Programming Logic and Design
Ninth Edition

Chapter 10

Object-Oriented Programming

Objectives

In this chapter, you will learn about:

- The principles of object-oriented programming
- Classes
- Public and private access
- Ways to organize classes
- Instance methods
- Static methods
- Using objects

Principles of Object-Oriented Programming

- **Object-oriented programming** (OOP)
  - A programming model that focuses on an application’s components and data and the methods to manipulate them
- Uses all of the familiar concepts from modular procedural programming
  - Variables, methods, passing arguments
  - Sequence, selection, and looping structures
  - But involves a different way of thinking

Principles of Object-Oriented Programming (continued)

- Important features of object-oriented languages
  - Classes
  - Objects
  - Polymorphism
  - Inheritance
  - Encapsulation

Classes and Objects

- **Class**
  - Describes a group or collection of objects with common attributes
- **Object - **One **instance** of a class
  - Sometimes called one **instantiation** of a class
  - When a program creates an object, it **instantiates** the object
- Example
  - Class name: dog
  - Attributes: name, age, hasShots
  - Methods: changeName, updateShots

Classes and Objects (continued -1)

- **Attributes**
  - Characteristics that define an object as part of a class
  - Example
    - Automobile’s attributes: make, model, year, and purchase price
- Methods
  - Actions that alter, use, or retrieve the attributes
  - Example
    - Methods for changing an automobile’s running status, gear, speed, and cleanliness

Classes and Objects (continued -2)

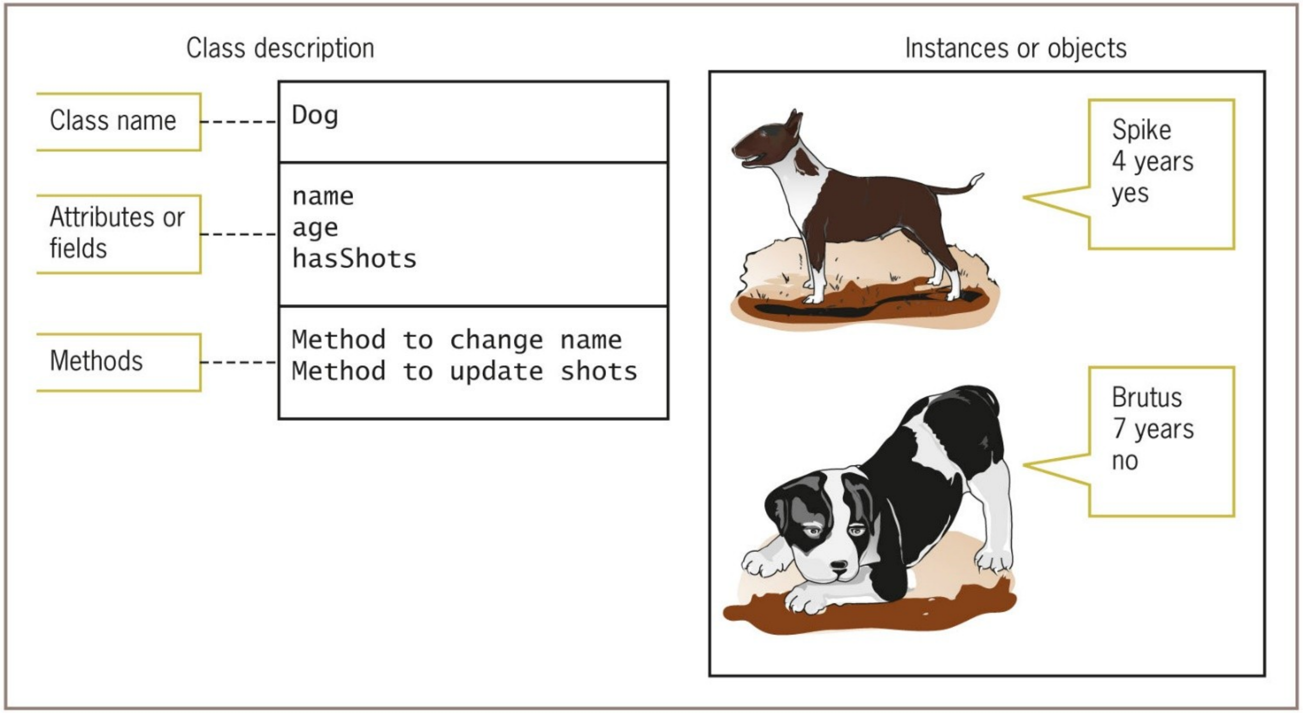


Figure 10-1 A Dog class and two instances

Classes and Objects (continued -3)

- Think in an object-oriented manner
  - Everything is an object
  - Every object is a member of a class
- **Is-a relationship**
  - “My oak desk with the scratch on top *is* a Desk”
- Class reusability
- Class’s **instance variables**
  - Data components of a class that belong to every instantiated object
  - Often called **fields**

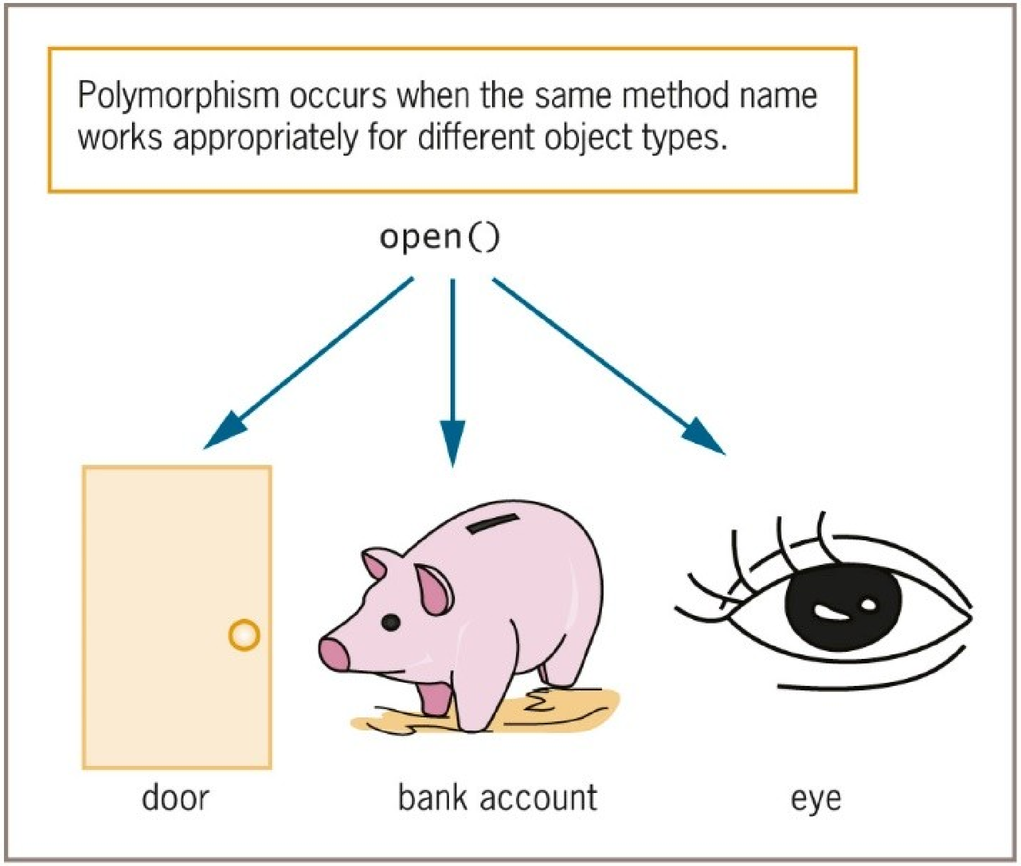
Classes and Objects (continued -4)

- **State**
  - A set of all the values or contents of a class object’s instance variables
- Every object that is an instance of a class possesses the same methods
- Create classes from which objects will be instantiated
- **Class client** or **class user**
  - A program or class that instantiates objects of another prewritten class

Polymorphism

- The world is full of objects
  - A door is an object that needs to be open or closed
  - But an “open” procedure works differently on different objects
    - Open a door
    - Open a drawer
    - Open a bank account
    - Open a file
    - Open your eyes
  - One “open” procedure can open anything if it gets the correct arguments

Polymorphism (continued)



Polymorphism occurs when the same method name works appropriately for different object types.

