Programming Logic and Design *Ninth Edition*

Chapter 11

More Object-Oriented Programming Concepts

Objectives

In this chapter, you will learn about:

- Constructors
- Destructors
- Composition
- Inheritance
- GUI objects
- Exception handling
- The advantages of object-oriented programming

Understanding Constructors

- **Constructor**
  - A method that has the same name as the class
  - Establishes an object
- Constructors fall into two categories:
  - **Default constructor**
    - Requires no arguments
  - **Non-default or parameterized constructor**
    - Requires arguments

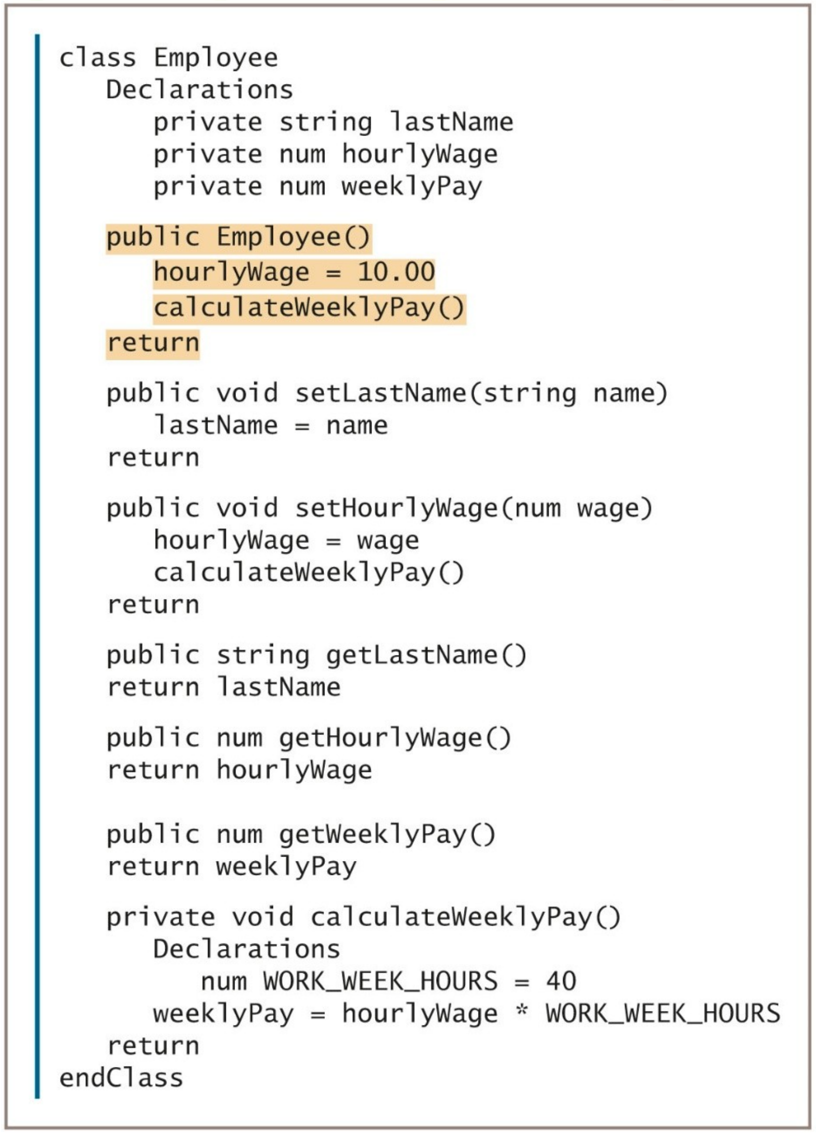
Understanding Constructors
(continued)

- A class can have three types of constructors:
  - **Default constructor** –
    1. automatically-created default constructor exists in a class in which the programmer has not explicitly written any constructors
    2. Programmer-written default constructor can reside in any class and replaces the automatically-created one
  - **Non-default** or **parameterized constructor**
    3. Written by programmer with one of more parameters

Default Constructors

- Default constructor for the `Employee` class
  - Establishes one `Employee` object with the identifier provided
- Declare constructors to be public so that other classes can instantiate objects that belong to the class
- Write any statement you want in a constructor
- Place the constructor anywhere inside the class
  - Often, programmers list the constructor first

Default Constructors (continued -1)



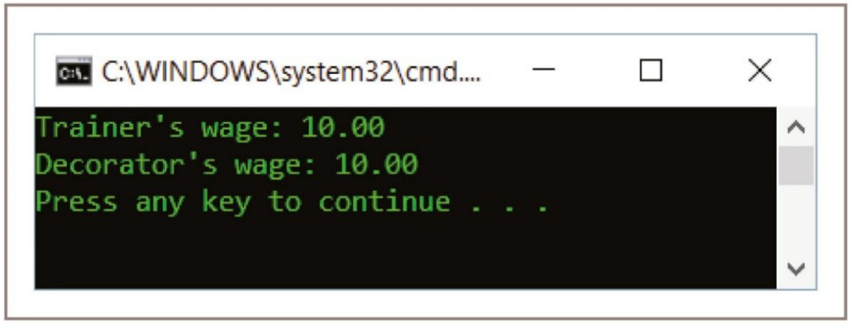
```
class Employee
   Declarations
      private string lastName
      private num hourlyWage
      private num weeklyPay

   public Employee()
      hourlyWage = 10.00
      calculateWeeklyPay()
   return

   public void setLastName(string name)
      lastName = name
   return

   public void setHourlyWage(num wage)
      hourlyWage = wage
      calculateWeeklyPay()
   return

   public string getLastName()
   return lastName

   public num getHourlyWage()
   return hourlyWage

   public num getWeeklyPay()
   return weeklyPay

   private void calculateWeeklyPay()
      Declarations
         num WORK_WEEK_HOURS = 40
      weeklyPay = hourlyWage * WORK_WEEK_HOURS
   return
endClass
```

Figure 11-1 Employee class with a default constructor that sets `hourlyWage` and `weeklyPay`

Default Constructors (continued -2)

```
start
   Declarations
      Employee myPersonalTrainer
      Employee myInteriorDecorator
   output "Trainer's wage: ",
      myPersonalTrainer.getHourlyWage()
   output "Decorator's wage: ",
      myInteriorDecorator.getHourlyWage()
stop
```

Figure 11-2 Program that declares Employee objects using class in Figure 11-1



```
Trainer's wage: 10.00
Decorator's wage: 10.00
Press any key to continue . . .
```

Figure 11-3 Output of program in Figure 11-2

```
public Employee()
   Declarations
      num DEFAULT_WAGE = 10.00
   setHourlyWage(DEFAULT_WAGE)
return
```

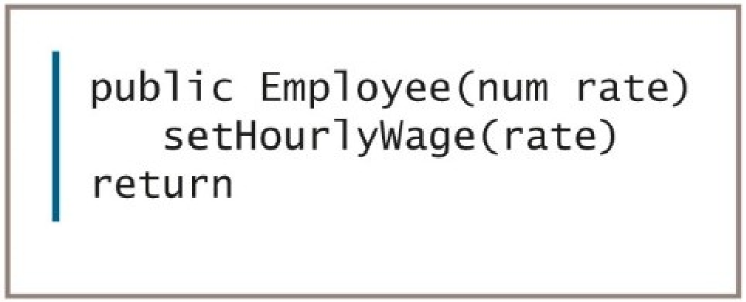
Figure 11-4 Improved version of the Employee class constructor

Nondefault Constructors

- Choose to create `Employee` objects with values that differ for each employee
  - Initialize each `Employee` with a unique `hourlyWage`
- Write constructors that receive arguments
  - `Employee partTimeWorker(8.81)`
  - `Employee partTimeWorker(valueEnteredByUser)`
- When the constructor executes
  - Numeric value within the constructor call is passed to `Employee()`

