

INTRODUCTION TO INHERITANCE

In this chapter, you will:

- ◆ Learn about the concept of inheritance
- ◆ Extend classes
- ◆ Override superclass methods
- ◆ Work with superclasses that have constructors
- ◆ Use superclass constructors that require arguments
- ◆ Access superclass methods
- ◆ Learn about information hiding
- ◆ Use methods you cannot override

You look exhausted,” Lynn Greenbrier says to you late one Friday afternoon.

“I am,” you reply. “Now that I know some Java, I am writing program after program for several departments in the company. It’s fun, but it’s a lot of work, and the worst thing is that I seem to be doing a lot of the same work over and over.”

“What do you mean?” Lynn asks.

“Well, the Event Planning Department asked me to develop several classes that will hold information for every event type handled by Event Handlers. There are weekday and weekend events, events with or without dinners, and events with or without guest speakers. Sure, these various types of events have differences, but all events have many things in common, such as an event number and a number of guests.”

“I see,” Lynn says. “You’d like to create a class based on an existing class, just by adding the specific new components the new class needs. You want to avoid rewriting the components that you already created.”

“Exactly!” you say. “But, since I can’t do that, I guess I’ll just have to get back to work.”

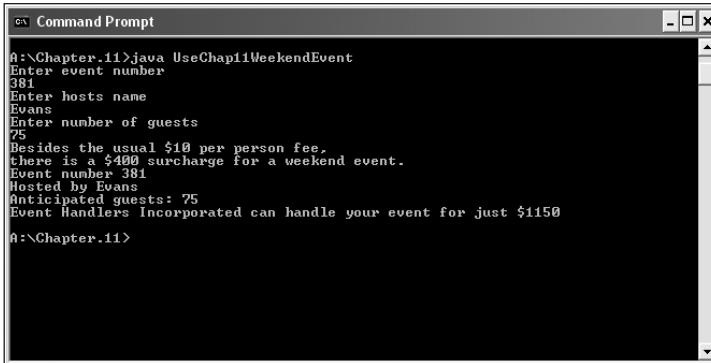
“Go home and relax!” Lynn says. “On Monday morning, I’ll teach you how to use inheritance to solve these problems.”

PREVIEWING AN EXAMPLE OF INHERITANCE

Weekend events hosted by Event Handlers Incorporated cost more than weekday events because staff members who work at the events are paid overtime rates. Weekend events have all the components of all other events, but they also possess this surcharge. You can use compiled versions of the `Chap11Event`, `Chap11WeekendEvent`, and `UseChap11WeekendEvent` class files that are saved in the `Chapter.11` folder on your Student Disk to run the `UseChap11WeekendEvent` program.

To use the `Chap11WeekendEvent` class:

1. Go to the command prompt for the `Chapter.11` folder on your Student Disk, type **java UseChap11WeekendEvent**, and then press **[Enter]**. This program allows you to supply data for a weekend event. The prompts will ask you for an event number, host name, and number of guests. You can supply any answers you want to these questions. The data you enter will echo to the screen, and you will see the charge for the event. The charge is calculated at \$10 per person, plus a \$400 surcharge for the weekend. Figure 11-1 shows a typical program run.



```
A:\Chapter.11>java UseChap11WeekendEvent
Enter event number
381
Enter hosts name
Evans
Enter number of guests
75
Besides the usual $10 per person fee,
there is a $400 surcharge for a weekend event.
Event number 381
Hosted by Evans
Anticipated guests: 75
Event Handlers Incorporated can handle your event for just $1150
A:\Chapter.11>
```

Figure 11-1 Output of the `UseChap11WeekendEvent` program

2. Open your text editor, and then open the **UseChap11WeekendEvent.java** file, or examine the code shown in Figure 11-2. The program creates a `Chap11WeekendEvent` object named `anEvent`, and then calls five methods.
3. Open the **Chap11WeekendEvent.java** file in your text editor, or examine the code shown in Figure 11-3. The class `Chap11WeekendEvent` does not contain any data fields, and it contains only one method, named `computePrice()`. However, when you ran the program `UseChap11WeekendEvent`, you provided input for several data fields. The program called several methods, and displayed several lines of output. The additional fields and methods were available because the `WeekendEvent` class inherited its additional components. You will create similar classes in this chapter.

```
public class UseChapter11WeekendEvent
{
    public static void main(String args[]) throws Exception
    {
        Chap11WeekendEvent anEvent = new Chapter11WeekendEvent();
        anEvent.setEventNum();
        anEvent.setEventHost();
        anEvent.setnumGuests();
        anEvent.computePrice();
        anEvent.printDetails();
    }
}
```

Figure 11-2 UseChap11WeekendEvent.java program

```
public class Chapter11WeekendEvent extends Chap11Event
{
    public void computePrice()
    {
        super.computePrice();
        quotedPrice += 400;
        System.out.println
            ("Besides the usual $10 per person fee,");
        System.out.println
            ("there is a $400 surcharge for a weekend event.");
    }
}
```

Figure 11-3 Chap11WeekendEvent class

LEARNING ABOUT THE CONCEPT OF INHERITANCE

Inheritance is the principle that allows you to apply your knowledge of a general category to more-specific objects. In Java, **inheritance** is a mechanism that enables one class to inherit both the behavior and the attributes of another class. You are familiar with the concept of inheritance from all sorts of nonprogramming situations.



In Chapter 3, you first learned about inheritance, where a class object can inherit all the attributes of an existing class. A functional new class can be created simply by indicating how it is different from the class from which it is derived.

When you use the term inheritance, you might think of genetic inheritance. You know from biology that your blood type and eye color are the product of inherited genes; you can say that many facts about you—or your data fields—are inherited. Similarly, you often can attribute your behaviors to inheritance. For example, your attitude toward saving money might be the same as your grandma's, and the odd way that you pull on your ear when you are tired might also be what your Uncle Steve does—thus your methods are inherited, too.

You also might choose plants and animals based on inheritance. You plant impatiens next to your house because of your shady street location; you adopt a Doberman pinscher because you need a watchdog. Every individual plant and pet has slightly different characteristics, but within a species, you can count on many consistent inherited attributes and behaviors. Similarly, the classes you create in object-oriented programming languages can inherit data and methods from existing classes. When you create a class by making it inherit from another class, you are provided with data fields and methods automatically.

From the first chapter of this book, you have been creating classes and instantiating objects that are members of those classes. For example, consider the simple `Employee` class shown in Figure 11-4. The class contains two data fields, `empNum` and `empSal`, and four methods, a `get` and `set` method for each field.

```
public class Employee
{
    private int empNum;
    private double empSal;
    public int getEmpNum()
    {
        return empNum;
    }
    public double getEmpSal()
    {
        return empSal;
    }
    public void setEmpNum(int num)
    {
        empNum = num;
    }
    public void setEmpSal(double sal)
    {
        empSal = sal;
    }
}
```

Figure 11-4 Employee class

After you create the `Employee` class, you can create specific `Employee` objects, such as `Employee receptionist = new Employee();` and `Employee deliveryPerson = new Employee();`. These `Employee` objects can eventually possess different numbers and salaries, but because they are `Employee` objects, you know that each `Employee` has *some* number and salary.

Suppose you hire a new `Employee` named `serviceRep`. A `serviceRep` object requires an employee number and a salary, but a `serviceRep` object also requires a data field to indicate territory served. You can create a new class with a name such as `EmployeeWithTerritory`, and provide the class three fields (`empNum`, `empSal`, and `empTerritory`) and six methods (`get` and `set` methods for each of the three fields).

However, when you do this, you are duplicating much of the work that you have already done for the `Employee` class. The wise, efficient alternative is to create the class `EmployeeWithTerritory` so it inherits all the attributes and methods of `Employee`. Then, you can add just the one field and two methods that are additions within `EmployeeWithTerritory` objects. Figure 11-5 shows a diagram of this relationship.