Figure 10-2 Examples of polymorphism

Inheritance

- **Inheritance**
  - The process of acquiring the traits of one’s predecessors
- Example
  - A door with a stained glass window inherits all the attributes (doorknob, hinges) and methods (open and close) of a door
- Once you create an object
  - Develop new objects that possess all the traits of the original object
  - Plus new traits

Encapsulation

- **Encapsulation**
  - The process of combining all of an object’s attributes and methods into a single package
- **Information hiding** (also called **data hiding**)
  - Other classes should not alter an object’s attributes
- Outside classes should only be allowed to make a request that an attribute be altered
  - It is up to the class’s methods to determine whether the request is appropriate

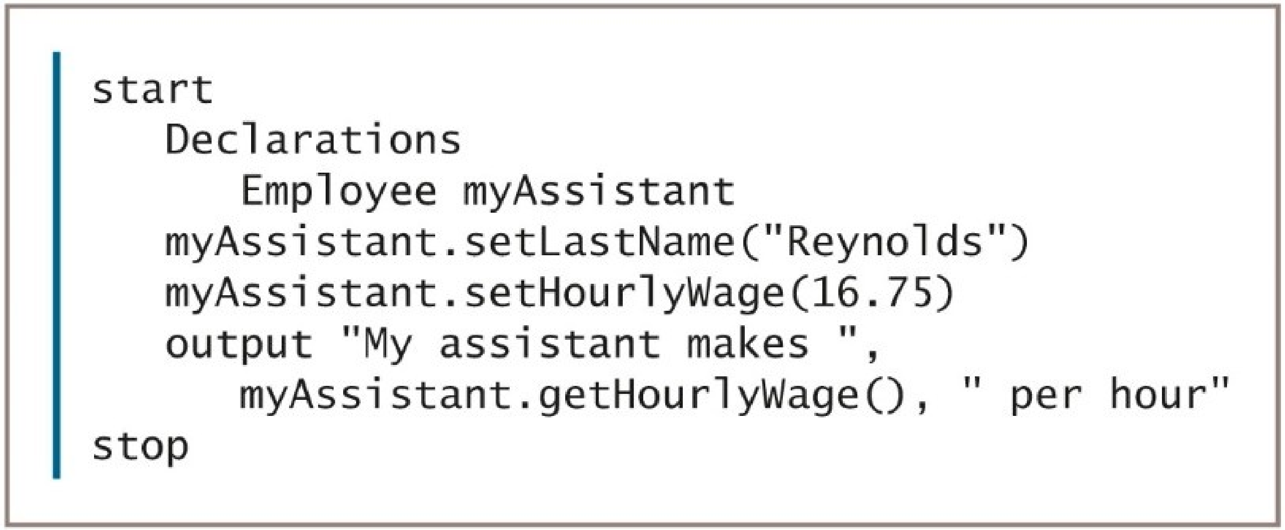
Defining Classes and Creating Class Diagrams

- **Class definition**
  - A set of program statements
  - Characteristics of the class’s objects and the methods that can be applied to its objects
- Three parts:
  - Every class has a name
  - Most classes contain data (not required)
  - Most classes contain methods (not required)

Defining Classes and Creating Class Diagrams (continued -1)

- Declaring a class
  - Does not create any actual objects
- After an object is instantiated
  - Methods can be accessed using an identifier, a dot, and a method call
  - `myAssistant.setHourlyWage(16.75)`
- `Employee myAssistant`
  - Declare the `myAssistant` object
  - Contains all the data fields
  - Access to all methods contained within the class

Defining Classes and Creating Class Diagrams (continued -2)



```
start
   Declarations
      Employee myAssistant
   myAssistant.setLastName("Reynolds")
   myAssistant.setHourlyWage(16.75)
   output "My assistant makes ",
      myAssistant.getHourlyWage(), " per hour"
stop
```

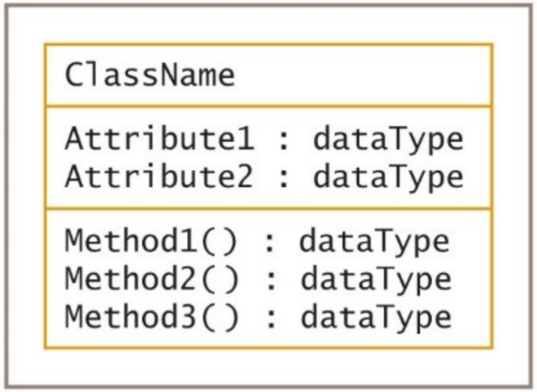
Figure 10-4 Application that declares and uses an Employee object

Defining Classes and Creating Class Diagrams (continued -3)

- Programmers call the classes they write **user-defined types**
  - More accurately called **programmer-defined types**
  - OOP programmers call them **abstract data types** (ADTs)
  - Simple numbers and characters are called **primitive data types**
- “Black box”
  - The ability to use methods without knowing the details of their contents
  - A feature of encapsulation

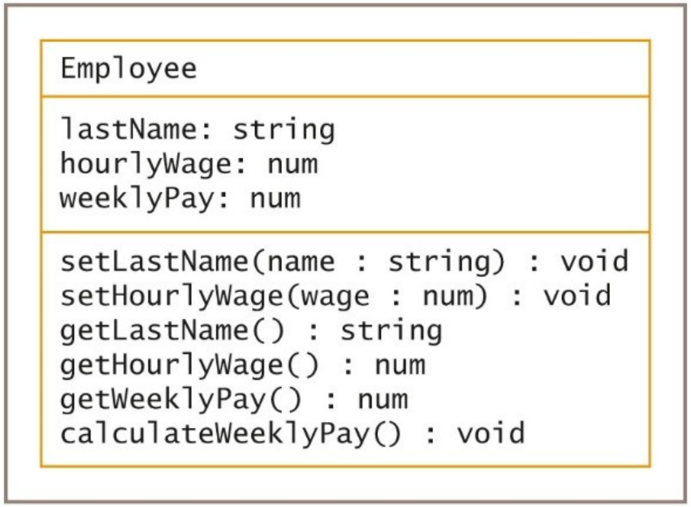
Creating Class Diagrams

- **Class diagram**
  - Three sections
    - Top: contains the name of the class
    - Middle: contains the names and data types of the attributes
    - Bottom: contains the



```
ClassName

Attribute1 : dataType
Attribute2 : dataType

Method1() : dataType
Method2() : dataType
Method3() : dataType
```

Figure 10-5 Generic class diagram



```
Employee

lastName: string
hourlyWage: num
weeklyPay: num

setLastName(name : string) : void
setHourlyWage(wage : num) : void
getLastName() : string
getHourlyWage() : num
getWeeklyPay() : num
calculateWeeklyPay() : void
```

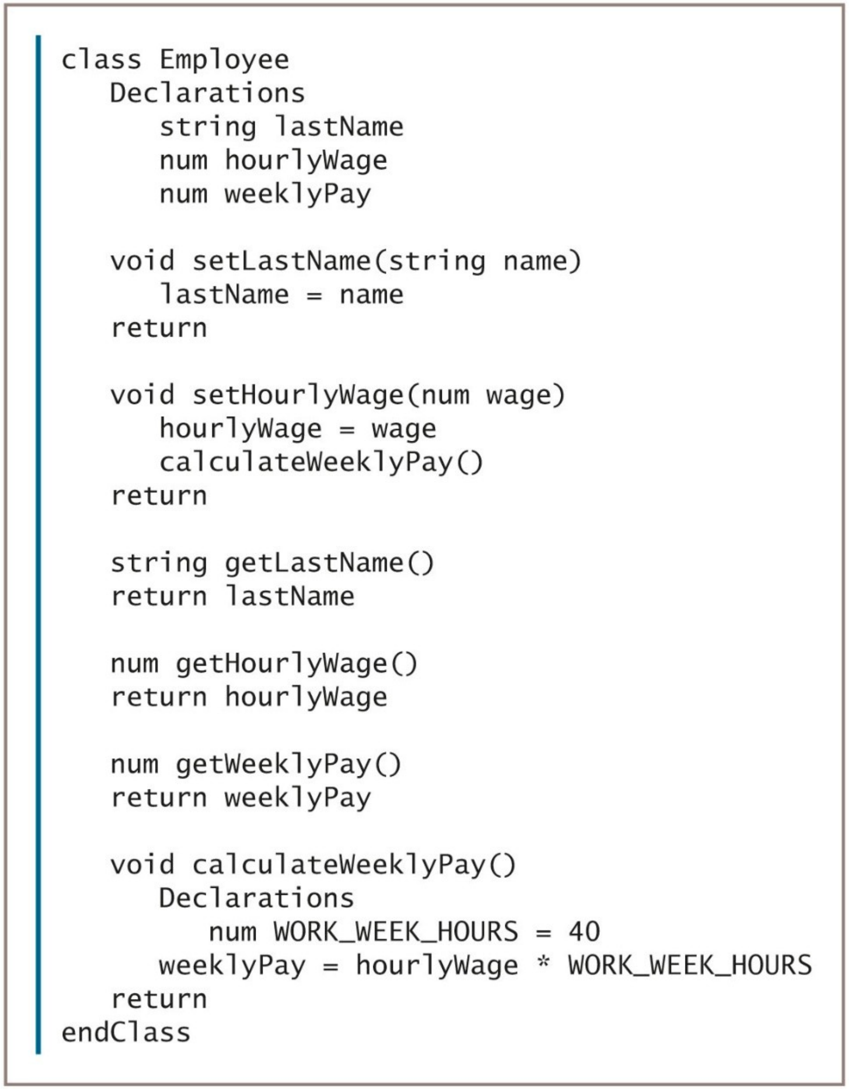
Figure 10-6 Employee class diagram

Creating Class Diagrams (continued - 1)

