10

Classes and Objects: A Deeper Look

- In Visual C# 2008, you can use **extension methods** to add functionality to an existing class without modifying the class's source code.
- Figure 10.25 uses extension methods to add functionality to class `Time` (from Section 10.17).

`TimeExtensionsTest.cs`

(1 of 3)

```
1  // Fig10.25: TimeExtensionsTest.cs
2  // Demonstrating extension methods.
3  using System;
4
5  class TimeExtensionsTest
6  {
7     static void Main( string[] args )
8     {
9        Time myTime = new Time(); // call Time constructor
10       myTime.SetTime( 11, 34, 15 ); // set the time to 11:34:15
11
```

Fig. 10.25 | Demonstrating extension methods. (Part 1 of 3.)

```
12         // test the DisplayTime extension method
13         Console.Write( "Use the DisplayTime method: " );
14      myTime.DisplayTime();
15
16         // test the AddHours extension method
17      Console.Write( "Add 5 hours to the Time object: " );
18      Time timeAdded = myTime.AddHours( 5 ); // add five hours
19      timeAdded.DisplayTime(); // display the new Time object
20
21         // add hours and display the time in one statement
22      Console.Write( "Add 15 hours to the Time object: " );
23      myTime.AddHours( 15 ).DisplayTime(); // add hours and display time
24
25         // use fully qualified extension-method name to display the time
26      Console.Write( "Use fully qualified extension-method name: " );
27      TimeExtensions.DisplayTime( myTime );
28   } // end Main
29 } // end class TimeExtensionsTest
30
31 // extension-methods class
32 static class TimeExtensions
33 {
34    // display the Time object in console
```

TimeExtensionsTest.cs

(2 of 3)

An extension method is called on an object of the class that it extends as if it were a members of the class. The compiler implicitly passes the object that is used to call the method as the extension method's first argument.

Fig. 10.25 | Demonstrating extension methods. (Part 2 of 3.)

```
35    public static void DisplayTime( this Time aTime )
36    {
37       Console.WriteLine( aTime.ToString() );
38    } //  end method DisplayTime
39
40     //  add the specified  number of hours to the time
41     //  and return  a new Time object
42    public static Time AddHours( this Time aTime, int hours )
43    {
44       Time newTime = new Time(); //  create  a new Time object
45       newTime.Minute = aTime.Minute; //  set  the minutes
46       newTime.Second = aTime.Second; //  set  the seconds
47
48          //  add the specified  number of hours to the given time
49       newTime.Hour = ( aTime.Hour + hours ) % 24;
50
51       return newTime; //  return  the new Time object
52    } //  end method AddHours
53 } //  end class  TimeExtensions
```

```
Use the DisplayTime method: 11:34:15 AM
Add 5 hours to the Time object:  4:34:15 PM
Add 15 hours to the Time object:  2:34:15 AM
Use fully  qualified  extension-method name: 11:34:15 AM
```

TimeExtensionsTest.cs

(3 of 3)

The `this` keyword before a method's first parameter notifies the compiler that the method extends an existing class.

Fig. 10.25 | Demonstrating extension methods. (Part 3 of 3.)

10.18 Time Class Case Study: Extension Methods (Cont.)

- The `this` keyword before a method's first parameter notifies the compiler that the method extends an existing class.
- An extension method is called on an object of the class that it extends as if it were a members of the class. The compiler implicitly passes the object that is used to call the method as the extension method's first argument.
- The type of an extension method's first parameter specifies the class that is being extended—extension methods must define at least one parameter.
- Extension methods must be defined as `static` methods in a `static` top-level class.

10.18 Time Class Case Study: Extension Methods (Cont.)

- *IntelliSense* displays extension methods with the extended class's instance methods and identifies them with a distinct icon (Fig. 10.26).