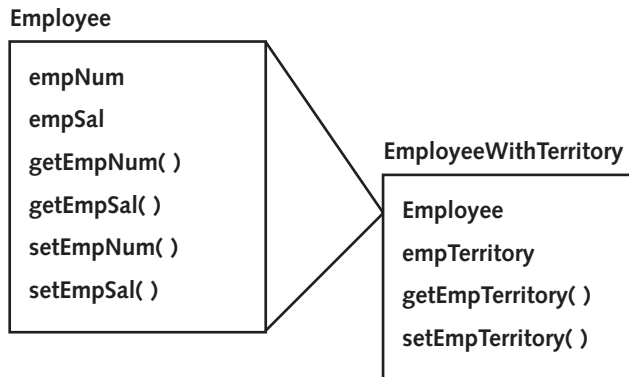


Figure 11-5 `EmployeeWithTerritory` class inherits from `Employee` class

When you use inheritance to create the `EmployeeWithTerritory` class, you:

- Save time, because the `Employee` fields and methods already exist
- Reduce errors, because the `Employee` methods already have been used and tested
- Ease understanding, because you have used the `Employee` methods on simpler objects and already understand how they work

The ability to use inheritance in the Java programming language makes programs easier to write, less error prone, and easier to understand. Imagine that besides creating `EmployeeWithTerritory`, you also want to create several other specific `Employee` classes (perhaps `EmployeeEarningCommission` including a commission rate, or `DismissedEmployee` including a reason for dismissal). By using inheritance, you can develop each new class correctly and more quickly.



The concept of a class inheritance is useful because it makes class code reusable.

A class that is used as a basis for inheritance, such as `Employee`, is called a **base class**. When you create a class that inherits from a base class (such as `EmployeeWithTerritory`), it is a **derived class**. When confronted with two classes that inherit from each other, you can tell which class is the base class and which class is the derived class by using the two classes in a sentence with the phrase “is a”. A derived class always “is a” case or instance of the more general base class. For example, a `Tree` class may be a base class to

an Evergreen class. An Evergreen “is a” Tree, so Tree is the base class; however, it is not true for all Trees that “a Tree is an Evergreen”. Similarly, an EmployeeWithTerritory “is an” Employee—but not the other way around—so Employee is the base class.

You can use the terms **superclass** and **subclass** as synonyms for base class and derived class. Thus, Evergreen can be called a subclass of the Tree superclass. You can also use the terms **parent class** and **child class**. An EmployeeWithTerritory is a child to the Employee parent. Use the pair of terms with which you are most comfortable; all of these terms will be used interchangeably throughout the book.

As an alternate way to discover which of two classes is the base class or subclass, you can try saying the two class names together. When people say their names together, they state the more-specific name before the all-encompassing family name, as in “Ginny Kroening”. Similarly, with classes, the order that “makes more sense” is the child-parent order. “Evergreen Tree” makes more sense than “Tree Evergreen”, thus Evergreen is the child class.

Finally, you usually can distinguish superclasses from their subclasses by size. Although it is not required, in general a subclass is larger than a superclass because it usually has additional fields and methods. A subclass description might look small, but any subclass contains all the fields and methods of its superclass, as well as the new, more-specific fields and methods you add to that subclass.

EXTENDING CLASSES

You use the keyword **extends** to achieve inheritance within the Java programming language. For example, the class header `public class EmployeeWithTerritory extends Employee` creates a superclass-subclass relationship between Employee and EmployeeWithTerritory. Each EmployeeWithTerritory automatically receives the data fields and methods of the superclass Employee; you then add new fields and methods to the newly created subclass. Figure 11-6 shows an EmployeeWithTerritory class.



You used the phrase **extends JApplet** throughout Chapters 9 and 10. Every Swing applet that you write is a child of the JApplet class.

When you write a program that instantiates an object using the statement `EmployeeWithTerritory northernRep = new EmployeeWithTerritory();`, then you can use any of the following statements to get field values for northernRep:

- `northernRep.getEmpNum();`
- `northernRep.getEmpSal();`
- `northernRep.getTerritoryNum();`

```
public class EmployeeWithTerritory extends Employee
{
    private int territoryNum;
    public int getTerritoryNum()
    {
        return territoryNum;
    }
    public void setTerritoryNum(int num)
    {
        territoryNum = num;
    }
}
```

Figure 11-6 EmployeeWithTerritory class

The northernRep object has access to all three methods—two methods that it inherits from Employee and one method that belongs to EmployeeWithTerritory.

Similarly, any of the following statements are legal:

- `northernRep.setEmpNum(915);`
- `northernRep.setEmpSal(210.00);`
- `northernRep.setTerritoryNum(5);`

Inheritance is a one-way proposition; a child inherits from a parent, not the other way around. If a program instantiates an Employee object, as in `Employee aClerk = new Employee;`, then the Employee object does not have access to the EmployeeWithTerritory methods. Employee is the parent class, and aClerk is an object of the parent class. It makes sense that a parent class object does not have access to its child's data and methods. When you create the parent class, you will not know how many future subclasses there might be, or what their data or methods might look like. In addition, subclasses are more specific. An Orthodontist class and Periodontist class are children of the Dentist class. You do not expect all members of the general parent class Dentist to have the Orthodontist's `braces()` method or the Periodontist's `deepClean()` method. However, Orthodontist objects and Periodontist objects have access to the more general Dentist methods `takeXRays()` and `billPatients()`.

Next you will create a working example of inheritance. You will create this example in four stages:

1. First, you will create a general Event class for Event Handlers Incorporated. This Event class will be small—it will hold just one data field and two methods.
2. After you create the general Event class, you will write a program to demonstrate its use.
3. Then you will create a more-specific DinnerEvent subclass that inherits the attributes of the Event class.

4. Finally, you will modify the demonstration program to add an example using the `DinnerEvent` class.

To create the general Event class:

1. Open a new file in your text editor, and then enter the following first few lines for a simple Event class. The class will host one integer data field—the number of guests expected at the event.

```
public class Event
{
    private int eventGuests;
```

2. To the Event class, add the following method that displays the number of eventGuests:

```
public void printEventGuests()
{
    System.out.println("Event guests: " + eventGuests);
}
```

3. Add a second method that prompts the user for the number of guests and stores the response in the eventGuests field. Begin by typing the following method header, which includes the `throws Exception` clause that handles data entry:

```
public void setEventGuests() throws Exception
{
```



You first learned about `throws Exception` in Chapter 5.

4. Enter the following code to add an integer variable to hold each character as it is read from the keyboard and a String variable to which you will add each numeric character the user enters:

```
char inChar;
String guestsString = new String("");
```

5. Prompt the user for the guest number and read the response using the following code:

```
System.out.print
("Enter the number of guests at your event ");
inChar = (char)System.in.read();
```

6. Enter the following while loop. While the user continues to enter digits, add each to a String.

```
while(inChar >= '0' && inChar <= '9')
{
```



```

    guestsString = guestsString + inChar;
    inChar = (char)System.in.read();
}

```

7. When the user finishes entering digits, use the `parseInt()` method to assign the `String` value to the `eventGuests` data field. In addition, add one more `read()` statement to absorb the extra byte from [Enter].

```

    eventGuests = Integer.parseInt(guestsString);
    System.in.read();
}

```

8. Save the file as **Event.java** in the Chapter.11 folder on your Student Disk. At the command prompt, compile the class using the **javac Event.java** command. If necessary, correct any errors and compile again.

Now that you have created a class, you can use it in an application or Swing applet. A very simple application creates an `Event` object, sets a value for the data field, and then displays the results.

To write a simple application that uses the `Event` class:

1. Open a new file in your text editor.
2. Write a `UseSimpleEvent` program that has one method—a `main()` method. Enter the following `main()` method, which declares an `Event` object, supplies it with a value, and then prints the value. Add a closing curly brace for the class:

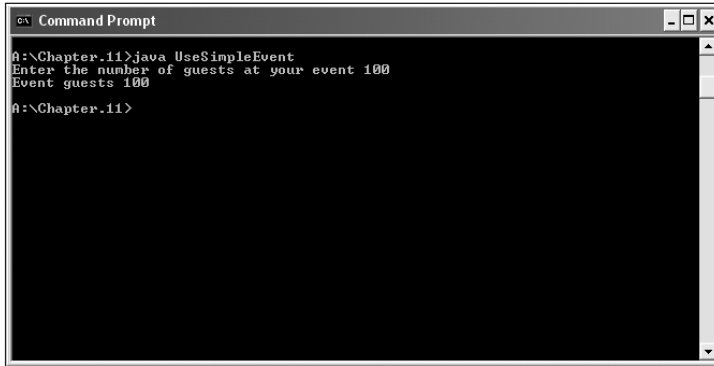
```

public class UseSimpleEvent
{
    public static void main(String[] args) throws Exception
    {
        Event anEvent = new Event();
        anEvent.setEventGuests();
        anEvent.printEventGuests();
    }
}

```

3. Save the file as **UseSimpleEvent.java** in the Chapter.11 folder on your Student Disk. Compile the program using the **javac UseSimpleEvent.java** command. After the program compiles without errors, run the program by typing **java UseSimpleEvent**, type **100** and press [Enter], type **Evans** and press [Enter], then type **75** and press [Enter]. The program's output appears in Figure 11-7.