Nondefault Constructors (continued)

- Once you write a constructor for a class, you no longer receive the automatically written default constructor
- If a class's only constructor requires an argument, you must provide an argument for every object of that class you create



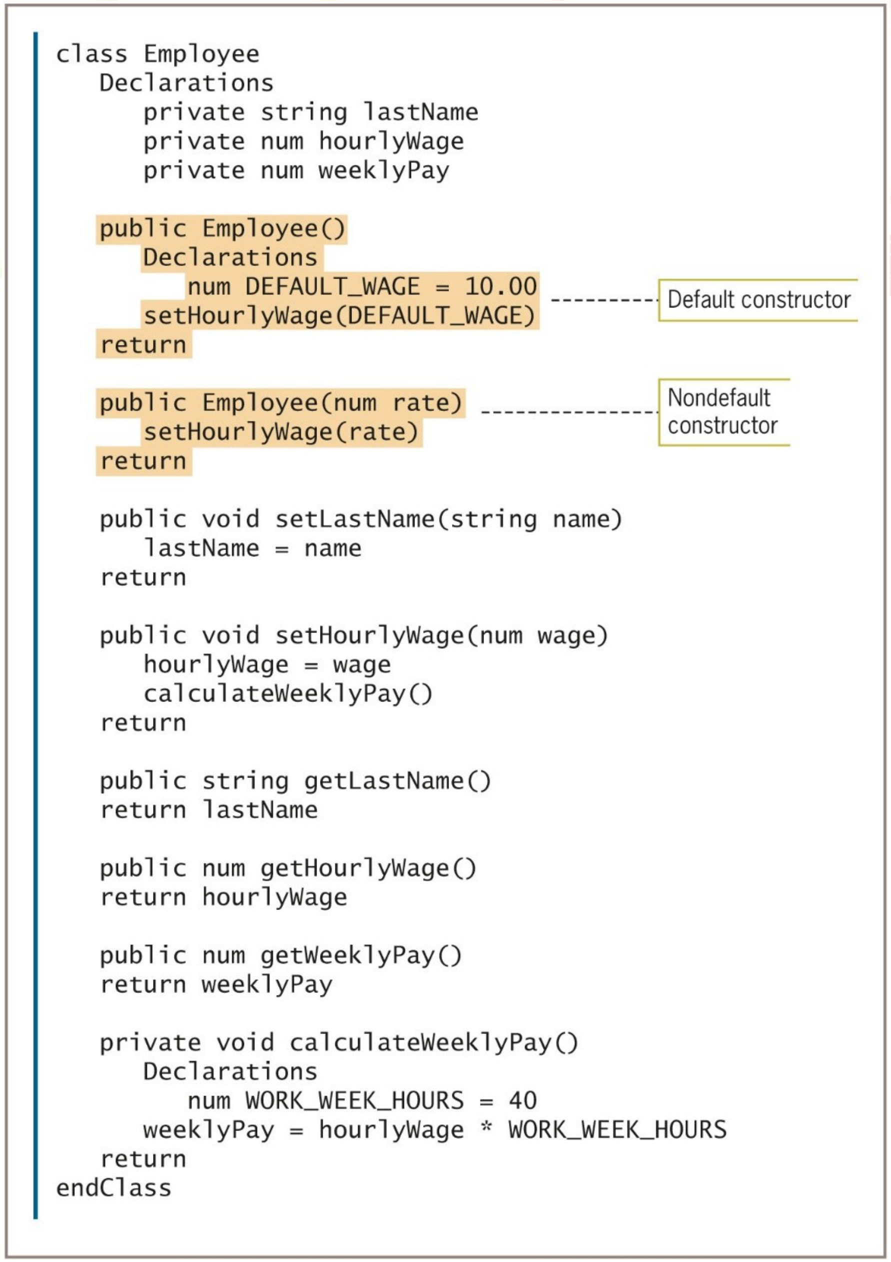
```
public Employee(num rate)
   setHourlyWage(rate)
return
```

Figure 11-5 Employee constructor that accepts a parameter

Overloading Instance Methods and Constructors

- Overload methods
  - Write multiple versions of a method with the same name but different argument lists
- Any method or constructor in a class can be overloaded
- Can provide as many versions as you want

Overloading Instance Methods and Constructors (continued -1)

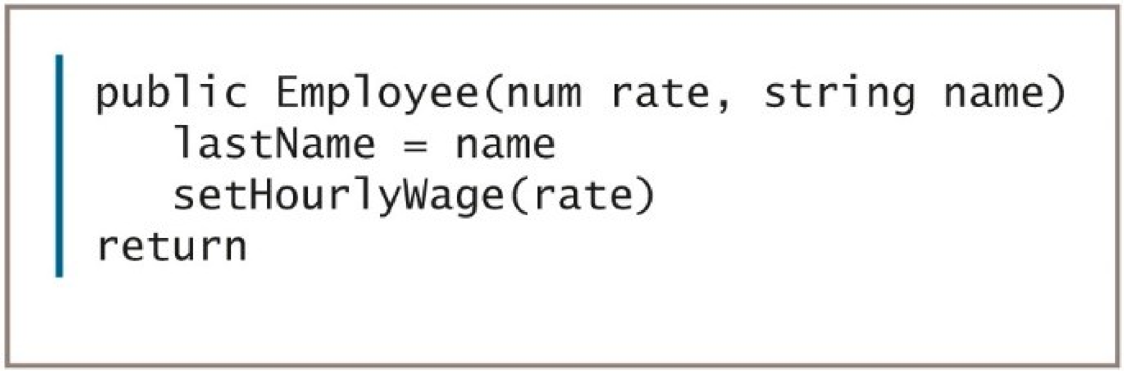


```
class Employee
   Declarations
      private string lastName
      private num hourlyWage
      private num weeklyPay

   public Employee()
      Declarations
         num DEFAULT_WAGE = 10.00
      setHourlyWage(DEFAULT_WAGE)
   return

   public Employee(num rate)
      setHourlyWage(rate)
   return

   public void setLastName(string name)
      lastName = name
   return

   public void setHourlyWage(num wage)
      hourlyWage = wage
      calculateWeeklyPay()
   return

   public string getLastName()
   return lastName

   public num getHourlyWage()
   return hourlyWage

   public num getWeeklyPay()
   return weeklyPay

   private void calculateWeeklyPay()
      Declarations
         num WORK_WEEK_HOURS = 40
      weeklyPay = hourlyWage * WORK_WEEK_HOURS
   return
endClass
```

Figure 11-6 Employee class with overloaded constructors

Overloading Instance Methods and Constructors

(continued -2)



```
public Employee(num rate, string name)
   lastName = name
   setHourlyWage(rate)
return
```

Figure 11-7 A third possible Employee class constructor

Understanding Destructors

- **Destructor**
  - A method that contains the actions you require when an instance of a class is destroyed
- Instance destroyed
  - When the object goes out of scope
- If you do not explicitly create a destructor for a class, one is provided automatically
- Declare a destructor
  - Identifier consists of a tilde (˜) followed by the class name

Understanding Destructors
(continued -1)

- Cannot provide any parameters to a destructor
  - Empty parameter list
- Destructors cannot be overloaded
  - Only one destructor per class
- A destructor has no return type
- Programs never explicitly call a destructor
  - Invoked automatically
- The last object created is the first object destroyed

