CHAPTER 9

INHERITANCE

Jason Hosking/Getty Images, Inc.

CHAPTER GOALS

To learn about inheritance

To implement subclasses that inherit and override superclass methods

To understand the concept of polymorphism

To be familiar with the common superclass `Object` and its methods

CHAPTER CONTENTS

Objects from related classes usually share common behavior. For example, cars, bicycles, and buses all provide transportation. In this chapter, you will learn how the notion of inheritance expresses the relationship between specialized and general classes. By using inheritance, you will be able to share code between classes and provide services that can be used by multiple classes.

9.1 Inheritance Hierarchies

A subclass inherits data and behavior from a superclass.

In object-oriented design, **inheritance** is a relationship between a more general class (called the **superclass**) and a more specialized class (called the **subclass**). The subclass inherits data and behavior from the superclass. For example, consider the relationships between different kinds of vehicles depicted in Figure 1.

Every car *is a* vehicle. Cars share the common traits of all vehicles, such as the ability to transport people from one place to another. We say that the class `Car` inherits from the class `Vehicle`. In this relationship, the `Vehicle` class is the superclass and the `Car` class is the subclass. In Figure 2, the superclass and subclass are joined with an arrow that points to the superclass.

When you use inheritance in your programs, you can reuse code instead of duplicating it. This reuse comes in two forms. First, a subclass inherits the methods of the superclass. For example, if the `Vehicle` class has a `drive` method, then a subclass `Car` automatically inherits the method. It need not be duplicated.

You can always use a subclass object in place of a superclass object.

The second form of reuse is more subtle. You can reuse algorithms that manipulate `Vehicle` objects. Because a car is a special kind of vehicle, we can use a `Car` object in such an algorithm, and it will work correctly. The **substitution principle** states that

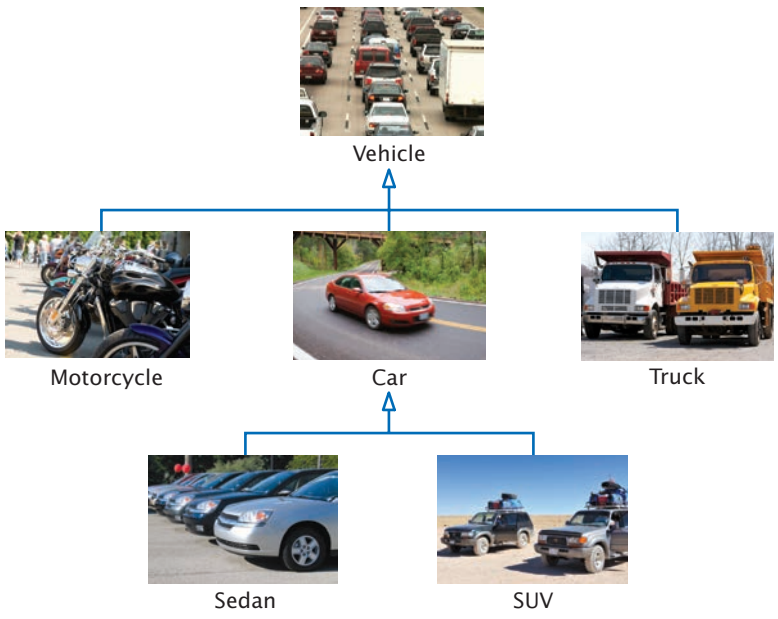
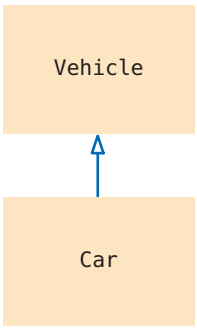


Figure 1
An Inheritance Hierarchy of Vehicle Classes

Figure 2
An Inheritance Diagram



you can always use a subclass object when a superclass object is expected. For example, consider a method that takes an argument of type `Vehicle`:

```
void processVehicle(Vehicle v)
```

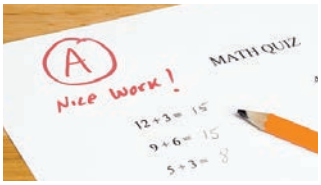
Because `Car` is a subclass of `Vehicle`, you can call that method with a `Car` object:

```
Car myCar = new Car(. . .);
processVehicle(myCar);
```

Why provide a method that processes `Vehicle` objects instead of `Car` objects? That method is more useful because it can handle *any* kind of vehicle (including `Truck` and `Motorcycle` objects).

In this chapter, we will consider a simple hierarchy of classes. Most likely, you have taken computer-graded quizzes. A quiz consists of questions, and there are different kinds of questions:

- Fill-in-the-blank
- Choice (single or multiple)
- Numeric (where an approximate answer is ok; e.g., 1.33 when the actual answer is 4/3)
- Free response



© paul kline/iStockphoto.

We will develop a simple but flexible quiz-taking program to illustrate inheritance.

Figure 3 shows an inheritance hierarchy for these question types.

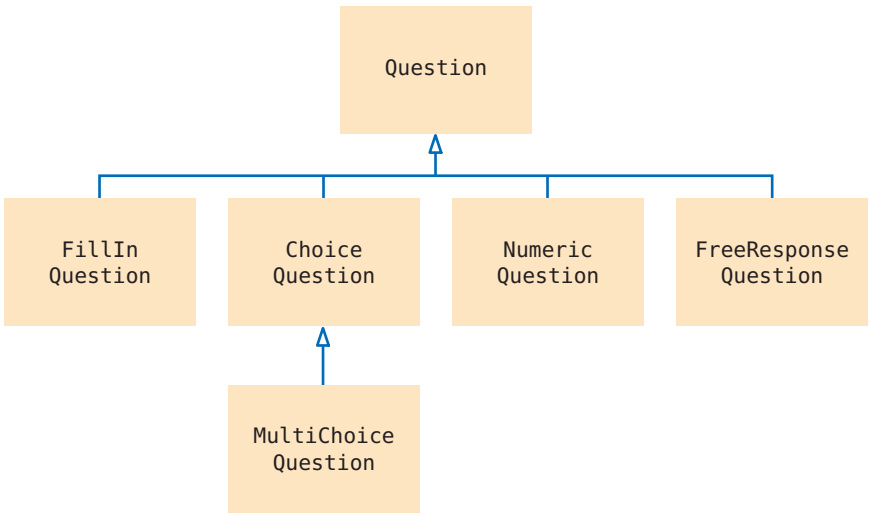


Figure 3
Inheritance Hierarchy of Question Types

At the root of this hierarchy is the `Question` type. A question can display its text, and it can check whether a given response is a correct answer.

sec01/Question.java

```
1  /**
2     A question with a text and an answer.
3  */
4  public class Question
5  {
6     private String text;
7     private String answer;
8
9     /**
10       Constructs a question with empty question and answer.
11    */
12    public Question()
13    {
14       text = "";
15       answer = "";
16    }
17
18    /**
19       Sets the question text.
20       @param questionText the text of this question
21    */
22    public void setText(String questionText)
23    {
24       text = questionText;
25    }
26
27    /**
28       Sets the answer for this question.
29       @param correctResponse the answer
30    */
31    public void setAnswer(String correctResponse)
32    {
33       answer = correctResponse;
34    }
35
36    /**
37       Checks a given response for correctness.
38       @param response the response to check
39       @return true if the response was correct, false otherwise
40    */
41    public boolean checkAnswer(String response)
42    {
43       return response.equals(answer);
44    }
45
46    /**
47       Displays this question.
48    */
49    public void display()
50    {
51       System.out.println(text);
52    }
53 }
```

This `Question` class is very basic. It does not handle multiple-choice questions, numeric questions, and so on. In the following sections, you will see how to form subclasses of the `Question` class.