- Purpose of `Employee` class methods
  - Two of the methods accept values from the outside world
  - Three of the methods send data to the outside world
  - One method performs work within the class

Creating Class Diagrams

(continued -2)



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

   void setLastName(string name)
      lastName = name
   return

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

   string getLastName()
   return lastName

   num getHourlyWage()
   return hourlyWage

   num getWeeklyPay()
   return weeklyPay

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

Figure 10-7 Pseudocode for Employee class described in the class diagram in Figure 10-6

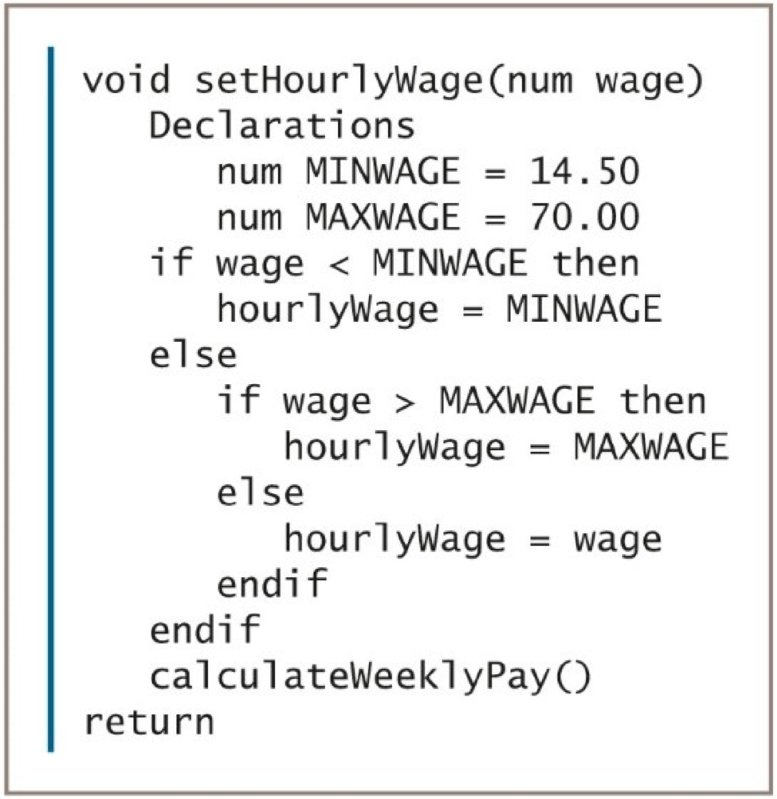
The Set Methods

- **Set method** (also called **mutator method**)
  - Sets the values of data fields within the class

```
void setLastName(string name)
   lastName = name
return
mySecretary.setLastName("Johnson")
```

  - No requirement that such methods start with the *set* prefix
  - Some languages allow you to create a **property** to set field values instead of creating a set method

The Set Methods (continued)



```
void setHourlyWage(num wage)
   Declarations
      num MINWAGE = 14.50
      num MAXWAGE = 70.00
   if wage < MINWAGE then
      hourlyWage = MINWAGE
   else
      if wage > MAXWAGE then
         hourlyWage = MAXWAGE
      else
         hourlyWage = wage
      endif
   endif
   calculateWeeklyPay()
return
```

Figure 10-8 A version of the `setHourlyWage()` method including validation

The Get Methods

- **Get method** (also called **accessor method**)
- Purpose is to return a value to the world outside the class

```
string getLastName()
return lastName
```

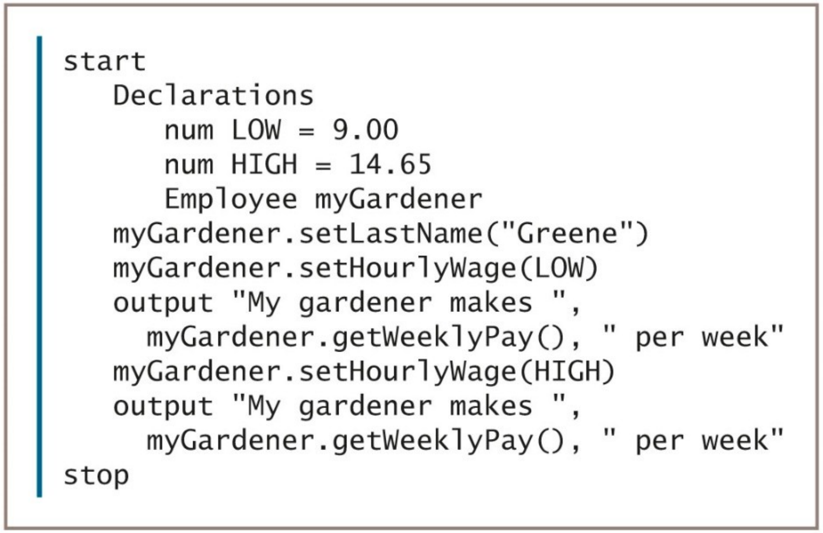
- Value returned from a get method can be used as any other variable of its type would be used

Work Methods

- **Work method** (also called **help method**, or **facilitator**)
  - performs tasks within a class

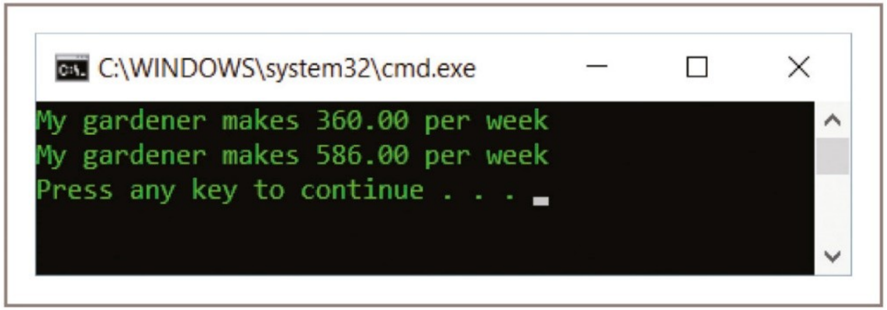
```
void calculateWeeklyPay()
  Declarations
        num WORK_WEEK_HOURS = 40
  weeklyPay = hourlyWage * WORK_WEEK_HOURS
return
```

Work Methods (continued)



```
start
   Declarations
      num LOW = 9.00
      num HIGH = 14.65
      Employee myGardener
   myGardener.setLastName("Greene")
   myGardener.setHourlyWage(LOW)
   output "My gardener makes ",
     myGardener.getWeeklyPay(), " per week"
   myGardener.setHourlyWage(HIGH)
   output "My gardener makes ",
     myGardener.getWeeklyPay(), " per week"
stop
```

Figure 10-9 Program that sets and displays `Employee` data two times



```
My gardener makes 360.00 per week
My gardener makes 586.00 per week
Press any key to continue . . .
```