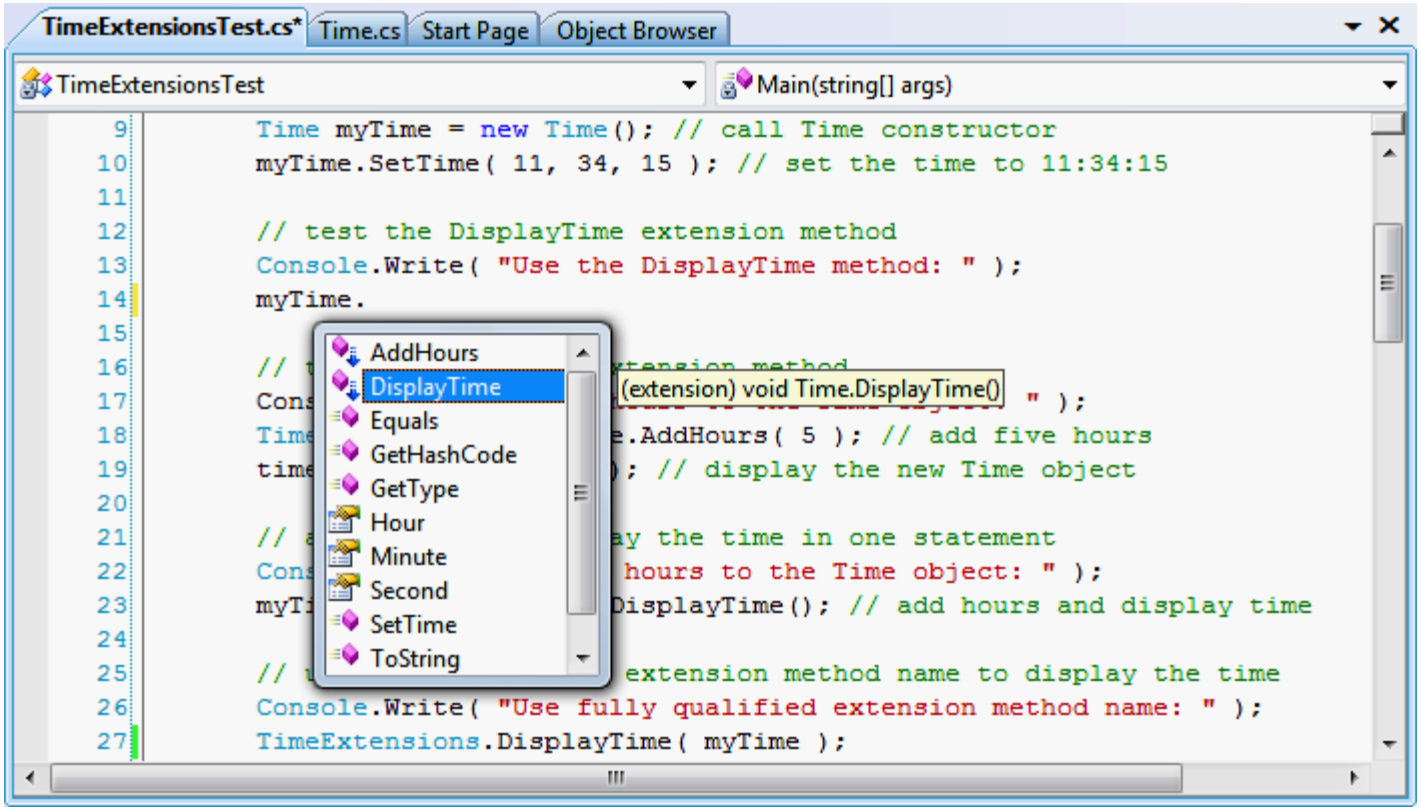

Fig. 10.26 | *IntelliSense* support for extension methods.

10.18 Time Class Case Study: Extension Methods (Cont.)

- Extension methods, as well as instance methods, allow **cascaded method calls**—that is, invoking multiple methods in the same statement.
- Cascaded method calls are performed from left to right.
- When using the fully qualified method name to call an extension method, you must specify an argument for extension method's first parameter. This use of the extension method resembles a call to a `static` method.
- If the type being extended defines an instance method with the same name as your extension method and a compatible signature, the instance method will shadow the extension method.

- A **delegate** is an object that holds a reference to a method.
- Delegates allow you to treat methods as data—via delegates, you can assign methods to variables, and pass methods to and from other methods.
- You can also call methods through variables of delegate types.
- A delegate type is declared by preceeding a method header with keyword `delegate` (placed after any access specifiers, such as `public` or `private`).
- Figure 10.27 uses delegates to customize the functionality of a method that filters an `int` array.

Example Using a Delegate

See Figure 10.27

Line 9: Define a delegate type named NumberPredicate. This variable can store a reference to any method that takes an int argument and returns a bool.

Line 16: Create a particular instance of the NumberPredicate delegate type

Line 20: Use the delegate to call the method that it references

Line 23: Pass the delegate as a parameter to a function

10.20 Lambda Expressions

- A lambda expression begins with a parameter list, which is followed by the => **lambda operator** and an expression that represents the body of the function.
- The value produced by the expression is implicitly returned by the lambda expression.
- The return type can be inferred from the return value or, in some cases, from the delegate's return type.
- A delegate can hold a reference to a lambda expression whose signature is compatible with the delegate type.
- Lambda expressions are often used as arguments to methods with parameters of delegate types, rather than defining and referencing a separate method.

10.20 Lambda Expressions (Cont.)

- A lambda expression can be called via the variable that references it.
- A lambda expression's input parameter `number` can be explicitly typed.
- Lambda expressions that have an expression to the right of the lambda operator are called **expression lambdas**.
- **Statement lambdas** contain a statement block—a set of statements enclosed in braces (`{}`)—to the right of the lambda operator.
- Lambda expressions can help reduce the size of your code and the complexity of working with delegates.
- Lambda expressions are particularly powerful when combined with the `where` clause in LINQ queries.

- **Lambda expressions** allow you to define simple, **anonymous functions**.
- Figure 10.28 uses lambda expressions to reimplement the previous example that introduced delegates.

Lambdas.cs

(1 of 4)

```
1  // Fig10.28: Lambdas.cs
2  // Using lambda expressions.
3  using System;
4  using System.Collections.Generic;
5
6  class Lambdas
7  {
8     // delegate for a function that receives an int and returns a bool
9     public delegate bool NumberPredicate( int number );
10
11    static void Main( string[] args )
12    {
13       int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
14
15       // create an instance of the NumberPredicate delegate type using an
16       // implicit lambda expression
17       NumberPredicate evenPredicate = number => ( number % 2 == 0 );
```

A lambda expression begins with a parameter list, which is followed by the => **lambda operator** and an expression that represents the body of the function.

Fig. 10.28 | Using lambda expressions. (Part 1 of 4.)

Lambdas.cs

(2 of 4)

```
18
19        // call a lambda expression through a variable
20        Console.WriteLine( "Use a lambda-expression variable: {0}",
21           evenPredicate( 4 ) );
22
23        // filter the even numbers using a lambda expression
24        List< int > evenNumbers = FilterArray( numbers, evenPredicate );
25
26        // display the result
27        DisplayList( "Use a lambda expression to filter even numbers: ",
28           evenNumbers );
29
30        // filter the odd numbers using an explicitly typed lambda
31        // expression
32        List< int > oddNumbers = FilterArray( numbers,
33           ( int number ) => ( number % 2 == 1 ) );
34
35        // display the result
36        DisplayList( "Use a lambda expression to filter odd numbers: ",
37           oddNumbers );
38
```

A lambda expression can be called via the variable that references it.