Next you will create a new class named `DinnerEvent`. A `DinnerEvent` “is a” type of `Event` at which dinner is served, so `DinnerEvent` will be a child class of `Event`.



```

C:\Chapter.11>java UseSimpleEvent
Enter the number of guests at your event 100
Event guests 100
A:\Chapter.11>

```

Figure 11-7 Output of the UseSimpleEvent program

To create a DinnerEvent class that extends Event:

1. Open a new file in your text editor, and type the following header for the DinnerEvent class:
2. A DinnerEvent contains a number of guests, but you do not have to define the variable here. The variable is already defined in Event, which is the superclass of this class. You only need to add any variables that are particular to a DinnerEvent. Enter the following code to add a character to hold the dinner menu choice, which will be b or c (for beef or chicken) for each DinnerEvent object: **char dinnerChoice;**
3. The Event class already contains methods to set and print the number of guests, so DinnerEvent only needs methods to print and set the dinnerChoice variable. To keep this example simple, you will not validate the input character to ensure that it is b or c; you can add this improvement to the method later. The printDinnerChoice() method will assume that if the choice is not beef, it must be chicken. Type the printDinnerChoice() method as follows:

```

public void printDinnerChoice()
{
    if(dinnerChoice == 'b')
        System.out.println("Dinner choice is beef");
    else
        System.out.println("Dinner choice is chicken");
}

```

4. Enter the following setDinnerChoice() method, which prompts the user for the choice of entrées at the event, and then add a closing curly brace for the class:

```

public void setDinnerChoice() throws Exception
{
    System.out.println("Enter dinner choice");
}

```

```
        System.out.print("b for beef, c for chicken ");
        dinnerChoice = (char)System.in.read();
        System.in.read(); System.in.read();
    }
}
```

5. Save the file as **DinnerEvent.java** in the Chapter.11 folder on your Student Disk, and then compile it.

Now you can modify the UseSimpleEvent program so that it creates a DinnerEvent as well as a plain Event.

To modify the UseSimpleEvent program:

1. Open the **UseSimpleEvent.java** file in your text editor.
2. Change the class name from UseSimpleEvent to **UseDinnerEvent**.
3. Position your insertion point at the end of the line that constructs anEvent, and then press **[Enter]** to start a new line. Type the following println() statement so that when you run the program you will know that you are using the Event class to create the event:

```
System.out.println("A plain event");
```

4. Position your insertion point at the end of the line that prints the event guests (just before the closing curly braces), and then press **[Enter]** to start a new line. Add the following two new statements—one constructs a Dinner event, and the other prints a line so that when you run the program you will understand you are creating a DinnerEvent:

```
DinnerEvent aDinnerEvent = new DinnerEvent();
System.out.println("An event with dinner");
```

5. Add the following method calls to set the number of guests and dinner choice for the DinnerEvent object. Even though the DinnerEvent class does not contain a setEventGuests() method, its parent class (Event) does, so aDinnerEvent can use the setEventGuests() method.

```
aDinnerEvent.setEventGuests();
aDinnerEvent.setDinnerChoice();
```

6. Enter the following code to call the methods that print the entered data:

```
aDinnerEvent.printEventGuests();
aDinnerEvent.printDinnerChoice();
```

7. Save the file as **UseDinnerEvent.java** in the Chapter.11 folder on your Student Disk. Compile the program and run it using the values shown in Figure 11-8. The DinnerEvent object successfully uses the data field and methods of its superclass, as well as its own data field and methods.

```

A:\Chapter.11>java UseDinnerEvent
A plain event
Enter the number of guests at your event 60
Event guests 60
An event with dinner
Enter the number of guests at your event 40
Enter dinner choice
b for beef, c for chicken b
Event guests 40
Dinner choice is beef
A:\Chapter.11>_

```

Figure 11-8 Output of the UseDinnerEvent program

OVERRIDING SUPERCLASS METHODS

When you create a new subclass by extending an existing class, the new subclass contains data and methods that were defined in the original superclass. Sometimes those superclass data fields and methods are not entirely appropriate for the subclass objects.

When you use the English language, you often use the same method name to indicate diverse meanings. For example, if you think of *MusicalInstrument* as a class, you can think of `play()` as a method of that class. If you think of various subclasses such as *Guitar* and *Drum*, you know that you carry out the `play()` method quite differently for each subclass. Using the same method name to indicate different implementations is called **polymorphism**. The word polymorphism means “many forms”; many forms of action take place, even though you use the same word to describe the action. In other words, there are many forms of the same word depending on the object associated with the word.

For another example, you can create an *Employee* superclass containing data fields such as `firstName`, `lastName`, `socialSecurityNumber`, `dateOfHire`, `rateOfPay`, and so on. The methods contained in the *Employee* class include the usual set and get methods. If your usual time period for payment to each *Employee* object is weekly, then your `getRateOfPay()` method might include a statement such as `System.out.println("Pay is " + rateOfPay + " per week");`.

Imagine your company has a few *Employees* who are not paid weekly. Maybe some are paid by the hour, and others are *Employees* whose work is contracted on a job-to-job basis. Because each *Employee* type requires different paycheck-calculating procedures, you might want to create subclasses of *Employee*, such as *HourlyEmployee* and *ContractEmployee*.

When you call the `getRateOfPay()` method for an *HourlyEmployee* object, you want the display to include the phrase “per hour”, as in “Pay is \$8.75 per hour”. When you call the `getRateOfPay()` method for a *ContractEmployee*, include “per contract” as in “Pay is \$2000 per contract”. Each class—the *Employee* superclass and the two subclasses—requires its own `getRateOfPay()` method. Fortunately, if you create

separate `getRateOfPay()` methods for each class, then each class's objects will use the appropriate method for that class.



It is important to note that each subclass method overrides any method in the parent class that has the same name and argument list.

If you could not override superclass methods, you could always create unique names for each subclass method, such as `getRateOfPayForHourly()` and `getRateOfPayForContractual()`, but the classes you create are easier to write and understand if you use one reasonable name for methods that do essentially the same thing. Because you are attempting to get the rate of pay for each object, `getRateOfPay()` is an excellent method name for all three object types.



You already have overridden methods in your Swing applets. When you write your own `init()` or `start()` method within a Swing applet, you are overriding the automatically supplied superclass version you get when you use the phrase `extends JApplet`.



If a superclass and its subclass have methods with the same name but different argument lists, you are overloading methods, and not overriding them. You learned about overloading methods in Chapter 4.

Next you will create two methods with the same name, `printHeader()`, with one version in the `Event` superclass and another in the `DinnerEvent` subclass. When you call the `printHeader()` method, the correct version will execute based on the object you use.

To demonstrate that subclass methods override superclass methods with the same name:

1. In your text editor, open the **Event.java** file in the Chapter.11 folder on your Student Disk. Change the class name from `Event` to **EventWithHeader** because this new class will contain a method that allows you to print a header line of explanatory text with each class object. Save the file as **EventWithHeader.java** in the Chapter.11 folder on your Student Disk. In addition to providing a descriptive name, changing the class name serves another purpose. By giving the class a new name, you retain the original class on your disk so you can study the differences later.
2. Position the insertion point at the end of the line that contains the closing curly brace for the `printEventGuests()` method, and then press **[Enter]** to start a new line.
3. Enter the following `printHeader()` method:

```
public void printHeader()
{
    System.out.println("Simple event: ");
}
```

4. Save the file, and then compile it.
5. Open the **DinnerEvent.java** file in the Chapter.11 folder. Change the class name from **DinnerEvent** to **DinnerEventWithHeader**, and change the class from which it extends—**Event**—to **EventWithHeader**. Save the file as **DinnerEventWithHeader.java** in the Chapter.11 folder on your Student Disk.
6. Position your insertion point at the end of the line that contains the closing curly brace for the `printDinnerChoice()` method, and then press **[Enter]** to start a new line of text. Add the following `printHeader()` method to this class:

```
public void printHeader()
{
    System.out.println("Dinner event: ");
}
```

7. Save the file, and then compile it.

You just created a `DinnerEventWithHeader` class that contains a `printHeader()` method. Then you extended the class by creating a `DinnerEvent` subclass containing a method with the same name. Now you will write a program that demonstrates that the correct method executes depending on the object.