Figure 10-10 Execution of program in Figure 10-9

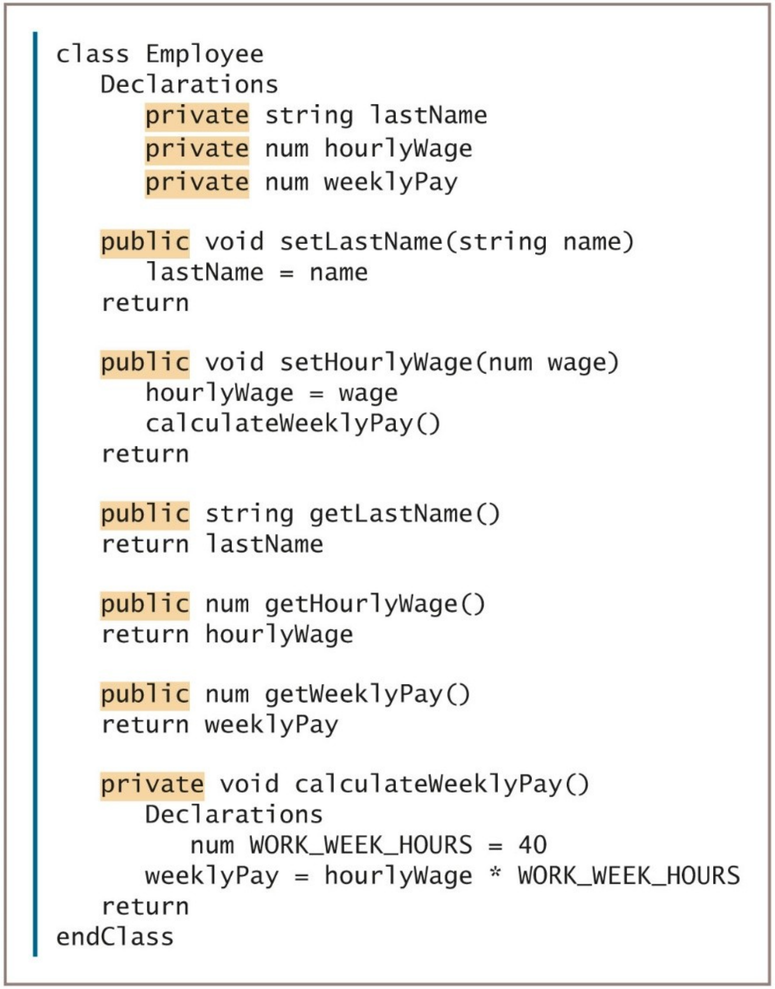
Understanding Public and Private Access

- You do not want any outside programs or methods to alter your class's data fields unless you have control over the process
- Prevent outsiders from changing your data
  - Force other programs and methods to use a method that is part of the class
- Specify that data fields have **private access**
  - Data cannot be accessed by any method that is not part of the class

Understanding Public and Private Access (continued -1)

- **Public access**
  - Other programs and methods may use the methods that control access to the private data
- **Access specifier**
  - Also called an access modifier
  - An adjective defining the type of access that outside classes will have to the attribute or method
    - `public` or `private`

Understanding Public and Private Access (continued -2)



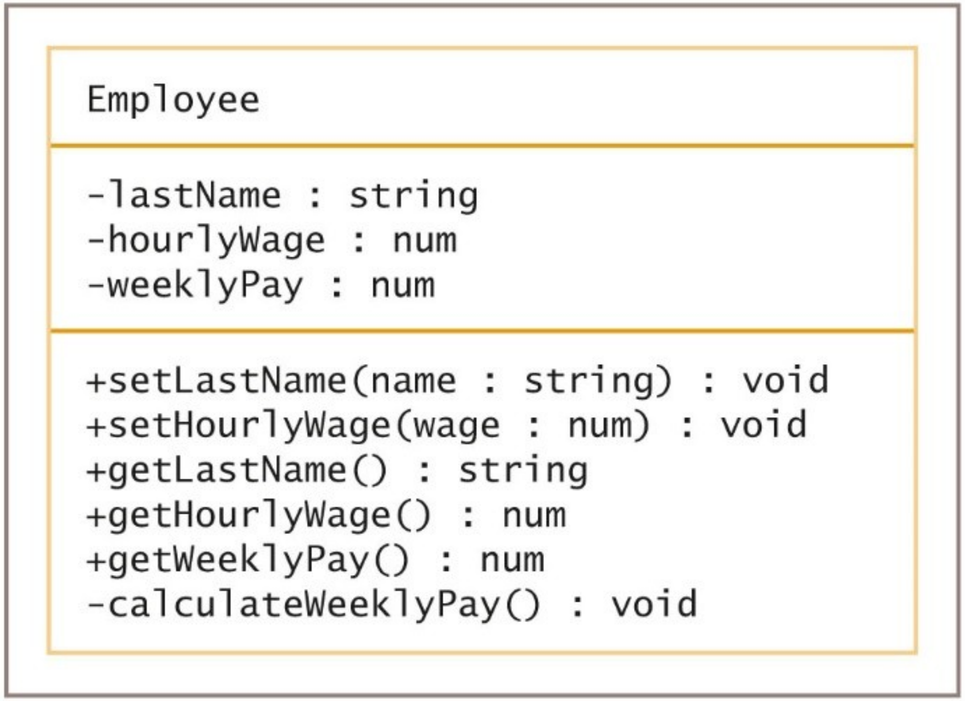
```
class Employee
   Declarations
      private string lastName
      private num hourlyWage
      private num weeklyPay

   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 10-11 Employee class including `public` and `private` access specifiers

Understanding Public and Private Access (continued -3)

- Don’t do it:
  - `myAssistant.hourlyWage = 15.00`
- Instead:
  - `myAssistant.setHourlyWage(15.00)`
- Methods may be private; don’t do it:
  - `myAssistant.calculateWeeklyPay()`

Understanding Public and Private Access (continued -4)



```
Employee

-lastName : string
-hourlyWage : num
-weeklyPay : num

+setLastName(name : string) : void
+setHourlyWage(wage : num) : void
+getLastName() : string
+getHourlyWage() : num
+getWeeklyPay() : num
-calculateWeeklyPay() : void
```

Figure 10-12 Employee class diagram with public and private access specifiers

Organizing Classes

- Most programmers place data fields in logical order at the beginning of a class
  - An ID number is most likely used as a unique identifier
    - Primary key
  - Flexibility in how you position data fields
- In some languages, you can organize a class’s data fields and methods in any order

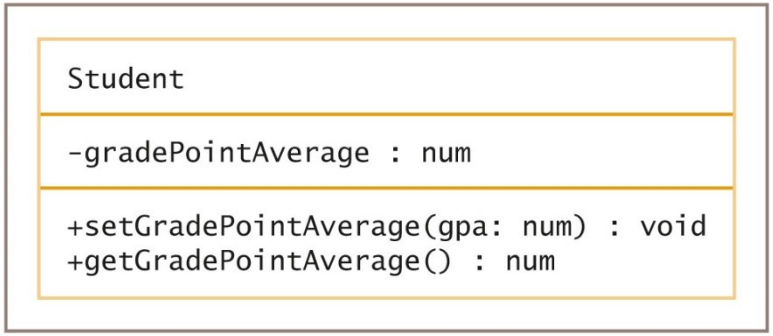
Organizing Classes (continued)

- Class method ordering
  - Alphabetical
  - Pairs of get and set methods
  - Same order as the data fields are defined
  - All accessor (get) methods together and all mutator (set) methods together

Understanding Instance Methods

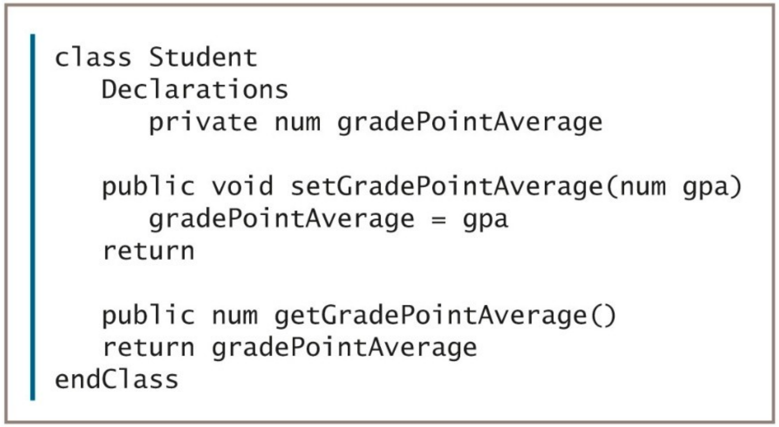
- Every object that is an instance of a class is assumed to possess the same data and have access to the same methods

Understanding Instance Methods (continued -1)



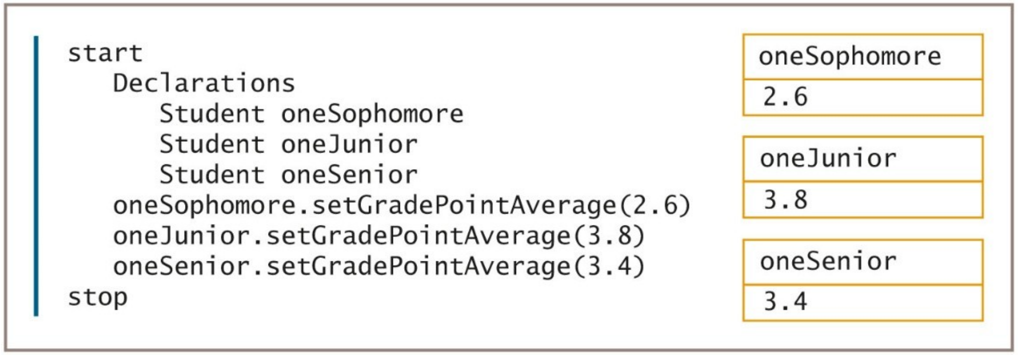
Student
-gradePointAverage : num
+setGradePointAverage(gpa: num) : void
+getGradePointAverage() : num

Figure 10-13 Class diagram for `Student` class



