11

Object-Oriented Programming: Inheritance

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

- Using `protected` instance variables creates several potential problems.
  - The derived-class object can set an inherited variable's value directly without validity checking.
  - Derived-class methods would need to be written to depend on the base class's data implementation.
- You should be able to change the base-class implementation while still providing the same services to the derived classes.

Software Engineering Observation 11.5

Declaring base-class instance variables private enables the base-class implementation of these instance variables to change without affecting derived-class implementations.

- Class `CommissionEmployee` (Fig. 11.13) is modified to declare `private` instance variables and provide `public` properties.

CommissionEmployee.cs

(1 of 5)

```
// Fig11.13: CommissionEmployee.cs
// CommissionEmployee class represents a commission employee.
public class CommissionEmployee
{
  private string firstName;
  private string lastName;
  private string socialSecurityNumber;
  private decimal grossSales; // gross weekly sales
  private decimal commissionRate; // commission percentage

  // five-parameter constructor
  public CommissionEmployee( string first, string last, string ssn,
    decimal sales, decimal rate )
  {
```

Fig. 11.13 | `CommissionEmployee` class represents a commission employee. (Part 1 of 5.)

Outline

`CommissionEmployee.cs`

(2 of 5)

```
      // implicit call to object constructor occurs here
         firstName = first;
         lastName = last;
         socialSecurityNumber = ssn;
         GrossSales = sales; // validate gross sales via property
         CommissionRate = rate; // validate commission rate via property
      } // end five-parameter CommissionEmployee constructor

   // read-only property that gets commission employee's first name
   public string FirstName
   {
      get
      {
         return firstName;
      } // end get
   } // end property FirstName

   // read-only property that gets commission employee's last name
   public string LastName
   {
      get
      {
         return lastName;
      } // end get
   } // end property LastName

```

Fig. 11.13 | `CommissionEmployee` class represents a commission employee. (Part 2 of 5.)

Outline

```
// read-only property that gets
// commission employee's social security number
public string SocialSecurityNumber
{
  get
  {
    return socialSecurityNumber;
  } // end get
} // end property SocialSecurityNumber

// property that gets and sets commission employee's gross sales
public decimal GrossSales
{
  get
  {
    return grossSales;
  } // end get
  set
  {
    grossSales = ( value < 0 ) ? 0 : value;
  } // end set
} // end property GrossSales
```

Commission Employee.cs

(3 of 5)

Fig. 11.13 | `CommissionEmployee` class represents a commission employee. (Part 3 of 5.)

Outline

`CommissionEmployee.cs`

(4 of 5)

```

   // property that gets and sets commission employee's commission rate
   public decimal CommissionRate
   {
      get
      {
         return commissionRate;
      } // end get
      set
      {
         commissionRate = ( value > 0 && value < 1 ) ? value : 0;
      } // end set
   } // end property CommissionRate

   // calculate commission employee's pay
   public virtual decimal Earnings()
   {
      return CommissionRate * GrossSales;
   } // end method Earnings
```

Use the class's properties to obtain the values of its instance variables.

Fig. 11.13 | `CommissionEmployee` class represents a commission employee. (Part 4 of 5.)

Outline

CommissionEmployee.cs

(5 of 5)

```

 // return string representation of CommissionEmployee object
 public override string ToString()
 {
  return string.Format(
   "{0}: {1} {2}\n{3}: {4}\n{5}: {6:C}\n{7}: {8:F2}",
   "commission employee", FirstName, LastName,
   "social security number", SocialSecurityNumber,
   "gross sales", GrossSales, "commission rate", CommissionRate );
 } // end method ToString
} // end class CommissionEmployee
```

Fig. 11.13 | `CommissionEmployee` class represents a commission employee. (Part 5 of 5.)

Outline

- Class `BasePlusCommissionEmployee` (Fig. 11.14) has several changes to its method implementations.

BasePlusCommissionEmployee.cs

(1 of 3)

```
// Fig11.14: BasePlusCommissionEmployee.cs
// BasePlusCommissionEmployee inherits from CommissionEmployee and has
// access to CommissionEmployee's private data via
// its public properties.
public class BasePlusCommissionEmployee : CommissionEmployee
{
  private decimal baseSalary; // base salary per week

  // six-parameter derived class constructor
  // with call to base class CommissionEmployee constructor
  public BasePlusCommissionEmployee( string first, string last,
    string ssn, decimal sales, decimal rate, decimal salary )
    : base( first, last, ssn, sales, rate )
  {
    BaseSalary = salary; // validate base salary via property
  } // end six-parameter BasePlusCommissionEmployee constructor
```

Fig. 11.14 | `BasePlusCommissionEmployee` inherits from `CommissionEmployee` and has access to `CommissionEmployee's` `private` data via its `public` properties. (Part 1 of 3.)

Outline

BasePlusCommissionEmployee.cs

(2 of 3)

```

   // property that gets and sets
   // base-salaried commission employee's base salary
   public decimal BaseSalary
   {
      get
      {
         return baseSalary;
      } // end get
      set
      {
         baseSalary = ( value < 0 ) ? 0 : value;
      } // end set
   } // end property BaseSalary
```

Fig. 11.14 | `BasePlusCommissionEmployee` inherits from `CommissionEmployee` and has access to `CommissionEmployee`'s `private` data via its `public` properties. (Part 2 of 3.)