To create a program that demonstrates that the correct `printHeader()` method executes depending on the object:

1. Open a new file in your text editor, and then enter the first few lines of a `UseEventWithHeader` class:

```
public class UseEventWithHeader
{
    public static void main(String[] args) throws Exception
    {
```

2. Enter the following code to create two objects, `EventWithHeader` and `DinnerEventWithHeader`:

```
EventWithHeader anEvent = new EventWithHeader();
DinnerEventWithHeader aDinnerEvent =
    new DinnerEventWithHeader();
```

3. Enter the following code to call the three `EventWithHeader` methods:

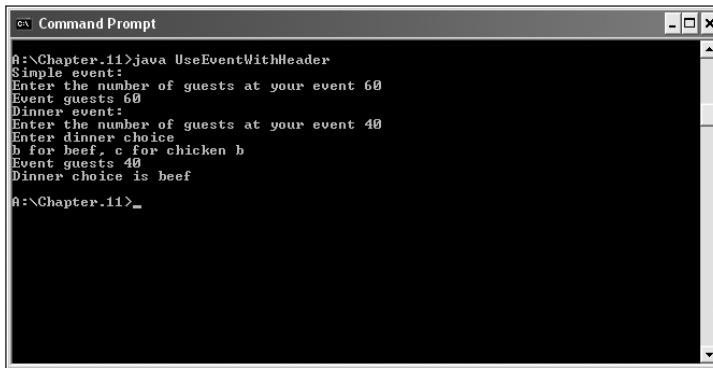
```
anEvent.printHeader();
anEvent.setEventGuests();
anEvent.printEventGuests();
```

4. Enter the following code to call the five `DinnerEventWithHeader` methods, and then add the closing curly brace for the class:

```
aDinnerEvent.printHeader();
aDinnerEvent.setEventGuests();
aDinnerEvent.setDinnerChoice();
aDinnerEvent.printEventGuests();
```

```
        aDinnerEvent.printDinnerChoice();  
    }  
}
```

5. Save the file as **UseEventWithHeader.java** in the Chapter.11 folder on your Student Disk. Compile and run the program. Input the values shown in Figure 11-9. Be sure to notice that for each type of object, the correct method executes.



```
Command Prompt  
A:\Chapter.11>java UseEventWithHeader  
Simple event:  
Enter the number of guests at your event 60  
Event guests 60  
Dinner event:  
Enter the number of guests at your event 40  
Enter dinner choice  
b for beef, c for chicken b  
Event guests 40  
Dinner choice is beef  
A:\Chapter.11>
```

Figure 11-9 Output of the UseEventWithHeader program

WORKING WITH SUPERCLASSES THAT HAVE CONSTRUCTORS

When you create any object, as in `SomeClass anObject = new SomeClass();`, you are calling a class constructor method that has the same name as the class itself. When you instantiate an object that is a member of a subclass, you are actually calling at least two constructors: the constructor for the base class and the constructor for the extended, derived class. When you create any subclass object, the superclass constructor must execute first, and *then* the subclass constructor executes.

In the examples of inheritance so far in this chapter, each class contained default constructors, so their execution was transparent. However, you should realize that when you create an object using `HourlyEmployee clerk = new HourlyEmployee();` (where `HourlyEmployee` is a subclass of `Employee`), *both* the `Employee()` and `HourlyEmployee()` constructors execute.

You can create a class whose constructor does nothing but print a message. Then, when you extend the class, you can create a subclass constructor that prints a different message. When the program is run, both classes will print messages confirming that the constructor in each class is called.

To demonstrate that a subclass constructor calls the superclass constructor first:

1. Open a new file in your text editor.

2. Create the following superclass, whose constructor prints a message on the screen:

```
public class ASuperClass
{
    public ASuperClass()
    {
        System.out.println("In superclass constructor");
    }
}
```

3. Save the file as **ASuperClass.java** in the Chapter.11 folder on your Student Disk, and then compile it.
4. Open a new file in your text editor, and then enter the following program to create a derived subclass that extends the superclass. The constructor of the subclass also prints a message.

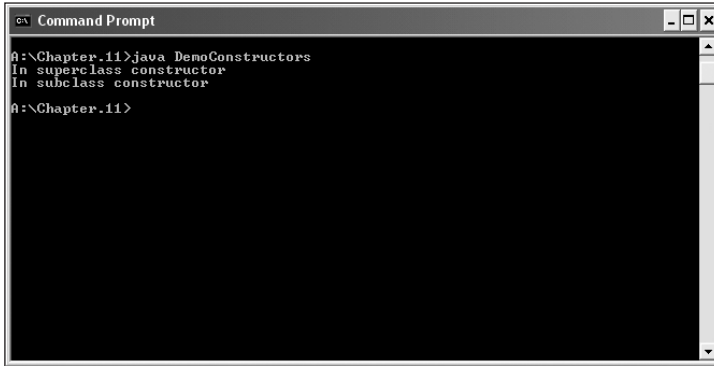
```
public class ASubClass extends ASuperClass
{
    public ASubClass()
    {
        System.out.println("In subclass constructor");
    }
}
```

5. Save the file as **ASubClass.java** in the Chapter.11 folder on your Student Disk, and then compile it.
6. Open a new text file and enter the following program that does just one thing—it creates a child class object:

```
public class DemoConstructors
{
    public static void main(String[] args)
    {
        ASubClass child = new ASubClass();
    }
}
```

7. Save this file as **DemoConstructors.java** in the Chapter.11 folder on your Student Disk. Compile the file, and then execute the program. The output appears as shown in Figure 11-10. Even though you create only one subclass object, two separate messages print—one from the superclass constructor and one from the subclass constructor.

Of course, most constructors perform many more tasks than printing a message to inform you that they exist. When constructors initialize variables, you usually want the superclass constructor to take care of initializing the data fields that originate in the superclass. The subclass constructor only needs to initialize the data fields that are specific to the subclass.



```
Command Prompt
A:\Chapter.11>java DemoConstructors
In superclass constructor
In subclass constructor
A:\Chapter.11>
```

Figure 11-10 Output of the DemoConstructors program

Next you will add a constructor to the Event class you created for Event Handlers Incorporated. When you instantiate a subclass DinnerEvent object, the superclass Event constructor will execute.

To add a constructor to the Event class:

1. Open the **EventWithHeader.java** file in your text editor. Use your text editor's Save As command to save the file as **EventWithConstructor.java** in the Chapter.11 folder on your Student Disk. Be sure to change the class name from EventWithHeader to **EventWithConstructor**.
2. Position your insertion point to the right of the statement that declares the eventGuests data field, and then press **[Enter]** to start a new line. Type the following constructor that initializes the number of guests to zero:

```
public EventWithConstructor()
{
    eventGuests = 0;
}
```

3. Save the file and compile it.
4. In your text editor, open the **DinnerEventWithHeader.java** file from the Chapter.11 folder. Change the class header so that both the class name and the parent class name read as follows:

```
public class DinnerEventWithConstructor
    extends EventWithConstructor
```

5. Save the file as **DinnerEventWithConstructor.java** in the Chapter.11 folder on your Student Disk, and then compile it.
6. In your text editor, open a new file so you can write a program to demonstrate the use of the base class constructor with both a base class object and an extended class object. To begin the class, you create a class header, a main()

method header, and definitions of two objects—one is a member of the base class, and the other is a member of the extended class:

```
public class UseEventsWithConstructors
{
    public static void main(String[] args)
    {
        EventWithConstructor anEvent =
            new EventWithConstructor();
        DinnerEventWithConstructor aDinnerEvent =
            new DinnerEventWithConstructor();
```

