

Introduction to Inheritance

In this chapter you will:

- ⦿ Learn about inheritance
- ⦿ Extend classes
- ⦿ Use the `protected` access specifier
- ⦿ Override base class members
- ⦿ Understand how a derived class object “is an” instance of the base class
- ⦿ Learn about the `Object` class
- ⦿ Work with base class constructors
- ⦿ Create and use abstract classes
- ⦿ Create and use interfaces
- ⦿ Use extension methods
- ⦿ Recognize inheritance in GUI applications and understand the benefits of inheritance

Understanding classes helps you organize objects. Understanding inheritance helps you organize them more precisely. If you have never heard of a *Braford*, for example, you would have a hard time forming a picture of one in your mind. When you learn that a Braford is an animal, you gain some understanding of what it must be like. That understanding grows when you learn it is a mammal, and the understanding is almost complete when you learn it is a cow. When you learn that a Braford is a cow, you understand it has many characteristics that are common to all cows. To identify a Braford, you must learn only relatively minor details—its color or markings, for example. Most of a Braford’s characteristics, however, derive from its membership in a particular hierarchy of classes: animal, mammal, and cow.

All object-oriented programming languages make use of inheritance for the same reasons—to organize the objects used by programs, and to make new objects easier to understand based on your knowledge of their inherited traits. In this chapter, you will learn to make use of inheritance with your C# objects.

Understanding Inheritance

Inheritance is the principle that you can apply your knowledge of a general category to more specific objects. You are familiar with the concept of inheritance from all sorts of situations. When you use the term *inheritance*, you might think of genetic inheritance. You know from biology that your blood type and eye color are the products of inherited genes. You can say that many other facts about you (your attributes) are inherited. Similarly, you often can attribute your behaviors to inheritance; for example, the way you handle money might be similar to the way your grandmother handles it, and your gait might be the same as your father’s—so your methods are inherited, too.

You also might choose to own plants and animals based on their inherited attributes. You plant impatiens next to your house because they thrive in the shade; you adopt a poodle because you know poodles don’t shed. Every plant and pet has slightly different characteristics, but within a species, you can count on many consistent inherited attributes and behaviors. In other words, you can reuse the knowledge you gain about general categories and apply it to more specific categories. Similarly, the classes you create in object-oriented programming languages can inherit data and methods from existing classes. When you create a class by making it inherit from another class, you are provided with data fields and methods automatically; you can reuse fields and methods that are already written and tested.

You already know how to create classes and how to instantiate objects from those classes. For example, consider the `Employee` class in Figure 10-1. The class contains two data fields, `idNum` and `salary`, as well as properties that contain accessors for each field and a method that creates an `Employee` greeting.

```
class Employee
{
    private int idNum;
    private double salary;
    public int IdNum
    {
        get
        {
            return idNum;
        }
        set
        {
            idNum = value;
        }
    }
    public double Salary
    {
        get
        {
            return salary;
        }
        set
        {
            salary = value;
        }
    }
    public string GetGreeting()
    {
        string greeting = "Hello. I am employee #" + IdNum;
        return greeting;
    }
}
```

Figure 10-1 The Employee class

After you create the `Employee` class, you can create specific `Employee` objects, as in the following:

```
Employee receptionist = new Employee();
Employee deliveryPerson = new Employee();
```

These `Employee` objects can eventually possess different numbers and salaries, but because they are `Employee` objects, you know that each possesses *some* ID number and salary.

Suppose that you hire a new type of `Employee` who earns a commission as well as a salary. You can create a class with a name such as `CommissionEmployee`, and provide this class with three fields (`idNum`, `salary`, and `commissionRate`), three properties (with accessors

to get and set each of the three fields), and a greeting method. However, this work would duplicate much of the work that you already have done for the `Employee` class. The wise and efficient alternative is to create the class `CommissionEmployee` so it inherits all the attributes and methods of `Employee`. Then, you can add just the single field and property with two accessors that are additions within `CommissionEmployee` objects. Figure 10-2 depicts these relationships.

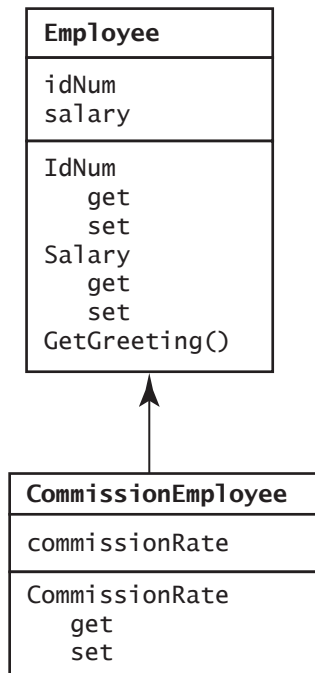


Figure 10-2
`CommissionEmployee`
inherits from `Employee`



In class diagrams, the convention is to provide three sections for the class name, its data, and its methods, respectively. The upward-pointing arrow between the classes in Figure 10-2 indicates that the class on the bottom inherits from the one on the top. Using an arrow in this way is conventional in **Unified Modeling Language (UML) diagrams**, which are graphical tools that programmers and analysts use to describe systems.

When you use inheritance to create the `CommissionEmployee` class, you acquire the following benefits:

- You save time, because you need not re-create the `Employee` fields, properties, and methods.
- You reduce the chance of errors, because the `Employee` properties and methods have already been used and tested.
- You make it easier for anyone who has used the `Employee` class to understand the `CommissionEmployee` class because such users can concentrate on the new features only.

The ability to use inheritance makes programs easier to write, easier to understand, and less prone to errors. Imagine that besides `CommissionEmployee`, you want to create several other specific `Employee` classes (perhaps `PartTimeEmployee`, including a field for hours worked, or `DismissedEmployee`, including a reason for dismissal). By using inheritance, you can develop each new class correctly and more quickly.



In part, the concept of class inheritance is useful because it makes class code reusable and development faster. However, you do not use inheritance simply to save work. When properly used, inheritance always involves a general-to-specific relationship.

Understanding Inheritance Terminology

A class that is used as a basis for inheritance, like `Employee`, is called a **base class**. When you create a class that inherits from a base class (such as `CommissionEmployee`), it is a **derived class** or **extended class**. When presented with two classes that have a parent-child relationship, you can tell which class is the base class and which is the derived class by using the two classes in a sentence with the phrase “is a.” A derived class always “is a” case or instance of the more general base class. For example, a `Tree` class may be a base class to an `Evergreen` class. Every `Evergreen` “is a” `Tree`; however, it is not true that every `Tree` is an `Evergreen`. Thus, `Tree` is the base class, and `Evergreen` is the derived class. Similarly, a `CommissionEmployee` “is an” `Employee`—not always the other way around—so `Employee` is the base class and `CommissionEmployee` is derived.

You can use the terms **superclass** and **subclass** as synonyms for base class and derived class. Thus, `Evergreen` can be called a subclass of the `Tree` superclass. You also can use the terms **parent class** and **child class**. A `CommissionEmployee` is a child to the `Employee` parent. Use the pair of terms with which you are most comfortable; all of these terms will be used interchangeably in this book.

As an alternative way to discover which of two classes is the base class and which is the derived class, you can try saying the two class names together (although this technique might not work with every superclass-subclass pair). When people say their names together in the English language, they state the more specific name before the all-encompassing family name, such as *Ginny Kroening*. Similarly, with classes, the order that “makes more sense” is the child-parent order. Thus, because *Evergreen Tree* makes more sense than *Tree Evergreen*, you can deduce that `Evergreen` is the child class. It also is convenient to think of a derived class as building upon its base class by providing the “adjectives” or additional descriptive terms for the “noun.” Frequently, the names of derived classes are formed in this way, as in `CommissionEmployee`.

Finally, you usually can distinguish base classes from their derived classes by size. A derived class is larger than a base class, in the sense that it usually has additional fields and methods. A derived class description may look small, but any subclass contains all the fields and methods of its superclass as well as its own more specific fields and methods. Do not think of a subclass as a “subset” of another class—in other words, possessing only parts of its superclass. In fact, a derived class contains everything in the superclass, plus any new attributes and methods.

A derived class can be further extended. In other words, a subclass can have a child of its own. For example, after you create a `Tree` class and derive `Evergreen`, you might derive a `Spruce` class from `Evergreen`. Similarly, a `Poodle` class might derive from `Dog`, `Dog` from

`DomesticPet`, and `DomesticPet` from `Animal`. The entire list of parent classes from which a child class is derived constitutes the **ancestors** of the subclass.

After you create the `Spruce` class, you might be ready to create `Spruce` objects. For example, you might create `theTreeInMyBackYard`, or you might create an array of 1000 `Spruce` objects for a tree farm. Similarly, one `Poodle` object might be `myPetDogFiFi`.

Inheritance is **transitive**, which means a child inherits all the members of all its ancestors. In other words, when you declare a `Spruce` object, it contains all the attributes and methods of both an `Evergreen` and a `Tree`. As you work with C#, you will encounter many examples of such transitive chains of inheritance.



When you create your own transitive inheritance chains, you want to place fields and methods at their most general level. In other words, a method named `Grow()` rightfully belongs in a `Tree` class, whereas `LeavesTurnColor()` does not, because the method applies to only some of the `Tree` child classes. Similarly, a `LeavesTurnColor()` method would be better located in a `Deciduous` class than separately within the `Oak` or `Maple` child class.

TWO TRUTHS & A LIE

Understanding Inheritance

1. When you use inheritance to create a class, you save time because you can copy and paste fields, properties, and methods that have already been created for the original class.
2. When you use inheritance to create a class, you reduce the chance of errors because the original class's properties and methods have already been used and tested.
3. When you use inheritance to create a class, you make it easier for anyone who has used the original class to understand the new class because such users can concentrate on the new features.

The false statement is #1. When you use inheritance to create a class, you save time because you need not re-create fields, properties, and methods that have already been created for the original class. You do not copy these class members; you inherit them.

Extending Classes

When you create a class that is an extension or child of another class, you use a single colon between the derived class name and its base class name. For example, the following class header creates a subclass-superclass relationship between `CommissionEmployee` and `Employee`.

```
class CommissionEmployee : Employee
```

Each `CommissionEmployee` object automatically contains the fields and methods of the base class; you then can add new fields and methods to the new derived class. Figure 10-3 shows a `CommissionEmployee` class.

```
class CommissionEmployee : Employee
{
    private double commissionRate;
    public double CommissionRate
    {
        get
        {
            return commissionRate;
        }
        set
        {
            commissionRate = value;
        }
    }
}
```

Figure 10-3 The `CommissionEmployee` class

Although you see only one field defined in the `CommissionEmployee` class in Figure 10-3, it contains three fields: `idNum` and `salary`, inherited from `Employee`, and `commissionRate`, which is defined within the `CommissionEmployee` class. Similarly, the `CommissionEmployee` class contains three properties and a method—two properties and the method are inherited from `Employee`, and one property is defined within `CommissionEmployee` itself. When you write a program that instantiates an object using the following statement, then you can use any of the next statements to set field values for the `salesperson`:

```
CommissionEmployee salesperson = new CommissionEmployee();
salesperson.IdNum = 234;
salesperson.Salary = Convert.ToDouble(ReadLine());
salesperson.CommissionRate = 0.07;
```

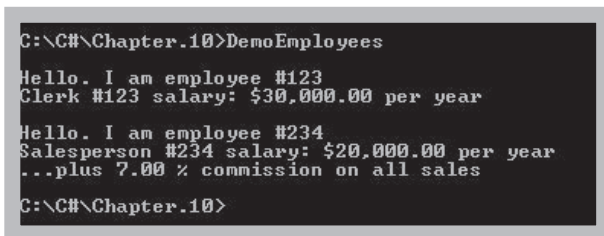
The `salesperson` object has access to all three `set` accessors (two from its parent and one from its own class) because it is both a `CommissionEmployee` and an `Employee`. Similarly, the object has access to three `get` accessors and the `GetGreeting()` method. Figure 10-4 shows a

Main() method that declares Employee and CommissionEmployee objects and shows all the properties and methods that can be used with each. Figure 10-5 shows the program output.