Outline

BasePlusCommissionEmployee.cs

(3 of 3)

```
   // calculate earnings
   public override decimal Earnings()
   {
      return BaseSalary + base.Earnings();
   } // end method Earnings

   // return string representation of BasePlusCommissionEmployee
   public override string ToString()
   {
      return string.Format( "base-salaried {0}\nbase salary: {1:C}",
         base.ToString(), BaseSalary );
   } // end method ToString
} // end class BasePlusCommissionEmployee
```

Use `CommissionEmployee`’s `Earnings` method to calculate the commission pay, and add it to the `BaseSalary`.

Fig. 11.14 | `BasePlusCommissionEmployee` inherits from `CommissionEmployee` and has access to `CommissionEmployee's` `private` data via its `public` properties. (Part 3 of 3.)

Outline

BasePlusCommissionEmployeeTest.cs

(1 of 3)

- Figure 11.15 performs the same manipulations on a `BasePlusCommissionEmployee` object.
- By using inheritance and properties, we have efficiently and effectively constructed a well-engineered class.

```
// Fig11.15: BasePlusCommissionEmployeeTest.cs
// Testing class BasePlusCommissionEmployee.
using System;

public class BasePlusCommissionEmployeeTest
{
  public static void Main( string[] args )
  {
      // instantiate BasePlusCommissionEmployee4 object
    BasePlusCommissionEmployee employee =
      new BasePlusCommissionEmployee( "Bob", "Lewis",
      "333-33-3333", 5000.00M, .04M, 300.00M );

      // display base-salaried commission-employee data
    Console.WriteLine(
      "Employee information obtained by properties and methods: \n" );
```

Fig. 11.15 | Testing class BasePlusCommissionEmployee4. (Part 1 of 3.)

Outline

BasePlusCommission EmployeeTest.cs

(2 of 3)

```
      Console.WriteLine( "First name is {0}", employee.FirstName );
      Console.WriteLine( "Last name is {0}", employee.LastName );
      Console.WriteLine( "Social security number is {0}",
         employee.SocialSecurityNumber );
      Console.WriteLine( "Gross sales are {0:C}", employee.GrossSales );
      Console.WriteLine( "Commission rate is {0:F2}",
         employee.CommissionRate );
      Console.WriteLine( "Earnings are {0:C}", employee.Earnings() );
      Console.WriteLine( "Base salary is {0:C}", employee.BaseSalary );

      employee.BaseSalary = 1000.00M; // set base salary

      Console.WriteLine( "\n{0}:\n\n{1}",
         "Updated employee information obtained by ToString", employee );
      Console.WriteLine( "earnings: {0:C}", employee.Earnings() );
   } // end Main
} // end class BasePlusCommissionEmployeeTest
```

Fig. 11.15 | Testing class `BasePlusCommissionEmployee4`. (Part 2 of 3.)

Outline

BasePlusCommission EmployeeTest.cs

(3 of 3)

```
Employee information obtained by properties and methods:

First name is Bob
Last name is Lewis
Social security number is ٣٣٣-٣٣-٣٣٣٣
Gross sales are $٥,٠٠٠.٠٠
Commission rate is ٠.٠٤
Earnings are $٥٠٠.٠٠
Base salary is $٣٠٠.٠٠

Updated employee information obtained by ToString:

base-salaried commission employee: Bob Lewis
social security number: ٣٣٣-٣٣-٣٣٣٣
gross sales: $٥,٠٠٠.٠٠
commission rate: ٠.٠٤
base salary: $١,٠٠٠.٠٠
earnings: $١,٢٠٠.٠٠
```

Fig. 11.15 | Testing class `BasePlusCommissionEmployee4`. (Part 3 of 3.)

11.7 Class `object`

- All classes inherit directly or indirectly from the `object` class.
- Figure 11.19 summarizes `object`'s methods.

Method	Description
`Equals`	This method compares two objects for equality and returns `true` if they are equal and `false` otherwise.
`Finalize`	`Finalize` is called by the garbage collector before it reclaims an object's memory.
`GetHashCode`	The hashcode value returned can be used by a hashtable to determine the location at which to insert the corresponding value.

Fig. 11.19 | `object` methods that are inherited directly or indirectly by all classes. (Part 1 of 2.)

11.7 Class object (Cont.)

Method	Description
`GetType`	Returns an object of class `Type` that contains information about the object's type.
`MemberwiseClone`	This `protected` method makes a copy of the object on which it is called. Instance-variable values in one object are copied into another object of the same type. For reference types, only the references are copied.
`ReferenceEquals`	This `static` method returns `true` if two objects are the same instance or if they are `null` references.
`ToString`	Returns a `string` representation of an object. The default implementation returns the namespace and class name.

Fig. 11.19 | `object` methods that are inherited directly or indirectly by all classes. (Part 2 of 2.)

12

Polymorphism, Interfaces & Operator Overloading

12.1 Introduction

- **Polymorphism** enables you to write applications that process objects that share the same base class in a class hierarchy as if they were all objects of the base class.
- Polymorphism can improve extensibility.