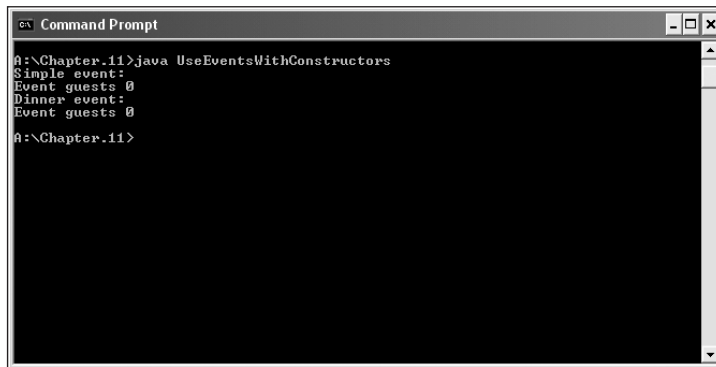
7. Add the following statements that print a header and the number of guests for the parent class member:

```
anEvent.printHeader();
anEvent.printEventGuests();
```

8. Add statements that print a header and the number of guests for the child class member, and then add a closing curly brace for the class:

```
aDinnerEvent.printHeader();
aDinnerEvent.printEventGuests();
    }
}
```

9. Save the program as **UseEventsWithConstructors.java** in the Chapter.11 folder on your Student Disk. Then compile and run the program. The output is shown in Figure 11-11. The guest number is initialized correctly for objects of both classes.



```
c:\ Command Prompt
A:\Chapter.11>java UseEventsWithConstructors
Simple event:
Event guests: 0
Dinner event:
Event guests: 0
A:\Chapter.11>
```

Figure 11-11 Output of the UseEventsWithConstructors program

USING SUPERCLASS CONSTRUCTORS THAT REQUIRE ARGUMENTS

When you create a class and do not provide a constructor, Java automatically supplies you with one that never requires arguments. When you write your own constructor, you replace the automatically supplied version. Depending on your needs, the constructor you create for a class might require arguments. When you use a class as a superclass, and the class has a constructor that requires arguments, then you must make sure that any subclasses provide the superclass constructor with what it needs.



Don't forget that a class can have many constructors. As soon as you create at least one constructor for a class, you no longer can use an automatic version.

When a superclass constructor requires arguments, you must include a constructor for each subclass you create. Your subclass constructor can contain any number of statements, but the first statement within the constructor must call the superclass constructor. Even if you have no other reason to create a subclass constructor, you must write the subclass constructor so it can call its superclass' constructor.

The format of the statement that calls a superclass constructor is `super(list of arguments);`. The keyword **super** always refers to the superclass of the class in which you use it. Suppose that you create an `Employee` class with a constructor that requires three arguments—a character, a double, and an integer—and you create an `HourlyEmployee` class that is a subclass of `Employee`. The following code shows a valid constructor for `HourlyEmployee`:

```
public HourlyEmployee()  
{  
    super('P', 12.35, 40);  
    // Other statements can go here  
}
```

The `HourlyEmployee` constructor requires no arguments, but it passes three arguments to its superclass constructor. A different `HourlyEmployee` constructor can require arguments. It could then pass the appropriate arguments to the superclass constructor. For example:

```
public HourlyEmployee(char dept, double rate, int hours)  
{  
    super(dept, rate, hours);  
    // Other statements can go here  
}
```



Except for any comments, the `super()` statement must be the first statement in the subclass constructor. Not even data field definitions can precede it.



Although it seems that you should be able to use the superclass constructor name to call the superclass constructor, Java does not allow this. You must use the keyword `super`.

Next you will modify the `Event` class so that its constructor requires arguments. Then, to show how the superclass and subclass constructors work, you will create a subclass object that calls its superclass constructor.

To demonstrate how inheritance works when class constructors require arguments:

1. Open the **EventWithConstructor.java** file in your text editor, and then change the class name to **EventWithConstructorArg**.
2. Change the current constructor name so it matches the class name, and then change the constructor argument list so it requires an integer argument. In other words, change `public EventWithConstructor()` to `public EventWithConstructorArg(int guests)`.
3. Change the constructor statement that sets `eventGuests` to zero as follows so that it sets the event guests field to the constructor's argument value:
`eventGuests = guests;`
4. Save the file as **EventWithConstructorArg.java** in the Chapter.11 folder on your Student Disk, and then compile it.

Next you will add a constructor to the `DinnerEvent` class so it can call its parent's constructor. The child class constructor requires an integer argument, which it then passes to the parent class constructor.

To create the child class:

1. Open the **DinnerEventWithConstructor.java** file in your text editor.
2. Change the class header as follows so that the name of this class is **`DinnerEventWithConstructorArg`**, and thus inherits from `EventWithConstructorArg`:

```
public class DinnerEventWithConstructorArg
extends EventWithConstructorArg
```

3. Position your insertion point after the declaration of the `dinnerChoice` field, and then press **[Enter]** to start a new line. Create the following constructor that requires an integer argument and passes it to the superclass constructor:

```
public DinnerEventWithConstructorArg(int guests)
{
    super(guests);
}
```

4. Save the file as **DinnerEventWithConstructorArg.java** in the Chapter.11 folder on your Student Disk, and then compile it.

Now you can create a program to demonstrate creating parent and child class objects when the parent constructor needs an argument.

To create the program:

1. Open a new file in your text editor, and then enter the following first few lines of a program that demonstrates using `super()`:

```
public class UseEventsWithConstructorArg
{
    public static void main(String[] args)
    {
```

2. Enter the following code to create an `EventWithConstructorArg` object and give it a value for the number of guests:

```
        EventWithConstructorArg anEvent =
            new EventWithConstructorArg(45);
```

3. Add the following code to create a `DinnerEventWithConstructorArg` object. This constructor also requires an integer argument.

```
        DinnerEventWithConstructorArg aDinnerEvent =
            new DinnerEventWithConstructorArg(65);
```

4. Add the following statements to print explanations and guest fields for each object, and then add the closing curly brace for the class:

```
        anEvent.printHeader();
        anEvent.printEventGuests();
        aDinnerEvent.printHeader();
        aDinnerEvent.printEventGuests();
    }
}
```

5. Save the file as **UseEventsWithConstructorArg.java** in the Chapter.11 folder on your Student Disk, then compile and execute the program. The output appears in Figure 11-12. Each object is correctly initialized because the superclass constructor was correctly called in each case.

```
Command Prompt
A:\Chapter.11>java UseEventsWithConstructorsArg
Simple event:
Event guests: 45
Dinner event:
Event guests: 65
A:\Chapter.11>
```

Figure 11-12 Output of the `UseEventsWithConstructorsArg` program

ACCESSING SUPERCLASS METHODS

Earlier in this chapter, you learned that a subclass could contain a method with the same name and arguments as a method in its parent class. When this happens, using the subclass method overrides the superclass method. However, you might want to use the superclass method within a subclass. If so, you can use the keyword **super** to access the parent class method. To demonstrate, you will create a simple subclass that has a method with the same name as a method that is part of its superclass.

To access a superclass method from within a subclass:

1. Open a new file in your text editor, then create the following parent class with a single method:
2. Save the file as **AParentClass.java** in the Chapter.11 folder on your Student Disk, and then compile it.
3. Open a new text file, then create the following child class that inherits from the parent. The child has one method. The method has the same name as the parent's method, but the child can call the parent's method without conflict by using the keyword **super**.

```
public class AParentClass
{
    private int aVal;
    public void printClassName()
    {
        System.out.println("AParentClass");
    }
}
```

4. Save the file as **AChildClass.java** in the Chapter.11 folder on your Student Disk, and then compile it.
5. Finally, open a new text file and enter the following demonstration program to show that the child class can call its parent's method:

```
public class DemoSuper
{
```

```
public static void main(String[] args)
{
    AChildClass child = new AChildClass();
    child.printClassName();
}
}
```

6. Save the file as **DemoSuper.java** in the Chapter.11 folder on your Student Disk, then compile and execute the program. As the output in Figure 11-13 shows, even though the child and parent classes have methods with the same name, the child class can use the parent class method correctly by employing the keyword **super**.



Figure 11-13 Output of the DemoSuper program



You can use the keyword **this** as the opposite of **super**. For example, if a superclass and its subclass each have a method named `someMethod()`, then within the subclass, `super.someMethod()` refers to the superclass version of the method. Both `someMethod()` and `this.someMethod()` refer to the subclass version.