A lambda expression's input parameter `number` can be explicitly typed.

Fig. 10.28 | Using lambda expressions. (Part 2 of 4.)

```
39      // filter numbers greater than 5 using an implicit lambda statement
40      List< int > numbersOver5 = FilterArray( numbers,
41         number => { return number > 5; } );
42
43      // display the result
44      DisplayList( "Use a lambda expression to filter numbers over 5: "
45         numbersOver5 );
46   } // end Main
47
48   // select an array's elements that satisfy the predicate
49   private static List< int > FilterArray( int[] intArray,
50      NumberPredicate predicate )
51   {
52      // hold the selected elements
53      List< int > result = new List< int >();
54
55      // iterate over each element in the array
56      foreach ( int item in intArray )
57      {
58         // if the element satisfies the predicate
59         if ( predicate( item ) )
```

Lambdas.cs

(3 of 4)

Statement lambdas contain a statement block—a set of statements enclosed in braces ({})—to the right of the lambda operator.

Fig. 10.28 | Using lambda expressions. (Part 3 of 4.)

```
66 60  // display the elements of a List
67     private static void DisplayList( string description, List< int > list )
68     {
69        Console.Write( description ); // display the output's description
70
71        // iterate over each element in the List
72        foreach ( int item in list )
73           Console.Write( "{0} ", item ); // print item followed by a space
74
75        Console.WriteLine(); // add a new line
76     } // end method DisplayList
77  } //  end class Lambdas
```

Lambdas.cs

(4 of 4)

```
Use a lambda-expression variable: True
Use a lambda expression to filter even numbers: 2 4 6 8 10
Use a lambda expression to filter odd numbers: 1 3 5 7 9
Use a lambda expression to filter numbers over 5: 6 7 8 9 10
```

Fig. 10.28 | Using lambda expressions. (Part 4 of 4.)

10.21 Anonymous Types

- An anonymous type declaration begins with the keyword `new` followed by a member-initializer list in braces (`{}`).
- The compiler generates a new class definition that contains the properties specified in the member-initializer list.
- All properties of an anonymous type are `public` and immutable.
- Anonymous type properties are read-only—you cannot modify a property's value once the object is created.
- Each property's type is inferred from the values assigned to it.
- Because they are anonymous, you must use implicitly typed local variables to reference objects of anonymous types.

10.21 Anonymous Types (Cont.)

- The compiler defines the `ToString` method that returns a `string` in curly braces containing a comma-separated list of *Property-Name* = *value* pairs.
- Two anonymous objects that specify the same property names and types, in the same order, use the same anonymous class definition and are considered to be of the same type.
- The anonymous type's `Equals` method compares the properties of two anonymous objects.

Anonymous Types in LINQ

- Anonymous types are frequently used in LINQ queries to select specific properties from the items being queried.

- **Anonymous types** allow you to create simple classes used to store data without writing a class definition.
- Anonymous type declarations—known formally as **anonymous object-creation expressions**—are demonstrated in Fig. 10.29.

AnonymousTypes.cs

(1 of 3)

```
1  // Fig10.29: AnonymousTypes.cs
2  // Using anonymous types.
3  using System;
4
5  class AnonymousTypes
6  {
7     static void Main( string[] args )
8     {
9        // create a "person" object using an anonymous type
10       var bob = new { Name = "Bob Smith", Age = 37 };
11
12       // display Bob's information
13       Console.WriteLine( "Bob: " + bob.ToString() );
14
```

An anonymous type declaration begins with the keyword `new` followed by a member-initializer list in braces ({}).

Fig. 10.29 | Using anonymous types. (Part 1 of 4.)

AnonymousTypes.cs

(2 of 3)

```
15    // create another "person" object using the same anonymous type
16    var steve = new { Name = "Steve Jones", Age = 26 };
17
18    // display Steve's information
19    Console.WriteLine( "Steve: " + steve.ToString() );
20
21    // determine if objects of the same anonymous type are equal
22    Console.WriteLine( "\nBob and Steve are {0}",
23       ( bob.Equals( steve ) ? "equal" : "not equal" ) );
24
25    // create a "person" object using an anonymous type
26    var bob2 = new { Name = "Bob Smith", Age = 37 };
27
28    // display Bob's information
29    Console.WriteLine( "\nBob2: " + bob2.ToString() );
30
31    // determine whether objects of the same anonymous type are equal
32    Console.WriteLine( "\nBob and Bob2 are {0}\n",
```

Because they are anonymous, you must use implicitly typed local variables to reference objects of anonymous types.

The anonymous type's `Equals` method compares the properties of two anonymous objects.

Fig. 10.29 | Using anonymous types. (Part 2 of 3.)

AnonymousTypes.cs

(3 of 3)

```
33          ( bob.Equals( bob2 ) ? "equal" : "not equal" ) );
34    } // end Main
35 } // end class AnonymousTypes
```

```
Bob: { Name = Bob Smith, Age = 37 }
Steve: { Name = Steve Jones, Age = 26 }

Bob and Steve are not equal
Bob2: { Name = Bob Smith, Age = 37 }
Bob and Bob2 are equal
```

Fig. 10.29 | Using anonymous types. (Part 3 of 3.)