12.2 Polymorphism Examples

- If class `Rectangle` is derived from class `Quadrilateral`, then a `Rectangle` is a more specific version of a `Quadrilateral`.
- Any operation that can be performed on a `Quadrilateral` object can also be performed on a `Rectangle` object.
- These operations also can be performed on other `Quadrilateral`s, such as `Square`s, `Parallelogram`s and `Trapezoid`s.
- The polymorphism occurs when an application invokes a method through a base-class variable.

12.2 Polymorphism Examples (Cont.)

- As another example, suppose we design a video game that manipulates objects of many different types, including objects of classes `Martian`, `Venusian`, `Plutonian`, `SpaceShip` and `La-serBeam`.
- Each class inherits from the common base class `SpaceObject`, which contains method `Draw`.
- A screen-manager application maintains a collection (e.g., a `SpaceObject` array) of references to objects of the various classes.
- To refresh the screen, the screen manager periodically sends each object the same message—namely, `Draw`, while object responds in a unique way.

12.3 Demonstrating Polymorphic Behavior

- In a method call on an object, the type of the *actual referenced object*, not the type of the *reference*, determines which method is called.
- An object of a derived class can be treated as an object of its base class.
- A base-class object is not an object of any of its derived classes.
- The *is-a* relationship applies from a derived class to its direct and indirect base classes, but not vice versa.

12.3 Demonstrating Polymorphic Behavior (Cont.)

- The compiler allows the assignment of a base-class reference to a derived-class variable *if* we explicitly cast the base-class reference to the derived-class type
- If an application needs to perform a derived-class-specific operation on a derived-class object referenced by a base-class variable, the base-class reference must be **downcasted** to a derived-class reference

Outline

PolymorphismTest.cs

(1 of 3)

- The example in Fig. 12.1 demonstrates three ways to use base-class and derived-class variables.

```csharp
// Fig12.1: PolymorphismTest.cs
// Assigning base-class and derived-class references to base-class and
// derived-class variables.
using System;

public class PolymorphismTest
{
  public static void Main( string[] args )
  {
    // assign base-class reference to base-class variable
    CommissionEmployee3 commissionEmployee = new CommissionEmployee3(
      "Sue", "Jones", "222-22-2222", 10000.00M, .06M );

     // assign derived-class reference to derived-class variable
    BasePlusCommissionEmployee4 basePlusCommissionEmployee =
      new BasePlusCommissionEmployee4( "Bob", "Lewis",
      "333-33-3333", 5000.00M, .04M, 300.00M );

     // invoke ToString and Earnings on base-class object
```

Create a new CommissionEmployee3 object and assign its reference to a CommissionEmployee3 variable.

Fig. 12.1 | Assigning base-class and derived-class references to base-class and derived-class variables. (Part 1 of 3.)

Outline

PolymorphismTest.cs

```
      // using base-class variable
      Console.WriteLine( "{0} {1}:\n\n{2}\n{3}:{4:C}\n",
        "Call CommissionEmployee3's ToString with base-class reference",
        "to base class object", commissionEmployee.ToString(),
        "earnings", commissionEmployee.Earnings() );

      // invoke ToString and Earnings on derived-class object
      // using derived-class variable
    Console.WriteLine( "{0} {1}:\n\n{2}\n{3}:{4:C}\n",
      "Call BasePlusCommissionEmployee4's ToString with derived class",
      "reference to derived-class object",
      basePlusCommissionEmployee.ToString(),
      "earnings", basePlusCommissionEmployee.Earnings() );

      // invoke ToString and Earnings on derived-class object
      // using base-class variable
    CommissionEmployee3 commissionEmployee2 =
      basePlusCommissionEmployee;
    Console.WriteLine( "{0} {1}:\n\n{2}\n{3}:{4:C}",
      "Call BasePlusCommissionEmployee4's ToString with base class",
      "reference to derived-class object",
      commissionEmployee2.ToString(), "earnings",
      commissionEmployee2.Earnings() );
  } // end Main
} // end class PolymorphismTest
```

Use the reference `commissionEmployee` to invoke methods `ToString` and `Earnings`. Because `commissionEmployee` refers to a `CommissionEmployee3` object, base class `Commission-Employee3`'s version of the methods are called.

Assign the reference to derived-class object `basePlusCommissionEmployee` to a base-class `Commission-Employee3` variable.

Invoke methods `ToString` and `Earnings` on the base-class `CommisionEmployee3`, but the overriding derived-class's (`BasePlusCommissionEmployee4`'s) version of the methods are actually called.

Fig. 12.1 | Assigning base-class and derived-class references to base-class and derived-class variables. (Part 2 of 3.)

Outline

```
Call CommissionEmployee3's ToString with base-class reference to base-class object:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: $10,000.00
commission rate: 0.06
earnings: $600.00

Call BasePlusCommissionEmployee4's ToString with derived-class reference to derive
class object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: $5,000.00
commission rate: 0.04
base salary: $300.00
earnings: $500.00

Call BasePlusCommissionEmployee4's ToString with base-class reference to derived-
class object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: $5,000.00
commission rate: 0.04
base salary: $300.00
earnings: $500.00
```

`PolymorphismTest.cs`

(3 of 3)

Fig. 12.1 | Assigning base-class and derived-class references to base-class and derived-class variables. (Part 3 of 3.)