430

```
using static System.Console;
class DemoEmployees
{
    static void Main()
    {
        Employee clerk = new Employee();
        CommissionEmployee salesperson = new CommissionEmployee();
        clerk.IdNum = 123;
        clerk.Salary = 30000.00;
        salesperson.IdNum = 234;
        salesperson.Salary = 20000.00;
        salesperson.CommissionRate = 0.07;
        WriteLine("\n" + clerk.GetGreeting());
        WriteLine("Clerk #{0} salary: {1} per
            year", clerk.IdNum,
            clerk.Salary.ToString("C"));
        WriteLine("\n" + salesperson.GetGreeting());
        WriteLine("Salesperson #{0} salary: {1} per year",
            salesperson.IdNum, salesperson.Salary.ToString("C"));
        WriteLine("...plus {0} commission on all sales",
            salesperson.CommissionRate.ToString("P"));
    }
}
```

Figure 10-4 The DemoEmployees class that declares Employee and CommissionEmployee objects



```
C:\C#\Chapter.10>DemoEmployees
Hello. I am employee #123
Clerk #123 salary: $30,000.00 per year
Hello. I am employee #234
Salesperson #234 salary: $20,000.00 per year
...plus 7.00 % commission on all sales
C:\C#\Chapter.10>
```

Figure 10-5 Output of the DemoEmployees program

Inheritance works only in one direction: A child inherits from a parent—not the other way around. If a program instantiates an Employee object as in the following statement, the Employee object does *not* have access to the CommissionEmployee properties or methods.


```
Employee clerk = new Employee();
clerk.CommissionRate = 0.1;
```

Don't Do It

This statement is invalid—Employee objects don't have a CommissionRate.

Employee is the parent class, and clerk is an object of the parent class. It makes sense that a parent class object does not have access to its child's data and methods. When you create the parent class, you do not know how many future child classes might be created, or what their data or methods might look like. In addition, child classes are more specific. A HeartSurgeon class and an Obstetrician class are children of a Doctor class. You do not expect all members of the general parent class Doctor to have the HeartSurgeon's RepairValve() method or the Obstetrician's DeliverBaby() method. However, HeartSurgeon and Obstetrician objects have access to the more general Doctor methods TakeBloodPressure() and BillPatients(). As with doctors, it is convenient to think of derived classes as *specialists*. That is, their fields and methods are more specialized than those of the base class.



Watch the video *Inheritance*.

TWO TRUTHS & A LIE

Extending Classes

1. The following class header indicates that Dog is a subclass of Pet :

```
public class Pet : Dog
```
2. If class X has four fields and class Y derives from it, then class Y also contains at least four fields.
3. Inheritance works only in one direction: A child inherits from a parent—not the other way around.

The false statement is #1. The following class header indicates that Dog is a subclass of Pet:

```
public class Dog : Pet
```



You Do It

In this section, you create a working example of inheritance. You create this example in four parts:

1. You create a general `Loan` class that holds data pertaining to a bank loan—a loan number, a customer name, and the amount borrowed.
2. After you create the general `Loan` class, you write a program to instantiate and use a `Loan` object.
3. You create a more specific `CarLoan` derived class that inherits the attributes of the `Loan` class but adds information about the automobile that serves as collateral for the loan.
4. You modify the `Loan` demonstration program to add a `CarLoan` object and demonstrate its use.

Creating the `Loan` Class and a Program That Uses It

1. Open a new project named **DemoLoan**, and then enter the following first few lines for a `Loan` class. The class contains three auto-implemented properties for the loan number, the last name of the customer, and the value of the loan.

```
class Loan
{
    public int LoanNumber {get; set;}
    public string LastName {get; set;}
    public double LoanAmount {get; set;}
}
```

2. At the top of the file, enter the following code to add a `DemoLoan` class that contains a `Main()` method. The class declares a `Loan` object and shows how to set each field and display the results.

```
using static System.Console;
class DemoLoan
{
    static void Main()
    {
        Loan aLoan = new Loan();
        aLoan.LoanNumber = 2239;
        aLoan.LastName = "Mitchell";
        aLoan.LoanAmount = 1000.00;
        WriteLine("Loan #{0} for {1} is for {2}",
            aLoan.LoanNumber, aLoan.LastName,
```

(continues)

(continued)

```

        aLoan.LoanAmount.ToString("C2"));
    }
}

```

3. Save the file, and then compile and execute the program. The output looks like Figure 10-6. There is nothing unusual about this class or how it operates; it is similar to many you saw in the last chapter before you learned about inheritance.



```

C:\C#\Chapter.10>DemoLoan
Loan #2239 for Mitchell is for $1,000.00
C:\C#\Chapter.10>

```

Figure 10-6 Output of the DemoLoan program

Extending a Class

Next, you create a class named `CarLoan`. A `CarLoan` “is a” type of `Loan`. As such, it has all the attributes of a `Loan`, but it also has the year and make of the car that the customer is using as collateral for the loan. Therefore, `CarLoan` is a subclass of `Loan`.

1. Save the `DemoLoan` file as **DemoCarLoan**. Change the `DemoLoan` class name to **DemoCarLoan**. Begin the definition of the `CarLoan` class after the closing curly brace for the `Loan` class. `CarLoan` extends `Loan` and contains two properties that hold the year and make of the car.

```

class CarLoan : Loan
{
    public int Year {get; set;}
    public string Make {get; set;}
}

```

2. Within the `Main()` method of the `DemoCarLoan` class, just after the declaration of the `Loan` object, declare a `CarLoan` as follows:


```
CarLoan aCarLoan = new CarLoan();
```
3. After the three property assignments for the `Loan` object, insert five assignment statements for the `CarLoan` object.

```

aCarLoan.LoanNumber = 3358;
aCarLoan.LastName = "Jansen";
aCarLoan.LoanAmount = 20000.00;

```

(continues)

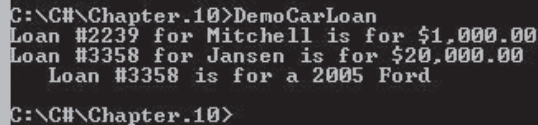
(continued)

```
aCarLoan.Make = "Ford";
aCarLoan.Year = 2005;
```

- Following the `WriteLine()` statement that displays the `Loan` object data, insert two `WriteLine()` statements that display the `CarLoan` object's data.

```
WriteLine("Loan #{0} for {1} is for {2}",
    aCarLoan.LoanNumber, aCarLoan.LastName,
    aCarLoan.LoanAmount.ToString("C2"));
WriteLine("    Loan #{0} is for a {1} {2}",
    aCarLoan.LoanNumber, aCarLoan.Year,
    aCarLoan.Make);
```

- Save the program, and then compile and execute it. The output looks like Figure 10-7. The `CarLoan` object correctly uses its own fields and properties as well as those of the parent `Loan` class.



```
C:\C#\Chapter.10>DemoCarLoan
Loan #2239 for Mitchell is for $1,000.00
Loan #3358 for Jansen is for $20,000.00
    Loan #3358 is for a 2005 Ford
C:\C#\Chapter.10>
```

Figure 10-7 Output of the `DemoCarLoan` program

Using the protected Access Specifier

The `Employee` class in Figure 10-1 is a typical C# class in that its data fields are `private` and its properties and methods are `public`. In the chapter “Using Classes and Objects,” you learned that this scheme provides for information hiding—protecting your `private` data from alteration by methods outside the data's own class. When a program is a client of the `Employee` class (that is, it instantiates an `Employee` object), the client cannot alter the data in any `private` field directly. For example, when you write a `Main()` method that creates an `Employee` named `clerk`, you cannot change the `Employee`'s `idNum` or `salary` directly using a statement such as `clerk.idNum = 2222;`. Instead, you must use the `public IdNum` property to set the `idNum` field of the `clerk` object.

When you use information hiding, you are assured that your data will be altered only by the properties and methods you choose and only in ways that you can control. If outside classes could alter an `Employee`'s `private` fields, then the fields could be assigned values that the `Employee` class couldn't control. In such a case, the principle of information hiding would be destroyed, causing the behavior of the object to be unpredictable.

Any derived class you create, such as `CommissionEmployee`, inherits all the data and methods of its base class. However, even though a child of `Employee` has `idNum` and `salary` fields, the `CommissionEmployee` methods cannot alter or use those `private` fields directly. If a new class could simply extend your `Employee` class and “get to” its data fields without “going through the proper channels,” then information hiding would not be operating.

On some occasions, you do want to access parent class data from a child class. For example, suppose that the `Employee` class `Salary` property `set` accessor has been written so that no `Employee`'s salary is ever set to less than 15000, as follows:

```
set
{
    double MINIMUM = 15000;
    if(value < MINIMUM)
        salary = MINIMUM;
    else
        salary = value;
}
```

Also assume that a `CommissionEmployee` draws commission only and no regular salary; that is, when you set a `CommissionEmployee`'s `commissionRate` field, the `salary` should become 0. You would write the `CommissionEmployee` class `CommissionRate` property `set` accessor as follows:

```
set
{
    commissionRate = value;
    Salary = 0;
}
```

Using this implementation, when you create a `CommissionEmployee` object and set its `CommissionRate`, 0 is sent to the `set` accessor for the `Employee` class `Salary` property. Because the value of the salary is less than 15000, the salary is forced to 15000 in the `Employee` class `set` accessor, even though you want it to be 0.

A possible alternative would be to rewrite the `set` accessor for the `CommissionRate` property in the `CommissionEmployee` class using the field `salary` instead of the property `Salary`, as follows:

```
set
{
    commissionRate = value;
    salary = 0;
}
```

In this `set` accessor, you bypass the parent class's `Salary` `set` accessor and directly use the `salary` field. However, when you include this accessor in a program and compile it, you receive an error message: *Employee.salary is inaccessible due to its protection level*. In other words, `Employee.salary` is `private`, and no other class can access it, even a child class of `Employee`.

So, in summary:

- Using the `public set` accessor in the parent class does not work because of the minimum salary requirement.
- Using the `private` field in the parent class does not work because it is inaccessible.
- Making the parent class field `public` would work, but doing so would violate the principle of information hiding.

Fortunately, there is a fourth option. The solution is to create the `salary` field with **protected access**, which provides you with an intermediate level of security between `public` and `private` access. A `protected` data field or method can be used within its own class or in any classes extended from that class, but it cannot be used by “outside” classes. In other words, `protected` members can be used “within the family”—by a class and its descendents.



Some sources say that `private`, `public`, and `protected` are *access specifiers*, while other class designations, such as `static`, are *access modifiers*. However, Microsoft developers, who created C#, use the terms interchangeably in their documentation.

Figure 10-8 shows how you can declare `salary` as `protected` within the `Employee` class so that it becomes legal to access it directly within the `CommissionRate set` accessor of the `CommissionEmployee` derived class. Figure 10-9 shows a program that instantiates a `CommissionEmployee` object, and Figure 10-10 shows the output. Notice that the `CommissionEmployee`'s salary initially is set to 20000 in the program, but the salary becomes 0 when the `CommissionRate` is set later.

```
class Employee
{
    private int idNum;
    protected double salary;
    public int IdNum
    {
        get
        {
            return idNum;
        }
        set
        {
            idNum = value;
        }
    }
}
```

Figure 10-8 The `Employee` class with a `protected` field and the `CommissionEmployee` class (*continues*)

(continued)

```

public double Salary
{
    get
    {
        return salary;
    }
    set
    {
        double MINIMUM = 15000;
        if(value < MINIMUM)
            salary = MINIMUM;
        else
            salary = value;
    }
}
public string GetGreeting()
{
    string greeting = "Hello. I am employee #" + IdNum;
    return greeting;
}
}
class CommissionEmployee : Employee
{
    private double commissionRate;
    public double CommissionRate
    {
        get
        {
            return commissionRate;
        }
        set
        {
            commissionRate = value;
            salary = 0;
        }
    }
}

```

The protected salary field is accessible in the child class.

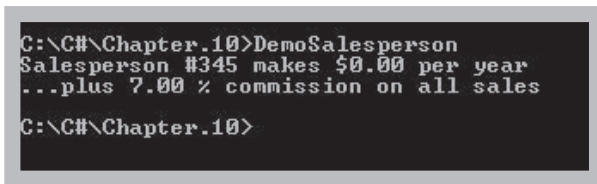
Figure 10-8 The Employee class with a protected field and the CommissionEmployee class

```

using static System.Console;
class DemoSalesperson
{
    static void Main()
    {
        CommissionEmployee salesperson = new CommissionEmployee();
        salesperson.IdNum = 345;
        salesperson.Salary = 20000;
        salesperson.CommissionRate = 0.07;
        WriteLine("Salesperson #{0} makes {1} per year",
            salesperson.IdNum, salesperson.Salary.ToString("C"));
        WriteLine("...plus {0} commission on all sales",
            salesperson.CommissionRate.ToString("P"));
    }
}

```

Figure 10-9 The DemoSalesperson program



```

C:\C#\Chapter.10>DemoSalesperson
Salesperson #345 makes $0.00 per year
...plus 7.00 % commission on all sales
C:\C#\Chapter.10>

```

Figure 10-10 Output of the DemoSalesperson program



If you set the salesperson's `CommissionRate` first in the `DemoSalesperson` program, then set `Salary` to a nonzero value, `Salary` will not be reduced to 0. If your intention is to always create `CommissionEmployee`s with salaries of 0, then the `Salary` property should also be overridden in the derived class.

