBox();
            }
        }
    }

    private void btnClose_Click(object sender, System.EventArgs e)
    {
        this.Close();
    }
}
```

Figure 12-13 The code for the Product Maintenance form

The code for the New Product form

```
public class frmNewProduct : System.Windows.Forms.Form
{
    private Product product = null;

    public Product GetNewProduct()
    {
        this.ShowDialog();
        return product;
    }

    private void btnSave_Click(object sender, System.EventArgs e)
    {
        if (IsValidData())
        {
            product = new Product(txtCode.Text,
                txtDescription.Text, Convert.ToDecimal(txtPrice.Text));
            this.Close();
        }
    }

    private bool IsValidData()
    {
        return Validator.IsPresent(txtCode) &&
               Validator.IsPresent(txtDescription) &&
               Validator.IsPresent(txtPrice) &&
               Validator.IsDecimal(txtPrice);
    }

    private void btnCancel_Click(object sender, System.EventArgs e)
    {
        this.Close();
    }

}
```

Figure 12-14 The code for the New Product form

The code for the Validator class

```
public class Validator
{
    private static string title = "Entry Error";

    public static string Title
    {
        get
        {
            return title;
        }
        set
        {
            title = value;
        }
    }

    public static bool IsPresent(TextBox textBox)
    {
        if (textBox.Text == "")
        {
            MessageBox.Show(textBox.Tag + " is a required field.", Title);
            textBox.Focus();
            return false;
        }
        return true;
    }

    public static bool IsDecimal(TextBox textBox)
    {
        try
        {
            Convert.ToDecimal(textBox.Text);
            return true;
        }
        catch (FormatException)
        {
            MessageBox.Show(textBox.Tag + " must be a decimal number.", Title);
            textBox.Focus();
            return false;
        }
    }