Here is a simple test program for the `Question` class.

sec01/QuestionDemo1.java

```
1  import java.util.Scanner;
2
3  /**
4     This program shows a simple quiz with one question.
5  */
6  public class QuestionDemo1
7  {
8     public static void main(String[] args)
9     {
10       Scanner in = new Scanner(System.in);
11
12       Question q = new Question();
13       q.setText("Who was the inventor of Java?");
14       q.setAnswer("James Gosling");
15
16       q.display();
17       System.out.print("Your answer: ");
18       String response = in.nextLine();
19       System.out.println(q.checkAnswer(response));
20    }
21 }
```

Program Run

```
Who was the inventor of Java?
Your answer: James Gosling
true
```

Programming Tip 9.1

Use a Single Class for Variation in Values, Inheritance for Variation in Behavior

The purpose of inheritance is to model objects with different *behavior*. When students first learn about inheritance, they have a tendency to overuse it, by creating multiple classes even though the variation could be expressed with a simple instance variable.

Consider a program that tracks the fuel efficiency of a fleet of cars by logging the distance traveled and the refueling amounts. Some cars in the fleet are hybrids. Should you create a subclass `HybridCar`? Not in this application. Hybrids don't behave any differently than other cars when it comes to driving and refueling. They just have a better fuel efficiency. A single `Car` class with an instance variable

```
double milesPerGallon;
```

is entirely sufficient.

However, if you write a program that shows how to repair different kinds of vehicles, then it makes sense to have a separate class `HybridCar`. When it comes to repairs, hybrid cars behave differently from other cars.

9.2 Implementing Subclasses

In this section, you will see how to form a subclass and how a subclass automatically inherits functionality from its superclass.

Suppose you want to write a program that handles questions such as the following:

```
In which country was the inventor of Java born?
1. Australia
2. Canada
3. Denmark
4. United States
```

You could write a `ChoiceQuestion` class from scratch, with methods to set up the question, display it, and check the answer. But you don't have to. Instead, use inheritance and implement `ChoiceQuestion` as a subclass of the `Question` class (see Figure 4).

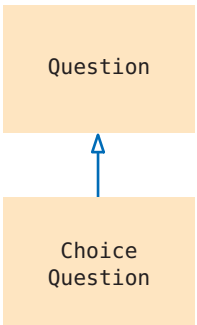


Figure 4
The `ChoiceQuestion` Class is a Subclass of the `Question` Class

A subclass inherits all methods that it does not override.

In Java, you form a subclass by specifying what makes the subclass *different from* its superclass.

Subclass objects automatically have the instance variables that are declared in the superclass. You only declare instance variables that are not part of the superclass objects.

A subclass can override a superclass method by providing a new implementation.

The subclass inherits all public methods from the superclass. You declare any methods that are *new* to the subclass, and *change* the implementation of inherited methods if the inherited behavior is not appropriate. When you supply a new implementation for an inherited method, you **override** the method.

A `ChoiceQuestion` object differs from a `Question` object in three ways:

- Its objects store the various choices for the answer.
- There is a method for adding answer choices.
- The `display` method of the `ChoiceQuestion` class shows these choices so that the respondent can choose one of them.

Like the manufacturer of a stretch limo, who starts with a regular car and modifies it, a programmer makes a subclass by modifying another class.

Media Bakery.

Syntax 9.1 Subclass Declaration

Syntax

```
public class SubclassName extends SuperclassName
{
   instance variables
   methods
}
```

```
public class ChoiceQuestion extends Question
{
   private ArrayList<String> choices;

   public void addChoice(String choice, boolean correct) { . . . }

   public void display() { . . . }
}
```

The reserved word `extends` *denotes inheritance.*

Subclass

Superclass

Declare instance variables that are added to the subclass.

Declare methods that are added to the subclass.

Declare methods that the subclass overrides.

When the `ChoiceQuestion` class inherits from the `Question` class, it needs to spell out these three differences:

```
public class ChoiceQuestion extends Question
{
   // This instance variable is added to the subclass
   private ArrayList<String> choices;

   // This method is added to the subclass
   public void addChoice(String choice, boolean correct) { . . . }

   // This method overrides a method from the superclass
   public void display() { . . . }
}
```

The `extends` reserved word indicates that a class inherits from a superclass.

The reserved word `extends` denotes inheritance. Figure 5 shows how the methods and instance variables are captured in a UML diagram.

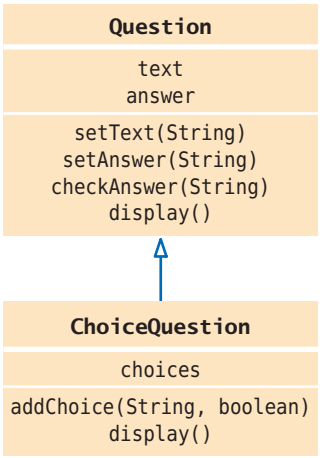


Figure 5
The `ChoiceQuestion` Class Adds an Instance Variable and a Method, and Overrides a Method

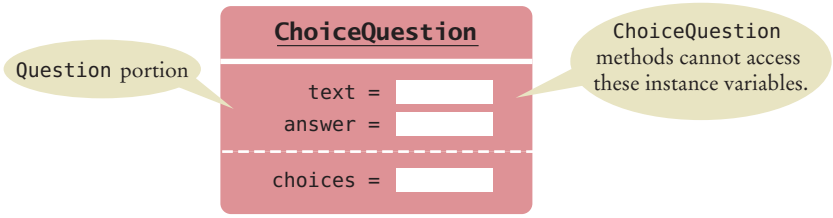


Figure 6 Data Layout of a Subclass Object

Figure 6 shows the layout of a `ChoiceQuestion` object. It has the `text` and `answer` instance variables that are declared in the `Question` superclass, and it adds an additional instance variable, `choices`.

The `addChoice` method is specific to the `ChoiceQuestion` class. You can only apply it to `ChoiceQuestion` objects, not general `Question` objects.

In contrast, the `display` method is a method that already exists in the superclass. The subclass overrides this method, so that the choices can be properly displayed.

All other methods of the `Question` class are automatically inherited by the `ChoiceQuestion` class.

You can call the inherited methods on a subclass object:

```
choiceQuestion.setAnswer("2");
```

However, the private instance variables of the superclass are inaccessible. Because these variables are private data of the superclass, only the superclass has access to them. The subclass has no more access rights than any other class.

In particular, the `ChoiceQuestion` methods cannot directly access the instance variable `answer`. These methods must use the public interface of the `Question` class to access its private data, just like every other method.

To illustrate this point, let's implement the `addChoice` method. The method has two arguments: the choice to be added (which is appended to the list of choices), and a Boolean value to indicate whether this choice is correct.

For example,

```
choiceQuestion.addChoice("Canada", true);
```

The first argument is added to the `choices` variable. If the second argument is true, then the `answer` instance variable becomes the number of the current choice. For example, if `choices.size()` is 2, then `answer` is set to the string `"2"`.

```
public void addChoice(String choice, boolean correct)
{
   choices.add(choice);
   if (correct)
   {
      // Convert choices.size() to string
      String choiceString = "" + choices.size();
      setAnswer(choiceString);
   }
}
```

You can't just access the `answer` variable in the superclass. Fortunately, the `Question` class has a `setAnswer` method. You can call that method. On which object? The question that you are currently modifying—that is, the implicit parameter of the `ChoiceQuestion.addChoice` method. Remember, if you invoke a method on the implicit

parameter, you don't have to specify the implicit parameter and can write just the method name:

```
setAnswer(choiceString);
```

If you prefer, you can make it clear that the method is executed on the implicit parameter:

```
this.setAnswer(choiceString);
```

EXAMPLE CODE See sec02 of your eText or companion code for a program that shows a simple `Car` class extending a `Vehicle` class.

Common Error 9.1

Replicating Instance Variables from the Superclass

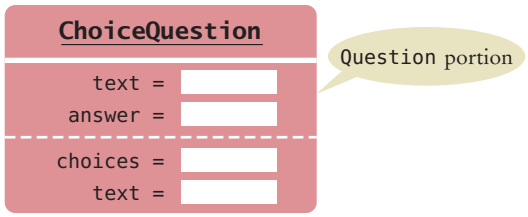
A subclass has no access to the private instance variables of the superclass.

```
public ChoiceQuestion(String questionText)
{
   text = questionText; // Error—tries to access private superclass variable
}
```

When faced with a compiler error, beginners commonly "solve" this issue by adding *another* instance variable with the same name to the subclass:

```
public class ChoiceQuestion extends Question
{
   private ArrayList<String> choices;
   private String text; // Don't!
   . . .
}
```

Sure, now the constructor compiles, but it doesn't set the correct text! Such a `ChoiceQuestion` object has two instance variables, both named `text`. The constructor sets one of them, and the `display` method displays the other. The correct solution is to access the instance variable of the superclass through the public interface of the superclass. In our example, the `ChoiceQuestion` constructor should call the `setText` method of the `Question` class.