11

Object-Oriented Programming: Inheritance

11.1 Introduction

- **Inheritance** allows a new class to absorb an existing class's members.
- A derived class normally adds its own fields and methods to represent a more specialized group of objects.
- Inheritance saves time by reusing proven and debugged high-quality software.

11.1 Introduction (Cont.)

- The **direct base class** is the base class which the derived class explicitly inherits.
- An **indirect base class** is any class above the direct base class in the **class hierarchy**.
- The class hierarchy begins with class `object`.

11.1 Introduction (Cont.)

- The *is-a* **relationship** represents inheritance.
- For example, a car *is a* vehicle, and a truck *is a* vehicle.
- New classes can inherit from thousands of pre-built classes in **class libraries**.

11.2 Base Classes and Derived Classes

- Figure 11.1 lists several simple examples of base classes and derived classes.
- Note that base classes are "more general," and derived classes are "more specific."

Base class	Derived classes
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff, HourlyWorker, CommissionWorker
BankAccount	CheckingAccount, SavingsAccount

Fig. 11.1 | Inheritance examples.

11.2 Base Classes and Derived Classes (Cont.)

- Now consider the `Shape` inheritance hierarchy in Fig. 11.3.
- We can follow the arrows to identify several *is-a* relationships.

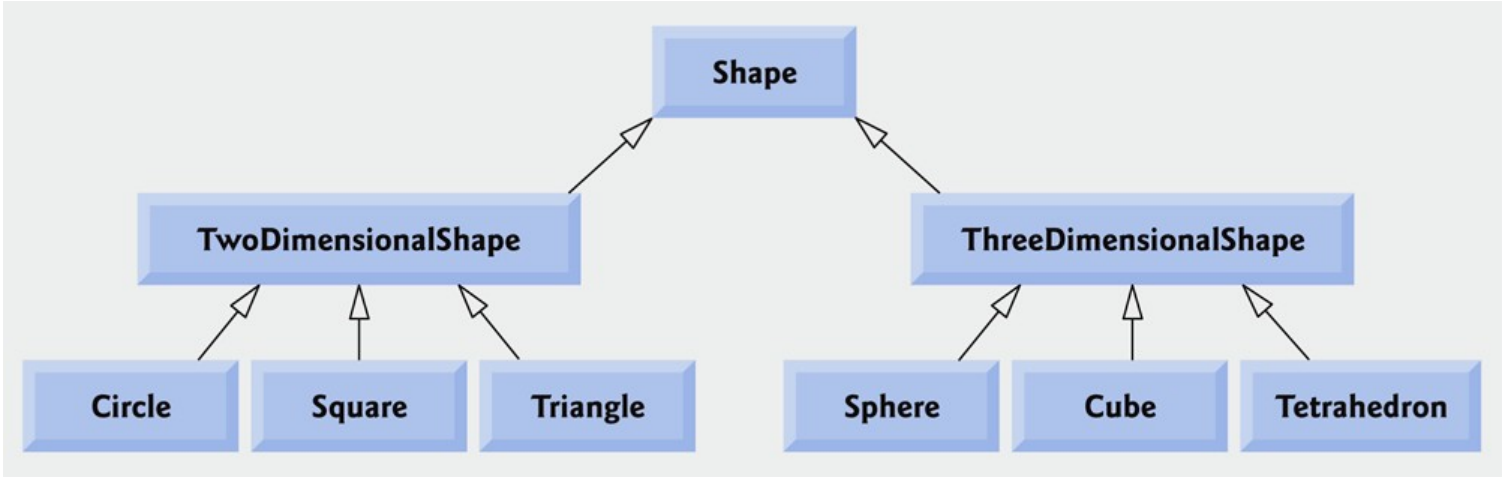

Fig. 11.3 | UML class diagram showing an inheritance hierarchy for Shapes.

11.2 Base Classes and Derived Classes (Cont.)

- Objects of all classes that extend a common base class can be treated as objects of that base class.
- However, base-class objects cannot be treated as objects of their derived classes.
- When a derived class needs a customized version of an inherited method, the derived class can **override** the base-class method.

11.3 protected Members

- A base class's `private` members are inherited by derived classes, but are not directly accessible by derived-class methods and properties.
- A base class's `protected` members can be accessed by members of that base class *and* by members of its derived classes.
- A base class's `protected internal` members can be accessed by members of a base class, the derived classes *and* by any class in the same assembly.

11.4 Relationship between Base Classes and Derived Classes (Cont.)

- A colon (`:`) followed a class name at the end of the class declaration header indicates that the class extends the class to the right of the colon.
- Every C# class directly or indirectly inherits `object`'s methods.
- If a class does not specify that it inherits from another class, it implicitly inherits from `object`.

11.4 Relationship between Base Classes and Derived Classes (Cont.)

- Declaring instance variables as `private` and providing `public` properties to manipulate and validate them helps enforce good software engineering.
- Constructors are not inherited.
- Either explicitly or implicitly, a call to the base-class constructor is made.
- Class `object`'s default (empty) constructor does nothing.
- Note that even if a class does not have constructors, the default constructor will call the base class's default or parameterless constructor.

11.4 Relationship between Base Classes and Derived Classes (Cont.)

- Method `ToString` is special—it is one of the methods that every class inherits directly or indirectly from class `object`.
- Method `ToString` returns a `string` representing an object.
- Class `object`'s `ToString` method is primarily a placeholder that typically should be overridden by a derived class.
- To override a base-class method, a derived class must declare a method with keyword **override**.
- The method must have the same signature (method name, number of parameters and parameter types) and return type as the base-class method.