LEARNING ABOUT INFORMATION HIDING

The `AStudent` class shown in Figure 11-14 is a typical construction for Java classes. The keyword **private** precedes each data field, and the keyword **public** precedes each method. As a matter of fact, the four get and set methods are necessary within the `AStudent` class specifically because the data fields are **private**. Without the **public** get and set methods, there would be no way to access these private data fields.

```

public class AStudent
{
    private int idNum;
    private double semesterTuition;
    public int getIdNum()
    {
        return idNum;
    }
    public double getTuition()
    {
        return semesterTuition;
    }
    public void setIdNum(int num)
    {
        idNum = num;
    }
    public void setTuition(double amt)
    {
        semesterTuition = amt;
    }
}

```

Figure 11-14 AStudent class

When a program is a class user of AStudent (that is, it instantiates the AStudent object), then the user cannot directly alter the data in any **private** field. For example, when you write a `main()` method that creates a AStudent as `AStudent someStudent = new AStudent();`, you cannot change the AStudent's `idNum` with a statement such as `someStudent.idNum = 812;`. The `idNum` of the `someStudent` object is not accessible in the `main()` program that uses the AStudent object because `idNum` is **private**. Only methods that are part of the AStudent class itself are allowed to alter AStudent data. To alter an AStudent's `idNum`, you must use the **public** method `setIdNum()`, as in `someStudent.setIdNum(812);`.

The concept of keeping data private is known as **information hiding**. When you employ information hiding, your data can be altered only by the methods you choose and only in ways that you can control. For example, you might want the `setIdNum()` method to check to make sure the `idNum` is within a specific range of values. If a class other than the AStudent class itself could alter `idNum`, then `idNum` could be assigned a value that the AStudent class couldn't control.



You first learned about information hiding and using the **public** and **private** keywords in Chapter 3. You may want to review these concepts.

When a class serves as a superclass to other classes you create, your subclasses inherit all the data and methods of the superclass, with one exception: **private** members of the

parent class are not inherited. If you could use **private** data outside its class, you would use the advantages of information hiding. If you intend the `AStudent` class data field `idNum` to be **private**, then you don't want any outside classes using the field. If a new class could simply extend your `AStudent` class and get to its data fields without going through the proper channels, then information hiding would not be operating.

There are occasions when you want to access parent class data from within a subclass. For example, suppose you create two child classes that extend the `AStudent` class: `PartTimeStudent` and `FullTimeStudent`. If you want the subclass methods to be able to access `idNum` and `semesterTuition`, then those data fields cannot be **private**. However, if you don't want other, nonchild classes to access those data fields, then they cannot be **public**. To solve this problem, you can create the fields using the modifier **protected**. Using the keyword **protected** provides you with an intermediate level of security between public and private access. If you create a protected data field or method, it can be used within its own class or in any classes extended from that class, but it cannot be used by "outside" classes. In other words, protected members are those that can be used by a class and its descendents.

Next you will create a superclass with a protected field using the **protected** access modifier with a superclass data field so that you can access the field within a subclass method. Then you will create a method to set the number of event guests, and create a simple program to test this class.

To create a superclass with a protected field:

1. In your text editor, open the **EventWithHeader.java** file. For simplicity, you will use the file that you created before adding constructors. Change the class name to **EventWithProtectedData**.
2. Change the modifier on the `eventGuests` field from **private** to **protected**.
3. Save the file as **EventWithProtectedData.java** in the Chapter.11 folder on your Student Disk, and then compile it.
4. Open the **DinnerEventWithHeader.java** file. Change its name and its parent's name so the class header reads as follows:

```
public class DinnerEventWithProtectedData
extends EventWithProtectedData
```

Assume that Event Handlers Incorporated requires at least 10 guests for an event with dinner, but there is no minimum guest number for other event types. To ensure that DinnerEvents (unlike plain Events) have at least 10 guests, the subclass `setEventGuests()` method will override the `setEventGuests()` method in the superclass. The subclass version of the method will call the superclass method, but if the user does not enter a guest number of at least 10, the subclass method will call the superclass method again.

5. To create the subclass `setEventGuests()` method, position your insertion point at the end of the closing curly brace of the `setDinnerChoice()` method, press **[Enter]** to start a new line, and then type the following header for the method and the method's opening curly brace:

```
public void setEventGuests() throws Exception
{
```

6. Call the superclass method with the same name:

```
super.setEventGuests();
```

7. Check the value of the `eventGuests` data field using the following `while` loop. Because the field is `protected` in the base class, it can be accessed here in the derived class. If the guest number continues to be too low, issue an error message and call the superclass method.

```
while(eventGuests < 10)
{
    System.out.print("Minimum required for dinner: ");
    System.out.println("10 guests!");
    super.setEventGuests();
}
```



If `eventGuests` had not been inherited (that is, if it was still `private`), you would need to use `public getEventGuests()` to access its value.

8. Add the closing curly brace for the method, save the file as **DinnerEventWithProtectedData.java** in the Chapter.11 folder on your Student Disk, and then compile it.
9. Next, create a simple program to test this class. Open a new file in your text editor and then enter the following first few lines of a demonstration program:

```
public class UseProtected
{
    public static void main(String[] args)
        throws Exception
    {
```

10. Create the `aDinnerEvent` object by entering the following code:

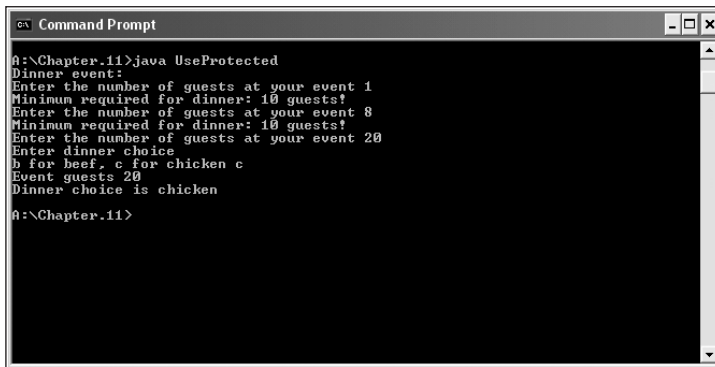
```
DinnerEventWithProtectedData aDinnerEvent =
    new DinnerEventWithProtectedData();
```

11. Using the newly created object, print an explanation, set the field values, then print the field values by entering the following code:

```
aDinnerEvent.printHeader();
aDinnerEvent.setEventGuests();
aDinnerEvent.setDinnerChoice();
```

```
aDinnerEvent.printEventGuests();  
aDinnerEvent.printDinnerChoice();
```

12. Add the closing curly brace for the `main()` method and the closing curly brace for the class, and then save the file as **UseProtected.java** in the Chapter.11 folder on your Student Disk.
13. Compile and execute the program. When you run the program, make several attempts to set the number of dinner guests to values below 10. The program will continue to prompt you until your guest number meets the required minimum. A sample program output appears in Figure 11-15.



```
A:\Chapter.11>java UseProtected  
Dinner event:  
Enter the number of guests at your event 1  
Minimum required for dinner: 10 guests!  
Enter the number of guests at your event 8  
Minimum required for dinner: 10 guests!  
Enter the number of guests at your event 20  
Enter dinner choice  
b for beef, c for chicken c  
Event guests 20  
Dinner choice is chicken  
A:\Chapter.11>
```

Figure 11-15 Sample output from the UseProtected program

USING METHODS YOU CANNOT OVERRIDE

There are four types of methods that you cannot override in a subclass:

- **private** methods
- **static** methods
- **final** methods
- Methods within **final** classes

You already know that when you create a **private** variable in a superclass, the variable is not available for use in a subclass. Similarly, if you create a **private** method in a superclass, the method is not available for use in any class extended from the superclass. You also know that a subclass can access **private** variables in the superclass through the use of nonprivate methods. Again, similarly, if a superclass has a nonprivate method that accesses a **private** method, then a child class can use the inherited nonprivate method to access the noninherited **private** method.