Common Error 9.2

Confusing Super- and Subclasses

If you compare an object of type `ChoiceQuestion` with an object of type `Question`, you find that

- The reserved word `extends` suggests that the `ChoiceQuestion` object is an extended version of a `Question`.
- The `ChoiceQuestion` object is larger; it has an added instance variable, `choices`.
- The `ChoiceQuestion` object is more capable; it has an `addChoice` method.

It seems a superior object in every way. So why is `ChoiceQuestion` called the *subclass* and `Question` the *superclass*?

The *super/sub* terminology comes from set theory. Look at the set of all questions. Not all of them are `ChoiceQuestion` objects; some of them are other kinds of questions. Therefore, the set of `ChoiceQuestion` objects is a *subset* of the set of all `Question` objects, and the set of `Question` objects is a *superset* of the set of `ChoiceQuestion` objects. The more specialized objects in the subset have a richer state and more capabilities.

9.3 Overriding Methods

An overriding method can extend or replace the functionality of the superclass method.

The subclass inherits the methods from the superclass. If you are not satisfied with the behavior of an inherited method, you **override** it by specifying a new implementation in the subclass.

Consider the `display` method of the `ChoiceQuestion` class. It overrides the superclass `display` method in order to show the choices for the answer. This method *extends* the functionality of the superclass version. This means that the subclass method carries out the action of the superclass method (in our case, displaying the question text), and it also does some additional work (in our case, displaying the choices). In other cases, a subclass method *replaces* the functionality of a superclass method, implementing an entirely different behavior.

Let us turn to the implementation of the `display` method of the `ChoiceQuestion` class. The method needs to

- *Display the question text.*
- *Display the answer choices.*

The second part is easy because the answer choices are an instance variable of the subclass.

```
public class ChoiceQuestion
{
   . . .
   public void display()
   {
      // Display the question text
      . . .
      // Display the answer choices
      for (int i = 0; i < choices.size(); i++)
      {
         int choiceNumber = i + 1;
         System.out.println(choiceNumber + ": " + choices.get(i));
      }
   }
}
```

But how do you get the question text? You can't access the `text` variable of the superclass directly because it is private.

Use the reserved word `super` to call a superclass method.

Instead, you can call the `display` method of the superclass, by using the reserved word `super`:

```
public void display()
{
   // Display the question text
   super.display(); // OK
   // Display the answer choices
   . . .
}
```

Syntax 9.2 Calling a Superclass Method

Syntax `super.methodName(parameters);`

```
public void deposit(double amount)
{
   transactionCount++;
   super.deposit(amount);
}
```

Calls the method of the superclass instead of the method of the current class.

If you omit `super`, *this method calls itself. See Common Error 9.4.*

If you omit the reserved word `super`, then the method will not work as intended.

```
public void display()
{
   // Display the question text
   display(); // Error—invokes this.display()
   . . .
}
```

Because the implicit parameter `this` is of type `ChoiceQuestion`, and there is a method named `display` in the `ChoiceQuestion` class, that method will be called—but that is just the method you are currently writing! The method would call itself over and over.

Note that `super`, unlike `this`, is *not* a reference to an object. There is no separate superclass object—the subclass object contains the instance variables of the superclass. Instead, `super` is simply a reserved word that forces execution of the superclass method.

Here is the complete program that lets you take a quiz consisting of two `ChoiceQuestion` objects. We construct both objects and pass them to a method `presentQuestion`. That method displays the question to the user and checks whether the user response is correct.

sec03/QuestionDemo2.java

```
1  import java.util.Scanner;
2
3  /**
4     This program shows a simple quiz with two choice questions.
5  */
6  public class QuestionDemo2
7  {
8     public static void main(String[] args)
9     {
10       ChoiceQuestion first = new ChoiceQuestion();
11       first.setText("What was the original name of the Java language?");
12       first.addChoice("*7", false);
13       first.addChoice("Duke", false);
14       first.addChoice("Oak", true);
15       first.addChoice("Gosling", false);
16
17       ChoiceQuestion second = new ChoiceQuestion();
18       second.setText("In which country was the inventor of Java born?");
19       second.addChoice("Australia", false);
20       second.addChoice("Canada", true);
```

```
21          second.addChoice("Denmark", false);
22          second.addChoice("United States", false);
23
24          presentQuestion(first);
25          presentQuestion(second);
26       }
27
28       /**
29          Presents a question to the user and checks the response.
30          @param q the question
31       */
32       public static void presentQuestion(ChoiceQuestion q)
33       {
34          q.display();
35          System.out.print("Your answer: ");
36          Scanner in = new Scanner(System.in);
37          String response = in.nextLine();
38          System.out.println(q.checkAnswer(response));
39       }
40    }
```

sec03/ChoiceQuestion.java

```
1  import java.util.ArrayList;
2
3  /**
4     A question with multiple choices.
5  */
6  public class ChoiceQuestion extends Question
7  {
8     private ArrayList<String> choices;
9
10    /**
11       Constructs a choice question with no choices.
12    */
13    public ChoiceQuestion()
14    {
15       choices = new ArrayList<String>();
16    }
17
18    /**
19       Adds an answer choice to this question.
20       @param choice the choice to add
21       @param correct true if this is the correct choice, false otherwise
22    */
23    public void addChoice(String choice, boolean correct)
24    {
25       choices.add(choice);
26       if (correct)
27       {
28          // Convert choices.size() to string
29          String choiceString = "" + choices.size();
30          setAnswer(choiceString);
31       }
32    }
33
34    public void display()
35    {
36       // Display the question text
37       super.display();
```

```
38         // Display the answer choices
39         for (int i = 0; i < choices.size(); i++)
40         {
41            int choiceNumber = i + 1;
42            System.out.println(choiceNumber + ": " + choices.get(i));
43         }
44      }
45   }
```

Program Run

```
What was the original name of the Java language?
1: *7
2: Duke
3: Oak
4: Gosling
Your answer: *7
false
In which country was the inventor of Java born?
1: Australia
2: Canada
3: Denmark
4: United States
Your answer: 2
true
```

Common Error 9.3

Accidental Overloading

In Java, two methods can have the same name, provided they differ in their parameter types. For example, the `PrintStream` class has methods called `println` with headers

```
void println(int x)
```

and

```
void println(String x)
```

These are different methods, each with its own implementation. The Java compiler considers them to be completely unrelated. We say that the `println` name is **overloaded**. This is different from overriding, where a subclass method provides an implementation of a method whose parameter variables have the *same* types.

If you mean to override a method but use a parameter variable with a different type, then you accidentally introduce an overloaded method. For example,

```
public class ChoiceQuestion extends Question
{
   . . .
   public void display(PrintStream out)
   // Does not override void display()
   {
      . . .
   }
}
```

The compiler will not complain. It thinks that you want to provide a method just for `PrintStream` arguments, while inheriting another method `void display()`.

When overriding a method, be sure to check that the types of the parameter variables match exactly.

Common Error 9.4

Forgetting to Use `super` When Invoking a Superclass Method

A common error in extending the functionality of a superclass method is to forget the reserved word `super`. For example, to compute the salary of a manager, get the salary of the underlying `Employee` object and add a bonus:

```
public class Manager
{
   . . .
   public double getSalary()
   {
      double baseSalary = getSalary();
         // Error: should be super.getSalary()
      return baseSalary + bonus;
   }
}
```

Here `getSalary()` refers to the `getSalary` method applied to the implicit parameter of the method. The implicit parameter is of type `Manager`, and there is a `getSalary` method in the `Manager` class. Calling that method is a recursive call, which will never stop. Instead, you must tell the compiler to invoke the superclass method.