Using the `protected` access specifier for a field can be convenient, and it also slightly improves program performance by using a field directly instead of “going through” property accessors. Also, using the `protected` access specifier is occasionally necessary. However, `protected` data members should be used sparingly. Whenever possible, the principle of information hiding should be observed, and even child classes should have to go through accessors to “get to” their parent’s private data. When child classes are allowed direct access to a parent’s fields, the likelihood of future errors increases. Classes that depend on field names from parent classes are said to be **fragile** because they are prone to errors—that is, they are easy to “break.”

TWO TRUTHS & A LIE

Using the protected Access Specifier

1. A child class does not contain the `private` members of its parent.
2. A child class cannot use the `private` members of its parent.
3. A child class can use the `protected` members of its parent, but outside classes cannot.

The false statement is #1. A child class contains the `private` members of its parent, but cannot use them directly.

439

Overriding Base Class Members

When you create a derived class by extending an existing class, the new derived class contains properties and methods that were defined in the original base class. Sometimes, the superclass features are not entirely appropriate for the subclass objects. Using the same method or property name to indicate different implementations is called polymorphism. The word *polymorphism* means “many forms”—in programming, it means that many forms of action take place, even though you use the same method name. The specific method executed depends on the object.



You first learned the term *polymorphism* in Chapter 1.

Everyday cases provide many examples of polymorphism:

- Although both are musical instruments and have a `Play()` method, a guitar is played differently than a drum.
- Although both are vehicles and have an `Operate()` method, a bicycle is operated differently than a truck.
- Although both are schools and have a `SatisfyGraduationRequirements()` method, a preschool’s requirements are different from those of a college.

You understand each of these methods based on the context in which it is used. In a similar way, `C#` understands your use of the same method name based on the type of object associated with it.

For example, suppose that you have created a `Student` class as shown in Figure 10-11. `Students` have `names`, `credits` for which they are enrolled, and `tuition` amounts. You can

set a Student's name and credits by using the `set` accessors in the `Name` and `Credits` properties, but you cannot set a Student's tuition directly because there is no `set` accessor for the `Tuition` property. Instead, `tuition` is calculated based on a standard `RATE` (of \$55.75) for each credit that the Student takes.



In Figure 10-11, the Student fields that hold credits and tuition are declared as protected because a child class will use them.

```
class Student
{
    private const double RATE = 55.75;
    private string name;
    protected int credits;
    protected double tuition;
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
    public virtual int Credits
    {
        get
        {
            return credits;
        }
        set
        {
            credits = value;
            tuition = credits * RATE;
        }
    }
    public double Tuition
    {
        get
        {
            return tuition;
        }
    }
}
```

Figure 10-11 The Student class

In Figure 10-11, `Credits` is declared to be virtual (see shading). A **virtual method** (or property) is one that can be overridden by a method with the same signature in a child class. In Chapter 9, you learned that when a method overrides another, it takes precedence over the method.

Suppose that you derive a subclass from `Student` called `ScholarshipStudent`, as shown in Figure 10-12. A `ScholarshipStudent` has a `name`, `credits`, and `tuition`, but the `tuition` is not calculated in the same way as it is for a `Student`; instead, `tuition` for a `ScholarshipStudent` should be set to 0. You want to use the `Credits` property to set a `ScholarshipStudent`'s `credits`, but you want the property to behave differently than the parent class `Student`'s `Credits` property. As a child of `Student`, a `ScholarshipStudent` possesses all the attributes, properties, and methods of a `Student`, but its `Credits` property behaves differently.

```
class ScholarshipStudent : Student
{
    public override int Credits
    {
        set
        {
            credits = value;
            tuition = 0;
        }
    }
}
```

Figure 10-12 The `ScholarshipStudent` class



In C#, you can use either `new` or `override` when defining a derived class member that has the same name as a base class member, but you cannot use both together. When you write a statement such as `ScholarshipStudent s1 = new ScholarshipStudent();`, you won't notice the difference. However, if you use `new` when defining the derived class `Credits` property and write a statement such as `Student s2 = new ScholarshipStudent();` (using `Student` as the type), then `s2.Credits` accesses the base class property. On the other hand, if you use `override` when defining `Credits` in the derived class, then `s2.Credits` uses the derived class property.

In the child `ScholarshipStudent` class in Figure 10-12, the `Credits` property is declared with the `override` modifier (see shading) because it has the same header (that is, the same signature—the same name and parameter list) as a property in its parent class. The `Credits` property overrides and **hides** its counterpart in the parent class. (You could do the same thing with methods.) If you omit `override`, the program will still operate correctly, but you will receive a warning that you are hiding an inherited member with the same name in the base class. Using the keyword `override` eliminates the warning and makes your intentions clear. When you use the `Name` property with a `ScholarshipStudent` object, a program uses the parent class property `Name`; it is not hidden. However, when you use `Credits` to set a value for a `ScholarshipStudent` object, the program uses the new, overriding property from its own class.



If `Credits` and `Tuition` had been declared as `private` within the `Student` class, then `ScholarshipStudent` would not be able to use them.

442

You are not required to override a virtual method or property in a derived class; a derived class can simply use the base class version. A base class member that is not hidden by the derived class is **visible** in the derived class.

Figure 10-13 shows a program that uses `Student` and `ScholarshipStudent` objects. Even though each object assigns the `Credits` property with the same number of credit hours (in the two shaded statements), the calculated `Tuition` values are different because each object uses a different version of the `Credits` property. Figure 10-14 shows the execution of the program.

```
using static System.Console;
class DemoStudents
{
    static void Main()
    {
        Student payingStudent = new Student();
        ScholarshipStudent freeStudent = new ScholarshipStudent();
        payingStudent.Name = "Megan";
        payingStudent.Credits = 15;
        freeStudent.Name = "Luke";
        freeStudent.Credits = 15;
        WriteLine("{0}'s tuition is {1}",
            payingStudent.Name, payingStudent.Tuition.ToString("C"));
        WriteLine("{0}'s tuition is {1}",
            freeStudent.Name, freeStudent.Tuition.ToString("C"));
    }
}
```

Figure 10-13 The `DemoStudents` program

```
C:\C#\Chapter.10>DemoStudents
Megan's tuition is $836.25
Luke's tuition is $0.00

C:\C#\Chapter.10>
```

Figure 10-14 Output of the `DemoStudents` program

If a base class and a derived class have methods with the same names but different parameter lists, then the derived class method does not override the base class method; instead,

it *overloads* the base class method. For example, if a base class contains a method with the header `public void Display()`, and its child contains a method with the header `public void Display(string s)`, then the child class would have access to both methods. (You learned about overloading methods in the chapter “Advanced Method Concepts.”)

Accessing Base Class Methods and Properties from a Derived Class

When a derived class contains a method or property that overrides a parent class method or property, you might have occasion to use the parent class version within the subclass. If so, you can use the keyword `base` to access the parent class method or property.

For example, recall the `GetGreeting()` method that appears in the `Employee` class in Figure 10-8. If its child, `CommissionEmployee`, also contains a `GetGreeting()` method, as shown in Figure 10-15, then within the `CommissionEmployee` class you can call `base.GetGreeting()` to access the base class version of the method. Figure 10-16 shows an application that uses the method with a `CommissionEmployee` object. Figure 10-17 shows the output.

```
class CommissionEmployee : Employee
{
    private double commissionRate;
    public double CommissionRate
    {
        get
        {
            return commissionRate;
        }
        set
        {
            commissionRate = value;
            salary = 0;
        }
    }
    new public string GetGreeting()
    {
        string greeting = base.GetGreeting();
        greeting += "\nI work on commission.";
        return greeting;
    }
}
```

Figure 10-15 The `CommissionEmployee` class with a `GetGreeting()` method

```
using static System.Console;
class DemoSalesperson2
{
    static void Main()
    {
        CommissionEmployee salesperson = new CommissionEmployee();
        salesperson.IdNum = 345;
        WriteLine(salesperson.GetGreeting());
    }
}
```

Figure 10-16 The DemoSalesperson2 program

```
C:\C#\Chapter.10>DemoSalesperson2
Hello, I am employee #345
I work on commission.

C:\C#\Chapter.10>
```

Figure 10-17 Output of the DemoSalesperson2 program

In Figure 10-15, the child class, the `GetGreeting()` method uses the keyword `new` in its header to eliminate a compiler warning. Then, within the method, the parent's version of the `GetGreeting()` method is called. The returned string is stored in the `greeting` variable, and then an "I work on commission." statement is added to it before the complete message is returned to the calling program. By overriding the base class method in the child class, the duplicate typing to create the first part of the message is eliminated. Additionally, if the first part of the message is altered in the future, it will be altered in only one place—in the base class.



Watch the video *Handling Methods and Inheritance*.

TWO TRUTHS & A LIE

Overriding Base Class Members

1. When you override a parent class method in a child class, the methods have the same name.
2. When you override a parent class method in a child class, the methods have the same parameter list.
3. When you override a parent class method in a child class and then use the child class method, the parent class method executes first, followed by the child class method.

The false statement is #3. When you override a parent class method in a child class and then use the child class method, the child class method executes instead of the parent class version.



You Do It

In the previous sections, you created `Loan` and `CarLoan` classes and objects. Suppose that the bank adopts new rules as follows:

- No regular loan will be made for less than \$5000.
- No car loan will be made for any car older than model year 2006.
- Although `Loans` might have larger loan numbers, `CarLoans` will have loan numbers that are no more than three digits. If a larger loan number is provided, the program will use only the last three digits for the loan number.

Using Base Class Members in a Derived Class

1. Open the `DemoCarLoan` file, and immediately save it as **DemoCarLoan2**. Also change the class name from `DemoCarLoan` to **DemoCarLoan2**.
2. Within the `Loan` class, add a new constant that represents the minimum loan value:

```
public const double MINIMUM_LOAN = 5000;
```

(continues)

(continued)

3. Add a field for the `loanAmount` because the `LoanAmount` property, which was previously an auto-implemented property, will need to use it:

```
protected double loanAmount;
```

The field is protected so that `CarLoan` objects will be able to access it as well as `Loan` objects.

4. Replace the auto-implemented property for `LoanAmount` in the `Loan` class with `get` and `set` accessors as follows. This change ensures that no loan is made for less than the minimum allowed value.

```
public double LoanAmount
{
    set
    {
        if(value < MINIMUM_LOAN)
            loanAmount = MINIMUM_LOAN;
        else
            loanAmount = value;
        }
    get
    {
        return loanAmount;
    }
}
```

5. Within the `CarLoan` class, add two new constants to hold the earliest year for which car loans will be given and the lowest allowed loan number:

```
private const int EARLIEST_YEAR = 2006;
private const int LOWEST_INVALID_NUM = 1000;
```

6. Also within the `CarLoan` class, add a field for the year of the car, and replace the existing auto-implemented `Year` property with one that contains coded `get` and `set` accessors. The `Year` property `set` accessor not only sets the `year` field, it sets `loanAmount` to 0 when a car's year is less than 2006.

```
private int year;
public int Year
{
    set
    {
        if(value < EARLIEST_YEAR)
        {
            year = value;
            loanAmount = 0;
        }
    }
```

(continues)

(continued)

```

        else
            year = value;
    }
    get
    {
        return year;
    }
}

```

If `loanAmount` were `private` in the parent `Loan` class, you would not be able to set its value in the child `CarLoan` class, as you do here. You could use the `public` property `LoanAmount` to set the value, but the parent class `set` accessor would force the value to 5000.

- Suppose that unique rules apply for issuing loan numbers for cars. Within the `CarLoan` class, just before the closing curly brace, change the inherited `LoanNumber` property to accommodate the new rules. If a car loan number is three digits or fewer, pass it on to the base class property. If not, obtain the last three digits by calculating the remainder when the loan number is divided by 1000, and pass the new number to the base class property. Add the following property after the definition of the `Make` property.

```

public new int LoanNumber
{
    get
    {
        return base.LoanNumber;
    }
    set
    {
        if(value < LOWEST_INVALID_NUM)
            base.LoanNumber = value;
        else
            base.LoanNumber = value % LOWEST_INVALID_NUM;
    }
}

```

If you did not use the keyword `base` to access the `LoanNumber` property within the `CarLoan` class, you would be telling this version of the `LoanNumber` property to call itself. Although the program would compile, it would run continuously in an infinite loop until it ran out of memory and issued an error message.

(continues)

(continued)



A method that calls itself is a **recursive** method. Recursive methods are sometimes useful, but they require specialized code to avoid infinite loops and are not appropriate in this case.