```
class Student
   Declarations
      private num gradePointAverage

   public void setGradePointAverage(num gpa)
      gradePointAverage = gpa
   return

   public num getGradePointAverage()
   return gradePointAverage
endClass
```

Figure 10-14 Pseudocode for the `Student` class



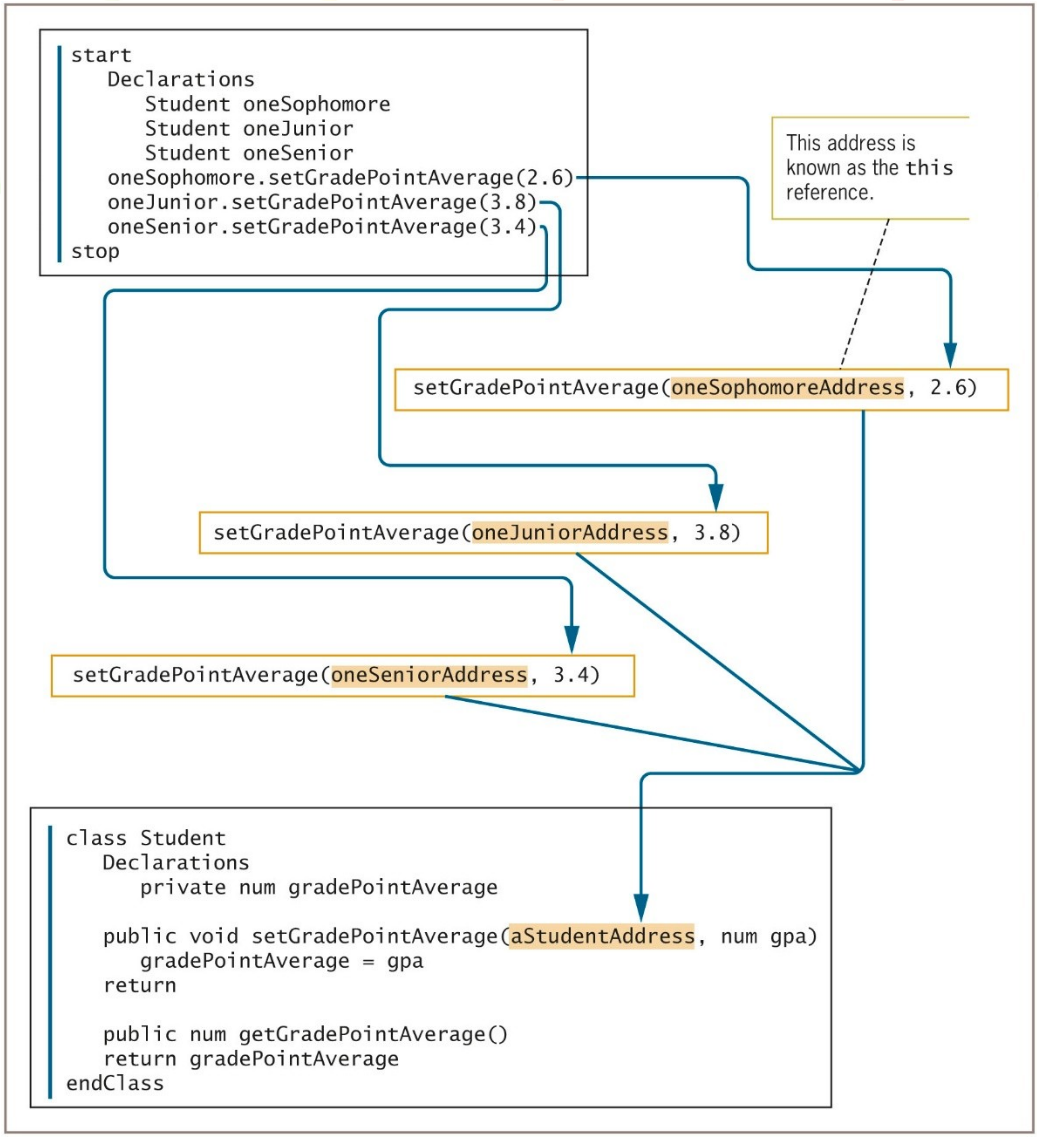
```
start
   Declarations
      Student oneSophomore
      Student oneJunior
      Student oneSenior
   oneSophomore.setGradePointAverage(2.6)
   oneJunior.setGradePointAverage(3.8)
   oneSenior.setGradePointAverage(3.4)
stop
```

Figure 10-15 Program that creates three `Student` objects and picture of how they look in memory

Understanding Instance Methods (continued -2)

- **Instance method**
  - Method that works appropriately with different objects
  - If you create 100 `Students` and assign grade point averages to each of them, you would need 100 storage locations in computer memory
- Only one copy of each instance method is stored in memory
  - The computer needs a way to determine whose `gradePointAverage` is being set or retrieved

Understanding Instance Methods (continued-3)



```
start
   Declarations
      Student oneSophomore
      Student oneJunior
      Student oneSenior
   oneSophomore.setGradePointAverage(2.6)
   oneJunior.setGradePointAverage(3.8)
   oneSenior.setGradePointAverage(3.4)
stop
```

This address is known as the `this` reference.

```
setGradePointAverage(oneSophomoreAddress, 2.6)
```

```
setGradePointAverage(oneJuniorAddress, 3.8)
```

```
setGradePointAverage(oneSeniorAddress, 3.4)
```

```
class Student
   Declarations
      private num gradePointAverage

   public void setGradePointAverage(aStudentAddress, num gpa)
      gradePointAverage = gpa
   return

   public num getGradePointAverage()
   return gradePointAverage
endClass
```

Figure 10-16 How `Student` object memory addresses are passed from an application to an instance method of the `Student` class

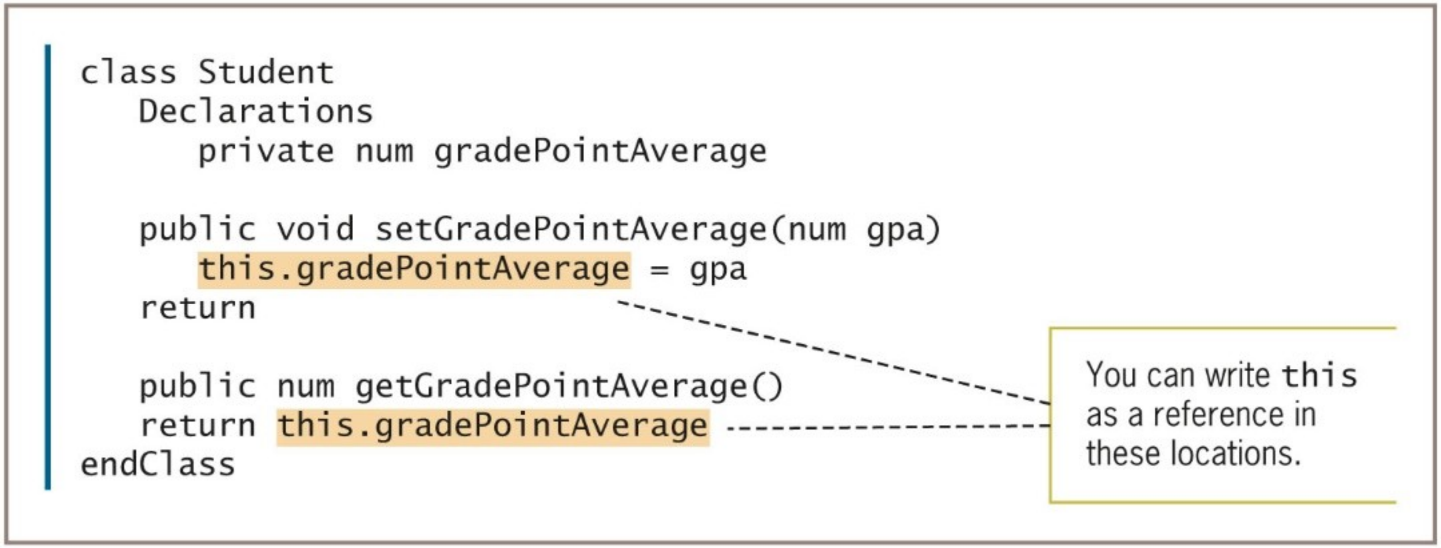
Understanding Instance Methods (continued -4)

- **`this` reference**
  - An automatically created variable
  - Holds the address of an object
  - Passes it to an instance method whenever the method is called
  - Refers to “this particular object” using the method
  - Implicitly passed as a parameter to each instance method

Understanding Instance Methods (continued -5)

- Identifiers within the method always mean exactly the same thing
  - Any field name defined in the class
  - `this`, followed by a dot, followed by the same field name
- Example of an occasion when you might use the `this` reference explicitly

Understanding Instance Methods (continued -6)