Understanding Destructors

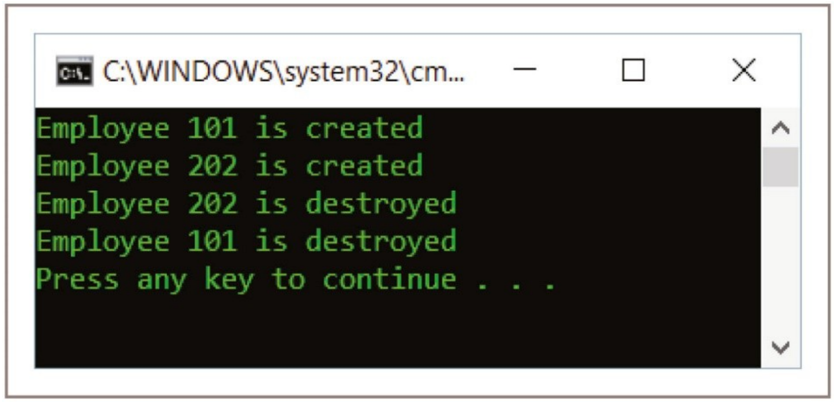
(continued -2)

```
class Employee
   Declarations
      private string idNumber
   public Employee(string empID)
      idNumber = empId
      output "Employee ", idNumber, " is created"
   return
   public ~Employee()
      output "Employee ", idNumber, " is destroyed"
   return
endClass
```

Figure 11-8 Employee class with destructor

```
start
   Declarations
      Employee aWorker("101")
      Employee anotherWorker("202")
stop
```

Figure 11-9 Program that declares two Employee objects



```
Employee 101 is created
Employee 202 is created
Employee 202 is destroyed
Employee 101 is destroyed
Press any key to continue . . .
```

Figure 11-10 Output of program in Figure 11-9

Understanding Composition

- When a class contains objects of another class as data fields, the relationship is called a **whole-part relationship** or **composition**
- Example
  - `Date` that contains a month, day, and year
  - `Employee` class has two `Date` fields to hold `Employee`’s birth date and hire date
- Composition
  - Placing a class object within another class object
  - Called **has-a relationship** because one class “has an” instance of another

Understanding Composition
(continued)

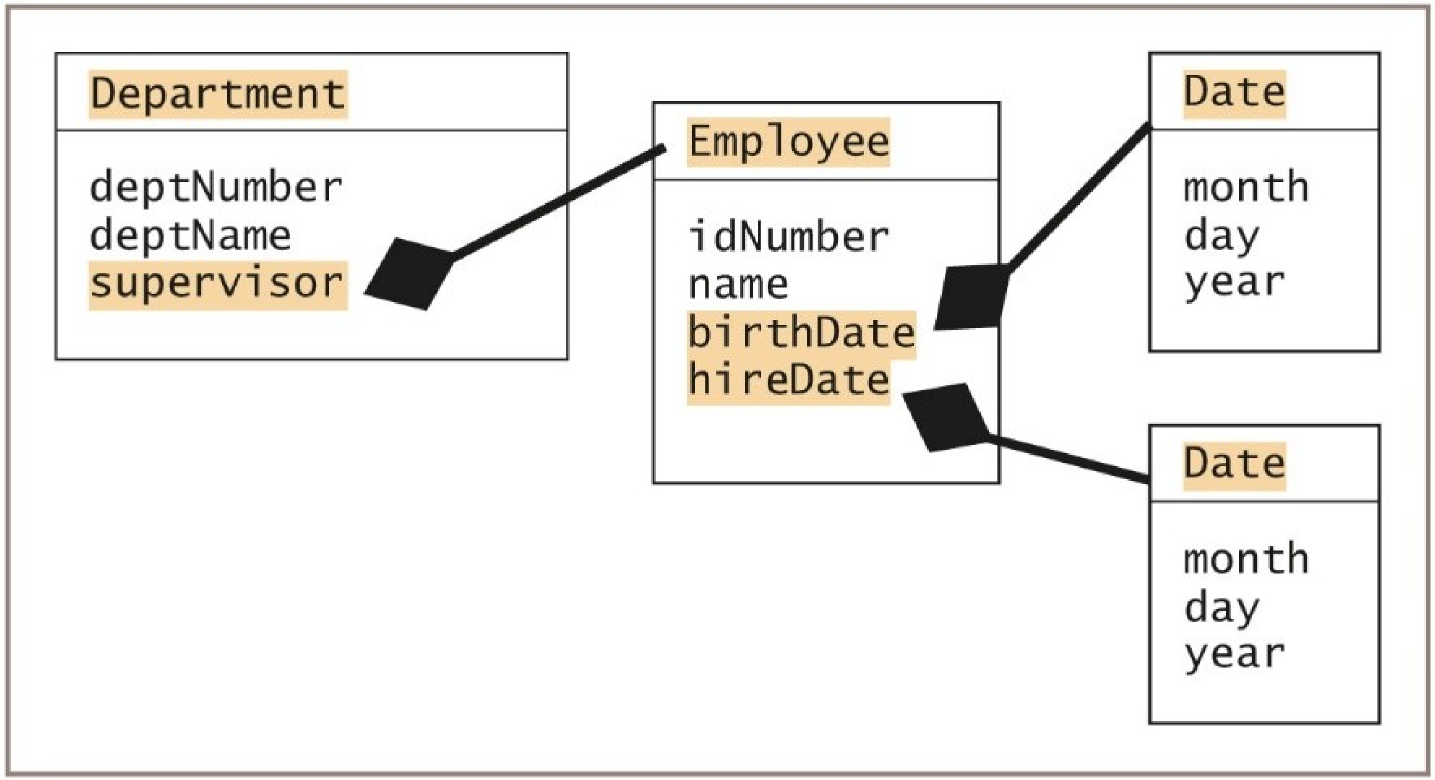


Figure 11-11 Diagram of typical composition relationships

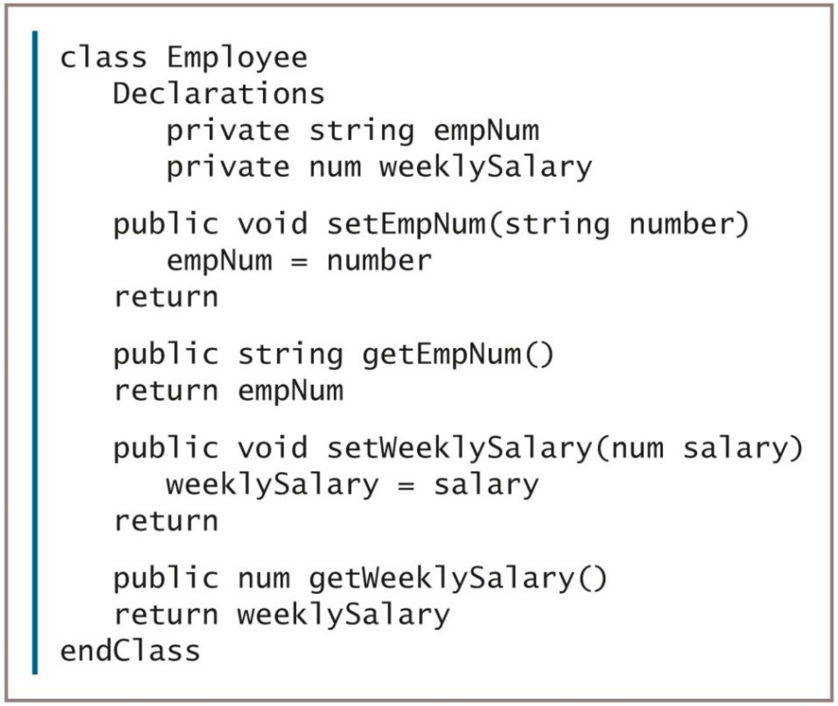
Understanding Inheritance

- Understanding classes helps you organize objects in real life
- Inheritance
  - A principle that enables you to apply your knowledge of a general category to more specific objects
- Reuse the knowledge you gain about general categories and apply it to more specific categories