8. Save the file. Compile it, and correct any errors. When you execute the program, the output looks like Figure 10-18. Compare the output to Figure 10-7. Notice that the \$1000 bank loan has been forced to \$5000. Also notice that the car loan number has been shortened to three digits and the value of the loan is \$0 because of the age of the car.

```
C:\C#\Chapter.10>DemoCarLoan2
Loan #2239 for Mitchell is for $5,000.00
Loan #358 for Jansen is for $0.00
    Loan #358 is for a 2005 Ford
C:\C#\Chapter.10>
```

Figure 10-18 Output of the DemoCarLoan2 program

9. Change the assigned values within the DemoCarLoan2 class to combinations of early and late years and valid and invalid loan numbers. After each change, save the program, compile and execute it, and confirm that the program operates as expected.

Understanding Implicit Reference Conversions

Every derived object “is a” specific instance of both the derived class and the base class. In other words, myCar “is a” Car as well as a Vehicle, and myDog “is a” Dog as well as a Mammal. You can assign a derived class object to an object of any of its superclass types. When you do, C# makes an **implicit conversion** from derived class to base class.

You have already learned that C# also makes implicit conversions when casting one data type to another. For example, in the statement `double money = 10;`, the integer value 10 is implicitly converted (or cast) to a `double`. When a derived class object is assigned to its ancestor’s data type, the conversion can more specifically be called an **implicit reference conversion**. This term is more accurate because it emphasizes the difference between numerical conversions and reference objects. When you assign a derived class object to a base class type, the object is treated as though it had only the characteristics defined in the base class and not those added in the child class definition.

For example, when a `CommissionEmployee` class inherits from `Employee`, an object of either type can be passed to a method that accepts an `Employee` parameter. In Figure 10-19, an `Employee` is passed to `DisplayGreeting()` in the first shaded statement, and a `CommissionEmployee` is passed in the second shaded statement. Each is referred to as `emp` within the method, and each is used correctly, as shown in Figure 10-20.

```
using static System.Console;
class DemoSalesperson3
{
    static void Main()
    {
        Employee clerk = new Employee();
        CommissionEmployee salesperson = new CommissionEmployee();
        clerk.IdNum = 234;
        salesperson.IdNum = 345;
        DisplayGreeting(clerk);
        DisplayGreeting(salesperson);
    }
    public static void DisplayGreeting(Employee emp)
    {
        WriteLine("Hi there from #" + emp.IdNum);
        WriteLine(emp.GetGreeting());
    }
}
```

Figure 10-19 The `DemoSalesperson3` program

```
C:\C#\Chapter.10>DemoSalesperson3
Hi there from #234
Hello. I am employee #234
Hi there from #345
Hello. I am employee #345
C:\C#\Chapter.10>
```

Figure 10-20 Output of the `DemoSalesperson3` program

TWO TRUTHS & A LIE

Understanding Implicit Reference Conversions

1. You can assign a derived class object to an object of any of its superclass types.
2. You can assign a base class object to an object of any of its derived types.
3. An implicit conversion from one type to another is an automatic conversion.

The false statement is #2. You can assign a derived class object to an object of any of its superclass types but not the other way around.

Using the Object Class

Every class you create in C# derives from a single class named `System.Object`. In other words, the **object** (or `Object`) class type in the `System` namespace is the ultimate base class, or **root class**, for all other types. When you create a class such as `Employee`, you usually use the header `class Employee`, which implicitly, or automatically, descends from the `Object` class. Alternatively, you could use the header `class Employee : Object` to explicitly show the name of the base class, but it would be extremely unusual to see such a format in a C# program.

The keyword `object` is an alias for the `System.Object` class. You can use the lowercase and uppercase versions of the class interchangeably. The fact that `object` is an alias for `System.Object` should not surprise you. You already know, for example, that `int` is an alias for `Int32` and that `double` is an alias for `Double`.

Because every class descends from `Object`, every `object` “is an” `Object`. As proof, you can write a method that accepts an argument of type `Object`, and it will accept arguments of any type. Figure 10-21 shows a program that declares three objects using classes created earlier in this chapter—a `Student`, a `ScholarshipStudent`, and an `Employee`. Even though these types possess different attributes and methods (and one type, `Employee`, has nothing in common with the other two), each type can serve as an argument to the `DisplayObjectMessage()` because each type “is an” `Object`. Figure 10-22 shows the execution of the program.

```
using System;
using static System.Console;
class DiverseObjects
{
    static void Main()
    {
        Student payingStudent = new Student();
    }
}
```

Figure 10-21 The `DiverseObjects` program (*continues*)

(continued)

```

        ScholarshipStudent freeStudent = new ScholarshipStudent();
        Employee clerk = new Employee();
        Write("Using Student: ");
        DisplayObjectMessage(payingStudent);
        Write("Using ScholarshipStudent: ");
        DisplayObjectMessage(freeStudent);
        Write("Using Employee: ");
        DisplayObjectMessage(clerk);
    }
    public static void DisplayObjectMessage(Object o)
    {
        WriteLine("Method successfully called");
    }
}

```

Figure 10-21 The DiverseObjects program

```

C:\C#\Chapter.10>DiverseObjects
Using Student: Method successfully called
Using ScholarshipStudent: Method successfully called
Using Employee: Method successfully called

C:\C#\Chapter.10>

```

Figure 10-22 Output of the DiverseObjects program

When you create any child class, it inherits all the methods of all of its ancestors. Because all classes inherit from the `Object` class, all classes inherit the `Object` class methods. The `Object` class contains a constructor, a destructor, and four `public` instance methods, as summarized in Table 10-1.

Method	Explanation
<code>Equals()</code>	Determines whether two <code>Object</code> instances are equal
<code>GetHashCode()</code>	Gets a unique code for each object; useful in certain sorting and data management tasks
<code>GetType()</code>	Returns the type, or class, of an object
<code>ToString()</code>	Returns a <code>String</code> that represents the object

Table 10-1 The four `public` instance methods of the `Object` class



The `Object` class contains other nonpublic and noninstance (static) methods in addition to the four methods listed in Table 10-1. The C# documentation provides more details on these methods.

Using the `Object` Class's `GetType()` Method

The `GetType()` method returns an object's type, or class. For example, if you have created an `Employee` object named `someWorker`, then the following statement displays *Employee*:

```
WriteLine(someWorker.GetType());
```

If an object's class is defined in a namespace, then `GetType()` returns a string composed of the namespace, a dot, and the class name.

Using the `Object` Class's `ToString()` Method

The `Object` class methods are not very useful as they stand. For example, when you use the `Object` class's `ToString()` method with an object you create, it simply returns a string that holds the name of the class, just as `GetType()` does. That is, if `someWorker` is an `Employee`, then the following statement displays *Employee*:

```
WriteLine(someWorker.ToString());
```

When you create a class such as `Employee`, you often want to override the `Object` class's `ToString()` method with your own, more useful version—perhaps one that returns an `Employee`'s ID number, name, or combination of the two. Of course, you could create a differently named method to do the same thing—perhaps `GetEmployeeIdentification()` or `ConvertEmployeeToString()`. However, by naming your class method `ToString()`, you make the class easier for others to understand and use. Programmers know the `ToString()` method works with every object; when they use it with your objects, you can provide a useful set of information. A class's `ToString()` method is often a useful debugging aid.

For example, you might create an `Employee` class `ToString()` method, as shown in Figure 10-23. This method assumes that `IdNum` and `Name` are `Employee` properties with `get` accessors. The returned `string` will have a value such as *Employee: 234 Johnson*.

```
public override string ToString()
{
    return(getType() + ": " + IdNum + " " + Name);
}
```

Figure 10-23 An `Employee` class `ToString()` method



You have been using an overloaded version of the `ToString()` method since Chapter 2. There, you learned that you can format numeric output when you pass a string such as “F3” or “C2” to the `ToString()` method.

Using the Object Class’s `Equals()` Method

The `Equals()` method compares objects for reference equality. **Reference equality** occurs when two reference type objects refer to the same object. The `Equals()` method returns `true` if two `Objects` have the same memory address—that is, if one object is a reference to the other and both are literally the same object. For example, you might write the following:

```
if(oneObject.Equals(anotherObject))...
```

Like the `ToString()` method, this method might not be useful to you in its original form. For example, you might prefer to think of two `Employee` objects at unique memory addresses as equal if their ID numbers or first and last names are equal. You might want to override the `Equals()` method for any class you create if you anticipate that class clients will want to compare objects based on any of their field values.

If you overload the `Equals()` method, it should meet the following requirements by convention:

- Its header should be as follows (you can use any identifier for the `Object` parameter):

```
public override bool Equals(Object o)
```

- It should return `false` if the argument is `null`.
- It should return `true` if an object is compared to itself.
- It should return `true` only if both of the following are true:

```
oneObject.Equals(anotherObject)
anotherObject.Equals(oneObject)
```

- If `oneObject.Equals(anotherObject)` returns `true` and `oneObject.Equals(aThirdObject)` returns `true`, then `anotherObject.Equals(aThirdObject)` should also be `true`.



You first used the `Equals()` method to compare `String` objects in Chapter 2. When you use `Equals()` with `Strings`, you use the `String` class’s `Equals()` method that compares `String` contents as opposed to `String` addresses. In other words, the `Object` class’s `Equals()` method has already been overridden in the `String` class.

When you create an `Equals()` method to override the one in the `Object` class, the parameter must be an `Object`. For example, if you consider `Employee` objects equal when their `IdNum` properties are equal, then an `Employee` class `Equals()` method might be created as follows:

```
public override bool Equals(Object e)
{
    bool isEqual;
    Employee temp = (Employee)e;
    if(IdNum == temp.IdNum)
        isEqual = true;
    else
        isEqual = false;
    return isEqual;
}
```

In the shaded statement in the method, the `Object` parameter is cast to an `Employee` so the `Employee`'s `IdNum` can be compared. If you did not perform the cast and tried to make the comparison with `e.IdNum`, the method would not compile because an `Object` does not have an `IdNum` property.

An even better alternative is to ensure that compared objects are the same type before making any other decisions. For example, the `Equals()` method in Figure 10-24 uses the `GetType()` method with both the `this` object and the parameter before proceeding. If compared objects are not the same type, then the `Equals()` method should return `false`.

```
public override bool Equals(Object e)
{
    bool isEqual = true;
    if(this.GetType() != e.GetType())
        isEqual = false;
    else
    {
        Employee temp = (Employee)e;
        if(IdNum == temp.IdNum)
            isEqual = true;
        else
            isEqual = false;
    }
    return isEqual;
}
```

Figure 10-24 An `Equals()` method for the `Employee` class

Using the `Object` Class's `GetHashCode()` Method

When you override the `Equals()` method, you should also override the `GetHashCode()` method, because `Equals()` uses `GetHashCode()`, and two objects considered equal should

have the same hash code. A **hash code** is a number that should uniquely identify an object; you might use hash codes in some advanced C# applications. For example, Figure 10-25 shows an application that declares two `Employee`s from a class in which the `GetHashCode()` method has not been overridden. The output in Figure 10-26 shows a unique number for each object. (The number, however, is meaningless to you.) If you choose to override the `GetHashCode()` method, you should write this method so it returns a unique integer for every object—an `Employee` ID number, for example.



A hash code is sometimes called a *fingerprint* for an object because it uniquely identifies the object. In C#, the default implementation of the `GetHashCode()` method does not guarantee unique return values for different objects. However, if `GetHashCode()` is explicitly implemented in a derived class, it must return a unique hash code.

In cooking, hash is a dish that is created by combining ingredients. The term hash code derives from the fact that the code is sometimes created by mixing some of an object's data.

```
using static System.Console;
class TestHashCode
{
    static void Main()
    {
        Employee first = new Employee();
        Employee second = new Employee();
        WriteLine(first.GetHashCode());
        WriteLine(second.GetHashCode());
    }
}
```

Figure 10-25 The `TestHashCode` program

A screenshot of a terminal window showing the execution of the TestHashCode program. The prompt is 'C:\C#\Chapter.10>TestHashCode' and the output is '46104728' followed by '12289376'. The prompt is then 'C:\C#\Chapter.10>' again.

```
C:\C#\Chapter.10>TestHashCode
46104728
12289376
C:\C#\Chapter.10>
```

Figure 10-26 Output of the `TestHashCode` program

Although you can write an `Equals()` method for a class without overriding `GetHashCode()`, you receive a warning message. Additionally, if you overload `==` or `!=` for a class, you will receive warning messages if you do not also override both the `Equals()` and `GetHashCode()` methods.

TWO TRUTHS & A LIE

Using the Object Class

1. The `Object` class contains a method named `GetType()` that returns an object's type, or class.
2. If you do not override the `ToString()` method for a class, it returns the value of the first string declared within the class, if any.
3. The `Object` class's `Equals()` method returns `true` if two `Objects` have the same memory address—that is, if one object is a reference to the other and both are literally the same object.