Outline

- We now declare and test a separate class `BasePlusCommissionEmployee` (Fig. 11.6),

BasePlus Commission Employee.cs

(1 of 6)

```
1  // Fig11.6: BasePlusCommissionEmployee.cs
2  // BasePlusCommissionEmployee class represents an employee that receives
3  // a base salary in addition to a commission.
4  public class BasePlusCommissionEmployee
5  {
6     private string firstName;
7     private string lastName;
8     private string socialSecurityNumber;
9     private decimal grossSales; // gross weekly sales
10    private decimal commissionRate; // commission percentage
11    private decimal baseSalary; // base salary per week
12
13    // six-parameter constructor
14    public BasePlusCommissionEmployee( string first, string last,
15       string ssn, decimal sales, decimal rate, decimal salary )
16    {
```

Fig. 11.6 | `BasePlusCommissionEmployee` class represents an employee that receives a base salary in addition to a commission. (Part 1 of 6.)

Outline

```
17      // implicit call to object constructor occurs here
18         firstName = first;
19         lastName = last;
20         socialSecurityNumber = ssn;
21         GrossSales = sales; // validate gross sales via property
22         CommissionRate = rate; // validate commission rate via property
23      BaseSalary = salary; // validate base salary via property
24   } // end six-parameter BasePlusCommissionEmployee constructor
25
26   // read-only property that gets
27   // base-salaried commission employee's first name
28   public string FirstName
29   {
30      get
31      {
32         return firstName;
33      } // end get
34   } // end property FirstName
35
```

BasePlus Commission Employee.cs

(2 of 6)

Fig. 11.6 | `BasePlusCommissionEmployee` class represents an employee that receives a base salary in addition to a commission. (Part 2 of 6.)

Outline

BasePlus Commission Employee.cs

(3 of 6)

```
36    // read-only property that gets
37    // base-salaried commission employee's last name
38    public string LastName
39    {
40       get
41       {
42          return lastName;
43       } // end get
44    } // end property LastName
45
46    // read-only property that gets
47    // base-salaried commission employee's social security number
48    public string SocialSecurityNumber
49    {
50       get
51       {
52          return socialSecurityNumber;
53       } // end get
54    } // end property SocialSecurityNumber
55
56    // property that gets and sets
57    // base-salaried commission employee's gross sales
58    public decimal GrossSales
59    {
60       get
```

Fig. 11.6 | `BasePlusCommissionEmployee` class represents an employee that receives a base salary in addition to a commission. (Part 3 of 6.)

Outline

```
61        {
62          return grossSales;
63       } //  end get
64       set
65       {
66          grossSales = ( value < 0 ) ? 0 : value;
67       } //  end set
68    } //  end property  GrossSales
69
70    //  property  that  gets  and  sets
71    //  base-salaried  commission  employee's  commission  rate
72    public decimal CommissionRate
73    {
74       get
75       {
76          return commissionRate;
77       } //  end get
78       set
79       {
80          commissionRate = ( value > 0 && value < 1 ) ? value : 0;
```

BasePlus Commission Employee.cs

(4 of 6)

Fig. 11.6 | `BasePlusCommissionEmployee` class represents an employee that receives a base salary in addition to a commission. (Part 4 of 6.)

Outline

BasePlus Commission Employee.cs

(5 of 6)

```
81          //end set
82       }// end property CommissionRate
83
84    // property that gets and sets
85    // base-salaried commission employee's base salary
86    public decimal BaseSalary
87    {
88       get
89       {
90          return baseSalary;
91       } // end get
92       set
93       {
```

Fig. 11.6 | `BasePlusCommissionEmployee` class represents an employee that receives a base salary in addition to a commission. (Part 5 of 6.)

Outline

```
94          baseSalary = ( value < 0 ) ? 0 : value;
95       } //   end  set
96    } // end property BaseSalary
97
98    // calculate earnings
99    public decimal Earnings()
100   {
101      return BaseSalary + ( CommissionRate * GrossSales );
102   } // end method earnings
103
104   // return string representation of BasePlusCommissionEmployee
105   public override string ToString()
106   {
107      return string.Format(
108         "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}\n{9}: {10:C}",
109         "base-salaried commission employee", FirstName, LastName,
110         "social security number", SocialSecurityNumber,
111         "gross sales", GrossSales, "commission rate", CommissionRate,
112         "base salary", BaseSalary );
113   } // end method ToString
114} //  end class  BasePlusCommissionEmployee
```

BasePlus Commission Employee.cs

(6 of 6)

Fig. 11.6 | `BasePlusCommissionEmployee` class represents an employee that receives a base salary in addition to a commission. (Part 6 of 6.)

- Figure 11.7 tests class `BasePlusCommissionEmployee`.

BasePlusCommissionEmployeeTest.cs

(1 of 3)

```
1  // Fig11.7: BasePlusCommissionEmployeeTest.cs
2  // Testing class BasePlusCommissionEmployee.
3  using System;
4
5  public class BasePlusCommissionEmployeeTest
6  {
7     public static void Main( string[] args )
8     {
9        // instantiate BasePlusCommissionEmployee object
10       BasePlusCommissionEmployee employee =
11          new BasePlusCommissionEmployee( "Bob", "Lewis",
12          "333-33-3333", 5000.00M, .04M, 300.00M );
13
14       // display base-salaried commission-employee data
15       Console.WriteLine(
16          "Employee information obtained by properties and methods: \n" );
17       Console.WriteLine( "First name is {0}", employee.FirstName );
18       Console.WriteLine( "Last name is {0}", employee.LastName );
```

Fig. 11.7 | Testing class `BasePlusCommissionEmployee`. (Part 1 of 3.)