Understanding Inheritance
(continued -1)

- Create a class by making it inherit from another class
  - Provided with data fields and methods automatically
  - Reuse fields and methods that are already written and tested
- `Employee` class
  - `CommissionEmployee` inherits all the attributes and methods of `Employee`

Understanding Inheritance
(continued -2)



```
class Employee
   Declarations
      private string empNum
      private num weeklySalary

   public void setEmpNum(string number)
      empNum = number
   return

   public string getEmpNum()
   return empNum

   public void setWeeklySalary(num salary)
      weeklySalary = salary
   return

   public num getWeeklySalary()
   return weeklySalary
endClass
```

Figure 11-12 An Employee class

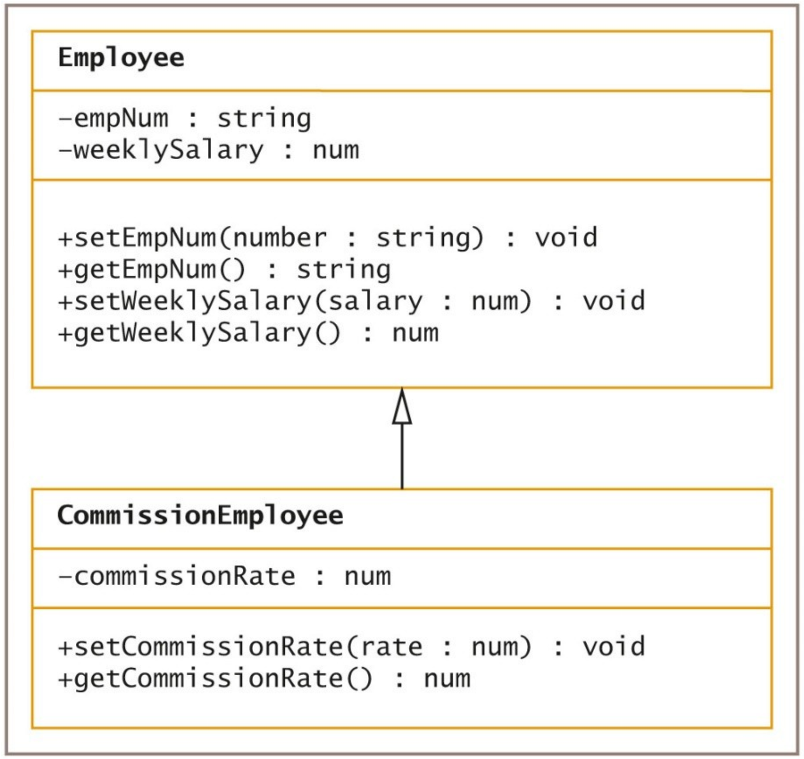
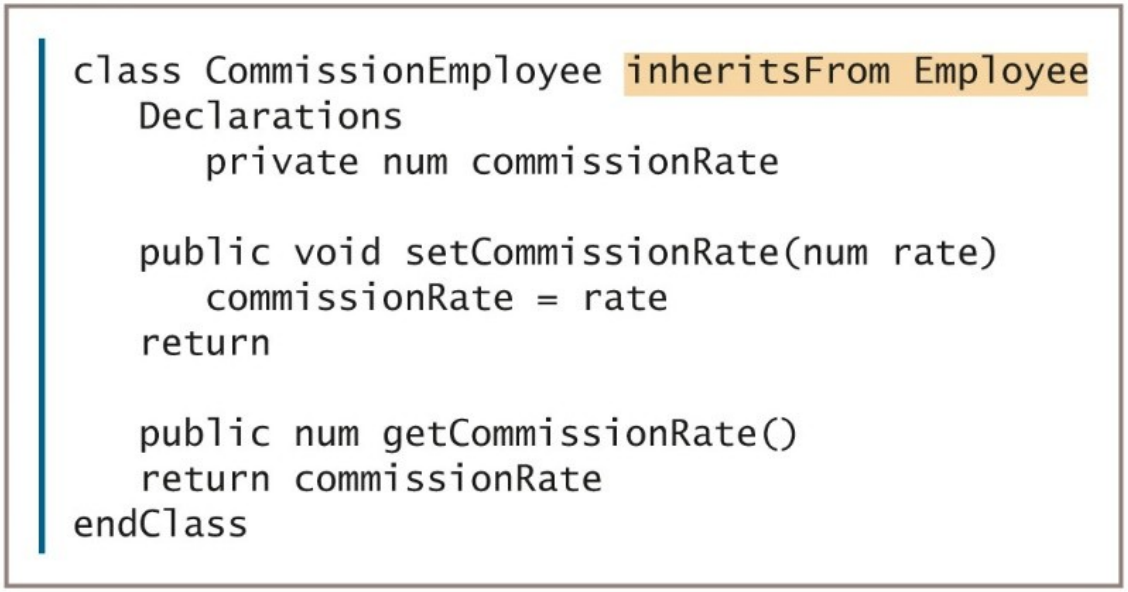

Employee
-empNum : string
-weeklySalary : num
+setEmpNum(number : string) : void
+getEmpNum() : string
+setWeeklySalary(salary : num) : void
+getWeeklySalary() : num
CommissionEmployee
-commissionRate : num
+setCommissionRate(rate : num) : void
+getCommissionRate() : num

Figure 11-13 CommissionEmployee inherits from Employee

Understanding Inheritance

(continued -3)



```
class CommissionEmployee inheritsFrom Employee
   Declarations
      private num commissionRate

   public void setCommissionRate(num rate)
      commissionRate = rate
   return

   public num getCommissionRate()
   return commissionRate
endClass
```

Figure 11-14 CommissionEmployee class

Understanding Inheritance
(continued -4)

- Benefits of using inheritance to create the CommissionEmployee class:
  - Save time
  - Reduce chance of errors
  - Make it easier to understand
  - Reduce errors and inconsistencies in shared fields

Understanding Inheritance Terminology

- **Base class**
  - A class that is used as a basis for inheritance
  - Also called: **superclass**, **parent class**
- **Derived class** or **extended class**
  - A class that inherits from a base class
  - Also called: **subclass**, **child class**

Understanding Inheritance Terminology (continued -1)

- Tell which class is the base class and which is the derived class
  - The base class is also called the superclass
  - The derived class is also called the subclass
  - Use the two classes in a sentence with the phrase “is a”
  - Try saying the two class names together
  - Compare the class sizes
- The derived class can be further extended

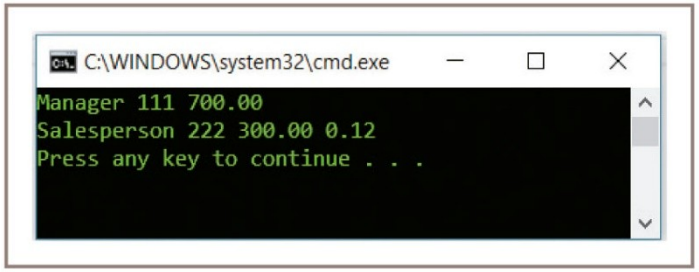
Understanding Inheritance Terminology (continued -2)

- **Ancestors**
  - The entire list of parent classes from which a child class is derived
  - A child inherits all the members of all its ancestors
  - A parent class does not gain any child class members