12.3 Demonstrating Polymorphic Behavior (Cont.)

- When the compiler encounters a method call made through a variable, it determines if the method can be called by checking the *variable's* class type.
- At execution time, *the type of the object to which the variable refers* determines the actual method to use.

12.4 Abstract Classes and Methods

- **Abstract classes**, or **abstract base classes** cannot be used to instantiate objects.
- Abstract base classes are too general to create real objects—they specify only what is common among derived classes.
- Classes that can be used to instantiate objects are called **concrete classes**.
- Concrete classes provide the specifics that make it reasonable to instantiate objects.

12.4 Abstract Classes and Methods (Cont.)

- An abstract class normally contains one or more **abstract methods**, which have the keyword `abstract` in their declaration.
- A class that contains abstract methods must be declared as an abstract class even if it contains concrete (nonabstract) methods.
- Abstract methods do not provide implementations.

12.5 Case Study: Payroll System Using Polymorphism

- In this section, we create an enhanced employee hierarchy to solve the following problem:

A company pays its employees on a weekly basis. The employees are of four types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, hourly employees are paid by the hour and receive overtime pay for all hours worked in excess of 40 hours, commission employees are paid a percentage of their sales, and salaried-commission employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward salaried-commission employees by adding 10% to their base salaries.

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

- We use `abstract` class `Employee` to represent the general concept of an employee.
- `SalariedEmployee`, `CommissionEmployee` and `HourlyEmployee` extend `Employee`.
- Class `BasePlusCommissionEmployee`—which extends `CommissionEmployee`—represents the last employee type.

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

- The UML class diagram in Fig. 12.2 shows the inheritance hierarchy for our polymorphic employee payroll application.