```
class Student
   Declarations
      private num gradePointAverage

   public void setGradePointAverage(num gpa)
      this.gradePointAverage = gpa
   return

   public num getGradePointAverage()
   return this.gradePointAverage
endClass
```

You can write this as a reference in these locations.

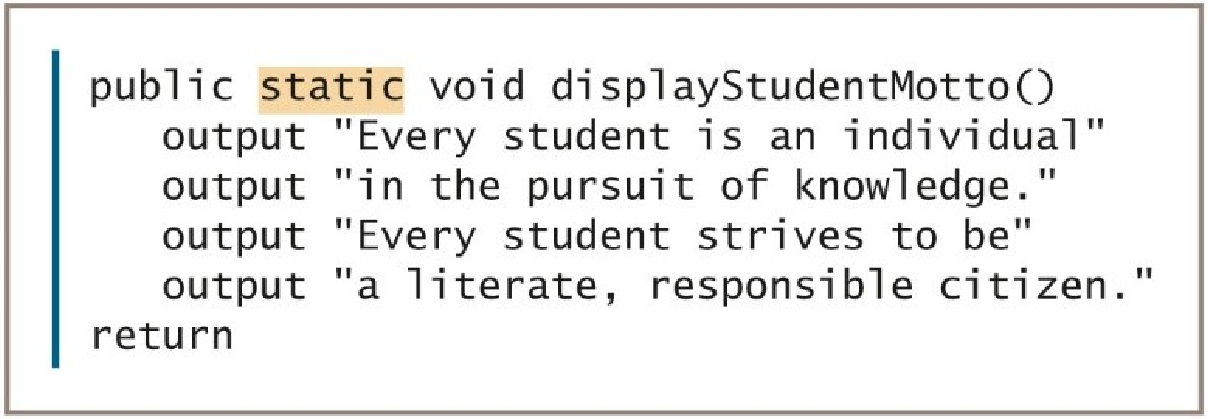
Figure 10-17 Explicitly using this in the Student class

Understanding Static Methods

- Some methods do not require a `this` reference
- `displayStudentMotto()`
  - A class method instead of an instance method
- Two types of methods
  - **Static methods** (also called **class methods**)
    - Methods for which no object needs to exist
  - **Nonstatic methods**
    - Methods that exist to be used with an object
- `Student.displayStudentMotto()`

Understanding Static Methods

(continued)



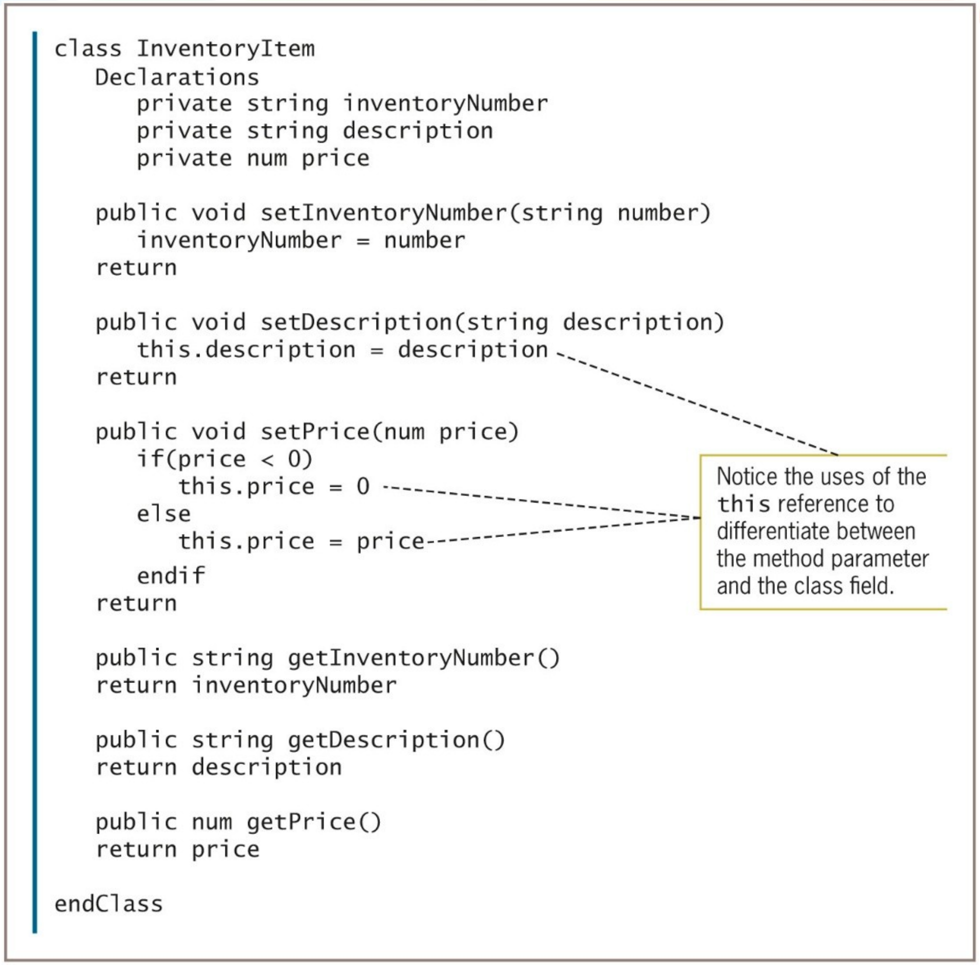
```
public static void displayStudentMotto()
   output "Every student is an individual"
   output "in the pursuit of knowledge."
   output "Every student strives to be"
   output "a literate, responsible citizen."
return
```

Figure 10-18 Student class displayStudentMotto() method

Using Objects

- You can use objects like you would use any other simpler data type
- `InventoryItem` class
  - Pass an object to a method
  - Return an object from a method
  - Use an array of objects

Using Objects (continued)

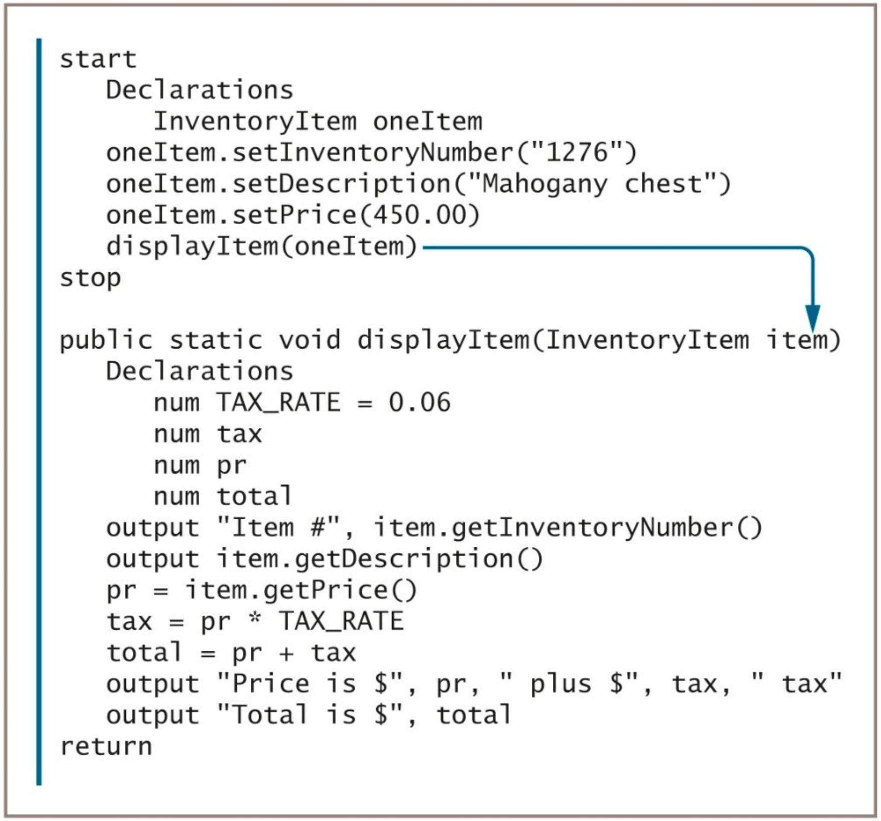