Whenever you call a superclass method from a subclass method with the same name, be sure to use the reserved word `super`.

Special Topic 9.1

Calling the Superclass Constructor

Consider the process of constructing a subclass object. A subclass constructor can only initialize the instance variables of the subclass. But the superclass instance variables also need to be initialized. Unless you specify otherwise, the superclass instance variables are initialized with the constructor of the superclass that has no arguments.

Unless specified otherwise, the subclass constructor calls the superclass constructor with no arguments.

Syntax 9.3 Constructor with Superclass Initializer

Syntax

```
public ClassName(parameterType parameterName, . . .)
{
   super(arguments);
   . . .
}
```

```
public ChoiceQuestion(String questionText)
{
   super(questionText);
   choices = new ArrayList<String>();
}
```

The superclass constructor is called first.

The constructor body can contain additional statements.

If you omit the superclass constructor call, the superclass constructor with no arguments is invoked.

In order to specify another constructor, you use the `super` reserved word, together with the arguments of the superclass constructor, as the *first statement* of the subclass constructor.

For example, suppose the `Question` superclass had a constructor for setting the question text. Here is how a subclass constructor could call that superclass constructor:

To call a superclass constructor, use the `super` reserved word in the first statement of the subclass constructor.

```
public ChoiceQuestion(String questionText)
{
   super(questionText);
   choices = new ArrayList<String>();
}
```

In our example program, we used the superclass constructor with no arguments. However, if all superclass constructors have arguments, you must use the `super` syntax and provide the arguments for a superclass constructor.

The constructor of a subclass can pass arguments to a superclass constructor, using the reserved word `super`.

When the reserved word `super` is followed by a parenthesis, it indicates a call to the superclass constructor. When used in this way, the constructor call must be *the first statement of the subclass constructor*. If `super` is followed by a period and a method name, on the other hand, it indicates a call to a superclass method, as you saw in the preceding section. Such a call can be made anywhere in any subclass method.

9.4 Polymorphism

In this section, you will learn how to use inheritance for processing objects of different types in the same program.

Consider our first sample program. It presented two `Question` objects to the user. The second sample program presented two `ChoiceQuestion` objects. Can we write a program that shows a mixture of both question types?

With inheritance, this goal is very easy to realize. In order to present a question to the user, we need not know the exact type of the question. We just display the question and check whether the user supplied the correct answer. The `Question` superclass has methods for displaying and checking. Therefore, we can simply declare the parameter variable of the `presentQuestion` method to have the type `Question`:

```
public static void presentQuestion(Question q)
{
   q.display();
   System.out.print("Your answer: ");
   Scanner in = new Scanner(System.in);
   String response = in.nextLine();
   System.out.println(q.checkAnswer(response));
}
```

A subclass reference can be used when a superclass reference is expected.

As discussed in Section 9.1, we can substitute a subclass object whenever a superclass object is expected:

```
ChoiceQuestion second = new ChoiceQuestion();
. . .
presentQuestion(second); // OK to pass a ChoiceQuestion
```

When the `presentQuestion` method executes, the object references stored in `second` and `q` refer to the same object of type `ChoiceQuestion` (see Figure 7).

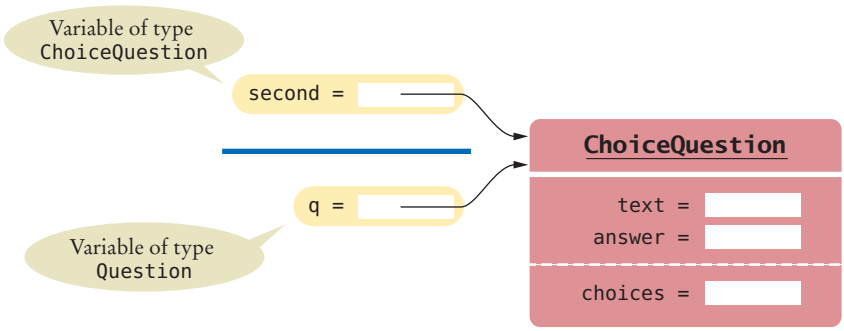


Figure 7 Variables of Different Types Referring to the Same Object

However, the *variable* `q` knows less than the full story about the object to which it refers (see Figure 8).

Because `q` is a variable of type `Question`, you can call the `display` and `checkAnswer` methods. You cannot call the `addChoice` method, though—it is not a method of the `Question` superclass.

This is as it should be. After all, it happens that in this method call, `q` refers to a `ChoiceQuestion`. In another method call, `q` might refer to a plain `Question` or an entirely different subclass of `Question`.

Now let's have a closer look inside the `presentQuestion` method. It starts with the call

```
q.display(); // Does it call Question.display or ChoiceQuestion.display?
```

When the virtual machine calls an instance method, it locates the method of the implicit parameter's class. This is called dynamic method lookup.

Which `display` method is called? If you look at the program output on page 322, you will see that the method called depends on the contents of the parameter variable `q`. In the first case, `q` refers to a `Question` object, so the `Question.display` method is called. But in the second case, `q` refers to a `ChoiceQuestion`, so the `ChoiceQuestion.display` method is called, showing the list of choices.

In Java, method calls *are always determined by the type of the actual object*, not the type of the variable containing the object reference. This is called **dynamic method lookup**.

Polymorphism ("having multiple forms") allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.

Dynamic method lookup allows us to treat objects of different classes in a uniform way. This feature is called **polymorphism**. We ask multiple objects to carry out a task, and each object does so in its own way.

Polymorphism makes programs *easily extensible*. Suppose we want to have a new kind of question for calculations, where we are willing to accept an approximate answer. All we need to do is to declare a new class `NumericQuestion` that extends `Question`, with its own `checkAnswer` method. Then we can call the `presentQuestion` method

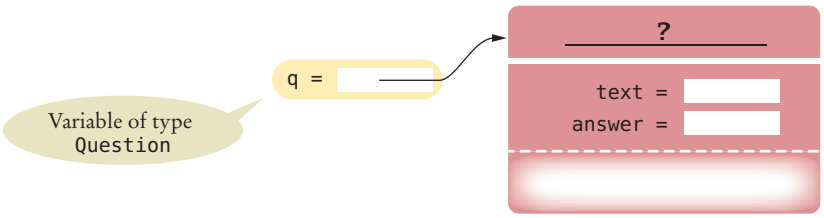


Figure 8 A `Question` Reference Can Refer to an Object of Any Subclass of `Question`

In the same way that vehicles can differ in their method of locomotion, polymorphic objects carry out tasks in different ways.

© Alpophoto/iStockphoto.

with a mixture of plain questions, choice questions, and numeric questions. The `presentQuestion` method need not be changed at all! Thanks to dynamic method lookup, method calls to the `display` and `checkAnswer` methods automatically select the correct method of the newly declared classes.

sec04/QuestionDemo3.java

```
1  import java.util.Scanner;
2
3  /**
4     This program shows a simple quiz with two question types.
5  */
6  public class QuestionDemo3
7  {
8     public static void main(String[] args)
9     {
10       Question first = new Question();
11       first.setText("Who was the inventor of Java?");
12       first.setAnswer("James Gosling");
13
14       ChoiceQuestion second = new ChoiceQuestion();
15       second.setText("In which country was the inventor of Java born?");
16       second.addChoice("Australia", false);
17       second.addChoice("Canada", true);
18       second.addChoice("Denmark", false);
19       second.addChoice("United States", false);
20
21       presentQuestion(first);
22       presentQuestion(second);
23    }
24
25    /**
26       Presents a question to the user and checks the response.
27       @param q the question
28    */
29    public static void presentQuestion(Question q)
30    {
31       q.display();
32       System.out.print("Your answer: ");
33       Scanner in = new Scanner(System.in);
```

```
34          String response = in.nextLine();
35          System.out.println(q.checkAnswer(response));
36       }
37    }
```

Program Run

```
Who was the inventor of Java?
Your answer: Bjarne Stroustrup
false
In which country was the inventor of Java born?
1: Australia
2: Canada
3: Denmark
4: United States
Your answer: 2
true
```

Special Topic 9.2

Dynamic Method Lookup and the Implicit Parameter

Suppose we add the `presentQuestion` method to the `Question` class itself:

```
void presentQuestion()
{
   display();
   System.out.print("Your answer: ");
   Scanner in = new Scanner(System.in);
   String response = in.nextLine();
   System.out.println(checkAnswer(response));
}
```

Now consider the call

```
ChoiceQuestion cq = new ChoiceQuestion();
cq.setText("In which country was the inventor of Java born?");
. . .
cq.presentQuestion();
```

Which `display` and `checkAnswer` method will the `presentQuestion` method call? If you look inside the code of the `presentQuestion` method, you can see that these methods are executed on the implicit parameter:

```
public class Question
{
   public void presentQuestion()
   {
      this.display();
      System.out.print("Your answer: ");
      Scanner in = new Scanner(System.in);
      String response = in.nextLine();
      System.out.println(this.checkAnswer(response));
   }
}
```

The implicit parameter `this` in our call is a reference to an object of type `ChoiceQuestion`. Because of dynamic method lookup, the `ChoiceQuestion` versions of the `display` and `checkAnswer` methods are called automatically. This happens even though the `presentQuestion` method is declared in the `Question` class, which has *no knowledge* of the `ChoiceQuestion` class.