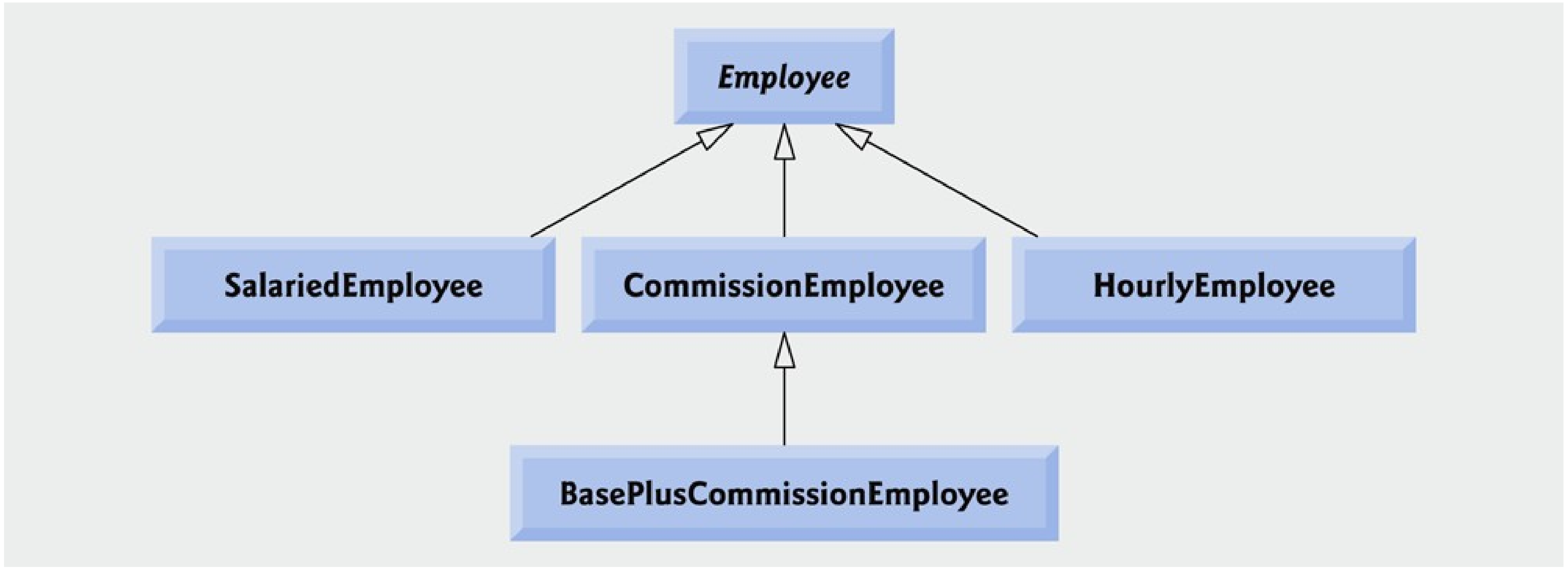


Fig. 12.2 | `Employee` hierarchy UML class diagram

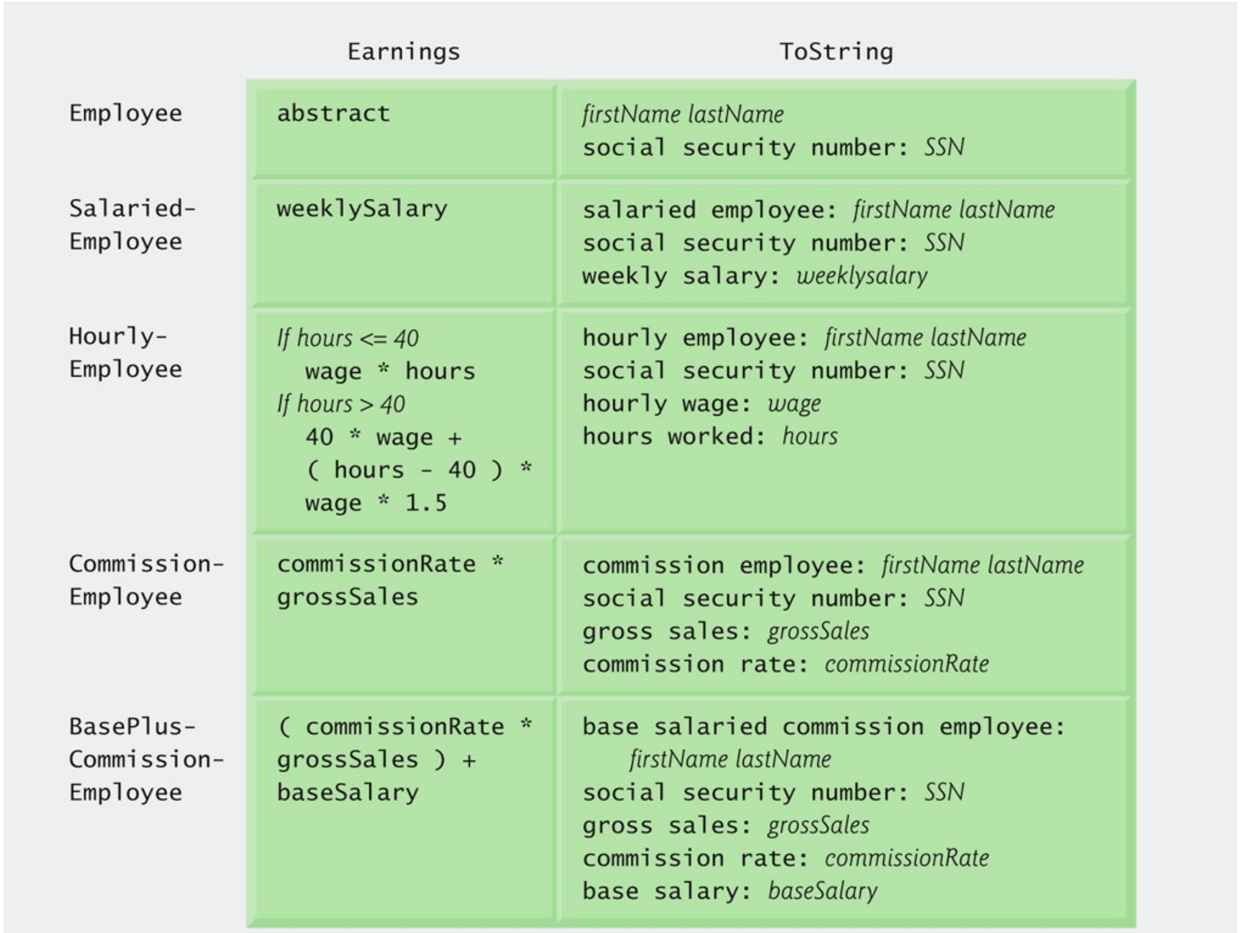
12.5 Case Study: Payroll System Using Polymorphism (Cont.)

12.5.1 Creating Abstract Base Class `Employee`

- Class `Employee` provides methods `Earnings` and `ToString`, in addition to the properties that manipulate `Employee`'s instance variables.
- Each earnings calculation depends on the employee's class, so we declare `Earnings` as `abstract`.
- The application iterates through the array and calls method `Earnings` for each `Employee` object. C# processes these method calls polymorphically.
- Each derived class overrides method `ToString` to create a string representation of an object of that class.

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

- The diagram in Fig. 12.3 shows each of the five classes in the hierarchy down the left side and methods `Earnings` and `ToString` across the top.



	Earnings	ToString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* 40 * wage + (hours - 40) * wage * 1.5	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage* hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales* commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales* commission rate: *commissionRate* base salary: *baseSalary*

Fig. 12.3 | Polymorphic interface for the `Employee` hierarchy classes.

- The `Employee` class’s declaration is shown in Fig. 12.4.

`Employee.cs`

(1 of 2)

```
// Fig12.4: Employee.cs
// Employee abstract base class.
public abstract class Employee
{
   // read-only property that gets employee's first name
   public string FirstName { get; private set; }

   // read-only property that gets employee's last name
   public string LastName { get; private set; }

   // read-only property that gets employee's social security number
   public string SocialSecurityNumber { get; private set; }

   // three-parameter constructor
   public Employee( string first, string last, string ssn )
   {
      FirstName = first;
      LastName = last;
      SocialSecurityNumber = ssn;
   } // end three-parameter Employee constructor
```

Fig. 12.4 | `Employee` abstract base class. (Part 1 of 2.)

Outline

Employee.cs

(1 of 2)

```

   // return string representation of Employee object, using properties
   public override string ToString()
   {
      return string.Format( "{0} {1}\nsocial security number: {2}",
         FirstName, LastName, SocialSecurityNumber );
   } // end method ToString

   // abstract method overridden by derived classes
   public abstract decimal Earnings(); // no implementation here
} // end abstract class Employee
```

The Employee class includes an abstract method Earnings, which must be implemented by concrete derived classes.