```
class InventoryItem
   Declarations
      private string inventoryNumber
      private string description
      private num price

   public void setInventoryNumber(string number)
      inventoryNumber = number
   return

   public void setDescription(string description)
      this.description = description
   return

   public void setPrice(num price)
      if(price < 0)
         this.price = 0
      else
         this.price = price
      endif
   return

   public string getInventoryNumber()
   return inventoryNumber

   public string getDescription()
   return description

   public num getPrice()
   return price

endClass
```

Notice the uses of the this reference to differentiate between the method parameter and the class field.

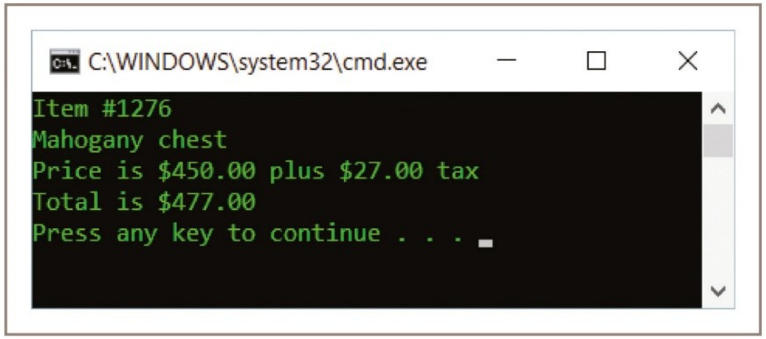
Figure 10-19 InventoryItem class

Passing an Object to a Method



```
start
   Declarations
      InventoryItem oneItem
   oneItem.setInventoryNumber("1276")
   oneItem.setDescription("Mahogany chest")
   oneItem.setPrice(450.00)
   displayItem(oneItem)
stop

public static void displayItem(InventoryItem item)
   Declarations
      num TAX_RATE = 0.06
      num tax
      num pr
      num total
   output "Item #", item.getInventoryNumber()
   output item.getDescription()
   pr = item.getPrice()
   tax = pr * TAX_RATE
   total = pr + tax
   output "Price is $", pr, " plus $", tax, " tax"
   output "Total is $", total
return
```

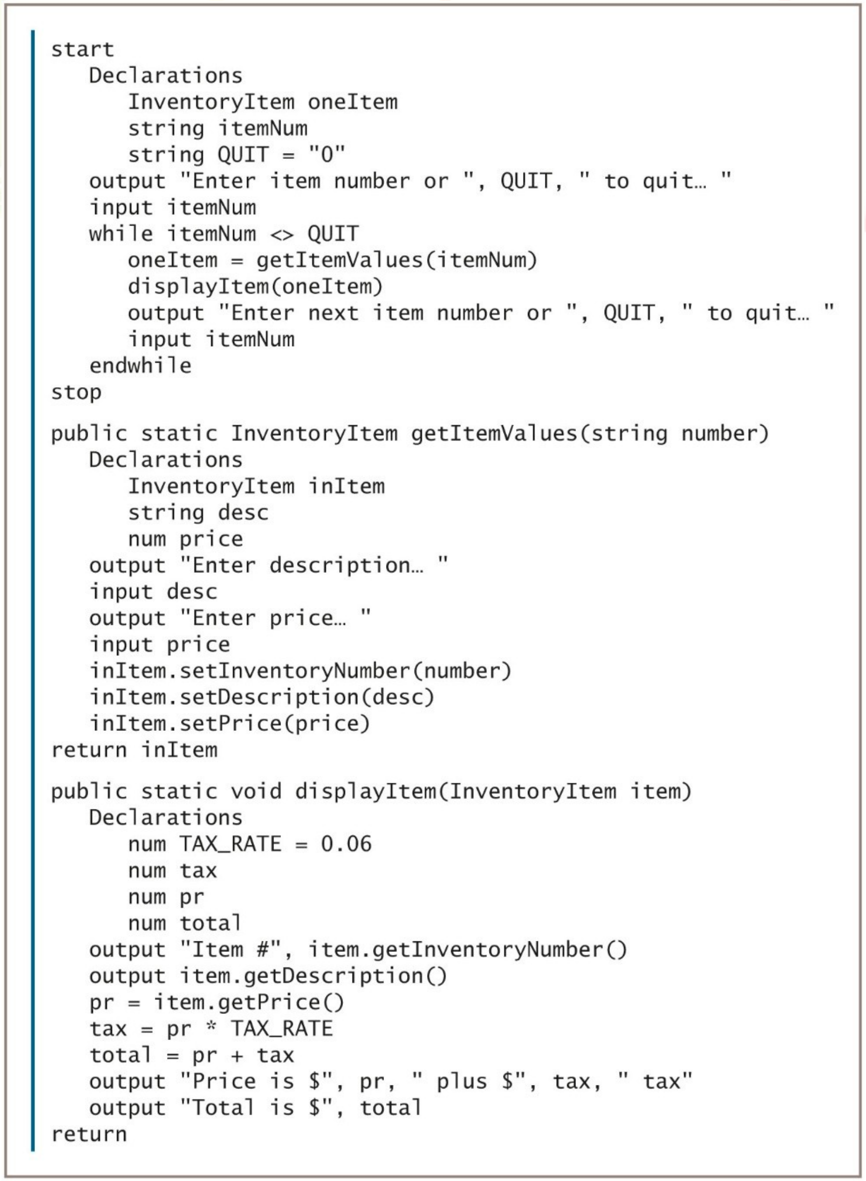
Figure 10-20 Application that declares and uses an `InventoryItem` object



```
Item #1276
Mahogany chest
Price is $450.00 plus $27.00 tax
Total is $477.00
Press any key to continue . . .
```

Figure 10-21 Execution of application in Figure 10-20

Returning an Object from a Method

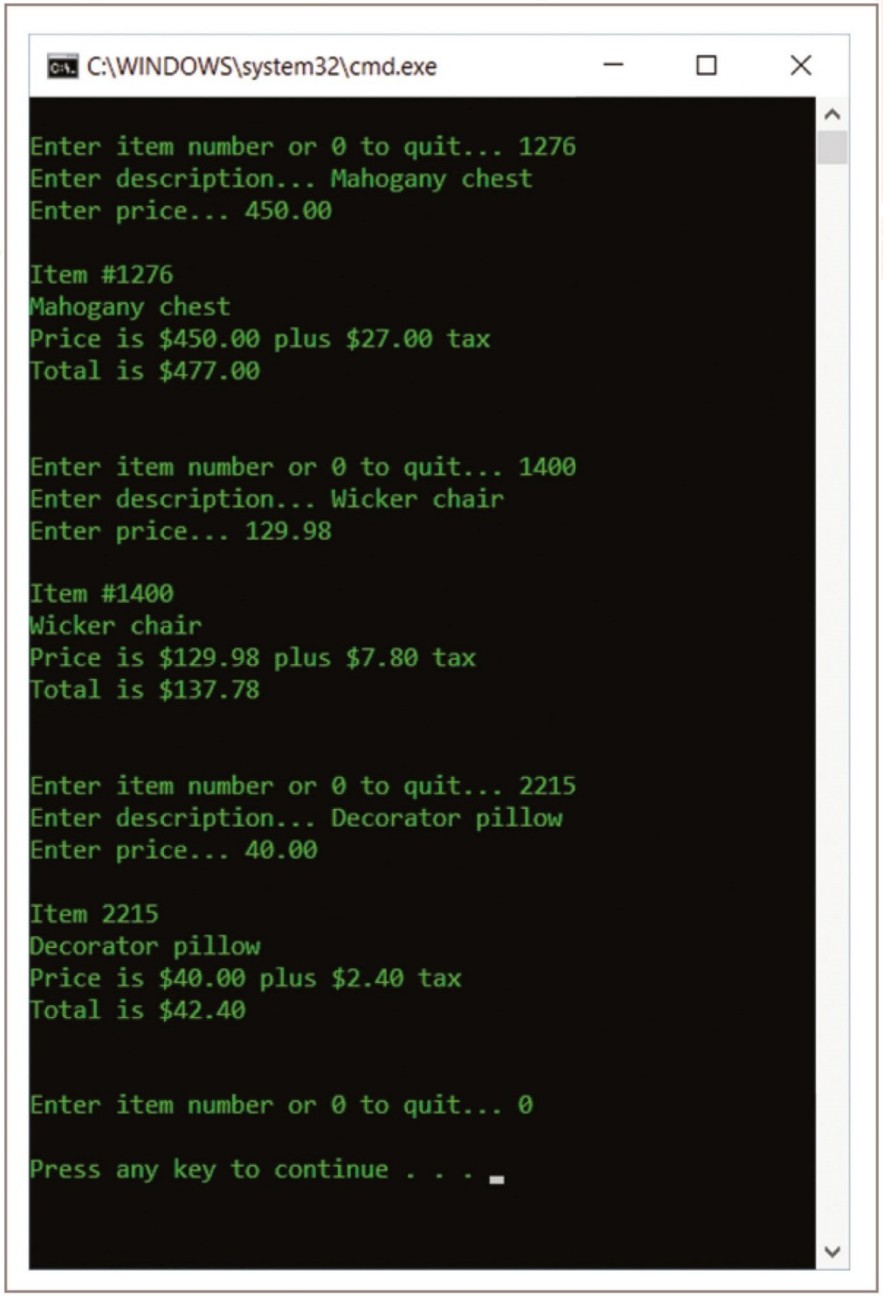