As you can see, polymorphism is a very powerful mechanism. The `Question` class supplies a `presentQuestion` method that specifies the common nature of presenting a question, namely to display it and check the response. How the displaying and checking are carried out is left to the subclasses.

Special Topic 9.3

Abstract Classes

When you extend an existing class, you have the choice whether or not to override the methods of the superclass. Sometimes, it is desirable to *force* programmers to override a method. That happens when there is no good default for the superclass and only the subclass programmer can know how to implement the method properly.

Here is an example: Suppose the First National Bank of Java decides that every account type must have some monthly fees. Therefore, a `deductFees` method should be added to the `Account` class:

```
public class Account
{
   public void deductFees() { . . . }
   . . .
}
```

But what should this method do? Of course, we could have the method do nothing. But then a programmer implementing a new subclass might simply forget to implement the `deductFees` method, and the new account would inherit the do-nothing method of the superclass. There is a better way—declare the `deductFees` method as an **abstract method**:

```
public abstract void deductFees();
```

An abstract method has no implementation. This forces the implementors of subclasses to specify concrete implementations of this method. (Of course, some subclasses might decide to implement a do-nothing method, but then that is their choice—not a silently inherited default.)

You cannot construct objects of classes with abstract methods. For example, once the `Account` class has an abstract method, the compiler will flag an attempt to create a `new Account()` as an error.

A class for which you cannot create objects is called an **abstract class**. A class for which you can create objects is sometimes called a **concrete class**. In Java, you must declare all abstract classes with the reserved word `abstract`:

```
public abstract class Account
{
   public abstract void deductFees();
   . . .
}

public class SavingsAccount extends Account // Not abstract
{
   . . .
   public void deductFees() // Provides an implementation
   {
      . . .
   }
}
```

If a class extends an abstract class without providing an implementation of all abstract methods, it too is abstract.

```
public abstract class BusinessAccount
{
```

```
   // No implementation of deductFees
}
```

Note that you cannot construct an *object* of an abstract class, but you can still have an *object reference* whose type is an abstract class. Of course, the actual object to which it refers must be an instance of a concrete subclass:

```
Account anAccount; // OK
anAccount = new Account(); // Error—Account is abstract
anAccount = new SavingsAccount(); // OK
anAccount = null; // OK
```

When you declare a method as abstract, you force programmers to provide implementations in subclasses. This is better than coming up with a default that might be inherited accidentally.

Special Topic 9.4

Final Methods and Classes

In Special Topic 9.3 you saw how you can force other programmers to create subclasses of abstract classes and override abstract methods. Occasionally, you may want to do the opposite and *prevent* other programmers from creating subclasses or from overriding certain methods. In these situations, you use the `final` reserved word. For example, the `String` class in the standard Java library has been declared as

```
public final class String { . . . }
```

That means that nobody can extend the `String` class. When you have a reference of type `String`, it must contain a `String` object, never an object of a subclass.

You can also declare individual methods as final:

```
public class SecureAccount extends BankAccount
{
   . . .
   public final boolean checkPassword(String password)
   {
      . . .
   }
}
```

This way, nobody can override the `checkPassword` method with another method that simply returns `true`.

Special Topic 9.5

Protected Access

We ran into a hurdle when trying to implement the `display` method of the `ChoiceQuestion` class. That method wanted to access the instance variable `text` of the superclass. Our remedy was to use the appropriate method of the superclass to display the text.

Java offers another solution to this problem. The superclass can declare an instance variable as *protected:*

```
public class Question
{
   protected String text;
   . . .
}
```

Protected data in an object can be accessed by the methods of the object's class and all its subclasses. For example, `ChoiceQuestion` inherits from `Question`, so its methods can access the protected instance variables of the `Question` superclass.

Some programmers like the `protected` access feature because it seems to strike a balance between absolute protection (making instance variables private) and no protection at all (making instance variables public). However, experience has shown that protected instance variables are subject to the same kinds of problems as public instance variables. The designer of the superclass has no control over the authors of subclasses. Any of the subclass methods can corrupt the superclass data. Furthermore, classes with protected variables are hard to modify. Even if the author of the superclass would like to change the data implementation, the protected variables cannot be changed, because someone somewhere out there might have written a subclass whose code depends on them.

In Java, protected variables have another drawback—they are accessible not just by subclasses, but also by other classes in the same package (see Special Topic 8.4).

It is best to leave all data private. If you want to grant access to the data to subclass methods only, consider making the *accessor* method protected.

HOW TO 9.1

Developing an Inheritance Hierarchy

When you work with a set of classes, some of which are more general and others more specialized, you want to organize them into an inheritance hierarchy. This enables you to process objects of different classes in a uniform way.

As an example, we will consider a bank that offers customers the following account types:

- *A savings account that earns interest. The interest compounds monthly and is computed on the minimum monthly balance.*
- *A checking account that has no interest, gives you three free withdrawals per month, and charges a $1 transaction fee for each additional withdrawal.*

Problem Statement Design and implement a program that will manage a set of accounts of both types. It should be structured so that other account types can be added without affecting the main processing loop. Supply a menu

```
D)eposit  W)ithdraw  M)onth end  Q)uit
```

For deposits and withdrawals, query the account number and amount. Print the balance of the account after each transaction.

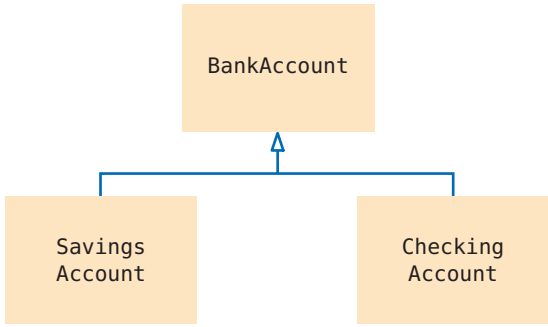
In the "Month end" command, accumulate interest or clear the transaction counter, depending on the type of the bank account. Then print the balance of all accounts.

Step 1 List the classes that are part of the hierarchy.

In our case, the problem description yields two classes: `SavingsAccount` and `CheckingAccount`. Of course, you could implement each of them separately. But that would not be a good idea because the classes would have to repeat common functionality, such as updating an account balance. We need another class that can be responsible for that common functionality. The problem statement does not explicitly mention such a class. Therefore, we need to discover it. Of course, in this case, the solution is simple. Savings accounts and checking accounts are special cases of a bank account. Therefore, we will introduce a common superclass `BankAccount`.

Step 2 Organize the classes into an inheritance hierarchy.

Draw an inheritance diagram that shows super- and subclasses. Here is one for our example:



Step 3 Determine the common responsibilities.