Fig. 12.4 | Employee abstract base class. (Part 2 of 2.)

Outline

SalariedEmployee.cs

(1 of 2)

```
// Fig12.5: SalariedEmployee.cs
// SalariedEmployee class that extends Employee.
public class SalariedEmployee : Employee
{
  private decimal weeklySalary;

  // four-parameter constructor
  public SalariedEmployee( string first, string last, string ssn,
    decimal salary ) : base( first, last, ssn )
  {
    WeeklySalary = salary; // validate salary via property
  } // end four-parameter SalariedEmployee constructor

  // property that gets and sets salaried employee's salary
  public decimal WeeklySalary
  {
    get
    {
      return weeklySalary;
    } // end get
```

SalariedEmployee extends Employee.

Using the base class constructor to initialize the private variables not inherited from the base class.

Fig. 12.5 | SalariedEmployee class that extends Employee. (Part 1 of 2.)

SalariedEmployee.cs

(2 of 2)

```
      set
      {
         weeklySalary = ( value >= 0 ) ? value : 0; // validation
      } // end set
   } // end property WeeklySalary

   // calculate earnings; override abstract method Earnings in Employee
   public override decimal Earnings()
   {
      return WeeklySalary;
   } // end method Earnings

   // return string representation of SalariedEmployee object
   public override string ToString()
   {
      return string.Format( "salaried employee: {0}\n{1}: {2:C}",
         base.ToString(), "weekly salary", WeeklySalary );
   } // end method ToString
} // end class SalariedEmployee
```

Method Earnings overrides Employee's abstract method Earnings to provide a concrete implementation that returns the SalariedEmployee's weekly salary.

Method ToString overrides Employee method ToString.

Fig. 12.5 | SalariedEmployee class that extends Employee. (Part 2 of 2.)

Outline

- Class `HourlyEmployee` (Fig. 12.6) also extends class `Employee`.

```
// Fig12.6: HourlyEmployee.cs
// HourlyEmployee class that extends Employee.
public class HourlyEmployee : Employee
{
  private decimal wage; // wage per hour
  private decimal hours; // hours worked for the week

  // five-parameter constructor
  public HourlyEmployee( string first, string last, string ssn,
    decimal hourlyWage, decimal hoursWorked )
    : base( first, last, ssn )
  {
    Wage = hourlyWage; // validate hourly wage via property
    Hours = hoursWorked; // validate hours worked via property
  } // end five-parameter HourlyEmployee constructor

  // property that gets and sets hourly employee's wage
  public decimal Wage
  {
    get
    {
      return wage;
    } // end get
```

`HourlyEmployee.cs`

(1 of 3)

Fig. 12.6 | `HourlyEmployee` class that extends `Employee`. (Part 1 of 3.)

`HourlyEmployee.cs`

(2 of 3)

```
      set
      {
         wage = ( value >= 0 ) ? value : 0; // validation
      } // end set
   } // end property Wage

   // property that gets and sets hourly employee's hours
   public decimal Hours
   {
      get
      {
         return hours;
      } // end get
      set
      {
         hours = ( ( value >= 0 ) && ( value <= 168 ) ) ?
            value : 0; // validation
      } // end set
   } // end property Hours
```

Method `ToString` overrides `Employee` method `ToString`.

The `set` accessor in property `Hours` ensures that `hours` is in the range `0`–`168` (the number of hours in a week).

Fig. 12.6 | `HourlyEmployee` class that extends `Employee`. (Part 2 of 3.)

Outline

`HourlyEmployee.cs`

(3 of 3)

```

   // calculate earnings; override Employee's abstract method Earnings
   public override decimal Earnings()
   {
      if ( Hours <= 40 ) // no overtime
         return Wage * Hours;
      else
         return ( 40 * Wage ) + ( ( Hours - 40 ) * Wage * 1.5M );
   } // end method Earnings

   // return string representation of HourlyEmployee object
   public override string ToString()
   {
      return string.Format(
         "hourly employee: {0}\n{1}: {2:C}; {3}: {4:F2}",
         base.ToString(), "hourly wage", Wage, "hours worked", Hours );
   } // end method ToString
} // end class HourlyEmployee
```

Fig. 12.6 | `HourlyEmployee` class that extends `Employee`. (Part 3 of 3.)

- Class `CommissionEmployee` (Fig. 12.7) extends class `Employee`.

```
// Fig12.7: CommissionEmployee.cs
// CommissionEmployee class that extends Employee.
public class CommissionEmployee : Employee
{
  private decimal grossSales; // gross weekly sales
  private decimal commissionRate; // commission percentage

  // five-parameter constructor
  public CommissionEmployee( string first, string last, string ssn,
    decimal sales, decimal rate ) : base( first, last, ssn )
  {
    GrossSales = sales; // validate gross sales via property
    CommissionRate = rate; // validate commission rate via property
  } // end five-parameter CommissionEmployee constructor

  // property that gets and sets commission employee's commission rate
  public decimal CommissionRate
  {
    get
    {
      return commissionRate;
    } // end get
```

`CommissionEmployee.cs`

(1 of 3)

Fig. 12.7 | `CommissionEmployee` class that extends `Employee`. (Part 1 of 3.)

