- 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.

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.

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.

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.

You Do It

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

1. You will 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 will write a program to instantiate and use a `Loan` object.

3. You will 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 will modify the `Loan` demonstration program to add a `CarLoan` object and demonstrate its use.

To create the Loan class:

1. Open a new file in your text editor, then enter the following first few lines for a `Loan` class. The class will contain 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 System;
public class DemoLoan
{
   public static void Main()
   {
      Loan aLoan = new Loan();
      aLoan.LoanNumber = 2239;
      aLoan.LastName = "Mitchell";
      aLoan.LoanAmount = 1000.00;
      Console.WriteLine("Loan #{0} for {1} is for {2}",
         aLoan.LoanNumber, aLoan.LastName,
         aLoan.LoanAmount.ToString("C2"));
   }
}
```

3. Save the file as **DemoLoan.cs**, then compile and execute the program. The output looks like Figure 10-40. 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.

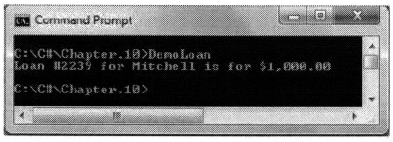

Figure 10-40 Output of the `DemoLoan` program

Extending a Class

Next, you will 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.

To create the CarLoan class that extends the Loan class:

1. Save the DemoLoan.cs file as **DemoCarLoan.cs**. 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;
aCarLoan.Make = "Ford";
aCarLoan.Year = 2005;
```

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

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

5. Save the program, then compile and execute it. The output looks like Figure 10-41. The CarLoan object correctly uses its own fields and properties as well as those of the parent Loan class.

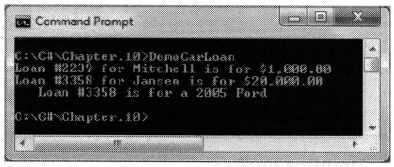

Figure 10-41 Output of the `DemoCarLoan` program

Using Base Class Members in a Derived Class

In the previous sections, you created `Loan` and `CarLoan` classes and objects. Suppose 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.

To implement the new `CarLoan` rules:

1. Open the DemoCarLoan.cs file and immediately save it as **DemoCarLoan2.cs**. 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;
   ```

3. Add a field for the `loanAmount` because the `LoanAmount` 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 standard `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;
      }
      else
         year = value;
   }
   get
   {
      return year;
   }
}
```

If `loanAmount` was `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.

7. Suppose there are unique rules 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;
   }
}
```

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.

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.

8. Save the file. Compile it and correct any errors. When you execute the program, the output looks like Figure 10-42. Compare the output to Figure 10-41. 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.

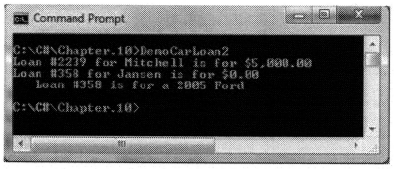

Figure 10-42 Output of the `DemoCarLoan2` program

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.

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 will add constructors to the Loan and CarLoan classes and demonstrate that they work as expected.

To add constructors to the classes:

1. Open the DemoCarLoan2 program and change the class name to DemoCarLoan3. Save the file as **DemoCarLoan3.cs**.
2. 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;
}
```

3. 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;
}
```

4. 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");
```

5. Remove the eight statements that assigned values to Loan and CarLoan, but retain the Console.WriteLine() statements that display the values.
6. Save the program, then compile and execute it. The output looks like Figure 10-43. 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.

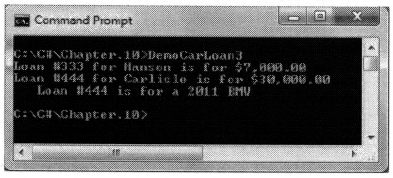

Figure 10-43 Output of the `DemoCarLoan3` program

Chapter Summary

- 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 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. On some occasions, however, you want to access parent class data from a derived class. For those occasions, 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 method but you want to access the parent class version of the method, 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