Understanding Inheritance Terminology (continued -3)

```
start
   Declarations
      Employee manager
      CommissionEmployee salesperson
   manager.setEmpNum("111")
   manager.setWeeklySalary(700.00)
   salesperson.setEmpNum("222")
   salesperson.setWeeklySalary(300.00)
   salesperson.setCommissionRate(0.12)
   output "Manager ", manager.getEmpNum(), manager.getWeeklySalary()
   output "Salesperson ", salesperson.getEmpNum(),
      salesperson.getWeeklySalary(), salesperson.getCommissionRate()
stop
```

Figure 11-15 EmployeeDemo application that declares two Employee objects



```
Manager 111 700.00
Salesperson 222 300.00 0.12
Press any key to continue . . .
```

Figure 11-16 Output of the program in Figure 11-15

Accessing Private Fields and Methods of a Parent Class

- It is common for methods to be public but for data to be private
- When a data field within a class is private:
  - No outside class can use it
  - Including a child class
    - Can be inconvenient
  - **Inaccessible**

Accessing Private Fields and Methods of a Parent Class (continued -1)

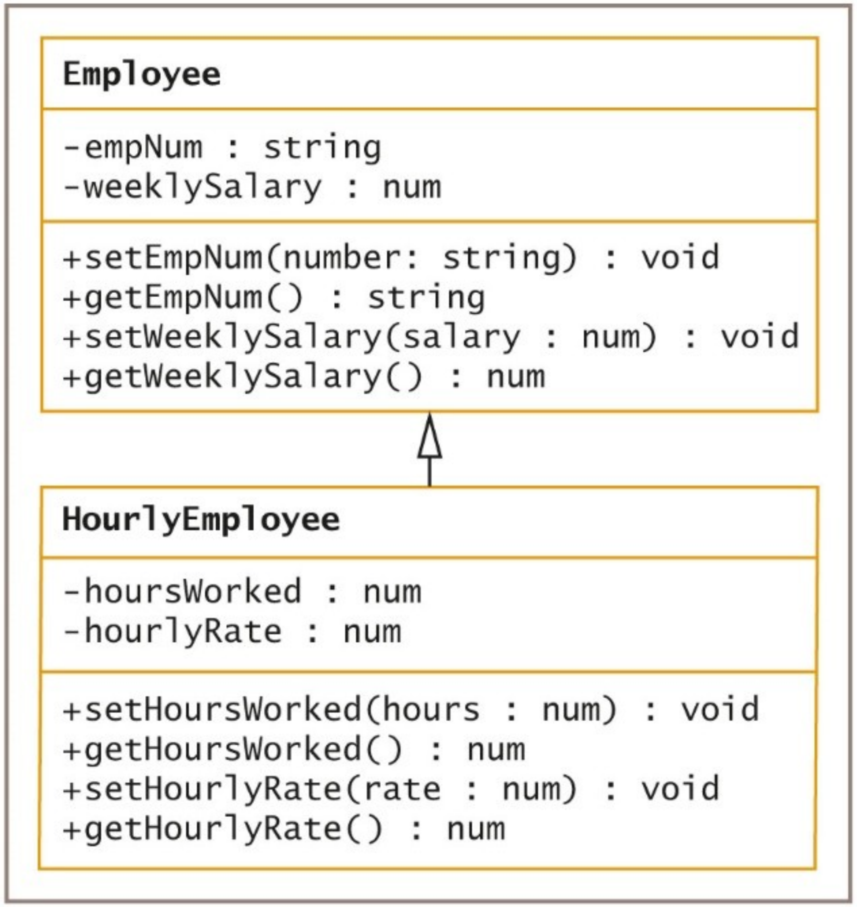
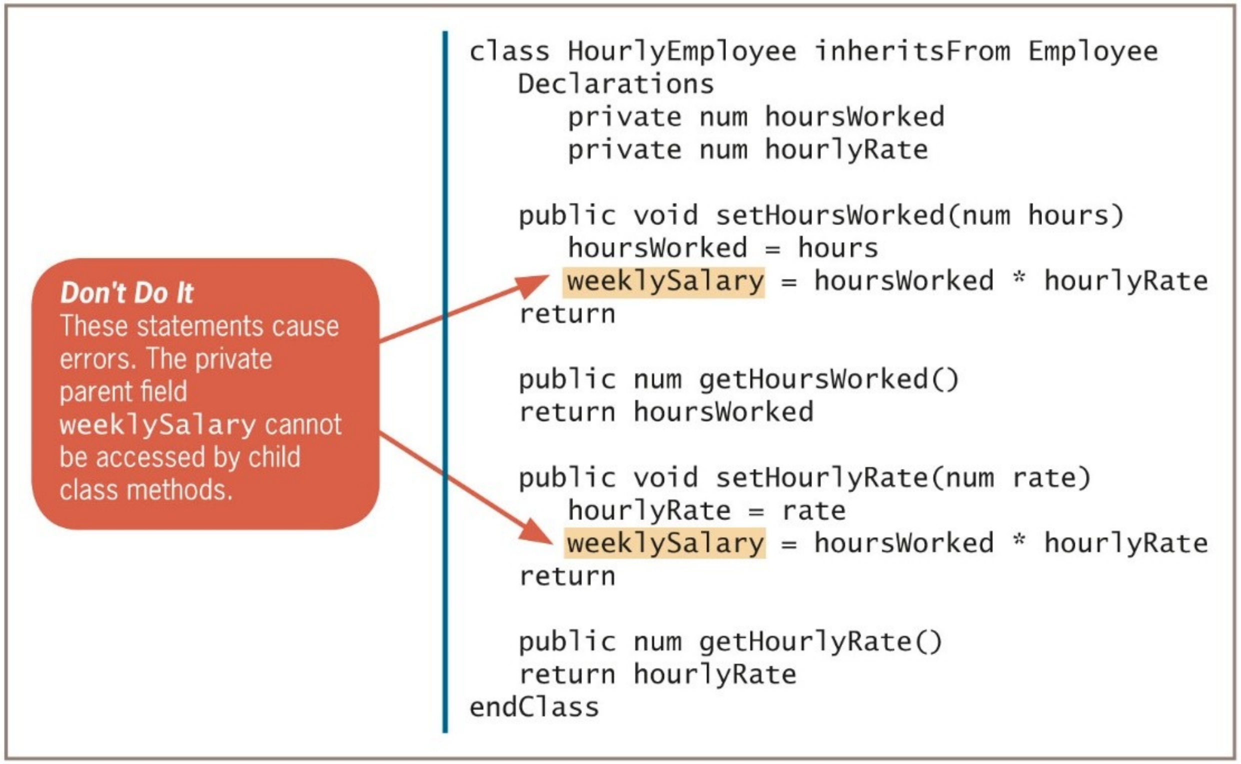


Figure 11-17 Class diagram for HourlyEmployee class

Accessing Private Fields and Methods of a Parent Class (continued -2)



Don't Do It
These statements cause errors. The private parent field weeklySalary cannot be accessed by child class methods.