Outline

BasePlusCommissionEmployeeTest.cs

(2 of 3)

```
19          Console.WriteLine( "Social security number is {0}",
20             employee.SocialSecurityNumber );
21          Console.WriteLine( "Gross sales are {0:C}", employee.GrossSales );
22          Console.WriteLine( "Commission rate is {0:F2}",
23             employee.CommissionRate );
24          Console.WriteLine( "Earnings are {0:C}", employee.Earnings() );
25          Console.WriteLine( "Base salary is {0:C}", employee.BaseSalary );
26
27          employee.BaseSalary = 1000.00M; // set base salary
28
29          Console.WriteLine( "\n{0}:\n\n{1}",
30             "Updated employee information obtained by ToString", employee );
31          Console.WriteLine( "earnings: {0:C}", employee.Earnings() );
32       } // end Main
33    } // end class BasePlusCommissionEmployeeTest
```

Fig. 11.7 | Testing class `BasePlusCommissionEmployee`. (Part 2 of 3.)

Outline

BasePlusCommissionEmployeeTest.cs

(3 of 3)

```
Employee information obtained by properties and methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales are $5,000.00
Commission rate is 0.04
Earnings are $500.00
Base salary is $300.00

Updated employee information obtained by ToString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: $5,000.00
commission rate: 0.04
base salary: $1,000.00
earnings: $1,200.00
```

Fig. 11.7 | Testing class `BasePlusCommissionEmployee`. (Part 3 of 3.)

11.4 Relationship between Base Classes and Derived Classes (Cont.)

- Much of the code for `BasePlusCommissionEmployee` is similar to the code for `CommissionEmployee`.

Error-Prevention Tip 11.1

Copying and pasting code from one class to another can spread errors across multiple source-code files. Use inheritance rather than the "copy-and-paste" approach.

Software Engineering Observation 11.4

With inheritance, the common members of all the classes in the hierarchy are declared in a base class. When changes are required for these common features, you need to make the changes only in the base class—derived classes then inherit the changes.

Outline

- Now we declare class `BasePlusCommissionEmployee` (Fig. 11.8), which extends class `CommissionEmployee` (Fig. 11.4).

BasePlusCommissionEmployee.cs

(1 of 3)

```
1  // Fig11.8: BasePlusCommissionEmployee.cs
2  // BasePlusCommissionEmployee inherits from class CommissionEmployee.
3  public class BasePlusCommissionEmployee : CommissionEmployee
4  {
5     private decimal baseSalary; // base salary per week
6
7     // six-parameter derived-class constructor
8     // with call to base class CommissionEmployee constructor
9     public BasePlusCommissionEmployee( string first, string last,
10       string ssn, decimal sales, decimal rate, decimal salary )
11       : base( first, last, ssn, sales, rate )
12    {
13       BaseSalary = salary; // validate base salary via property
14    } // end six-parameter BasePlusCommissionEmployee constructor
15
```

Class `BasePlusCommissionEmployee` has an additional instance variable `baseSalary`.

Invoke the `CommissionEmployee`'s five-parameter constructor using a constructor initializer.

Fig. 11.8 | `BasePlusCommissionEmployee` inherits from class `CommissionEmployee`. (Part 1 of 3.)

Outline

BasePlusCommissionEmployee.cs

(2 of 3)

```
16   // property that gets and sets
17   // base-salaried commission employee's base salary
18   public decimal BaseSalary
19   {
20      get
21      {
22         return baseSalary;
23      } // end get
24      set
25      {
26         baseSalary = ( value < 0 ) ? 0 : value;
27      } // end set
28   } // end property BaseSalary
29
30   // calculate earnings
31   public override decimal Earnings()
32   {
```

Fig. 11.8 | BasePlusCommissionEmployee inherits from class CommissionEmployee. (Part 2 of 3.)

Outline

```
33      // not allowed: commissionRate and grossSales private in base class
34      return baseSalary + ( commissionRate * grossSales );
35   } // end method Earnings
36
37   // return string representation of BasePlusCommissionEmployee
38   public override string ToString()
39   {
40      // not allowed: attempts to access private base-class members
41      return string.Format(
42         "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}\n{9}: {10:C}"
43         "base-salaried commission employee", firstName, lastName,
44         "social security number", socialSecurityNumber,
45         "gross sales", grossSales, "commission rate", commissionRate,
46         "base salary", baseSalary );
47   } // end method ToString
48 } //  end class BasePlusCommissionEmployee
```

BasePlusCommissionEmployee.cs

(3 of 3)

Error List

1 Error | 0 Warnings | 0 Messages

	Description	File	Line	Column	Project
1	'BasePlusCommissionEmployee.Earnings()': cannot override inherited member 'CommissionEmployee.Earnings()' because it is not marked virtual, abstract, or override	BasePlusCommissionE	31	28	BasePlusCommissionEmployee

Fig. 11.8 | `BasePlusCommissionEmployee` inherits from class `CommissionEmployee`. (Part 3 of 3.)

11.4 Relationship between Base Classes and Derived Classes (Cont.)

- A `BasePlusCommissionEmployee` object *is a* `CommissionEmployee`.
- A constructor initializer with keyword `base` invokes the base-class constructor.

Common Programming Error 11.2

A compilation error occurs if a derived-class constructor calls one of its base-class constructors with arguments that do not match the number and types of parameters specified in one of the base-class constructor declarations.