In Step 2, you will have identified a class at the base of the hierarchy. That class needs to have sufficient responsibilities to carry out the tasks at hand. To find out what those tasks are, write pseudocode for processing the objects.

```
For each user command
   If it is a deposit or withdrawal
      Deposit or withdraw the amount from the specified account.
      Print the balance.
   If it is month end processing
      For each account
         Call month end processing.
         Print the balance.
```

From the pseudocode, we obtain the following list of common responsibilities that every bank account must carry out:

```
Deposit money.
Withdraw money.
Get the balance.
Carry out month end processing.
```

Step 4 Decide which methods are overridden in subclasses.

For each subclass and each of the common responsibilities, decide whether the behavior can be inherited or whether it needs to be overridden. Be sure to declare any methods that are inherited or overridden in the root of the hierarchy.

```
public class BankAccount
{
   . . .
   /**
      Makes a deposit into this account.
      @param amount the amount of the deposit
   */
   public void deposit(double amount) { . . . }

   /**
      Makes a withdrawal from this account, or charges a penalty if
      sufficient funds are not available.
      @param amount the amount of the withdrawal
   */
   public void withdraw(double amount) { . . . }
```

```
   /**
      Carries out the end of month processing that is appropriate
      for this account.
   */
   public void monthEnd() { . . . }

   /**
      Gets the current balance of this bank account.
      @return the current balance
   */
   public double getBalance() { . . . }
}
```

The `SavingsAccount` and `CheckingAccount` classes will both override the `monthEnd` method. The `SavingsAccount` class must also override the `withdraw` method to track the minimum balance. The `CheckingAccount` class must update a transaction count in the `withdraw` method.

Step 5 Declare the public interface of each subclass.

Typically, subclasses have responsibilities other than those of the superclass. List those, as well as the methods that need to be overridden. You also need to specify how the objects of the subclasses should be constructed.

In this example, we need a way of setting the interest rate for the savings account. In addition, we need to specify constructors and overridden methods.

```
public class SavingsAccount extends BankAccount
{
   . . .
   /**
      Constructs a savings account with a zero balance.
   */
   public SavingsAccount() { . . . }

   /**
      Sets the interest rate for this account.
      @param rate the monthly interest rate in percent
   */
   public void setInterestRate(double rate) { . . . }

   // These methods override superclass methods
   public void withdraw(double amount) { . . . }
   public void monthEnd() { . . . }
}

public class CheckingAccount extends BankAccount
{
   . . .
   /**
      Constructs a checking account with a zero balance.
   */
   public CheckingAccount() { . . . }

   // These methods override superclass methods
   public void withdraw(double amount) { . . . }
   public void monthEnd() { . . . }
}
```

Step 6 Identify instance variables.

List the instance variables for each class. If you find an instance variable that is common to all classes, be sure to place it in the base of the hierarchy.

All accounts have a balance. We store that value in the `BankAccount` superclass:

```
public class BankAccount
{
   private double balance;
   . . .
}
```

The `SavingsAccount` class needs to store the interest rate. It also needs to store the minimum monthly balance, which must be updated by all withdrawals.

```
public class SavingsAccount extends BankAccount
{
   private double interestRate;
   private double minBalance;
   . . .
}
```

The `CheckingAccount` class needs to count the withdrawals, so that the charge can be applied after the free withdrawal limit is reached.

```
public class CheckingAccount extends BankAccount
{
   private int withdrawals;
   . . .
}
```

Step 7 Implement constructors and methods.

The methods of the `BankAccount` class update or return the balance.

```
public void deposit(double amount)
{
   balance = balance + amount;
}

public void withdraw(double amount)
{
   balance = balance - amount;
}

public double getBalance()
{
   return balance;
}
```

At the level of the `BankAccount` superclass, we can say nothing about end of month processing. We choose to make that method do nothing:

```
public void monthEnd()
{
}
```

In the `withdraw` method of the `SavingsAccount` class, the minimum balance is updated. Note the call to the superclass method:

```
public void withdraw(double amount)
{
   super.withdraw(amount);
   double balance = getBalance();
   if (balance < minBalance)
   {
      minBalance = balance;
   }
}
```

In the `monthEnd` method of the `SavingsAccount` class, the interest is deposited into the account. We must call the `deposit` method because we have no direct access to the `balance` instance variable. The minimum balance is reset for the next month.

```
public void monthEnd()
{
   double interest = minBalance * interestRate / 100;
   deposit(interest);
   minBalance = getBalance();
}
```

The `withdraw` method of the `CheckingAccount` class needs to check the withdrawal count. If there have been too many withdrawals, a charge is applied. Again, note how the method invokes the superclass method:

```
public void withdraw(double amount)
{
   final int FREE_WITHDRAWALS = 3;
   final int WITHDRAWAL_FEE = 1;

   super.withdraw(amount);
   withdrawals++;
   if (withdrawals > FREE_WITHDRAWALS)
   {
      super.withdraw(WITHDRAWAL_FEE);
   }
}
```

End of month processing for a checking account simply resets the withdrawal count.

```
public void monthEnd()
{
   withdrawals = 0;
}
```

Step 8 Construct objects of different subclasses and process them.

In our sample program, we allocate five checking accounts and five savings accounts and store their addresses in an array of bank accounts. Then we accept user commands and execute deposits, withdrawals, and monthly processing.

```
BankAccount[] accounts = . . .;
. . .
Scanner in = new Scanner(System.in);
boolean done = false;
while (!done)
{
   System.out.print("D)eposit  W)ithdraw  M)onth end  Q)uit: ");
   String input = in.next();
   if (input.equals("D") || input.equals("W")) // Deposit or withdrawal
   {
      System.out.print("Enter account number and amount: ");
      int num = in.nextInt();
      double amount = in.nextDouble();

      if (input.equals("D")) { accounts[num].deposit(amount); }
      else { accounts[num].withdraw(amount); }

      System.out.println("Balance: " + accounts[num].getBalance());
   }
   else if (input.equals("M")) // Month end processing
   {
```

```
            for (int n = 0; n < accounts.length; n++)
            {
               accounts[n].monthEnd();
               System.out.println(n + " " + accounts[n].getBalance());
            }
         }
         else if (input.equals("Q"))
         {
            done = true;
         }
      }
```

EXAMPLE CODE See how_to_1 of your companion code for the program with `BankAccount`, `SavingsAccount`, and `CheckingAccount` classes.

WORKED EXAMPLE 9.1

Implementing an Employee Hierarchy for Payroll Processing

Learn how to implement payroll processing that works for different kinds of employees. See your eText or visit `wiley.com/go/bjeo7`.

Jose Luis Pelaez Inc./Getty Images, Inc.

9.5 `Object`: The Cosmic Superclass

In Java, every class that is declared without an explicit `extends` clause automatically extends the class `Object`. That is, the class `Object` is the direct or indirect superclass of *every* class in Java (see Figure 9). The `Object` class defines several very general methods, including

- `toString`, which yields a string describing the object (Section 9.5.1).
- `equals`, which compares objects with each other (Section 9.5.2).
- `hashCode`, which yields a numerical code for storing the object in a set (see Special Topic 15.2).

9.5.1 Overriding the `toString` Method

The `toString` method returns a string representation for each object. It is often used for debugging.