The false statement is #2. If you do not override the `ToString()` method for a class, it returns a string that holds the name of the class.

Working with Base Class Constructors

When you create any object, you call a constructor that has the same name as the class itself. Consider the following example:

```
SomeClass anObject = new SomeClass();
```

When you instantiate an object that is a member of a derived class, the constructor for the base class executes first, and then the derived class constructor executes. In other words, when you create any object, you always implicitly call the `Object` constructor because all classes are derived from `Object`. So, when you create a base class and a derived class, and instantiate a derived class object, you call three constructors: one from the `Object` class, one from the base class, and one from the derived class.

In the examples of inheritance you have seen so far in this chapter, each class contained default constructors, so their execution was transparent. However, you should realize that when you create a subclass instance, both the base and derived constructors execute. For example, consider the abbreviated `Employee` and `CommissionEmployee` classes in Figure 10-27. `Employee` contains just two fields and a constructor; `CommissionEmployee` descends from `Employee` and contains a constructor as well. The `DemoSalesperson4` program in Figure 10-28 contains just one statement; it instantiates a `CommissionEmployee`. The output in Figure 10-29 shows that this one statement causes both constructors to execute.

```
class Employee
{
    private int idNum;
    protected double salary;
    public Employee()
    {
        WriteLine("Employee constructed");
    }
}
class CommissionEmployee : Employee
{
    private double commissionRate;
    public CommissionEmployee()
    {
        WriteLine("CommissionEmployee constructed");
    }
}
```

Figure 10-27 The `Employee` and `CommissionEmployee` classes with parameterless constructors

```
using static System.Console;
class DemoSalesperson4
{
    static void Main()
    {
        CommissionEmployee salesperson = new CommissionEmployee();
    }
}
```

Figure 10-28 The `DemoSalesperson4` program

```
C:\C#\Chapter.10>DemoSalesperson4
Employee constructed
CommissionEmployee constructed
C:\C#\Chapter.10>
```

Figure 10-29 Output of the `DemoSalesperson4` program

Of course, most constructors perform many more tasks than displaying a message to inform you that they exist. When constructors initialize variables, you usually want the base class constructor to initialize the data fields that originate in the base class. The derived class constructor needs to initialize only the data fields that are specific to the derived class.

Using Base Class Constructors That Require Arguments

When you create a class and do not provide a constructor, C# automatically supplies one that never requires arguments. When you write your own constructor for a class, you replace the automatically supplied version. Depending on your needs, the constructor you create for a class might require arguments. When you use a class as a base class and the class only has constructors that require arguments, you must make sure that any derived classes provide arguments so that one of the base class constructors can execute.

When all base class constructors require arguments, you must include a constructor for each derived class you create. Your derived class constructor can contain any number of statements; however, within the header of the constructor, you must provide values for any arguments required by the base class constructor that you use. Even if you have no other reason for creating a derived class constructor, you must write the derived class constructor so it can call its parent's constructor.

The header for a derived class constructor that calls a base class constructor includes a colon, the keyword `base`, and a list of arguments within parentheses. The keyword `base` always refers to the superclass of the class in which you use it. Although it seems that you should be able to use the base class constructor name to call the base class constructor, C# does not allow you to do so—you must use the keyword `base`.

For example, if you create an `Employee` class with a constructor that requires two arguments—an integer and a string—and you create a `CommissionEmployee` class that is a subclass of `Employee`, then the following code shows a valid constructor for `CommissionEmployee`:

```
public CommissionEmployee() : base(1234, "XXXX")
{
    // Other statements can go here
}
```

In this example, the `CommissionEmployee` constructor requires no arguments for its own execution, but it passes two arguments to its base class constructor. Every `CommissionEmployee` instantiation passes `1234` and `XXXX` to the `Employee` constructor. A different `CommissionEmployee` constructor might accept arguments; then it could pass the appropriate arguments on to the base class constructor, as in the following example:

```
public CommissionEmployee(int id, string name) : base(id, name)
{
    // Other statements can go here
}
```

Yet another `CommissionEmployee` constructor might require that some arguments be passed to the base class constructor and that some be used within `CommissionEmployee`. Consider the following example:

```
public CommissionEmployee(int id, string name, double rate) :
    base(id, name) // two parameters passed to base constructor
{
```

```

CommissionRate = rate;
// rate is used within child constructor
// Other statements can go here
}

```



Watch the video *Constructors and Inheritance*.

TWO TRUTHS & A LIE

Working with Base Class Constructors

1. When you create any derived class object, the base class constructor executes first, followed by the derived class constructor.
2. When a base class constructor requires arguments, you must include a constructor for each derived class you create.
3. When a derived class's constructor requires arguments, all of the arguments must be passed to the base class constructor.

The false statement is #3. When a derived class's constructor requires arguments, all of the arguments might be needed in the derived class, or perhaps all must be passed to the base class constructor. It also might be possible that some arguments are passed to the base class constructor and others are used within the derived class constructor.



You Do It

Adding Constructors to Base and Derived Classes

When a base class contains only constructors that require parameters, then any derived classes must provide for the base class constructor. In the next steps, you add constructors to the `Loan` and `CarLoan` classes and demonstrate that they work as expected.

1. Open the `DemoCarLoan2` program, and change the class name to **DemoCarLoan3**. Save the file as **DemoCarLoan3**.

(continues)

(continued)

- In the `Loan` class, just after the declaration of the `LoanAmount` field, add a constructor that requires values for all the `Loan`'s properties:

```
public Loan(int num, string name, double amount)
{
    LoanNumber = num;
    LastName = name;
    LoanAmount = amount;
}
```

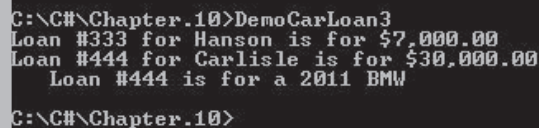
- In the `CarLoan` class, just after the declaration of the `year` field, add a constructor that takes five parameters. It passes three of the parameters to the base class constructor and uses the other two to assign values to the properties that are unique to the child class.

```
public CarLoan(int num, string name, double amount,
               int year, string make) : base(num, name, amount)
{
    Year = year;
    Make = make;
}
```

- In the `Main()` method of the `DemoCarLoan3` class, remove the existing declarations for `aLoan` and `aCarLoan`, and replace them with two declarations that use the arguments passed to the constructors.

```
Loan aLoan = new Loan(333, "Hanson", 7000.00);
CarLoan aCarLoan = new CarLoan(444, "Carlisle",
                               30000.00, 2011, "BMW");
```

- Remove the eight statements that assigned values to `Loan` and `CarLoan`, but retain the `writeLine()` statements that display the values.
- Save the program, and then compile and execute it. The output looks like Figure 10-30. Both constructors work as expected. The `CarLoan` constructor has called its parent's constructor to set the necessary fields before executing its own unique statements.



```
C:\C#\Chapter.10>DemoCarLoan3
Loan #333 for Hanson is for $7,000.00
Loan #444 for Carlisle is for $30,000.00
    Loan #444 is for a 2011 BMW
C:\C#\Chapter.10>
```

Figure 10-30 Output of the `DemoCarLoan3` program

Creating and Using Abstract Classes

Creating classes can become easier after you understand the concept of inheritance. When you create a child class, it inherits all the general attributes you need; you must create only the new, more specific attributes required by the child class. For example, a `Painter` and a `Sculptor` are more specific than an `Artist`. They inherit all the general attributes of `Artists`, but you must add the attributes and methods that are specific to `Painter` and `Sculptor`.

Another way to think about a superclass is to notice that it contains the features shared by its subclasses. The derived classes are more specific examples of the base class type; they add features to the shared, general features. Conversely, when you examine a derived class, you notice that its parent is more general.

Sometimes you create a parent class to be so general that you never intend to create any specific instances of the class. For example, you might never create “just” an `Artist`; each `Artist` is more specifically a `Painter`, `Sculptor`, `Illustrator`, and so on. A class that is used to instantiate objects is a **concrete class**. A class that you create only to extend from, but not to instantiate from, is an abstract class. An **abstract class** is one from which you cannot create concrete objects but from which you can inherit. You use the keyword `abstract` when you declare an abstract class. If you attempt to instantiate an object from an abstract class, you will receive a compiler error message.

Abstract classes are like regular classes in that they can contain data fields and methods. The difference is that you cannot create instances of abstract classes by using the `new` operator. Rather, you create abstract classes simply to provide a base class from which other objects may be derived.

Abstract classes usually contain abstract methods, and they also can contain nonabstract methods. However, they are not required to contain any methods. Recall from Chapter 9 that an abstract method has no statements. Any class derived from a class that contains an abstract method must override the abstract method by providing a body (an implementation) for it. (Alternatively, the derived class can declare the method to be abstract; in that case, the derived class’s children must implement the method.) You can create an `abstract` class with no `abstract` methods, but you cannot create an `abstract` method outside of an `abstract` class.



A method that is declared `virtual` is not required to be overridden in a child class, but a method declared `abstract` must be overridden.

When you create an abstract method, you provide the keyword `abstract` and the intended method type, name, and parameters, but you do not provide statements within the method; you do not even supply curly braces. When you create a derived class that inherits an abstract method from a parent, you must use the keyword **override** in the method header and provide

the actions, or implementation, for the inherited method within the derived class. In other words, you are required to code a derived class method to override any empty base class methods that are inherited.

462

For example, suppose that you want to create classes to represent different animals. You can create a generic, abstract class named `Animal` so you can provide generic data fields, such as the animal's name, only once. An `Animal` is generic, but each specific `Animal`, such as `Dog` or `Cat`, makes a unique sound. If you code an abstract `Speak()` method in the abstract `Animal` class, then you require all future `Animal` derived classes to override the `Speak()` method and provide an implementation that is specific to the derived class. Figure 10-31 shows an abstract `Animal` class that contains a data field for the name, a constructor that assigns a name, a `Name` property, and an abstract `Speak()` method.

```
abstract class Animal
{
    private string name;
    public Animal(string name)
    {
        this.name = name;
    }
    public string Name
    {
        get
        {
            return name;
        }
    }
    public abstract string Speak();
}
```

Figure 10-31 The `Animal` class

The `Animal` class in Figure 10-31 is declared to be **abstract**. (The keyword is shaded.) You cannot place a statement such as `Animal myPet = new Animal("Murphy");` within a program, because the program will not compile. Because `Animal` is an abstract class, no `Animal` objects can exist.

You create an abstract class like `Animal` so that you can extend it. For example, you can create `Dog` and `Cat` classes as shown in Figure 10-32. Because the `Animal` class contains a constructor that requires a `string` argument, both `Dog` and `Cat` must contain constructors that provide `string` arguments for their base class.


```
class Dog : Animal
{
    public Dog(string name) : base(name)
    {
    }
    public override string Speak()
    {
        return "woof";
    }
}
class Cat : Animal
{
    public Cat(string name) : base(name)
    {
    }
    public override string Speak()
    {
        return "meow";
    }
}
```

Figure 10-32 The Dog and Cat classes



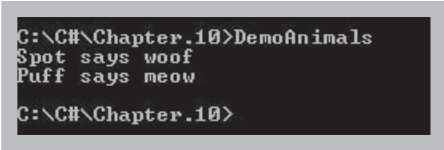
If a method that should be overridden in a child class has its own implementation, you declare the base class method to be `virtual`. If it does not have its own implementation, you declare the base class and the method to be `abstract`.

The `Dog` and `Cat` constructors perform no tasks other than passing out the name to the `Animal` constructor. The overriding `Speak()` methods within `Dog` and `Cat` are required because the abstract parent `Animal` class contains an abstract `Speak()` method. The keyword `override` (shaded) is required in each `Speak()` method header. You can code any statements you want within the `Dog` and `Cat` class `Speak()` methods, but the methods must exist.

Figure 10-33 shows a program that implements `Dog` and `Cat` objects. Figure 10-34 shows the output, in which `Speak()` operates correctly for each animal type.

```
using static System.Console;
class DemoAnimals
{
    static void Main()
    {
        Dog spot = new Dog("Spot");
        Cat puff = new Cat("Puff");
        WriteLine(spot.Name + " says " + spot.Speak());
        WriteLine(puff.Name + " says " + puff.Speak());
    }
}
```

Figure 10-33 The DemoAnimals program



```
C:\C#\Chapter.10>DemoAnimals
Spot says woof
Puff says meow
C:\C#\Chapter.10>
```

Figure 10-34 Output of the DemoAnimals program

Figure 10-35 shows an alternate way to create the DemoAnimals program. In this version the Dog and Cat objects are passed to a method that accepts an Animal parameter. The output is the same as in Figure 10-34. The Name property and Speak() method operate polymorphically, acting appropriately for each object type.

```
using static System.Console;
class DemoAnimals2
{
    static void Main()
    {
        Dog spot = new Dog("Spot");
        Cat puff = new Cat("Puff");
        DisplayAnimal(spot);
        DisplayAnimal(puff);
    }
    public static void DisplayAnimal(Animal creature)
    {
        WriteLine(creature.Name + " says " + creature.Speak());
    }
}
```

Figure 10-35 The DemoAnimals2 program

TWO TRUTHS & A LIE

Creating and Using Abstract Classes

1. An abstract class is one from which you cannot create concrete objects.
2. Unlike regular classes, abstract classes cannot contain methods.
3. When a base class contains an abstract method, the descendents of the base class must override the abstract method or declare the overriding method to be abstract.