CommissionEmployee
.cs

(2 of 3)

```
     set
        {
           commissionRate = ( value > 0 && value < 1 ) ?
                 value : 0; // validation
     } // end set
   } // end property CommissionRate

   // property that gets and sets commission employee's gross sales
   public decimal GrossSales
   {
     get
     {
       return grossSales;
     } // end get
     set
     {
       grossSales = ( value >= 0 ) ? value : 0; // validation
     } // end set
   } // end property GrossSales
```

Fig. 12.7 | `CommissionEmployee` class that extends `Employee`. (Part 2 of 3.)

Outline

CommissionEmployee
.cs

(3 of 3)

```
   // calculate earnings; override abstract method Earnings in Employee
   public override decimal Earnings()
   {
      return CommissionRate * GrossSales;
   } // end method Earnings

   // return string representation of CommissionEmployee object
   public override string ToString()
   {
      return string.Format( "{0}: {1}\n{2}: {3:C}\n{4}: {5:F2}",
         "commission employee", base.ToString(),
         "gross sales", GrossSales, "commission rate", CommissionRate );
   } // end method ToString
} // end class CommissionEmployee
```

Calling base-class method `ToString` to obtain the `Employee`-specific information.

Fig. 12.7 | `CommissionEmployee` class that extends `Employee`. (Part 3 of 3.)

Outline

- Class `BasePlusCommissionEmployee` (Fig. 12.8) extends class `Commission-Employee` and therefore is an indirect derived class of class `Employee`.

```
// Fig12.8: BasePlusCommissionEmployee.cs
// BasePlusCommissionEmployee class that extends CommissionEmployee.
public class BasePlusCommissionEmployee : CommissionEmployee
{
  private decimal baseSalary; // base salary per week

  // six-parameter constructor
  public BasePlusCommissionEmployee( string first, string last,
    string ssn, decimal sales, decimal rate, decimal salary )
    : base( first, last, ssn, sales, rate )
  {
    BaseSalary = salary; // validate base salary via property
  } // end six-parameter BasePlusCommissionEmployee constructor

  // property that gets and sets
  // base-salaried commission employee's base salary
  public decimal BaseSalary
  {
    get
    {
      return baseSalary;
    } // end get
```

`BasePlusCommission`
`Employee.cs`

(1 of 2)

Fig. 12.8 | `BasePlusCommissionEmployee` class that extends `CommissionEmployee`. (Part 1 of 2.)

Outline

BasePlusCommissionEmployee.cs

(2 of 2)

```
         set
         {
            baseSalary = ( value >= 0 ) ? value : 0; // validation
         } // end set
      } // end property BaseSalary

      // calculate earnings; override method Earnings in CommissionEmployee
      public override decimal Earnings()
      {
         return BaseSalary + base.Earnings();
      } // end method Earnings

      // return string representation of BasePlusCommissionEmployee object
      public override string ToString()
      {
         return string.Format( "base-salaried {0}; base salary: {1:C}",
            base.ToString(), BaseSalary );
      } // end method ToString
   } // end class BasePlusCommissionEmployee
```

Method `Earnings` calls the base class's `Earnings` method to calculate the commission-based portion of the employee's earnings.

`BasePlusCommissionEmployee`'s `ToString` method creates a string that contains `"base-salaried"`, followed by the `string` obtained by invoking base class `CommissionEmployee`'s `ToString` method (a good example of code reuse) then the base salary.

Fig. 12.8 | `BasePlusCommissionEmployee` class that extends `CommissionEmployee`. (Part 2 of 2.)

- The application in Fig. 12.9 tests our `Employee` hierarchy.

`PayrollSystemTest.cs`

(1 of 6)

```
// Fig12.9: PayrollSystemTest.cs
// Employee hierarchy test application.
using System;

public class PayrollSystemTest
{
   public static void Main( string[] args )
   {
      // create derived-class objects
      SalariedEmployee salariedEmployee =
         new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00M );
      HourlyEmployee hourlyEmployee =
         new HourlyEmployee( "Karen", "Price",
         "222-22-2222", 16.75M, 40.0M );
      CommissionEmployee commissionEmployee =
         new CommissionEmployee( "Sue", "Jones",
         "333-33-3333", 10000.00M, .06M );
      BasePlusCommissionEmployee basePlusCommissionEmployee =
         new BasePlusCommissionEmployee( "Bob", "Lewis",
         "444-44-4444", 5000.00M, .04M, 300.00M );

```

Create objects of each of the four concrete `Employee` derived classes.

Fig. 12.9 | `Employee` hierarchy test application. (Part 1 of 6.)

Outline

PayrollSystemTest.cs

(2 of 6)

```
      Console.WriteLine( "Employees processed individually:\n" );

      Console.WriteLine( "{0}\nearned: {1:C}\n",
        salariedEmployee, salariedEmployee.Earnings() );
      Console.WriteLine( "{0}\nearned: {1:C}\n",
        hourlyEmployee, hourlyEmployee.Earnings() );
      Console.WriteLine( "{0}\nearned: {1:C}\n",
        commissionEmployee, commissionEmployee.Earnings() );
      Console.WriteLine( "{0}\nearned: {1:C}\n",
        basePlusCommissionEmployee,
        basePlusCommissionEmployee.Earnings() );

      // create four-element Employee array
      Employee[] employees = new Employee[ 4 ];

      // initialize array with Employees of derived types
      employees[ 0 ] = salariedEmployee;
      employees[ 1 ] = hourlyEmployee;
      employees[ 2 ] = commissionEmployee;
      employees[ 3 ] = basePlusCommissionEmployee;

```

Each object's `ToString` method is called implicitly.