For example, Figure 11-16 shows a superclass named **Super** that contains two methods: one method is **public** and the other method is **private**.

```

public class Super
{
    public void printPublic()
    {
        System.out.println("This method is public");
        printPrivate();
    }
    private void printPrivate()
    {
        System.out.println("This method is private");
    }
}

```

Figure 11-16 Super class

Figure 11-17 shows a subclass that attempts to extend Super. If you compile the Sub class, you will receive the error message, “No method matching printPrivate() found in class Sub”. The statement that calls printPublic() works correctly because as a **public** method, printPublic() is inherited by the Sub class. The printPublic() method calls printPrivate(), which is perfectly legal because printPublic() and printPrivate() are methods within the same class. However, within the Sub class, because the printPrivate() method is “invisible”, the compiler tells you that it doesn’t exist. As a **private** method, printPrivate() is not inherited.

```

public class Sub extends Super
{
    public void printMessages()
    {
        printPublic();
        printPrivate();
    }
}

```

Figure 11-17 Incorrect Sub class with a printPrivate() method

Additionally, you cannot override a method that is **private** in a parent class. For example, examine the class in Figure 11-18. The Sub class extends the Super class shown in Figure 11-16 and adds its own printPrivate() method. When a main() program creates a Sub object, as in `Sub aSubObject = new Sub();`, and calls printPrivate() with `aSubObject.printMessages();`, then the screen will show “This method is private” from the Super class, and not “This will never print” from the Sub class method. This makes sense when you consider that printPublic() is a method that is part of the Super class. The printPublic() method calls the printPrivate() method from its own class. The subclass shown in Figure 11-18 compiles without error, but does so as if the printPrivate() method within the subclass does not exist.

```
public class Sub extends Super
{
    public void printMessages()
    {
        printPublic();
    }
    private void printPrivate()
    {
        System.out.println("This will never print");
    }
}
```

Figure 11-18 Sub class with a invisible method

In addition to **private** methods, you also cannot override **static** methods from a parent class. You learned in Chapter 4 that **static** methods are also called class methods and that they have no objects associated with them. You call a **static** method using the class name, not an object name. Recall that in Java the keyword **static** implies uniqueness; a **static** method is unique to the base class and all its descendants.



The **main()** methods in Java applications are always **static**.

11

Methods carrying the access modifier **final** cannot be overridden by subclass methods. For example, the statement **public final void getEmpNum()** is declared **final** by placing the modifier in the class declaration. If **getEmpNum()** is an automatically available method in a Java program, any attempt to create a method with the same name will result in an error. In Chapter 4, you learned that you can use the keyword **final** when you want to create a constant, as in **final double TAXRATE = .065**; You can also use the **final** modifier with methods when you don't want the method to be overridden. You use **static** as a method access modifier when you create class methods for which you want to prevent overriding; you use **final** as a method access modifier when you create instance methods for which you want to prevent overriding.



You can think of **private** and **static** methods as being implicitly **final**.

Because a **final** method's definition can never change, the compiler optimizes the program by removing the calls to **final** methods and replacing them with the expanded code of their definitions at each method call location. This process is called **inlining** the code. You are never aware that inlining is taking place; the compiler chooses to use this procedure to save the overhead of calling a method.



The compiler chooses to inline a `final` method only if it is a small method containing just one or two lines of code.

Finally, you can declare a class to be `final`. When you do so, all of its methods are `final`, regardless of which access modifier precedes the method name. A `final` class cannot be a parent.



Java's `Math` class, which you learned about in Chapter 4, is an example of a `final` class.

CHAPTER SUMMARY

- Inheritance is the principle that asserts that you can apply knowledge of a general category to more-specific objects. The classes you create in object-oriented programming languages can inherit data and methods from existing classes. When you create a class through inheritance, you are automatically provided with data fields and methods.
- A class that is used as a basis for inheritance is called a base class, a superclass, or a parent class. When you create a class that inherits from a base class, it is called a derived class, a subclass, or a child class. In general, a subclass is larger than a superclass because you add new, more-specific fields and methods to a subclass, as well as inherit fields and methods from the superclass.
- If you create separate methods for each subclass of a superclass, then each class's objects will use the appropriate method for that class. Each child class method overrides the method that has the same name in the parent class. Using the same method name to indicate different implementations is called polymorphism.
- When you instantiate an object that is a member of a subclass, you are actually calling two constructors: the constructor for the superclass and the constructor for the subclass. The base class constructor executes first, and then the subclass constructor executes.
- When constructors initialize variables, you usually want the base class constructor to take care of initializing the data fields that originate in the base class. The derived class constructor must initialize only the data fields that are specific to the derived class.
- When a superclass constructor requires arguments, you must create a constructor for each subclass you create. The first statement within the constructor must call the superclass constructor. The format of the statement that calls a superclass constructor is `super(list of arguments);`. The keyword `super` always refers to the superclass of the class in which you use it.

- When a program is a class user, it cannot directly alter the data in any **private** field. The concept of keeping data private is known as information hiding. Information hiding lets you control how data is used and altered.
- If you create a data field or method that uses the **protected** access modifier, then the field or method can be used within its own class or in any classes extended from that class, but cannot be used by “outside” classes.
- There are four types of methods that you cannot override in a subclass: **private**, **static**, **final**, and methods within **final** classes.
- If you create a **private** method in a superclass, the method is not available for use in any class extended from the superclass. If a superclass has a nonprivate method that accesses a **private** method, then a child class can use the inherited nonprivate method to access the noninherited private method.
- When you declare a class to be **final**, all of its methods are **final**, regardless of which access modifier precedes the method name. A **final** class cannot be a parent.

REVIEW QUESTIONS

1. _____ as an alternate way(s) to discover which of two classes is the base class or subclass.
 - a. Look at the class size
 - b. Try saying the two class names together
 - c. Use polymorphism
 - d. Both a and b are correct.
2. Employing inheritance reduces errors because _____.
 - a. the new classes have access to fewer data fields
 - b. the new classes have access to fewer methods
 - c. you can copy methods that you already created
 - d. many of the methods you need have already been used and tested
3. A base class can also be called a _____.
 - a. child class
 - b. subclass
 - c. derived class
 - d. superclass

4. Which of the following choices most closely describes a parent class/child class relationship?
 - a. Rose/Flower
 - b. Present/Gift
 - c. Dog/Poodle
 - d. Sparrow/Bird
5. The Java keyword that creates inheritance is _____.
 - a. `static`
 - b. `enlarge`
 - c. `extends`
 - d. `inherits`
6. A class named Building has a method named `getFloors()`. If School is a child class of Building, and ModelHigh is an object of type School, then which of the following statements is valid?
 - a. `Building.getFloors();`
 - b. `School.getFloors();`
 - c. `ModelHigh.getFloors();`
 - d. All of the above statements are valid.
7. Which of the following statements is false?
 - a. A child class inherits from a parent class.
 - b. A parent class inherits from a child class.
 - c. Both of the above statements are false.
 - d. Neither of the above statements is false.
8. When a subclass method has the same name and argument types as a superclass method, the subclass method can _____ the superclass method.
 - a. override
 - b. overuse
 - c. overload
 - d. overcompensate
9. When you instantiate an object that is a member of a subclass, the _____ constructor executes first.
 - a. subclass
 - b. child class
 - c. extended class
 - d. parent class

10. The keyword **super** always refers to the _____ of the class in which you use it.
 - a. child class
 - b. derived class
 - c. sub class
 - d. parent class
11. If a superclass constructor requires arguments, then its subclass _____.
 - a. must contain a constructor
 - b. must not contain a constructor
 - c. must contain a constructor that requires arguments
 - d. must not contain a constructor that requires arguments
12. If a superclass constructor requires arguments, any constructor of its subclasses must call the superclass constructor _____.
 - a. as the first statement
 - b. as the last statement
 - c. at some time
 - d. multiple times if multiple arguments are involved
13. A child class `Motorcycle` extends a parent class `Vehicle`. Each class constructor requires one `String` argument. The `Motorcycle` class constructor can call the `Vehicle` class constructor with the statement _____.
 - a. `Vehicle("Honda");`
 - b. `Motorcycle("Harley");`
 - c. `super("Suzuki");`
 - d. none of the above
14. In the Java programming language, the concept of keeping data private is known as _____.
 - a. polymorphism
 - b. information hiding
 - c. data deception
 - d. concealing fields
15. If you create a data field or method that is _____, it can be used within its own class or in any classes extended from that class.
 - a. `public`
 - b. `protected`
 - c. `private`
 - d. none of the above

16. Within a subclass, you cannot override _____ methods.
 - a. `public`
 - b. `private`
 - c. `protected`
 - d. constructor
17. You call a `static` method using a(n) _____ name.
 - a. class
 - b. superclass
 - c. object
 - d. none of the above
18. You use `final` as a method access modifier when you create _____ methods for which you want to prevent overriding.
 - a. class
 - b. superclass
 - c. subclass
 - d. instance
19. A compiler can decide to _____ a `final` method.
 - a. duplicate
 - b. inline
 - c. redline
 - d. beeline
20. You use _____ as a method access modifier when you create class methods for which you want to prevent overriding.
 - a. `final`
 - b. `static`
 - c. `private`
 - d. `public`

EXERCISES

1. Create a class named `Book` that contains data fields for title and number of pages. Include get and set methods for these fields. Next create a subclass named `Textbook`, which contains an additional field that holds a grade level for the `Textbook`, and additional methods to get and set the grade level field. Write a program that demonstrates using objects of each class. Save the programs as **Book.java**, **Textbook.java**, and **DemoBook.java** in the Chapter.11 folder on your Student Disk.