465

The false statement is #2. Abstract classes are like regular classes in that they can contain data fields and methods. The difference is that you cannot create instances of abstract classes by using the new operator. Rather, you create abstract classes simply to provide a base class from which other objects may be derived.

Creating and Using Interfaces

Some object-oriented programming languages, notably C++, allow a subclass to inherit from more than one parent class. For example, you might create an `Employee` class that contains data fields pertaining to each employee in your organization. You also might create a `Product` class that holds information about each product your organization manufactures. When you create a `Patent` class for each product for which your company holds a patent, you might want to include product information as well as information about the employee who was responsible for the invention. In this situation, it would be convenient to inherit fields and methods from both the `Product` and `Employee` classes. The ability to inherit from more than one class is called **multiple inheritance**.

Multiple inheritance is a difficult concept, and programmers encounter many problems when they use it. For example, variables and methods in the parent classes may have identical names, creating a conflict when the child class uses one of the names. Additionally, as you already have learned, a child class constructor must call its parent class constructor. When two or more parents exist, this becomes a more complicated task: To which class should `base` refer when a child class has multiple parents?

For all of these reasons, multiple inheritance is prohibited in C#. However, C# does provide an alternative to multiple inheritance, known as an interface. Much like an abstract class,

an **interface** is a collection of methods (and perhaps other members) that can be used by any class as long as the class provides a definition to override the interface's abstract definitions. Within an abstract class, some methods can be abstract, while others need not be. Within an interface, all methods are abstract.

466



You first learned about interfaces in the chapter “Using Classes and Objects” when you used the `IComparable` interface.

You create an interface much as you create an abstract class definition, except that you use the keyword `interface` instead of `abstract class`. For example, suppose that you create an `IWorkable` interface as shown in Figure 10-36. For simplicity, the `IWorkable` interface contains a single method named `Work()`.



Although not required, in C# it is customary to start interface names with an uppercase *I*. Other languages follow different conventions. Interface names frequently end with *able*.

```
public interface IWorkable
{
    string Work();
}
```

Figure 10-36 The `IWorkable` interface

When any class implements `IWorkable`, it must also include a `Work()` method that returns a `string`. Figure 10-37 shows two classes that implement `IWorkable`: the `Employee` class and the `Animal` class. Because each implements `IWorkable`, each must declare a `Work()` method. The `Employee` class implements `Work()` to return the *I do my job* string. The abstract `Animal` class defines `Work()` as an abstract method, meaning that descendants of `Animal` must implement `Work()`. Figure 10-37 also shows two child classes of `Animal`: `Dog` and `Cat`. Note how `Work()` is defined differently for each.

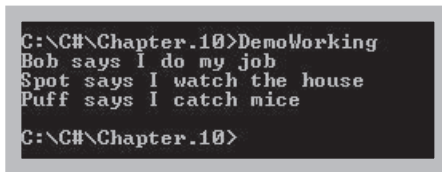
```
class Employee : IWorkable
{
    public Employee(string name)
    {
        Name = name;
    }
    public string Name {get; set;}
    public string Work()
    {
        return "I do my job";
    }
}
abstract class Animal : IWorkable
{
    public Animal(string name)
    {
        Name = name;
    }
    public string Name {get; set;}
    public abstract string Work();
}
class Dog : Animal
{
    public Dog(string name) : base(name)
    {
    }
    public override string Work()
    {
        return "I watch the house";
    }
}
class Cat : Animal
{
    public Cat(string name) : base(name)
    {
    }
    public override string Work()
    {
        return "I catch mice";
    }
}
```

Figure 10-37 The Employee, Animal, Cat, and Dog classes with the IWorkable interface

When you create a program that instantiates an Employee, a Dog, or a Cat, as in the DemoWorking program in Figure 10-38, each object type knows how to “Work()” appropriately. Figure 10-39 shows the output.

```
using static System.Console;
class DemoWorking
{
    static void Main()
    {
        Employee bob = new Employee("Bob");
        Dog spot = new Dog("Spot");
        Cat puff = new Cat("Puff");
        WriteLine(bob.Name + " says " + bob.Work());
        WriteLine(spot.Name + " says " + spot.Work());
        WriteLine(puff.Name + " says " + puff.Work());
    }
}
```

Figure 10-38 The DemoWorking program



```
C:\C#\Chapter.10>DemoWorking
Bob says I do my job
Spot says I watch the house
Puff says I catch mice
C:\C#\Chapter.10>
```

Figure 10-39 Output of the DemoWorking program

Abstract classes and interfaces are similar in that you cannot instantiate concrete objects from either one. Abstract classes differ from interfaces in that abstract classes can contain nonabstract methods, but all methods within an interface must be abstract. A class can inherit from only one base class (whether abstract or not), but it can implement any number of interfaces. For example, if you want to create a `Child` that inherits from a `Parent` class and implements two interfaces, `IWorkable` and `IPlayable`, you would define the class name and list the base class and interfaces separated by commas:

```
class Child : Parent, IWorkable, IPlayable
```

You implement an existing interface because you want a class to be able to use a method that already exists in other applications. For example, suppose that you have created a `Payroll` application that uses the `Work()` method in the interface class. Also suppose that you create a new class named `BusDriver`. If `BusDriver` implements the `IWorkable` interface, then `BusDriver` objects can be used by the existing `Payroll` program. As another example, suppose that you have written a game program that uses an `IAttackable` interface with methods that determine how

and when an object can attack. When you create new classes such as `MarsAlien`, `Vampire`, and `CivilWarSoldier`, and each implements the `IAttackable` interface, you must define how each one attacks and how each type of object can be added to the game. If you use these `IAttackable` classes, you are guaranteed that they can all determine how and when to attack.



You can think of an interface as a contract. A class that implements an interface must abide by the rules of the contract.

Beginning programmers sometimes find it difficult to decide when to create an abstract base class and when to create an interface. You can follow these guidelines:

- Typically, you create an abstract class when you want to provide some data or methods that derived classes can inherit, but you want the subclasses to override some specific methods that you declare to be **abstract**.
- You create an interface when you want derived classes to override every method.

In other words, you inherit from an abstract base class when the class you want to create “is a” subtype, and you use an interface when the class you want to create will act like the interface.

Interfaces provide you with a way to exhibit polymorphic behavior. If diverse classes implement the same interface in unique ways, then you can treat each class type in the same way using the same language. When various classes use the same interface, you know the names of the methods that are available with those classes, and `C#` classes adopt a more uniform functionality; this consistency helps you to understand new classes you encounter more easily. If you know, for example, the method names contained in the `IWorkable` interface, and you see that a class implements `IWorkable`, you have a head start in understanding how the class functions.



Now that you understand how to construct your own interfaces, you will benefit from rereading the section describing the `IComparable` interface in the chapter “Using Classes and Objects.”

TWO TRUTHS & A LIE

Creating and Using Interfaces

1. An interface is a collection of methods (and perhaps other members) that can be used by any class as long as the class provides a definition to override the interface's abstract definitions.
2. Abstract classes and interfaces differ in that all methods in abstract classes must be abstract, but interfaces can contain nonabstract methods.
3. A class can inherit from only one base class, but it can implement any number of interfaces.

The false statement is #2. Abstract classes and interfaces are similar in that you cannot instantiate concrete objects from either one. However, they differ in that abstract classes can contain nonabstract methods, but all methods within an interface must be abstract.

Using Extension Methods

When you write a C# program, sometimes you might wish a class had an additional method that would be useful to you. If you created the original class, you have two options:

- You could revise the existing class, including the new useful method.
- You could derive a child class from the existing class and provide it with a new method.

Sometimes, however, classes you use were created by others, and you might not be allowed to either revise or extend them. Of course, you could create an entirely new class that includes your new method, but that would duplicate a lot of the work already done when the first class was created. In these cases, the best option is to write an extension method. **Extension methods** are methods you can write to add to any type. Extension methods were introduced in C# 3.0.



Programmers sometimes define classes as sealed within the class header, as in `sealed class InventoryItem`. A **sealed class** cannot be extended. For example, the built-in `String` class is a sealed class.

For example, you have used the prewritten `Int32` class throughout this book to declare integers. Suppose that you work for a company that frequently uses customer account numbers, and that the company has decided to add an extra digit to each account number. For simplicity, assume that all account numbers are two digits and that the new, third digit should be the rightmost digit in the sum of the first two digits. You could handle this problem by creating a class named

`AccountNumber`, including a method to produce the extra digit, and redefining every instance of a customer's account number in your applications as an `AccountNumber` object. However, if you already have many applications that define the account number as an integer, you might prefer to create an extension method that extends the `Int32` class.



In Chapter 2 you learned that each C# intrinsic type, such as `int`, is an alias for a class in the `System` namespace, such as `Int32`.



When organizations append extra digits to account numbers, the extra digits are called check digits. *Check digits* help assure that all the digits in account numbers and other numbers are entered correctly. Check digits are calculated using different formulas. If a digit used to calculate the check digit is incorrect, then the resulting check digit is probably incorrect as well.

Figure 10-40 contains a method that extends the `Int32` class. The first parameter in an extension method specifies the type extended and must begin with the keyword `this`. For example, the first (and in this case, only) parameter in the `AddCheckDigit()` method is `this int num`, as shown in the shaded portion of the figure. Within the `AddCheckDigit()` method in Figure 10-40, the first digit is extracted from the two-digit account number by dividing by 10 and taking the resulting whole number, and the second digit is extracted by taking the remainder. Those two digits are added, and the last digit of that sum is returned from the method. For example, if 49 is passed into the method, `first` becomes 4, `second` becomes 9, and `third` becomes the last digit of 13, or 3. Then the original number (49) is multiplied by 10 and added to the third digit, resulting in 493.

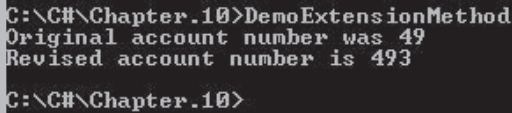
```
public static int AddCheckDigit(this int num)
{
    int first = num / 10;
    int second = num % 10;
    int third = (first + second) % 10;
    int result = num * 10 + third;
    return result;
}
```

Figure 10-40 The `AddCheckDigit()` extension method

An extension method must be static and must be stored in a static class. For example, the `DemoExtensionMethod` program in Figure 10-41 shows an application that is declared `static` in the shaded portion of the class header and uses the extension method in the second shaded statement. The static method `AddCheckDigit()` is used as if it were an instance method of the `Int32` class; in other words, it is attached to an `Int32` object with a dot, just as instance methods are when used with objects. No arguments are passed to the `AddCheckDigit()` method explicitly from the `DemoExtensionMethod` class. The parameter in the method is implied, just as these references are always implied in instance methods. Figure 10-42 shows the execution of the program.

```
using static System.Console;
static class DemoExtensionMethod
{
    static void Main()
    {
        int acctNum = 49;
        int revisedAcctNum = acctNum.AddCheckDigit();
        WriteLine("Original account number was {0}", acctNum);
        WriteLine("Revised account number is {0}", revisedAcctNum);
    }
    public static int AddCheckDigit(this int num)
    {
        int first = num / 10;
        int second = num % 10;
        int third = (first + second) % 10;
        int result = num * 10 + third;
        return result;
    }
}
```

Figure 10-41 The DemoExtensionMethod application



```
C:\C#\Chapter.10>DemoExtensionMethod
Original account number was 49
Revised account number is 493
C:\C#\Chapter.10>
```

Figure 10-42 Execution of the DemoExtensionMethod application

You can create extension methods for your own classes in the same way one was created for the `Int32` class in this example. Just like other outside methods, and unlike ordinary class instance methods, extension methods cannot access any private members of classes they extend. Furthermore, if a class contains an instance method with the same signature as an extension method, the instance method takes priority and will be the one that executes.

TWO TRUTHS & A LIE

Using Extension Methods

1. The first parameter in an extension method specifies the type extended and must be preceded by the keyword `this`.
2. Extension methods must be static methods.
3. When you write an extension method, it must be stored within the class to which it refers, along with the class's other instance methods.