For example, consider the `Rectangle` class in the standard Java library. Its `toString` method shows the state of a rectangle:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
String s = box.toString();
   // Sets s to "java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

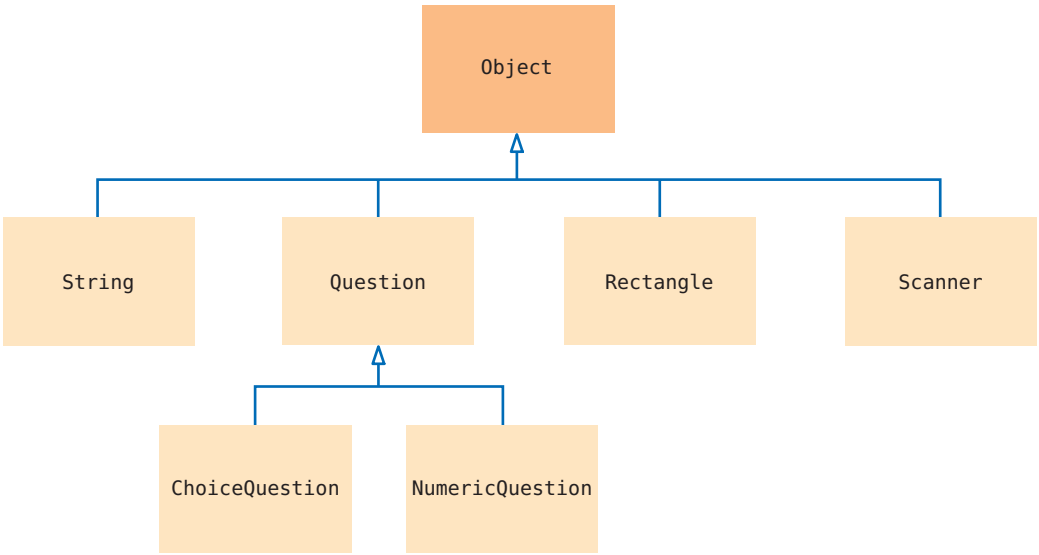


Figure 9 The `Object` Class Is the Superclass of Every Java Class

The `toString` method is called automatically whenever you concatenate a string with an object. Here is an example:

```
"box=" + box;
```

On one side of the `+` concatenation operator is a string, but on the other side is an object reference. The Java compiler automatically invokes the `toString` method to turn the object into a string. Then both strings are concatenated. In this case, the result is the string

```
"box=java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

The compiler can invoke the `toString` method, because it knows that *every* object has a `toString` method: Every class extends the `Object` class, and that class declares `toString`.

As you know, numbers are also converted to strings when they are concatenated with other strings. For example,

```
int age = 18;
String s = "Harry's age is " + age;
   // Sets s to "Harry's age is 18"
```

In this case, the `toString` method is *not* involved. Numbers are not objects, and there is no `toString` method for them. Fortunately, there is only a small set of primitive types, and the compiler knows how to convert them to strings.

Let's try the `toString` method for the `BankAccount` class:

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
   // Sets s to something like "BankAccount@d24606bf"
```

That's disappointing—all that's printed is the name of the class, followed by the **hash code**, a seemingly random code. The hash code can be used to tell objects apart—different objects are likely to have different hash codes. (See Special Topic 15.2 for the details.)

Override the `toString` method to yield a string that describes the object's state.

We don't care about the hash code. We want to know what is *inside* the object. But, of course, the `toString` method of the `Object` class does not know what is inside the `BankAccount` class. Therefore, we have to override the method and supply our own version in the `BankAccount` class. We'll follow the same format that the `toString` method of the `Rectangle` class uses: first print the name of the class, and then the values of the instance variables inside brackets.

```
public class BankAccount
{
   . . .
   public String toString()
   {
      return "BankAccount[balance=" + balance + "]";
   }
}
```

This works better:

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString(); // Sets s to "BankAccount[balance=5000]"
```

9.5.2 The equals Method

The equals method checks whether two objects have the same contents.

In addition to the `toString` method, the `Object` class also provides an `equals` method, whose purpose is to check whether two objects have the same contents:

```
if (stamp1.equals(stamp2)) . . .  // Contents are the same—see Figure 10
```

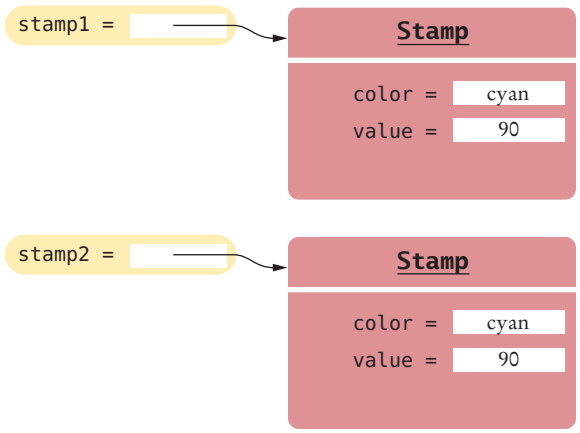


Figure 10
Two References to Equal Objects

This is different from the test with the == operator, which tests whether two references are identical, referring to the *same object:*

```
if (stamp1 == stamp2) . . .  // Objects are the same—see Figure 11
```

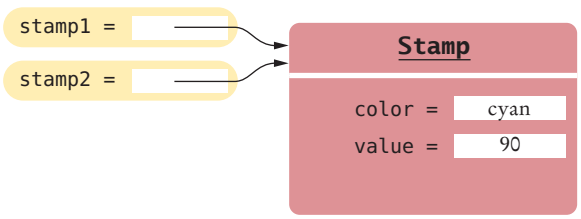


Figure 11
Two References to the Same Object

The `equals` *method checks whether two objects have the same contents.*

© Ken Brown/iStockphoto.

Let's implement the `equals` method for a `Stamp` class. You need to override the `equals` method of the `Object` class:

```
public class Stamp
{
   private String color;
   private int value;
   . . .
   public boolean equals(Object otherObject)
   {
      . . .
   }
   . . .
}
```

Now you have a slight problem. The `Object` class knows nothing about stamps, so it declares the `otherObject` parameter variable of the `equals` method to have the type `Object`. When overriding the method, you are not allowed to change the type of the parameter variable. Cast the parameter variable to the class `Stamp`:

```
Stamp other = (Stamp) otherObject;
```

Then you can compare the two stamps:

```
public boolean equals(Object otherObject)
{
   Stamp other = (Stamp) otherObject;
   return color.equals(other.color)
         && value == other.value;
}
```

Note that this `equals` method can access the instance variables of *any* `Stamp` object: the access `other.color` is perfectly legal.

9.5.3 The `instanceof` Operator

As you have seen, it is legal to store a subclass reference in a superclass variable:

```
ChoiceQuestion cq = new ChoiceQuestion();
Question q = cq; // OK
Object obj = cq; // OK
```

If you know that an object belongs to a given class, use a cast to convert the type.

Very occasionally, you need to carry out the opposite conversion, from a superclass reference to a subclass reference.

For example, you may have a variable of type `Object`, and you happen to know that it actually holds a `Question` reference. In that case, you can use a cast to convert the type:

```
Question q = (Question) obj;
```

The instanceof operator tests whether an object belongs to a particular type.

However, this cast is somewhat dangerous. If you are wrong, and `obj` actually refers to an object of an unrelated type, then a "class cast" exception is thrown.

To protect against bad casts, you can use the `instanceof` operator. It tests whether an object belongs to a particular type. For example,

```
obj instanceof Question
```

returns `true` if the type of `obj` is convertible to `Question`. This happens if `obj` refers to an actual `Question` or to a subclass such as `ChoiceQuestion`.

Using the `instanceof` operator, a safe cast can be programmed as follows:

```
if (obj instanceof Question)
{
   Question q = (Question) obj;
}
```

Note that `instanceof` is *not* a method. It is an operator, just like + or <. However, it does not operate on numbers. To the left is an object, and to the right a type name.

Do *not* use the `instanceof` operator to bypass polymorphism:

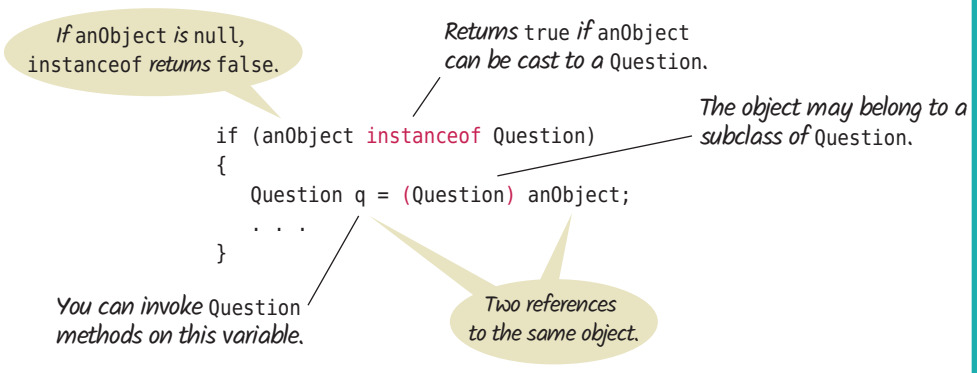
```
if (q instanceof ChoiceQuestion) // Don't do this—see Common Error 9.5
{
   // Do the task the ChoiceQuestion way
}
else if (q instanceof Question)
{
   // Do the task the Question way
}
```

In this case, you should implement a method `doTheTask` in the `Question` class, override it in `ChoiceQuestion`, and call