11.4 Relationship between Base Classes and Derived Classes (Cont.)

- The `virtual` and `abstract` keywords indicate that a base-class method can be overridden in derived classes.
- The `override` modifier declares that a derived-class method overrides a `virtual` or `abstract` base-class method.
- This modifier also implicitly declares the derived-class method `virtual`.
- We need to declare `CommissionEmployee`'s `Earnings` method `virtual`.

11.4 Relationship between Base Classes and Derived Classes (Cont.)

- The compiler generates additional errors because base class `CommissionEmployee`'s instance variables are `private`.
- The errors can be prevented by using the `public` properties inherited from class `CommissionEmployee`.

Error List

7 Errors | 0 Warnings | 0 Messages

	Description	File	Line	Column	Project
1	'CommissionEmployee.commissionRate' is inaccessible due to its protection level	BasePlusCommissionI	34	29	BasePlusCommission Employee
2	'CommissionEmployee.grossSales' is inaccessible due to its protection level	BasePlusCommissionI	34	46	BasePlusCommission Employee
3	'CommissionEmployee.firstName' is inaccessible due to its protection level	BasePlusCommissionI	43	47	BasePlusCommission Employee
4	'CommissionEmployee.lastName' is inaccessible due to its protection level	BasePlusCommissionI	43	58	BasePlusCommission Employee
5	'CommissionEmployee.socialSecurityNumber' is inaccessible due to its protection level	BasePlusCommissionI	44	36	BasePlusCommission Employee
6	'CommissionEmployee.grossSales' is inaccessible due to its protection level	BasePlusCommissionI	45	25	BasePlusCommission Employee
7	'CommissionEmployee.commissionRate' is inaccessible due to its protection level	BasePlusCommissionI	45	56	BasePlusCommission Employee

Fig. 11.9 | Compilation errors generated by `BasePlusCommissionEmployee` (Fig. 11.8) after declaring the `Earnings` method in Fig. 11.4 with keyword `virtual`.

Outline

- Class `CommissionEmployee` (Fig. 11.10) is modified to declare its instance variables as `protected` rather than `private` (Fig. 11.10).

CommissionEmployee.cs

```
1  // Fig11.10: CommissionEmployee.cs
2  // CommissionEmployee with protected instance variables.
3  public class CommissionEmployee
4  {
5     protected string firstName;
6     protected string lastName;
7     protected string socialSecurityNumber;
8     protected decimal grossSales; // gross weekly sales
9     protected decimal commissionRate; // commission percentage
10
11    // five-parameter constructor
12    public CommissionEmployee( string first, string last, string ssn,
13       decimal sales, decimal rate )
14    {
```


Fig. 11.10 | `CommissionEmployee` with `protected` instance variables. (Part 1 of 5.)

Outline

```
15      // implicit call to object constructor occurs here
16         firstName = first;
17         lastName = last;
18         socialSecurityNumber = ssn;
19         GrossSales = sales; // validate gross sales via property
20         CommissionRate = rate; // validate commission rate via property
21      } // end five-parameter CommissionEmployee constructor
22
23      // read-only property that gets commission employee's first name
24      public string FirstName
25      {
26         get
27         {
28            return firstName;
29         } // end get
30      } // end property FirstName
31
32      // read-only property that gets commission employee's last name
33      public string LastName
34      {
35         get
36         {
```

Commission Employee.cs

(2 of 5)

Fig. 11.10 | `CommissionEmployee` with `protected` instance variables. (Part 2 of 5.)

Outline

CommissionEmployee.cs

(3 of 5)

```
37        return lastName;
38     } // end get
39  } // end property LastName
40
41  // read-only property that gets
42  // commission employee's social security number
43  public string SocialSecurityNumber
44  {
45     get
46     {
47        return socialSecurityNumber;
48     } // end get
49  } // end property SocialSecurityNumber
50
51  // property that gets and sets commission employee's gross sales
52  public decimal GrossSales
53  {
54     get
55     {
56        return grossSales;
57     } // end get
58     set
59     {
```

Fig. 11.10 | `CommissionEmployee` with `protected` instance variables. (Part 3 of 5.)

Outline

CommissionEmployee.cs

(4 of 5)

```
60          grossSales = ( value < 0 ) ? 0 : value;
61       } // end set
62    } // end property GrossSales
63
64    // property that gets and sets commission employee's commission rate
65    public decimal CommissionRate
66    {
67       get
68       {
69          return commissionRate;
70       } // end get
71       set
72       {
73          commissionRate = ( value > 0 && value < 1 ) ? value : 0;
74       } // end set
75    } // end property CommissionRate
76
77    // calculate commission employee's pay
78    public virtual decimal Earnings()
79    {
80       return commissionRate * grossSales;
81    } // end method Earnings
82
83    // return string representation of CommissionEmployee object
84    public override string ToString()
85    {
```

Fig. 11.10 | `CommissionEmployee` with `protected` instance variables. (Part 4 of 5.)

Outline

CommissionEmployee.cs

(5 of 5)

```
86     return string.Format(
87        "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}"
88        "commission employee", firstName, lastName,
89        "social security number", socialSecurityNumber,
90        "gross sales", grossSales, "commission rate", commissionRate );
91     } // end method ToString
92  } // end class CommissionEmployee
```

Fig. 11.10 | `CommissionEmployee` with `protected` instance variables. (Part 5 of 5.)

- We also declare the `Earnings` method `virtual` in line 78 so that `BasePlusCommissionEmployee` can override the method.