The false statement is #3. Although you use an extension method like an instance method, any extension method you write must be stored in a static class.

Recognizing Inheritance in GUI Applications and Recapping the Benefits of Inheritance

When you create a Windows Forms application using Visual Studio's IDE, you automatically use inheritance. Every Form you create is a descendent of the Form class. Figure 10-43 shows a just-started project in which the programmer has double-clicked the automatically generated Form to expose the code. You can see that the automatically generated Form1 class extends Form.

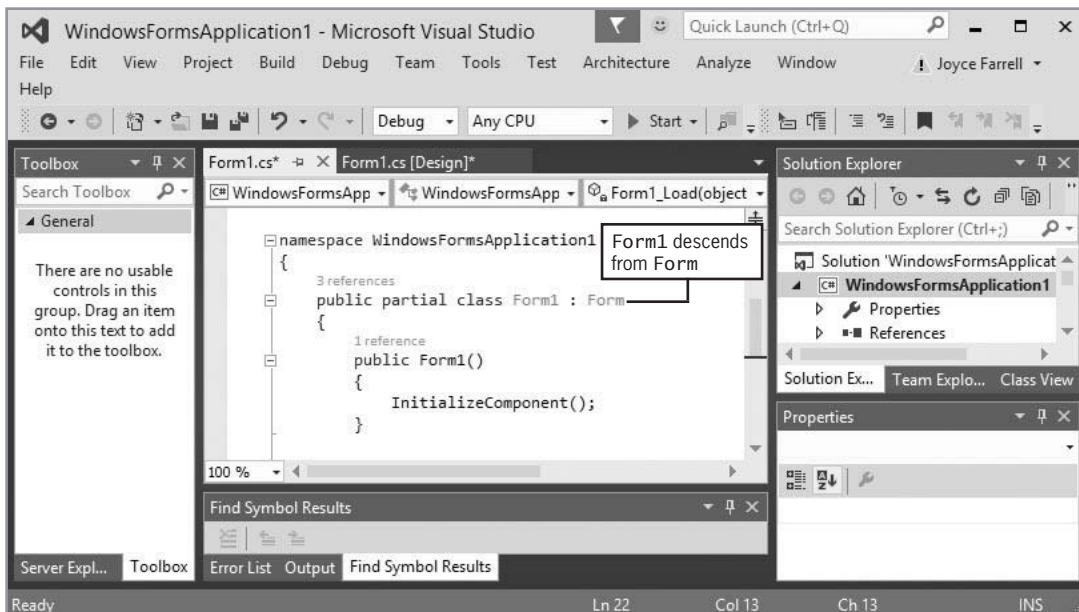


Figure 10-43 Automatically generated Form code in the IDE

The `Form` class descends from the `Object` class like all other `C#` classes but not directly. It is six generations removed from the `Object` class in the following line of descent:

- `Object`
- `MarshalByRefObject`
- `Component`
- `Control`
- `ScrollableControl`
- `ContainerControl`
- `Form`

Other GUI objects such as `Labels` and `Buttons` follow similar lengthy ancestry lines. You might guess that certain universal properties of GUI controls such as `Text` and `Visible` are inherited from ancestors. You will learn more about these hierarchies in the chapter “Using Controls,” but even though you have worked with only a few controls so far, you can understand the benefits inheritance provides.

When an automobile company designs a new car model, it does not build every component from scratch. The car might include a new feature—for example, some model contained the first air bag—but many of a new car’s features are simply modifications of existing features. The manufacturer might create a larger gas tank or a more comfortable seat, but these new features still possess many of the properties of their predecessors from older models. Most features of new car models are not even modified; instead, existing components, such as air filters and windshield wipers, are included on the new model without any changes.

Similarly, you can create many computer programs more easily if many of their components are used either “as is” or with slight modifications. Inheritance does not enable you to write any programs that you could not write if inheritance did not exist; you *could* create every part of a program from scratch, but reusing existing classes and interfaces makes your job easier.

You already have used many “as is” classes, such as `Console`, `Int32`, and `String`. Using these classes made it easier to write programs than if you had to invent the classes yourself. Now that you have learned about inheritance, you can extend existing classes as well as just use them. When you create a useful, extendable base class, you and other future programmers gain several advantages:

- Derived class creators save development time because much of the code that is needed for the class already has been written.
- Derived class creators save testing time because the base class code already has been tested and probably used in a variety of situations. In other words, the base class code is reliable.

- Programmers who create or use new derived classes already understand how the base class works, so the time it takes to learn the new class features is reduced.
- When you create a derived class in C#, the base class source code is not changed. Thus, the base class maintains its integrity.



Classes that are not intended to be instantiated and that contain only `static` members are declared as `static` classes. You cannot extend `static` classes. For example, `System.Console` is a `static` class.

When you think about classes, you need to think about the commonalities between them, and then you can create base classes from which to inherit. You might even be rewarded professionally when you see your own superclasses extended by others in the future.

TWO TRUTHS & A LIE

Recognizing Inheritance in GUI Applications and Recapping the Benefits of Inheritance

1. Inheritance enables you to create powerful computer programs more easily.
2. Without inheritance, you *could* create every part of a program from scratch, but reusing existing classes and interfaces makes your job easier.
3. Inheritance is frequently inefficient because base class code is seldom reliable when extended to a derived class.

The false statement is #3. Derived class creators save testing time because the base class code already has been tested and probably used in a variety of situations. In other words, the base class code is reliable.

Chapter Summary

476

- The classes you create in object-oriented programming languages can inherit data and methods from existing classes. The ability to use inheritance makes programs easier to write, easier to understand, and less prone to errors. A class that is used as a basis for inheritance is called a base class, superclass, or parent class. When you create a class that inherits from a base class, it is called a derived class, extended class, subclass, or child class.
- When you create a class that is an extension or child of another class, you use a single colon between the derived class name and its base class name. The child class inherits all the methods, properties, and fields of its parent. Inheritance works only in one direction—a child inherits from a parent, but not the other way around.
- If you could use private data outside of its class, the principle of information hiding would be destroyed. However, when you must access parent class data from a derived class, you declare parent class fields using the keyword **protected**, which provides you with an intermediate level of security between **public** and **private** access.
- When you declare a child class method with the same name and parameter list as a method within its parent class, you override the parent class method and allow your class objects to exhibit polymorphic behavior. You can use the keyword **new** or **override** with the derived class method. When a derived class overrides a parent class member but you want to access the parent class version, you can use the keyword **base**.
- Every derived class object “is a” specific instance of both the derived class and the base class. Therefore, you can assign a derived class object to an object of any of its base class types. When you do so, *C#* makes an implicit conversion from derived class to base class.
- Every class you create in *C#* derives from a single class named **System.Object**. Because all classes inherit from the **Object** class, all classes inherit the four **Object** class **public** instance methods: **Equals()**, **GetHashCode()**, **GetType()**, and **ToString()**.
- When you instantiate an object that is a member of a subclass, the base class constructor executes first, and then the derived class constructor executes.
- An abstract class is one from which you cannot create concrete objects but from which you can inherit. Usually, abstract classes contain abstract methods; an abstract method has no method statements. Any class derived from a class that contains an abstract method must override the abstract method by providing a body (an implementation) for it.
- An interface provides an alternative to multiple inheritance. Much like an abstract class, an interface is a collection of methods (and perhaps other members) that can be used by any class as long as the class provides a definition to override the interface’s abstract definitions. Within an abstract class, some methods can be abstract, while others need not be. Within an interface, all methods are abstract. A class can inherit from only one abstract base class, but it can implement any number of interfaces.

- Extension methods are methods you can write to add to any type. They are static methods, but they operate like instance methods. Their parameter lists begin with the keyword `this` and the data type being extended.
- GUI objects such as buttons and forms descend from several ancestors. You can create many computer programs more easily using inheritance because it saves development and testing time.

Key Terms

Inheritance is the application of your knowledge of a general category to more specific objects.

Unified Modeling Language (UML) diagrams are graphical tools that programmers and analysts use to describe systems.

A **base class** is a class that is used as a basis for inheritance.

A **derived class** or **extended class** is one that has inherited from a base class.

A **superclass** is a base class.

A **subclass** is a derived class.

A **parent class** is a base class.

A **child class** is a derived class.

The **ancestors** of a derived class are all the superclasses from which the subclass is derived.

Transitive describes the feature of inheritance in which a child inherits all the members of all its ancestors.

Protected access provides an intermediate level of security between public and private access; a **protected** data field or method can be used within its own class or in any classes extended from that class, but it cannot be used by “outside” classes.

Fragile describes classes that depend on field names from parent classes because they are prone to errors—that is, they are easy to “break.”

A **virtual method** is one whose behavior is determined by the implementation in a child class.

To **hide** a parent class member is to override it in a derived class.

Visible describes a base class member that is not hidden by a derived class.

Recursive describes a method that calls itself.

An **implicit conversion** occurs when a type is automatically converted to another upon assignment.

An **implicit reference conversion** occurs when a derived class object is assigned to its ancestor’s data type.

The **object** (or **Object**) class type in the **System** namespace is the root base class for all other types.

A **root class** is the first base class in a hierarchical ancestry tree.

Reference equality occurs when two reference type objects refer to the same object.

A **hash code** is a number that should uniquely identify an object.

A **concrete class** is a nonabstract class from which objects can be instantiated.

An **abstract class** is one from which you cannot create concrete objects but from which you can inherit.

The keyword **override** is used in method headers when you create a derived class that inherits an abstract method from a parent.

Multiple inheritance is the ability to inherit from more than one class.

An **interface** is a collection of abstract methods (and perhaps other members) that can be used by any class as long as the class provides a definition to override the interface's abstract definitions.

Extension methods are static methods that act like instance methods. You can write extension methods to add to any type.

A **sealed class** cannot be extended.

Review Questions

1. Specific types of objects assume features of more general classes through _____ .
 - a. polymorphism
 - b. encapsulation
 - c. inheritance
 - d. structure
2. Which of the following is *not* a benefit of using inheritance when creating a new class?
 - a. You save time, because you need not create fields and methods that already exist in a parent class.
 - b. You reduce the chance of errors, because the parent class methods have already been used and tested.
 - c. You make it easier for anyone who has used the parent class to understand the new class because the programmer can concentrate on the new features.
 - d. You save computer memory because when you create objects of the new class, storage is not required for parent class fields.

3. A child class is also called a(n) _____ .
- a. extended class
 - b. base class
 - c. superclass
 - d. delineated class
4. Assuming that the following classes are well named, which of the following is a parent class of **House**?
- a. **Apartment**
 - b. **Building**
 - c. **Victorian**
 - d. **myHouse**
5. A derived class usually contains _____ than its parent.
- a. more fields and methods
 - b. the same number of fields but fewer methods
 - c. fewer fields but more methods
 - d. fewer fields and methods
6. When you create a class that is an extension or child of another class, you use a(n) _____ between the derived class name and its base class name.
- a. ampersand
 - b. colon
 - c. dot
 - d. hyphen
7. A base class named **Garden** contains a private field **width** and a property **public int Width** that contains **get** and **set** accessors. A child class named **VegetableGarden** does not contain a **Width** property. When you write a class in which you declare an object as follows, what statement can you use to access the **VegetableGarden's width**?
- ```
VegetableGarden myGarden = new VegetableGarden();
```
- a. `myGarden.Width`
  - b. `myGarden.base.Width`
  - c. `VegetableGarden.Width`
  - d. You cannot use **Width** with a **VegetableGarden** object.
8. When a parent class contains a **private** data field, the field is \_\_\_\_\_ the child class.
- a. hidden in
  - b. not a member of
  - c. directly accessible in
  - d. **public** in

9. When a base class and a derived class contain a method with the same name and parameter list, and you call the method using a derived class object, \_\_\_\_\_ .
- you receive an error message
  - the base class version overrides the derived class version
  - the derived class version overrides the base class version
  - both method versions execute
10. Which of the following is an English-language form of polymorphism?
- seeing a therapist and seeing the point
  - moving friends with a compelling story and moving friends to a new apartment
  - both of these
  - neither of these
11. When base and derived classes contain a method with the same name and parameter list, you can use the base class method within the derived class by using the keyword \_\_\_\_\_ before the method name.
- |                    |                 |
|--------------------|-----------------|
| a. <b>new</b>      | c. <b>base</b>  |
| b. <b>override</b> | d. <b>super</b> |
12. In a program that declares a derived class object, you \_\_\_\_\_ assign it to an object of its base class type.
- |           |               |
|-----------|---------------|
| a. can    | c. must       |
| b. cannot | d. should not |
13. The root base class for all other class types is \_\_\_\_\_ .
- |                 |                  |
|-----------------|------------------|
| a. <b>Base</b>  | c. <b>Parent</b> |
| b. <b>Super</b> | d. <b>Object</b> |
14. All of the following are **Object** class methods *except* \_\_\_\_\_ .
- |                      |                         |
|----------------------|-------------------------|
| a. <b>ToString()</b> | c. <b>Print()</b>       |
| b. <b>Equals()</b>   | d. <b>GetHashCode()</b> |