}
```

Note

- The code for the IsInt32 and IsWithinRange methods isn't shown here because these methods aren't used by the Product Maintenance application.

Figure 12-15 The code for the Validator class

How to work with classes in Visual Studio

Now that you've seen a complete application that uses classes, the last two topics of this chapter introduce some additional Visual Studio features that are designed for working with classes. Although you don't have to use these features, they can make classes easier to work with in some cases.

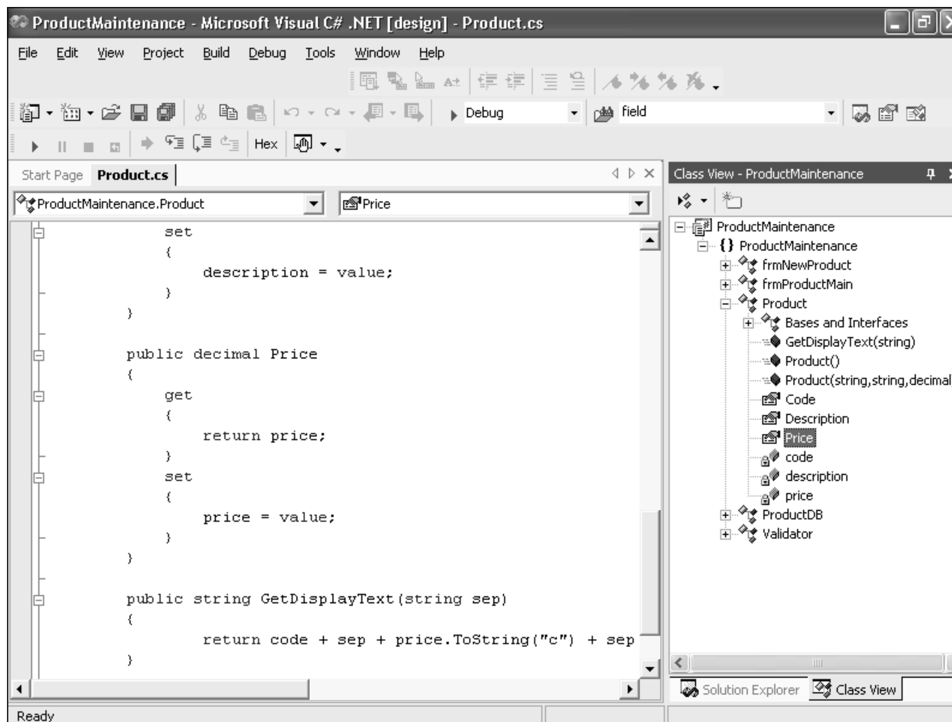
How to use the Class View window

The *Class View window* is a tabbed window that's displayed by default in the group with the Solution Explorer window. As figure 12-16 shows, this window contains a hierarchical view of the classes in your solution as well as the members of each class. In short, class view gives you an overall view of the structure of your application.

The solution shown here is for the Product Maintenance application that was presented in figures 12-11 through 12-15. Here, I've expanded the class list to show the individual members that make up the Product class. As you can see, the list of members for this class includes the GetDisplayText method, the two constructors for this class, the Code, Description, and Price properties, and the private fields named code, description, and price. Notice the padlock on the icons for the private fields. They indicate that these fields are not accessible from outside the class.

You can also use the Class View window to display the code for any item in the Code Editor window. To do that, just double-click the item in the Class View window. In this figure, for example, you can see the code that was displayed when I double-clicked the Price property.

The Class View window



Description

- The *Class View window* lets you browse the classes in a solution.
- The Class View window displays each class and its members, including properties, methods, events, and fields, in a hierarchical tree view. You can expand or collapse the tree by clicking the plus and minus signs.
- To display the Class View window, click the Class View tab that's grouped with the Solution Explorer window, or use the View→Class View command.
- You can double-click an item in the Class View window to display the code that defines that item in the Code Editor window.

Figure 12-16 How to use the Class View window

How to use wizards to create class members

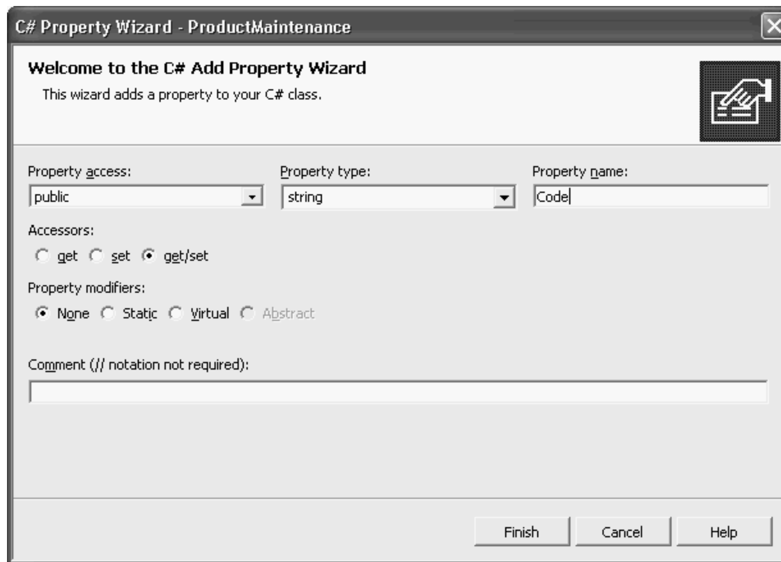
Instead of coding class members from scratch, you can use the wizards that Visual Studio provides to generate code skeletons for properties, methods, fields, and indexers. Then, you can add the code required to implement these members. Figure 12-17 shows how you can use wizards to create class members.

Each wizard displays a dialog box that lets you enter information about the member. For example, the dialog box shown here is for a property. As you can see, it lets you specify information such as the property's accessibility, its type, its name, and whether it has a get accessor, a set accessor, or both. In this example, I entered information to define the Code property of the Product class. Then, when I clicked the Finish button, the code shown in this figure was generated.

Although the dialog boxes for the other wizards aren't shown here, they work in much the same way. To create a method, for example, you provide information such as its name, accessibility, return type, and the parameters it accepts. And to create a field, you specify its name, accessibility, and type. Note that the Class View window must be active to use the wizards.

Whether or not you use the wizards is a matter of personal preference. When you first start coding your own classes, for example, the wizards may help you to code members quickly and easily. Once you get used to coding classes, however, you may find that the wizards don't save you much time.

The C# Add Property Wizard



The code generated for the property shown above