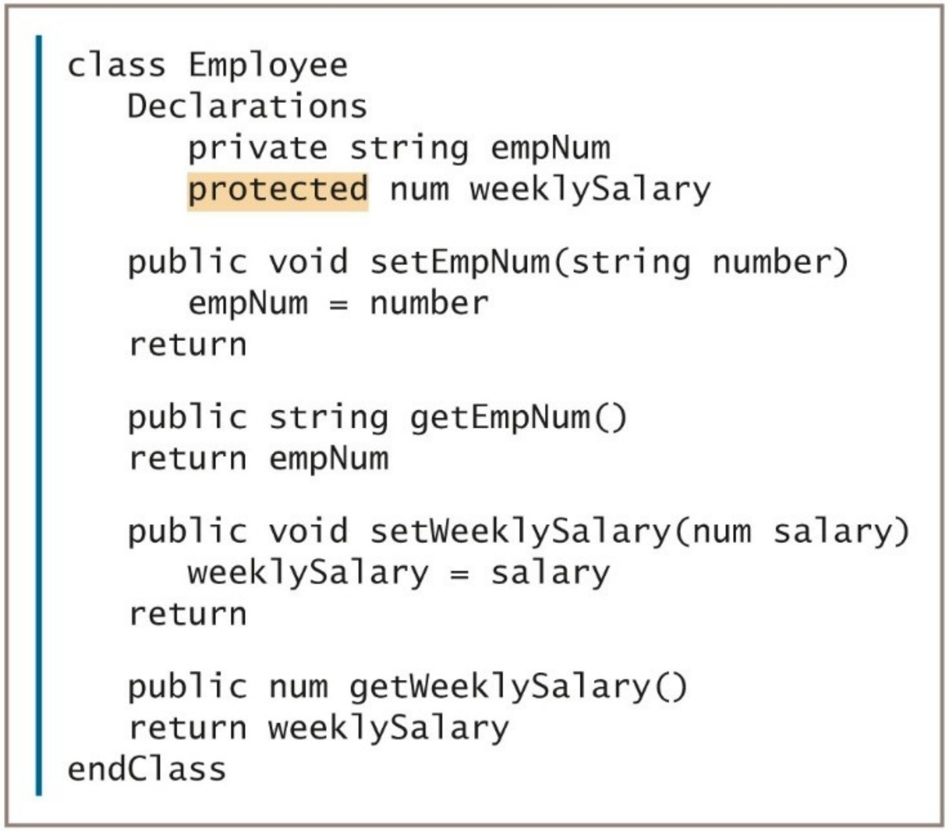
```
class HourlyEmployee inheritsFrom Employee
   Declarations
      private num hoursWorked
      private num hourlyRate

   public void setHoursWorked(num hours)
      hoursWorked = hours
      weeklySalary = hoursWorked * hourlyRate
   return

   public num getHoursWorked()
   return hoursWorked

   public void setHourlyRate(num rate)
      hourlyRate = rate
      weeklySalary = hoursWorked * hourlyRate
   return

   public num getHourlyRate()
   return hourlyRate
endClass
```

Figure 11-18 Implementation of HourlyEmployee class that attempts to access weeklySalary

Accessing Private Fields and Methods of a Parent Class (continued -3)

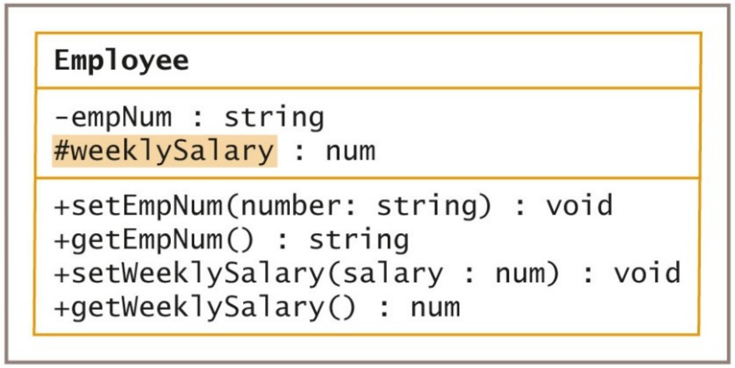
- `protected` **access specifier**
  - Used when you want no outside classes to be able to use a data field except classes that are children of the original class
- If the `Employee` class's creator did not foresee that a field would need to be accessible, then `weeklySalary` will remain private
  - Possible to correctly set an `HourlyEmployee`'s weekly pay using the `setWeeklySalary()` method

Accessing Private Fields and Methods of a Parent Class (continued -4)



```
class Employee
   Declarations
      private string empNum
      protected num weeklySalary

   public void setEmpNum(string number)
      empNum = number
   return

   public string getEmpNum()
   return empNum

   public void setWeeklySalary(num salary)
      weeklySalary = salary
   return

   public num getWeeklySalary()
   return weeklySalary
endClass
```

Figure 11-19 Employee class with a protected field



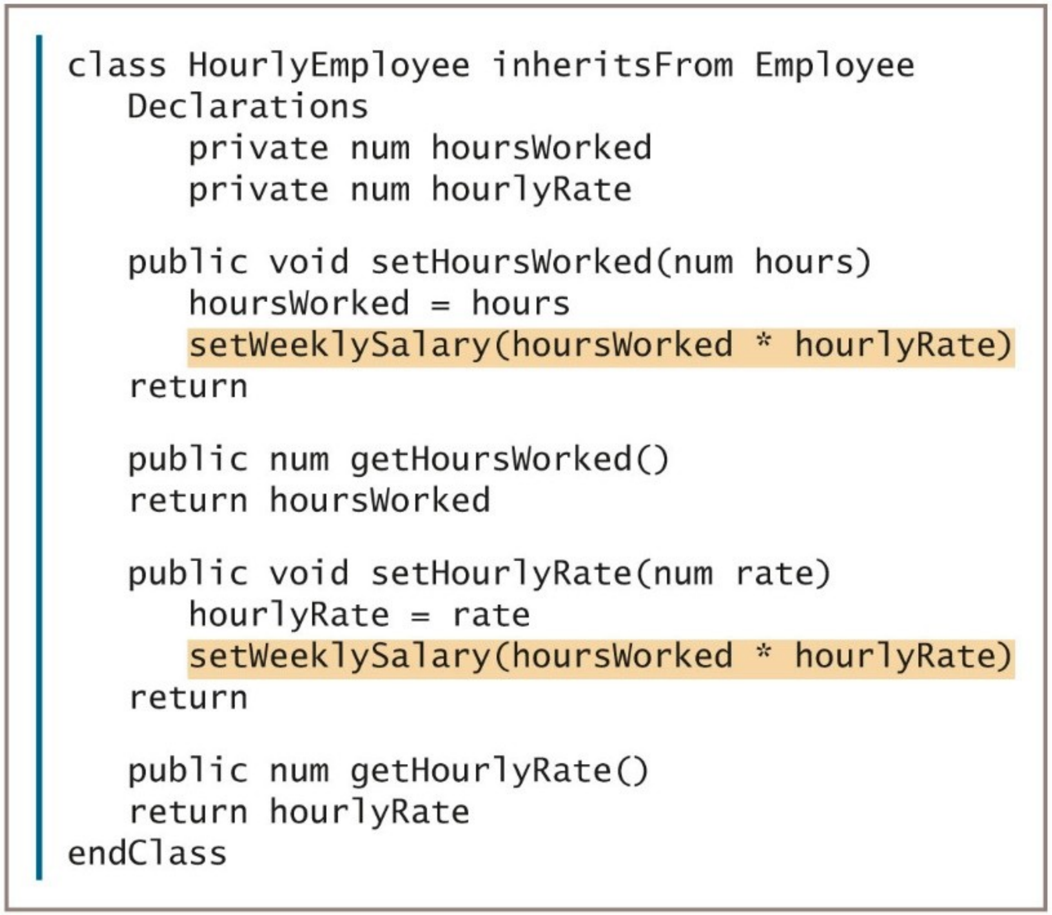
```
Employee
-empNum : string
#weeklySalary : num
+setEmpNum(number: string) : void
+getEmpNum() : string
+setWeeklySalary(salary : num) : void
+getWeeklySalary() : num
```

Figure 11-20 Employee class with protected member

Accessing Private Fields and Methods of a Parent Class (continued -5)

- When a child class accesses a private field of its parent's class:
  - Modify the parent class to make the field protected
  - The child class can use a public method within the parent class that modifies the field
- Protected access improves performance
  - Use protected data members sparingly
- Classes that depend on field names from parent classes are called **fragile** because they are prone to errors