15. When you create any derived class object, \_\_\_\_\_ .
- the base class and derived class constructors execute simultaneously
  - the base class constructor must execute first, and then the derived class constructor executes
  - the derived class constructor must execute first, and then the base class constructor executes
  - neither the base class constructor nor the derived class constructor executes
16. When a base class constructor requires arguments, then each derived class \_\_\_\_\_ .
- must include a constructor
  - must include a constructor that requires arguments
  - must include two or more constructors
  - must not include a constructor
17. When you create an abstract class, \_\_\_\_\_ .
- you can inherit from it
  - you can create concrete objects from it
  - Both of these are true.
  - None of these is true.
18. When you create an abstract method, you provide \_\_\_\_\_ .
- the keyword **abstract**
  - curly braces
  - method statements
  - all of these
19. Within an interface, \_\_\_\_\_ .
- no methods can be **abstract**
  - some methods might be **abstract**
  - some, but not all, methods must be **abstract**
  - all methods must be **abstract**

20. Abstract classes and interfaces are similar in that \_\_\_\_\_ .
- you can instantiate concrete objects from both
  - you cannot instantiate concrete objects from either one
  - all methods in both must be **abstract**
  - neither can contain nonabstract methods

## Exercises



### Programming Exercises

- Create an application class named **LetterDemo** that instantiates objects of two classes named **Letter** and **CertifiedLetter** and that demonstrates all their methods. The classes are used by a company to keep track of letters they mail to clients. The **Letter** class includes auto-implemented properties for the name of the recipient and the date mailed. Also, include a **ToString()** method that overrides the **Object** class's **Tostring()** method and returns a string that contains the name of the class (using **GetType()**) and the **Letter**'s data field values. Create a child class named **CertifiedLetter** that includes an auto-implemented property that holds a tracking number for the letter.
- Create an application class named **PhotoDemo** that demonstrates the methods of three related classes for a company that develops photographs. Create a class named **Photo** that includes fields for width and height in inches and properties for each field. Include a **protected** price field, and set it to \$3.99 for an 8-inch by 10-inch photo, \$5.99 for a 10-inch by 12-inch photo, and \$9.99 for any other size (because custom cutting is required). The price field requires a **get** accessor but no **set** accessor. Also include a **Tostring()** method that returns a string constructed from the return value of the object's **GetType()** method and the values of the fields. Derive two subclasses—**MattedPhoto** and **FramedPhoto**. The **MattedPhoto** class includes a string field to hold a color, and the **FramedPhoto** class includes two string fields that hold the frame's material (such as *silver*) and style (such as *modern*). The price for a **MattedPhoto** increases by \$10 over its base cost, and the price for a **FramedPhoto** increases by \$25 over its base cost. Each subclass should include a **Tostring()** method that overrides the parent class version.
- Create an application named **OrderDemo** that declares and uses **Order** objects. The **Order** class performs order processing of a single item that sells for \$19.95 each. The class has four variable fields that hold an order number, customer name, quantity ordered, and total price. Create a constructor that requires parameters for all the fields except total price. Include **public get** and **set** accessors for each field except the total price field; that field is calculated as quantity ordered times

unit price (19.95) whenever the quantity is set, so it needs only a `get` accessor. Also create the following for the class:

- An `Equals()` method that determines two `Orders` are equal if they have the same order number
- A `GetHashCode()` method that returns the order number
- A `ToString()` method that returns a string containing all order information

The `OrderDemo` application declares a few `Order` objects and sets their values. Make sure to create at least two orders with the same order number. Display the string from the `ToString()` method for each order. Write a method that compares two orders at a time and displays a message if they are equal. Send the `Orders` you created to the method two at a time and display the results.

- b. Using the `Order` class you created in Exercise 3a, write a new application named **OrderDemo2** that creates an array of five `Orders`. Prompt the user for values for each `Order`. Do not allow duplicate order numbers; force the user to reenter the order when a duplicate order number is entered. When five valid orders have been entered, display them all, plus a total of all orders.
  - c. Create a `ShippedOrder` class that derives from `Order`. A `ShippedOrder` has a \$4.00 shipping fee (no matter how many items are ordered). Override any methods in the parent class as necessary. Write a new application named **OrderDemo3** that creates an array of five `ShippedOrders`. Prompt the user for values for each, and do not allow duplicate order numbers—force the user to reenter the order when a duplicate order number is entered. When five valid orders have been entered, display them all, plus a total of all orders.
  - d. Make any necessary modifications to the `ShippedOrder` class so that it can be sorted by order number. Modify the `OrderDemo3` application so the displayed orders have been sorted. Save the application as **OrderDemo4**.
4. Create an application named **PackageDemo** that declares and demonstrates objects of the `Package` class and its descendents. The `Package` class includes auto-implemented properties for an ID number, recipient's name, and weight in ounces. The class also contains a delivery price field that is set when the weight is set as \$5 for the first 32 ounces and 12 cents per ounce for every ounce over 32.
    - b. Create a child class named **InsuredPackage**, which includes a field for the package's value. Override the method that sets a `Package`'s delivery price to include insurance, which is \$1 for packages valued up to \$20 and \$2.50 for packages valued \$20 or more.
  5. a. Create an application named **AutomobileDemo** that prompts a user for data for eight `Automobile` objects. The `Automobile` class includes auto-implemented properties for ID number, make, year, and price. Override the `ToString()` method to return all the details for a `Automobile`. During data entry, reprompt the user if any ID number is a duplicate. Sort the objects in ID number order, and display all their data as well as a total of all their prices.

- b. Using the **Automobile** class as a base, derive a **FinancedAutomobile** class that contains all the data of an **Automobile**, plus fields to hold the amount financed and interest rate. Override the parent class **ToString()** method to include the child class's additional data. Create a program named **AutomobileDemo2** that contains an array of four **FinancedAutomobile** objects. Prompt the user for all the necessary data, and do not allow duplicate ID numbers and do not allow the amount financed to be greater than the price of the automobile. Sort all the records in ID number order and display them with a total price for all **FinancedAutomobiles** and a total amount financed.
- c. Write an application named **AutomobileDemo3** that uses an extension method for the **FinancedAutomobile** class. The method computes and returns a **FinancedAutomobile**'s monthly payment due (1/24 of the amount financed). The application should allow the user to enter data for four objects and then display all the data for each.
6. Create an application named **ShapesDemo** that creates several objects that descend from an abstract class called **GeometricFigure**. Each **GeometricFigure** includes a height, a width, and an area. Provide **get** and **set** accessors for each field except area; the area is computed and is read only. Include an abstract method called **ComputeArea()** that computes the area of the **GeometricFigure**. Create three additional classes:
- A **Rectangle** is a **GeometricFigure** whose area is determined by multiplying width by height.
  - A **Square** is a **Rectangle** in which the width and height are the same. Provide a constructor that accepts both height and width, forcing them to be equal if they are not. Provide a second constructor that accepts just one dimension and uses it for both height and width. The **Square** class uses the **Rectangle**'s **ComputeArea()** method.
  - A **Triangle** is a **GeometricFigure** whose area is determined by multiplying the width by half the height.

In the **ShapesDemo** class, after each object is created, pass it to a method that accepts a **GeometricFigure** argument in which the figure's data is displayed. Change some dimensions of some of the figures, and pass each to the display method again.

7. Create an application named **RecoveringDemo** that declares objects of three types: **Patient**, **Upholsterer**, and **FootballPlayer**. Create an interface named **IRecoverable** that contains a single method named **Recover()**. Create the classes named **Patient**, **Upholsterer**, and **FootballPlayer** so that each implements **IRecoverable**. Create each class's **Recover()** method to display an appropriate message. For example, the **Patient**'s **Recover()** method might display "I am getting better."
8. Create an application named **TurningDemo** that creates instances of four classes: **Page**, **Corner**, **Pancake**, and **Leaf**. Create an interface named **ITurnable** that contains a single method named **Turn()**. The classes named **Page**, **Corner**,

`Pancake`, and `Leaf` implement `ITurnable`. Create each class's `Turn()` method to display an appropriate message. For example, the `Page`'s `Turn()` method might display "You turn a page in a book."

9. Write a program named **SalespersonDemo** that instantiates objects using classes named `RealEstateSalesperson` and `GirlScout`. Demonstrate that each object can use a `SalesSpeech()` method appropriately. Also, use a `MakeSale()` method two or three times with each object, and display the final contents of each object's data fields. First, create an abstract class named `Salesperson`. Fields include first and last names; the `Salesperson` constructor requires both these values. Include properties for the fields. Include a method that returns a string that holds the `Salesperson`'s full name—the first and last names separated by a space. Then perform the following tasks:
  - Create two child classes of `Salesperson`: `RealEstateSalesperson` and `GirlScout`. The `RealEstateSalesperson` class contains fields for total value sold in dollars and total commission earned (both of which are initialized to 0), and a commission rate field required by the class constructor. The `GirlScout` class includes a field to hold the number of boxes of cookies sold, which is initialized to 0. Include properties for every field.
  - Create an interface named `ISellable` that contains two methods: `SalesSpeech()` and `MakeSale()`. In each `RealEstateSalesperson` and `GirlScout` class, implement `SalesSpeech()` to display an appropriate one- or two-sentence sales speech that the objects of the class could use.

In the `RealEstateSalesperson` class, implement the `MakeSale()` method to accept an integer dollar value for a house, add the value to the `RealEstateSalesperson`'s total value sold, and compute the total commission earned. In the `GirlScout` class, implement the `MakeSale()` method to accept an integer representing the number of boxes of cookies sold and add it to the total field.



## Debugging Exercises

1. Each of the following files in the Chapter.10 folder of your downloadable student files has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with *Fixed*. For example, `DebugTen01.cs` will become `FixedDebugTen01.cs`.
  - a. `DebugTen01.cs`
  - b. `DebugTen02.cs`
  - c. `DebugTen03.cs`
  - d. `DebugTen04.cs`



## Case Problems

486

1. In Chapter 9, you created a **Contestant** class for the Greenville Idol competition. The class includes a contestant's name, talent code, and talent description. The competition has become so popular that separate contests with differing entry fees have been established for children, teenagers, and adults. Modify the **Contestant** class to contain a field that holds the entry fee for each category, and add **get** and **set** accessors.

Extend the **Contestant** class to create three subclasses: **ChildContestant**, **TeenContestant**, and **AdultContestant**. Child contestants are 12 years old and younger, and their entry fee is \$15. Teen contestants are between 13 and 17 years old, inclusive, and their entry fee is \$20. Adult contestants are 18 years old and older, and their entry fee is \$30. In each subclass, set the entry fee field to the correct value, and override the **ToString()** method to return a string that includes all the contestant data, including the age category and the entry fee.

Modify the **GreenvilleRevenue** program so that it performs the following tasks:

- The program prompts the user for the number of contestants in this year's competition, which must be between 0 and 30. The program continues to prompt the user until a valid value is entered.
  - The program prompts the user for names, ages, and talent codes for the contestants entered. Along with the prompt for a talent code, display a list of valid categories. Based on the age entered for each contestant, create an object of the correct type (adult, teen, or child), and store it in an array of **Contestant** objects.
  - After data entry is complete, display the total expected revenue, which is the sum of the entry fees for the contestants.
  - After data entry is complete, display the valid talent categories and then continuously prompt the user for talent codes, and display all the data for all the contestants in each category. Display an appropriate message if the entered code is not a character or a valid code.
2. In Chapter 9, you created a **Mural** class for Marshall's Murals. The class holds a customer's name, a mural code, and a description. Now, add a field to the **Mural** class that holds a price. Extend the **Mural** class to create subclasses named **InteriorMural** and **ExteriorMural**, and place statements that determine a mural's price within these classes. (A mural's price depends on the month, as described in the case problem in Chapter 9.) Also create **ToString()** methods for these subclasses that return a string containing all the pertinent data for a mural.



Modify the **MarshallsRevenue** program so that it performs the following tasks:

- The program prompts the user for the month, the number of interior murals scheduled, and the number of exterior murals scheduled. In each case, the program continues to prompt the user until valid entries are made.
- The program prompts the user for customer names and mural codes for interior and exterior murals. Along with the prompt for a mural code, display a list of valid categories.
- After data entry is complete, display the total revenue of interior murals, exterior murals, and all murals.
- After data entry is complete, the program displays the valid mural categories and then continuously prompts the user for codes and uses the `Tostring()` method to display all the details for the murals in each category. Appropriate messages are displayed if the entered code is not a character or a valid code.