```
start
   Declarations
      InventoryItem oneItem
      string itemNum
      string QUIT = "0"
   output "Enter item number or ", QUIT, " to quit… "
   input itemNum
   while itemNum <> QUIT
      oneItem = getItemValues(itemNum)
      displayItem(oneItem)
      output "Enter next item number or ", QUIT, " to quit… "
      input itemNum
   endwhile
stop

public static InventoryItem getItemValues(string number)
   Declarations
      InventoryItem inItem
      string desc
      num price
   output "Enter description… "
   input desc
   output "Enter price… "
   input price
   inItem.setInventoryNumber(number)
   inItem.setDescription(desc)
   inItem.setPrice(price)
return inItem

public static void displayItem(InventoryItem item)
   Declarations
      num TAX_RATE = 0.06
      num tax
      num pr
      num total
   output "Item #", item.getInventoryNumber()
   output item.getDescription()
   pr = item.getPrice()
   tax = pr * TAX_RATE
   total = pr + tax
   output "Price is $", pr, " plus $", tax, " tax"
   output "Total is $", total
return
```

Figure 10-22 Application that uses `InventoryItem` objects

Returning an Object from a Method

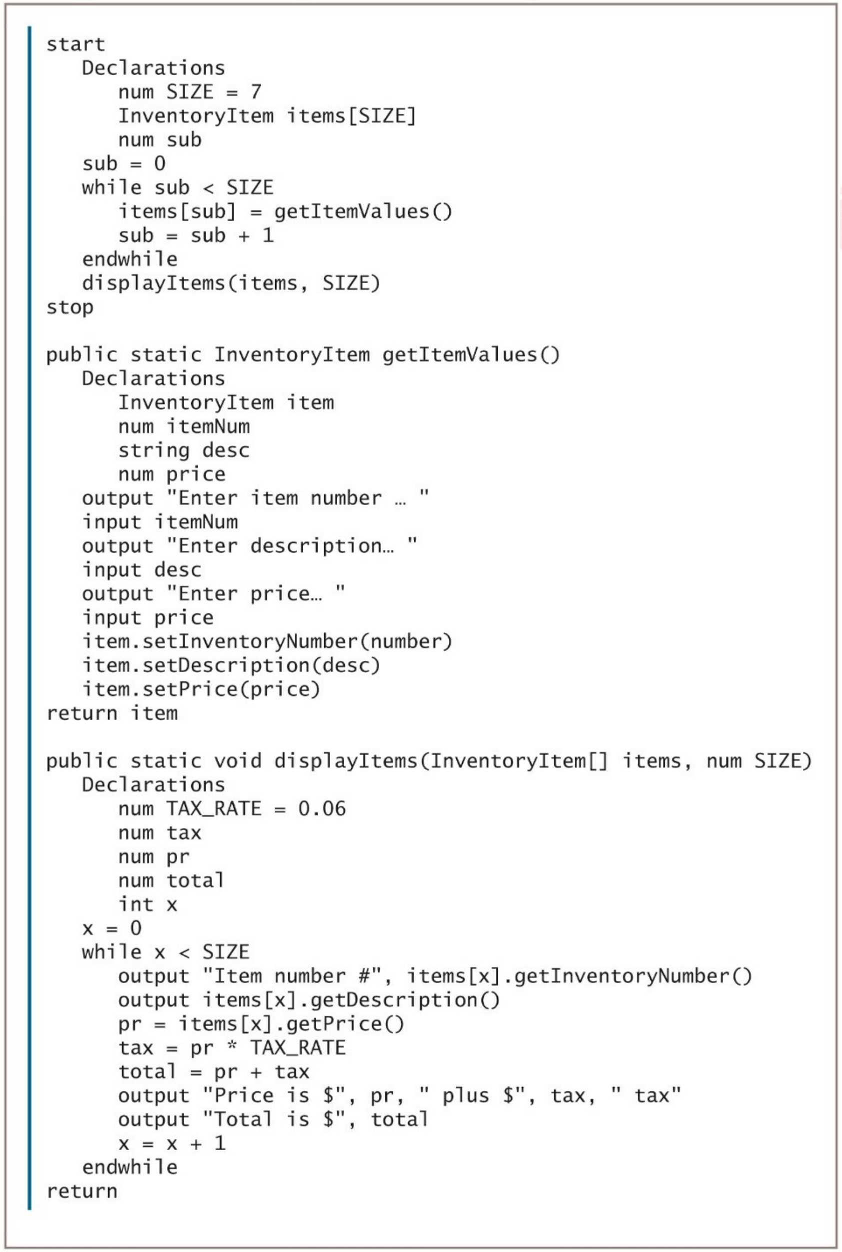
(continued)



```
Enter item number or 0 to quit... 1276
Enter description... Mahogany chest
Enter price... 450.00

Item #1276
Mahogany chest
Price is $450.00 plus $27.00 tax
Total is $477.00


Enter item number or 0 to quit... 1400
Enter description... Wicker chair
Enter price... 129.98

Item #1400
Wicker chair
Price is $129.98 plus $7.80 tax
Total is $137.78


Enter item number or 0 to quit... 2215
Enter description... Decorator pillow
Enter price... 40.00

Item 2215
Decorator pillow
Price is $40.00 plus $2.40 tax
Total is $42.40


Enter item number or 0 to quit... 0

Press any key to continue . . .
```

Figure 10-23 Typical execution of program in Figure 10-22

Using Arrays of Objects



```
start
   Declarations
      num SIZE = 7
      InventoryItem items[SIZE]
      num sub
   sub = 0
   while sub < SIZE
      items[sub] = getItemValues()
      sub = sub + 1
   endwhile
   displayItems(items, SIZE)
stop

public static InventoryItem getItemValues()
   Declarations
      InventoryItem item
      num itemNum
      string desc
      num price
   output "Enter item number … "
   input itemNum
   output "Enter description… "
   input desc
   output "Enter price… "
   input price
   item.setInventoryNumber(number)
   item.setDescription(desc)
   item.setPrice(price)
return item

public static void displayItems(InventoryItem[] items, num SIZE)
   Declarations
      num TAX_RATE = 0.06
      num tax
      num pr
      num total
      int x
   x = 0
   while x < SIZE
      output "Item number #", items[x].getInventoryNumber()
      output items[x].getDescription()
      pr = items[x].getPrice()
      tax = pr * TAX_RATE
      total = pr + tax
      output "Price is $", pr, " plus $", tax, " tax"
      output "Total is $", total
      x = x + 1
   endwhile
return
```

Figure 10-24 Application that uses an array of `InventoryItem` objects

Summary

- Classes
  - Basic building blocks of object-oriented programming
- Class definition
  - A set of program statements that tell you the characteristics of the class’s objects and the methods that can be applied to its objects
- Object-oriented programmers
  - Specify that their data fields will have private access
- As classes get more complex, organizing becomes more important

Summary (continued -1)

- Instance method operates correctly yet differently for every object instantiated from a class
- A class can contain two types of methods:
  - Static methods, which are also known as class methods and do not receive a this reference as an implicit parameter
  - Nonstatic methods, which are instance methods and do receive a this reference implicitly

Summary (continued -2)

- Objects can be used in many of the same ways you use items of simpler data types, such as passing them to and from methods and storing them in arrays