2. Create a class named `Square` that contains data fields for height, width, and surfaceArea, and a method named `computeSurfaceArea()`. Create a child class named `Cube`. `Cube` contains an additional data field named depth, and a `computeSurfaceArea()` method that overrides the parent method. Write a program that instantiates a `Square` object and a `Cube` object and displays the surface areas of the objects. Save the programs as **Cube.java**, **Square.java**, and **Demo Square.java** in the Chapter.11 folder on your Student Disk.
3. Create a class named `Order` that performs order processing of a single item. The superclass has four fields: customer name, customer number, quantity ordered, and unit price. Include set and get methods for each field. This class also needs methods to compute the total price (quantity times unit price) and to display the fields. Create a subclass that overrides `computePrice()` by adding a shipping and handling charge of \$4.00. Write a program that uses these classes. Save the programs as **Order.java**, **HandlingShipping.java**, and **UseHandlingShipping.java** in the Chapter.11 folder on your Student Disk.
4. Create a class named `Vacation` that computes the amount of vacation time an employee gets. If an employee has worked for more than five years, the total vacation time is three weeks annually; otherwise the employee gets two weeks annually. Use a superclass that contains an integer field that holds the number of vacation weeks, and get and set methods for the field. Use a superclass object for employees who have earned two weeks of vacation. Use a subclass for three-weeks vacation. Obtain the employee information from text fields in an applet. Use text boxes for employee number, employee name, and number of years. Display the number of weeks of vacation after computing the result. Save the programs as **Vacation.java**, **ExtraVacation.java**, **VacationHome.java**, **UseVacation.java**, and **TestUseVacation.html** in the Chapter.11 folder on your Student Disk.
5. a. Create a class named `Year` that contains a data field that holds the number of days in the year. Include a get method that displays the number of days and a constructor that sets the number of days to 365. Create a subclass named `LeapYear`. `LeapYear`'s constructor overrides `Year`'s constructor and sets the day field to 366. Write a program that instantiates one object of each class and displays their data. Save the programs as **Year.java**, **LeapYear.java**, and **UseYear.java** in the Chapter.11 folder on your Student Disk.
 - b. Add a method named `daysElapsed()` to the `Year` class you created in Exercise 5a. The `daysElapsed()` method accepts two arguments representing a month and a day; it returns an integer indicating the number of days that have elapsed since January 1 of the year. Create a `daysElapsed()` method for the `LeapYear` class that overrides the method in the `Year` class. Write a program that calculates the days elapsed on March 1 for a `Year` and for a `LeapYear`. Save the programs as **Year2.java**, **LeapYear2.java**, and **UseYear2.java** in the Chapter.11 folder on your Student Disk.

6. Create a class named `Computer` that contains data fields for processor model (for example, Pentium III) and clock speed in gigahertz (for example, 1.6). Include a get method for each field and a constructor that requires a parameter for each field. Create a subclass named `MultimediaComputer` that contains an additional integer field for the CD-ROM speed. The `MultiMedia` class also contains a get method for the new data field and a constructor that requires arguments for each of the three data fields. Write a program to demonstrate creating and using an object of each class. Save the programs as **`Computer.java`**, **`MultimediaComputer.java`**, and **`UseComputer.java`** in the `Chapter.11` folder on your Student Disk.
7. Create a class named `HotelRoom` that includes an integer field for the room number and a double field for the nightly rental rate. Include get methods for these fields, and a constructor that requires an integer argument representing the room number. The constructor sets the room rate based on the room number; rooms numbered 299 and below are \$69.95 per night, others are \$89.95 per night. Create an extended class named `Suite` whose constructor requires a room number and adds a \$40.00 surcharge to the regular hotel room rate based on the room number. Write a program to demonstrate creating and using an object of each class. Save the programs as **`HotelRoom.java`**, **`Suite.java`**, and **`UseHotelRoom.java`** in the `Chapter.11` folder on your Student Disk.
8. Create a class named `Package` with data fields for weight in ounces, shipping method, and shipping cost. The shipping method is a character: *A* for air, *T* for truck, or *M* for mail. The `Package` class contains a constructor that requires arguments for weight and shipping method. The constructor calls a `calculateCost()` method that determines the shipping cost based on the following:

Shipping Method (\$)

Weight (lb)	Air	Truck	Mail
1 to 8	2.00	1.50	.50
9 to 16	3.00	2.35	1.50
17 and over	4.50	3.25	2.15

The `Package` class also contains a `display()` method that displays the values in all four fields. Create a subclass named **`InsuredPackage`** that adds an insurance cost to the shipping cost based on the following:

Shipping Cost Before Insurance (\$)	Additional Cost (\$)
0 to 1.00	2.45
1.01 to 3.00	3.95
3.01 and over	5.55

- Write a program that instantiates at least three objects of each type (`Package` and `InsuredPackage`) using a variety of weights and shipping method codes. Display the results for each `Package` and `InsuredPackage`. Save the programs as **Package**, **InsuredPackage**, and **UsePackage** in the Chapter.11 folder on your Student Disk.
9. Write a program named `CarRental` that computes the cost of renting a car for a day, based on the size of the car: economy, medium, or full size. Include a constructor that requires the car size. Add a subclass to add the option of a car phone. Write a program to use these classes. Save the programs as **CarRental**, **CarPhone**, and **UseCarRentalAndPhone** in the Chapter.11 folder on your Student Disk.
 10. Write a program named `CollegeCourse` that computes the cost of taking a college course. Include a constructor that requires a course ID number. Add a subclass to compute a lab fee for a course that uses a lab. Write a program to use these classes. Save the programs as **CollegeCourse**, **Lab**, and **UseCourse** in the Chapter.11 folder on your Student Disk.
 11. Write a program named `Discount` that computes the price of an item. Include a constructor that requires the quantity, item name, and item number. Add a subclass to provide a discount based on the quantity ordered. Write a program to use these classes. Save the programs as **Discount**, **ComputeDiscount**, and **UseDiscount** in the Chapter.11 folder on your Student Disk.
 12. Write a program named `Vehicle` that acts as a superclass for vehicle types. The `Vehicle` class must contain private variables for the number of wheels and the average number of miles per gallon. The `Vehicle` class must also contain a constructor with integer arguments for the number of wheels and range in miles, and a `toString()` method to return the number of wheels and range in miles when called. Write subclass programs `Car` and `MotorCycle` that extend the `Vehicle` class. Each subclass should contain a **private static final** integer variable that sets the number of wheels for the subclass and a **private** variable to set the number of passengers. Each subclass must have a `toString()` method for returning the number of wheels, range, and passengers for the vehicle type. Write a `UseVehicle` class to instantiate the two vehicle objects and print the object's values. Save the programs as **Vehicle**, **Car**, **MotorCycle**, and **UseVehicle** in the Chapter.11 folder on your Student Disk.
 13. Create a class named `Course` that contains data fields for a course title and a boolean variable that indicates whether the class is offered online. Include get and set methods for these fields. Next create a subclass named `OnLine`, which contains an additional field that holds a grade level for students eligible to sign up for a course, and additional methods to get and set the grade level field. Write a program that demonstrates using objects of each class. Save the programs as **Course**, **OnLine**, and **DemoCourse** in the Chapter.11 folder on your Student Disk.

14. Each of the following files in the Chapter.11 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugEleven1.java will become FixDebugEleven1.java.
- a. DebugEleven1.java
 - b. DebugEleven2.java
 - c. DebugEleven3.java
 - d. DebugEleven4.java

CASE PROJECT



Wei's Tea Shop asks you to write an Inventory control program to keep track of the boxes of tea they serve to customers. The program will allow the user to enter the name of the tea, the number of boxes on hand, and the number of boxes used during a one-day period. The output will display the input data and the number of boxes remaining. Use dialog boxes for program input and output. The program names are **InventoryControl**, **InventorySold**, and **UseInventorySold**. The `InventorySold` class **extends** `InventoryControl` and prompts for the quantity of tea boxes sold.