Outline

- Class `BasePlusCommissionEmployee` (Fig. 11.11) is modified to extend `CommissionEmployee`.
- The instance variables are now `protected` members, so the compiler does not generate errors.

BasePlusCommissionEmployee.cs


```
1  // Fig11.11: BasePlusCommissionEmployee.cs
2  // BasePlusCommissionEmployee inherits from CommissionEmployee and has
3  // access to CommissionEmployee's protected members.
4  public class BasePlusCommissionEmployee : CommissionEmployee2
5  {
6    private decimal baseSalary; // base salary per week
7
8    // six-parameter derived-class constructor
9    // with call to base class CommissionEmployee constructor
10   public BasePlusCommissionEmployee( string first, string last,
11     string ssn, decimal sales, decimal rate, decimal salary )
12     : base( first, last, ssn, sales, rate )
13   {
14     BaseSalary = salary; // validate base salary via property
15   } // end six-parameter BasePlusCommissionEmployee constructor
16
```

`BasePlusCommissionEmployee`'s six-parameter constructor calls class `CommissionEmployee`'s five-parameter constructor with a constructor initializer.

Fig. 11.11 | `BasePlusCommissionEmployee` inherits from `CommissionEmployee` and has access to `CommissionEmployee`'s `protected` members. (Part 1 of 3.)

Outline

BasePlusCommissionEmployee.cs

(2 of 3)

```
17  // property that gets and sets
18  // base-salaried commission employee's base salary
19  public decimal BaseSalary
20  {
21     get
22     {
23        return baseSalary;
24     } // end get
25     set
26     {
27        baseSalary = ( value < 0 ) ? 0 : value;
28     } // end set
29  } // end property BaseSalary
30
31  // calculate earnings
32  public override decimal Earnings()
33  {
34     return baseSalary + ( commissionRate * grossSales );
35  } // end method Earnings
```

Fig. 11.11 | `BasePlusCommissionEmployee` inherits from `CommissionEmployee` and has access to `CommissionEmployee`'s `protected` members. (Part 2 of 3.)

Outline

BasePlusCommissionEmployee.cs

(3 of 3)

```
36
37    // return string representation of BasePlusCommissionEmployee
38    public override string ToString()
39    {
40       return string.Format(
41          "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}\n{9}: {10:C}"
42          "base-salaried commission employ", firstName, lastName,
43          "social security num", socialSecurityNumber,
44          "gross sale", grossSales, "commission rate", commissionRate,
45          "base salary", baseSalary );
46    } // end method ToString
47 } //  end class BasePlusCommissionEmployee
```

Fig. 11.11 | `BasePlusCommissionEmployee` inherits from `CommissionEmployee` and has access to `CommissionEmployee`'s `protected` members. (Part 3 of 3.)

Outline

- Figure 11.12 tests a `BasePlusCommissionEmployee` object.
- While the output is identical, there is less code repetition and overall this is a better implementation.

BasePlusCommissionEmployee.cs

(1 of 3)

```
1  // Fig11.12: BasePlusCommissionEmployee.cs
2  // Testing class BasePlusCommissionEmployee.
3  using System;
4
5  public class BasePlusCommissionEmployee
6  {
7     public static void Main( string[] args )
8     {
9        // instantiate BasePlusCommissionEmployee object
10       BasePlusCommissionEmployee basePlusCommissionEmployee =
11          new BasePlusCommissionEmployee( "Bob", "Lewis",
12          "333-33-3333", 5000.00M, .04M, 300.00M );
13
14       // display base-salaried commission-employee data
15       Console.WriteLine(
16          "Employee information obtained by properties and methods: \n" );
```

Fig. 11.12 | Testing class `BasePlusCommissionEmployee`. (Part 1 of 3.)

Outline

```
17          Console.WriteLine( "First name is {0}",
18             basePlusCommissionEmployee.FirstName );
19          Console.WriteLine( "Last name is {0}",
20             basePlusCommissionEmployee.LastName );
21          Console.WriteLine( "Social security number is {0}",
22             basePlusCommissionEmployee.SocialSecurityNumber );
23          Console.WriteLine( "Gross sales are {0:C}",
24             basePlusCommissionEmployee.GrossSales );
25          Console.WriteLine( "Commission rate is {0:F2}",
26             basePlusCommissionEmployee.CommissionRate );
27          Console.WriteLine( "Earnings are {0:C}",
28             basePlusCommissionEmployee.Earnings() );
29          Console.WriteLine( "Base salary is {0:C}",
30             basePlusCommissionEmployee.BaseSalary );
31
32          basePlusCommissionEmployee.BaseSalary = 1000.00M; // set base salary
33
34          Console.WriteLine( "\n{0}:\n\n{1}",
35             "Updated employee information obtained by ToString",
36             basePlusCommissionEmployee );
37          Console.WriteLine( "earnings: {0:C}",
38             basePlusCommissionEmployee.Earnings() );
39       } // end Main
40    } // end class BasePlusCommissionEmployee
```

BasePlusCommissionEmployee.cs

(2 of 3)

Fig. 11.12 | Testing class `BasePlusCommissionEmployee`. (Part 2 of 3.)

Outline

BasePlusCommissionEmployee.cs

(3 of 3)

```
Employee information obtained by properties and methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales are $5,000.00
Commission rate is 0.04
Earnings are $500.00
Base salary is $300.00

Updated employee information obtained by ToString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: $5,000.00
commission rate: 0.04
base salary: $1,000.00
earnings: $1,200.00
```

Fig. 11.12 | Testing class `BasePlusCommissionEmployee`. (Part 3 of 3.)