Accessing Private Fields and Methods of a Parent Class (continued -6)



```
class HourlyEmployee inheritsFrom Employee
   Declarations
      private num hoursWorked
      private num hourlyRate

   public void setHoursWorked(num hours)
      hoursWorked = hours
      setWeeklySalary(hoursWorked * hourlyRate)
   return

   public num getHoursWorked()
   return hoursWorked

   public void setHourlyRate(num rate)
      hourlyRate = rate
      setWeeklySalary(hoursWorked * hourlyRate)
   return

   public num getHourlyRate()
   return hourlyRate
endClass
```

Figure 11-21 The HourlyEmployee class when weeklySalary remains private

Accessing Private Fields and Methods of a Parent Class (continued -7)

- **Multiple inheritance**
  - The capability to inherit from more than one class
- **Abstract class**
  - A class from which you cannot create any concrete objects, but from which you can inherit

Overriding Parent Class Methods in a Child Class

- **Overriding**
  - The mechanism by which a child class method is used by default when a parent class contains a method with the same signature

Using Inheritance to Achieve Good Software Design

- You can create powerful computer programs more easily if many of their components are used either "as is" or with slight modifications
- Advantages of creating a useful, extendable superclass:
  - Subclass creators save development and testing time
  - Superclass code has been tested and is reliable
  - Programmers who create or use new subclasses already understand how the superclass works
  - Neither the superclass source code nor the translated superclass code is changed

An Example of Using Predefined Classes: Creating GUI Objects

- **Libraries** or **packages**
  - Collections of classes that serve related purposes
- Graphical user interface (GUI) objects
  - Created in a **visual development environment** known as an **IDE** (integrated development environment)
  - Frames, buttons, labels, and text boxes
  - Place within interactive programs so that users can manipulate them using input devices

An Example of Using Predefined Classes: Creating GUI Objects (continued)

- Disadvantages of creating your own GUI object classes:
  - Lots of work
  - Repetitious
  - Components would look different in various applications
- **Visual development environment**
  - Known as an **IDE** (Integrated Development Environment)
  - Create programs by dragging components such as buttons and labels onto a screen and arranging them visually

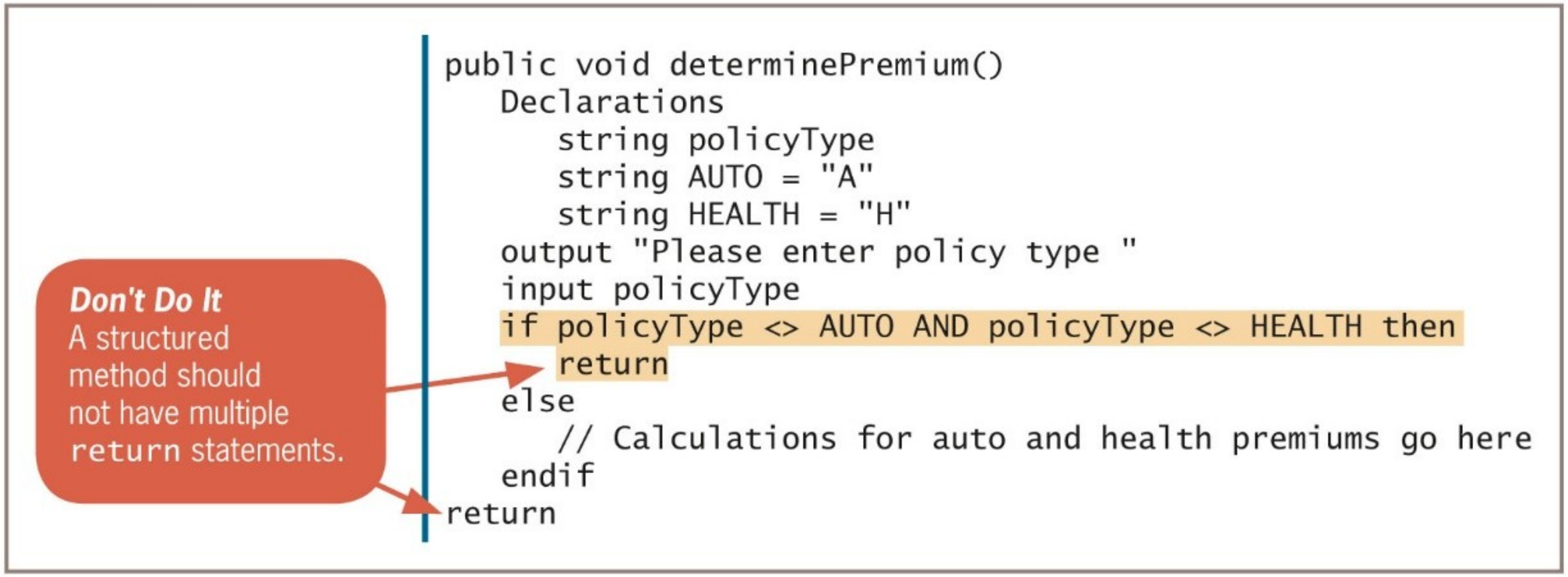
Understanding Exception Handling

- A lot of effort goes into checking data items to make sure they are valid and reasonable
- Procedural programs handled errors in various ways that were effective
  – Techniques had some drawbacks
- Object-oriented programming
  – New model called exception handling

Drawbacks to Traditional Error-Handling Techniques

- The most often used error-handling outcome in traditional programming was to terminate the program
  - Unforgiving
  - Unstructured
- Forcing the user to reenter data or limiting the number of chances the user gets to enter correct data
  - May allow no second chance at all

Drawbacks to Traditional Error-Handling Techniques
(continued -1)



```
public void determinePremium()
   Declarations
      string policyType
      string AUTO = "A"
      string HEALTH = "H"
   output "Please enter policy type "
   input policyType
   if policyType <> AUTO AND policyType <> HEALTH then
      return
   else
      // Calculations for auto and health premiums go here
   endif
return
```

Don't Do It
A structured method should not have multiple `return` statements.

Figure 11-22 A method that handles an error in an unstructured manner

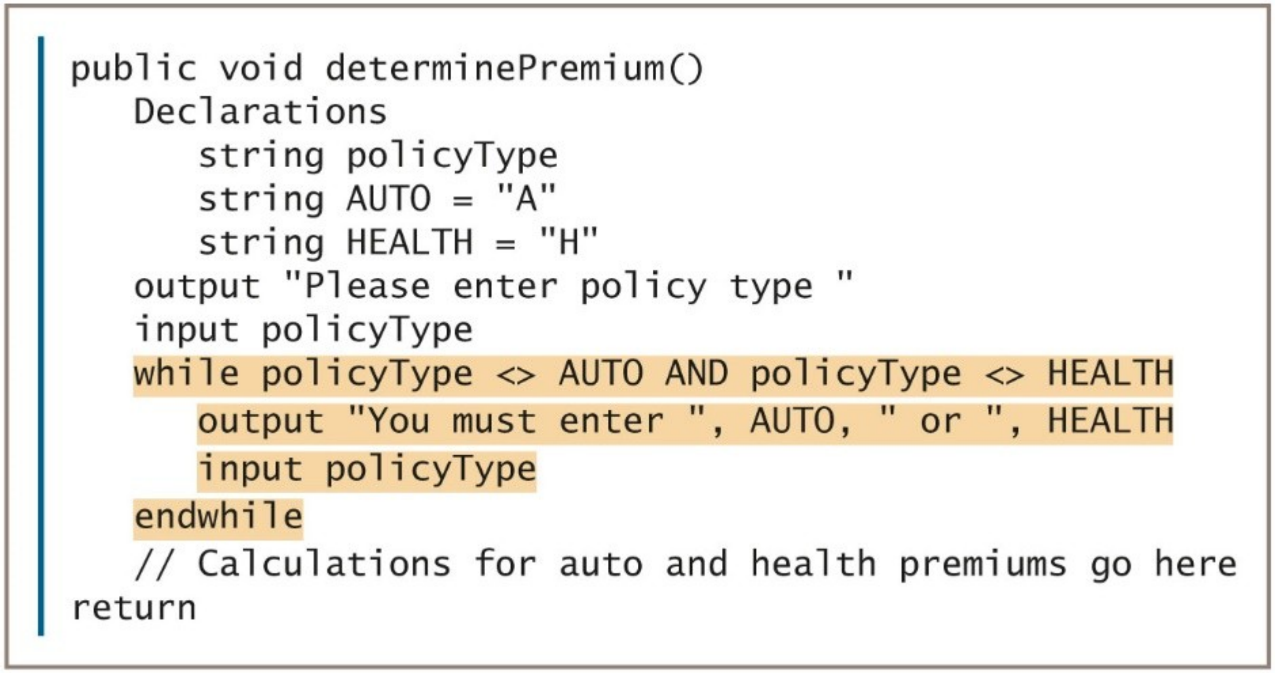
Drawbacks to Traditional Error-Handling Techniques