Fig. 12.9 | Employee hierarchy test application. (Part 2 of 6.)

Outline

PayrollSystemTest
.cs

(2 of 6)

```
         Console.WriteLine( "Employees processed polymorphically:\n" );

      // generically process each element in array employees
      foreach ( var currentEmployee in employees )
      {
         Console.WriteLine( currentEmployee ); // invokes ToString

         // determine whether element is a BasePlusCommissionEmployee
         if ( currentEmployee is BasePlusCommissionEmployee )
         {
            // downcast Employee reference to
            // BasePlusCommissionEmployee reference
            BasePlusCommissionEmployee employee =
               ( BasePlusCommissionEmployee ) currentEmployee;

            employee.BaseSalary *= 1.10M;
            Console.WriteLine(
               "new base salary with 10% increase is: {0:C}",
               employee.BaseSalary );
         } // end if
```

Method calls are resolved at execution time, based on the type of the object referenced by the variable.

The `is` operator is used to determine whether a particular `Employee` object's type is `BasePlusCommissionEmployee`.

Downcasting `currentEmployee` from type `Employee` to type `BasePlusCommissionEmployee`.

Fig. 12.9 | `Employee` hierarchy test application. (Part 3 of 6.)

Outline

```
         Console.WriteLine(
            "earned {0:C}\n", currentEmployee.Earnings() );
      } // end foreach

      // get type name of each object in employees array
      for ( int j = 0; j < employees.Length; j++ )
         Console.WriteLine( "Employee {0} is a {1}", j,
            employees[ j ].GetType() );
   } // end Main
} // end class PayrollSystemTest
```

```
Employees processed individually:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned: $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned: $670.00
```

(continued on next page…)

PayrollSystemTest.cs

(3 of 6)

Method calls are resolved at execution time, based on the type of the object referenced by the variable.

Method `GetType` returns an object of class `Type`, which contains information about the object's type.

Fig. 12.9 | Employee hierarchy test application. (Part 4 of 6.)

Outline

(continued from previous page…)

```
commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00
commission rate: 0.06
earned: $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00
commission rate: 0.04; base salary: $300.00
earned: $500.00


Employees processed polymorphically:

salaried employee: John Smith
social security number: 111-11-1111
```

```
weekly salary: $800.00
earned $800.00
```

(continued on previous page…)

`PayrollSystemTest.cs`

(5 of 6)

Fig. 12.9 | `Employee` hierarchy test application. (Part 5 of 6.)

(continued from previous page…)

```
weekly salary: $800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00
commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00
commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

PayrollSystemTest .cs

(6 of 6)

Fig. 12.9 | Employee hierarchy test application. (Part 6 of 6.)

- Class `BasePlusCommissionEmployee` (Fig. 12.8) extends class `Commission-Employee` and therefore is an indirect derived class of class `Employee`.

BasePlusCommission
Employee.cs

(1 of 2)

```
// Fig12.8: BasePlusCommissionEmployee.cs
// BasePlusCommissionEmployee class that extends CommissionEmployee.
public class BasePlusCommissionEmployee : CommissionEmployee
{
  private decimal baseSalary; // base salary per week

  // six-parameter constructor
  public BasePlusCommissionEmployee( string first, string last,
    string ssn, decimal sales, decimal rate, decimal salary )
    : base( first, last, ssn, sales, rate )
  {
    BaseSalary = salary; // validate base salary via property
  } // end six-parameter BasePlusCommissionEmployee constructor

  // property that gets and sets
  // base-salaried commission employee's base salary
  public decimal BaseSalary
  {
    get
    {
      return baseSalary;
    } // end get
```

Fig. 12.8 | BasePlusCommissionEmployee class that extends CommissionEmployee. (Part 1 of 2.)

Outline

BasePlusCommission
Employee.cs

(2 of 2)

```
      set
      {
         baseSalary = ( value >= 0 ) ? value : 0; // validation
      } // end set
   } // end property BaseSalary

   // calculate earnings; override method Earnings in CommissionEmployee
   public override decimal Earnings()
   {
      return BaseSalary + base.Earnings();
   } // end method Earnings

   // return string representation of BasePlusCommissionEmployee object
   public override string ToString()
   {
      return string.Format( "base-salaried {0}; base salary: {1:C}",
         base.ToString(), BaseSalary );
   } // end method ToString
} // end class BasePlusCommissionEmployee
```

Method `Earnings` calls the base class's `Earnings` method to calculate the commission-based portion of the employee's earnings.

`BasePlusCommissionEmployee`'s `ToString` method creates a string that contains `"base-salaried"`, followed by the `string` obtained by invoking base class `CommissionEmployee`'s `ToString` method (a good example of code reuse) then the base salary.

Fig. 12.8 | `BasePlusCommissionEmployee` class that extends `CommissionEmployee`. (Part 2 of 2.)