```
public string Code
{
    get
    {
        return null;
    }
    set
    {
    }
}
```

Description

- Visual Studio provides several wizards that you can use to add properties, methods, fields, and indexers to C# classes.
- To use a wizard, open the Class View window and select the class you want to add a member to. Then, choose the appropriate command from the Project menu (Add Method, Add Property, Add Field, or Add Indexer), enter the appropriate information into the dialog box that's displayed, and click the Finish button.
- You can also access a wizard by right-clicking a class in the Class View window and then choosing the Add Method, Add Property, Add Field, or Add Indexer command from the shortcut menu that's displayed.
- The wizards automatically insert the starting code for a member into your class. Then, you can enter any additional code that's required.

Figure 12-17 How to use wizards to create class members

Perspective

In this chapter, you've learned the basic skills for creating classes. However, there's a lot more to creating classes than what's presented here. In the next three chapters, then, you'll learn some skills for creating more complex classes.

Now that you've completed this chapter, you may be wondering why you should go to the extra effort of dividing an application into classes. The answer is twofold. First, dividing the code into classes makes it easier to use the classes in two or more applications. For example, any application that needs to work with the data in the Products file can use the ProductDB class. Second, using classes helps you separate the business logic and database processing of an application from the user interface elements. That can simplify the development of the application and make it easier to maintain and enhance later on.

At this point, you can continue in three different ways. First, you can read the next three chapters in this section to learn more about object-oriented programming. Second, you can skip to section 4 to learn how to develop database applications both with and without the use of business and database classes. And third, you can skip to chapters 21 and 22 to learn how to implement database classes with text, binary, and XML files. If you skip to section 4 or 5 after chapter 12, though, be sure to return to chapters 13 through 15 later on because they present the rest of the object-oriented skills that every programmer should have.

Summary

- In a *three-tiered architecture*, an application is separated into three layers: The *presentation layer* consists of the user interface; the *database layer* consists of the database and the database classes that work with it; and the *middle layer* provides an interface between the presentation layer and the database layer.
- An *object* contains *members*, such as *properties*, *methods*, and *events*. An object is *instantiated* from a *class*, which contains the code that defines the members of an object.
- *Encapsulation* lets you control the data and operations within a class that are exposed to other classes. When data is encapsulated within a class, it's called *data hiding*.
- The data that is stored in an object can be referred to as its *state*. Each object is a separate entity with its own state.
- A *field* is a variable that's defined at the class level. A *public field* can be accessed by other classes. A private field cannot.
- An *instance variable* is a field that's allocated when an object is instantiated. Each object has a separate copy of each instance variable.
- You use properties to get and set the data associated with an object. Each property can have a *get accessor* that retrieves the value of a property and a *set accessor* that sets the value of the property.

- A method defines the operations that an object can perform. You can *overload* methods by declaring them with the same name but with different parameter lists.
- A *constructor* is a special type of method that creates an instance of a class. A constructor typically initializes the instance variables defined by the class.
- A *static member* is a field, property, or method that is accessed directly from the class rather than from an instance of a class. If you create instances of a class that contains static members, all of the instances share those members.
- You can use the *Class View window* to browse the classes in a solution.
- You can use the wizards provided by Visual Studio to add starting code for properties, methods, fields or indexers.

Terms

multi-layered architecture
multi-tiered architecture
three-tiered architecture
presentation layer
database layer
middle layer
business rules layer
object
property
method
member
class
instantiation
encapsulation
data hiding
constructor
state
reference type
class file
field
instance variable
read-only field
public field
get accessor
set accessor
read/write property
read-only property
write-only property
signature
overloaded method
static member
Class View window

Objectives