(continued -2)

- More elegant solution
  - Looping until the data item becomes valid
- Shortcomings
  - Method is not very reusable
  - Method is not very flexible
  - Only works with interactive programs

Drawbacks to Traditional Error-Handling Techniques

(continued -3)



```
public void determinePremium()
   Declarations
      string policyType
      string AUTO = "A"
      string HEALTH = "H"
   output "Please enter policy type "
   input policyType
   while policyType <> AUTO AND policyType <> HEALTH
      output "You must enter ", AUTO, " or ", HEALTH
      input policyType
   endwhile
   // Calculations for auto and health premiums go here
return
```

Figure 11-23 A method that handles an error using a loop

The Object-Oriented Exception-Handling Model

- **Exception handling**
  - A specific group of techniques for handling errors in object-oriented programs
- **Exceptions**
  - The generic name used for errors in object-oriented languages
- **Try** some code that might **throw an exception**
- If an exception is thrown, it is passed to a block of code that can **catch the exception**

The Object-Oriented Exception-Handling Model

(continued -1)

- **`try` block**
  - A block of code you attempt to execute while acknowledging that an exception might occur
  - The keyword `try`, followed by any number of statements
  - If a statement in the block causes an exception, the remaining statements in the `try` block do not execute and the `try` block is abandoned
- **`throw` statement**
  - Sends an `Exception` object out of the current code block or method so it can be handled elsewhere

The Object-Oriented Exception-Handling Model

(continued -2)

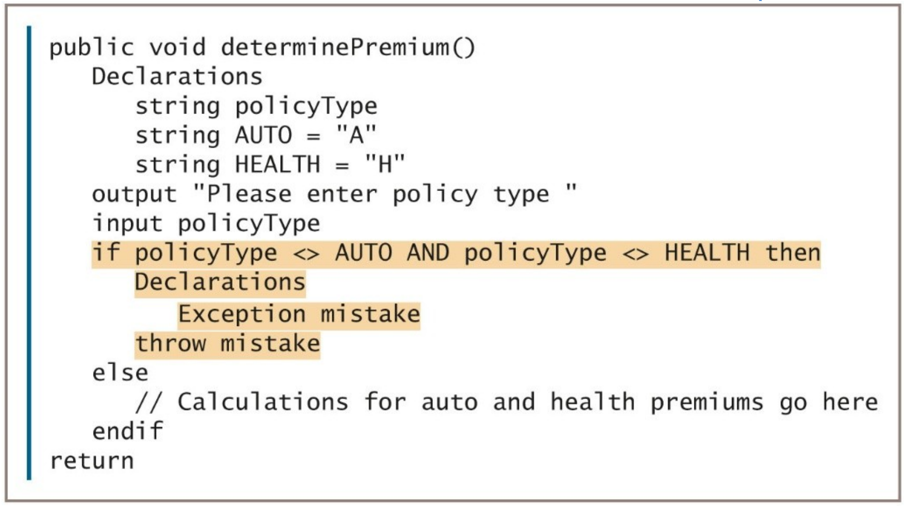
- **`catch` block**
  - A segment of code that can handle an exception
  - The keyword `catch`, followed by parentheses that contain an `Exception` type and an identifier
  - Statements that take the action to handle the error condition
  - The `endcatch` keyword indicates the end of the catch block in the pseudocode

The Object-Oriented Exception-Handling Model
(continued -3)

- General principle of exception handling:
  - The method that uses the data should be able to detect errors, but not be required to handle them
  - Handling should be left to the application that uses the object
- **Sunny day case**
  - When there are no exceptions and nothing goes wrong with a program

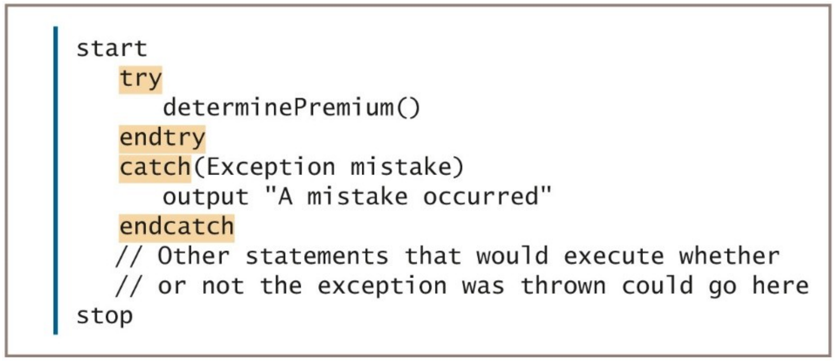
The Object-Oriented Exception-Handling Model

(continued -4)



```
public void determinePremium()
   Declarations
      string policyType
      string AUTO = "A"
      string HEALTH = "H"
   output "Please enter policy type "
   input policyType
   if policyType <> AUTO AND policyType <> HEALTH then
      Declarations
         Exception mistake
      throw mistake
   else
      // Calculations for auto and health premiums go here
   endif
return
```

Figure 11-24 A method that creates and throws an Exception object



```
start
   try
      determinePremium()
   endtry
   catch(Exception mistake)
      output "A mistake occurred"
   endcatch
   // Other statements that would execute whether
   // or not the exception was thrown could go here
stop
```

Figure 11-25 A program that contains a try...catch pair

Using Built-in Exceptions and Creating Your Own Exceptions

- Many OO languages provide many built-in `Exception` types
  - Built-in `Exceptions` in a programming language cannot cover every condition
- Create your own throwable `Exception`
  - Extend a built-in `Exception` class

Reviewing the Advantages of Object-Oriented Programming

- Save development time
  - Each object automatically includes appropriate, reliable methods and attributes
- Develop new classes more quickly
  - By extending classes that already exist and work
- Use existing objects
  - Concentrate only on the interface to those objects
- Polymorphism
  - Use reasonable, easy-to-remember names for methods

Summary

- A constructor establishes objects and a destructor destroys them
- Composition or aggregation (has-a relationship)is where a class can contain objects of another class as data members
- Creating a class by using inheritance provides you with prewritten and tested data fields and methods automatically. Using inheritance helps save time, reduces the chance of errors and inconsistencies, and makes it easier to understand your classes

Summary (continued)

- Libraries contain some of the most useful classes to create graphical user interface (GUI) objects
- Exception-handling techniques are used to handle errors in object-oriented programs
- Object-oriented programming saves development time
- Efficiency is achieved through both inheritance and polymorphism