- Given the specifications for an application that uses classes with any of the members presented in this chapter, develop the application and its classes.
- List and describe the three layers of a three-tiered application.
- Explain what encapsulation is and describe its main benefit.
- Explain what instantiation is and how it works.
- Explain what a field is and describe the two types of variables they can define.
- Explain how the get and set accessors of a property work.
- Describe the concept of overloading a method.
- Describe the function of a constructor.

- Explain what a static member is and how it's used.
- Describe the features that Visual Studio provides for working with classes.

Exercise 12-1 Create a Customer Maintenance application that uses classes

In this exercise, you'll create a Customer Maintenance application that uses three classes. To make this application easier to develop, we'll give you the starting forms with all the controls you'll need, a complete Validator class that you'll use to validate the data the user enters, and a complete CustomerDB class that you'll use to work with the data in a file of customers.

Open the project and add a Customer class

1. Open the project named CustomerMaintenance in the C:\C#.NET\Chapter 12\CustomerMaintenance directory.
2. Add a class named Customer to this project, and add the following properties, method, and constructors to this class:

Property	Description
`FirstName`	Gets or sets a string that contains the customer's first name.
`LastName`	Gets or sets a string that contains the customer's last name.
`Email`	Gets or sets a string that contains the customer's email address.
Method	**Description**
`GetDisplayText()`	Returns a string that contains the customer's name and email address formatted like this: Joanne Smith, jsmith@armaco.com.
Constructor	**Description**
`()`	Creates a customer object with default values.
`(firstName, lastName, email)`	Creates a customer object using the specified values.

Add code to implement the Add Customer form

3. Display the code for the Add Customer form, and declare a class variable named customer of type Customer with an initial value of null.
4. Add a public method named GetNewCustomer that displays the form as a dialog box and returns a Customer object.

5. Add an event handler for the Click event of the Save button that validates the data on the form using the Validator class (all three fields are required), and then creates a new customer object and closes the form if the data is valid.
6. Add an event handler for the Click event of the Cancel button that simply closes the form.

Add code to implement the Customer Maintenance form

7. Display the code for the Customer Maintenance form, and declare a class variable named customers of type ArrayList with an initial value of null.
8. Add an event handler for the Load event of the form that uses the GetCustomers method of the CustomerDB class to load the customers array list and then adds the customers to the Customers list box. Use the GetDisplayText method of the Customer class to format the customer data.
9. Add an event handler for the Click event of the Add button that creates a new instance of the Add Customer form and executes the GetNewCustomer method of that form. If the customer object that's returned by this method is not null, this event handler should add the new customer to the array list, call the SaveCustomers method of the CustomerDB class to save the array list, and then refresh the Customers list box.
10. Add an event handler for the Click event of the Delete button that removes the selected customer from the array list, calls the SaveProducts method of the CustomerDB class to save the array list, and refreshes the Customers list box. Be sure to confirm the delete operation.
11. Add an event handler for the Click event of the Exit button that closes the form.

Run and test the application

12. Run the application and test it to be sure that it works properly. When you're done, end the application, but leave the solution open if you're going to continue with the next exercise.

Exercise 12-2 Add a static method to the Validator class

In this exercise, you'll add a static method to the Validator class of the Customer Maintenance application that validates email addresses. Then, you'll modify the code for the Add Customer form so it uses this new method.

1. If it's not already open, open the project named CustomerMaintenance in the C:\C#.NET\Chapter 12\CustomerMaintenance directory.
2. Add a static method named IsValidEmail to the Validator class. This method should accept a text box as a parameter and then check the Text property of that text box to be sure that the email address includes both an @ character and a period. If the email address is invalid, this method should display an error message, set the focus to the text box, and return false. Otherwise, it should return true.
3. Modify the code in the Add Customer form so it uses the IsValidEmail method to validate the email address.
4. Run the application, and test it to be sure it works properly.