```
q.doTheTask();
```

Syntax 9.4 The instanceof Operator

Syntax *object* `instanceof` *TypeName*



EXAMPLE CODE See sec05 of your eText or companion code for a program that demonstrates the `toString` method and the `instanceof` operator.

Common Error 9.5

Don't Use Type Tests

Some programmers use specific type tests in order to implement behavior that varies with each class:

```
if (q instanceof ChoiceQuestion) // Don't do this
{
   // Do the task the ChoiceQuestion way
}
else if (q instanceof Question)
{
   // Do the task the Question way
}
```

This is a poor strategy. If a new class such as `NumericQuestion` is added, then you need to revise all parts of your program that make a type test, adding another case:

```
else if (q instanceof NumericQuestion)
{
   // Do the task the NumericQuestion way
}
```

In contrast, consider the addition of a class `NumericQuestion` to our quiz program. *Nothing* needs to change in that program because it uses polymorphism, not type tests.

Whenever you find yourself trying to use type tests in a hierarchy of classes, reconsider and use polymorphism instead. Declare a method `doTheTask` in the superclass, override it in the subclasses, and call

```
q.doTheTask();
```

Special Topic 9.6

Inheritance and the `toString` Method

You just saw how to write a `toString` method: Form a string consisting of the class name and the names and values of the instance variables. However, if you want your `toString` method to be usable by subclasses of your class, you need to work a bit harder.

Instead of hardcoding the class name, call the `getClass` method (which every class inherits from the `Object` class) to obtain an object that describes a class and its properties. Then invoke the `getName` method to get the name of the class:

```
public String toString()
{
   return getClass().getName() + "[balance=" + balance + "]";
}
```

Then the `toString` method prints the correct class name when you apply it to a subclass, say a `SavingsAccount`.

```
SavingsAccount momsSavings = . . . ;
System.out.println(momsSavings);
   // Prints "SavingsAccount[balance=10000]"
```

Of course, in the subclass, you should override `toString` and add the values of the subclass instance variables. Note that you must call `super.toString` to get the instance variables of the superclass—the subclass can't access them directly.

```
public class SavingsAccount extends BankAccount
{
```

```
   . . .
   public String toString()
   {
      return super.toString() + "[interestRate=" + interestRate + "]";
   }
}
```

Now a savings account is converted to a string such as `SavingsAccount[balance=10000][interestRate=5]`. The brackets show which variables belong to the superclass.

Special Topic 9.7

Inheritance and the `equals` Method

You just saw how to write an `equals` method: Cast the `otherObject` parameter variable to the type of your class, and then compare the instance variables of the implicit parameter and the explicit parameter.

But what if someone called `stamp1.equals(x)` where `x` wasn't a `Stamp` object? Then the bad cast would generate an exception. It is a good idea to test whether `otherObject` really is an instance of the `Stamp` class. The easiest test would be with the `instanceof` operator. However, that test is not specific enough. It would be possible for `otherObject` to belong to some subclass of `Stamp`. To rule out that possibility, you should test whether the two objects belong to the same class. If not, return false.

```
if (getClass() != otherObject.getClass()) { return false; }
```

Moreover, the Java language specification demands that the `equals` method return false when `otherObject` is `null`.

Here is an improved version of the `equals` method that takes these two points into account:

```
public boolean equals(Object otherObject)
{
   if (otherObject == null) { return false; }
   if (getClass() != otherObject.getClass()) { return false; }
   Stamp other = (Stamp) otherObject;
   return color.equals(other.color) && value == other.value;
}
```

When you implement `equals` in a subclass, you should first call `equals` in the superclass to check whether the superclass instance variables match. Here is an example:

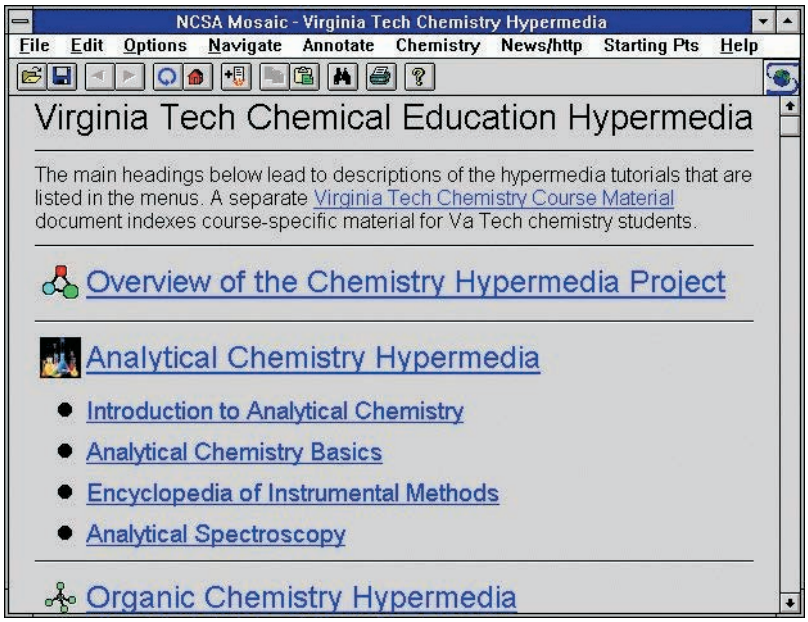
```
public CollectibleStamp extends Stamp
{
   private int year;
   . . .
   public boolean equals(Object otherObject)
   {
      if (!super.equals(otherObject)) { return false; }
      CollectibleStamp other = (CollectibleStamp) otherObject;
      return year == other.year;
   }
}
```

Computing & Society 9.1 Who Controls the Internet?

In 1962, J.C.R. Licklider was head of the first computer research program at DARPA, the Defense Advanced Research Projects Agency. He wrote a series of papers describing a "galactic network" through which computer users could access data and programs from other sites. This was well before computer networks were invented. By 1969, four computers—three in California and one in Utah—were connected to the ARPANET, the precursor of the Internet. The network grew quickly, linking computers at many universities and research organizations. It was originally thought that most network users wanted to run programs on remote computers. Using remote execution, a researcher at one institution would be able to access an underutilized computer at a different site. It quickly became apparent that remote execution was not what the network was actually used for. Instead, the "killer application" was electronic mail: the transfer of messages between computer users at different locations.

In 1972, Bob Kahn proposed to extend ARPANET into the *Internet:* a collection of interoperable networks. All networks on the Internet share common *protocols* for data transmission. Kahn and Vinton Cerf developed a protocol, now called TCP/IP (Transmission Control Protocol/Internet Protocol). On January 1, 1983, all hosts on the Internet simultaneously switched to the TCP/IP protocol (which is used to this day).

Over time, researchers, computer scientists, and hobbyists published increasing amounts of information on the Internet. For example, Project Gutenberg makes available the text of important classical books, whose copyright has expired, in computer-readable form (`http://www.gutenberg.org`). In 1989, Tim Berners-Lee, a computer scientist at CERN (the European organization for nuclear research) started work on hyperlinked documents, allowing users to browse by following links to related documents.



The NCSA Mosaic Browser

This infrastructure is now known as the World Wide Web.

The first interfaces to retrieve this information were, by today's standards, unbelievably clumsy and hard to use. In March 1993, WWW traffic was 0.1 percent of all Internet traffic. All that changed when Marc Andreesen, then a graduate student working for the National Center for Supercomputing Applications (NCSA), released Mosaic. Mosaic displayed web pages in graphical form, using images, fonts, and colors (see the figure). Andreesen went on to fame and fortune at Netscape, and Microsoft licensed the Mosaic code to create Internet Explorer. By 1996, WWW traffic accounted for more than half of the data transported on the Internet.

The Internet has a very democratic structure. Anyone can publish anything, and anyone can read whatever has been published. This does not always sit well with governments and corporations.

Many governments control the Internet infrastructure in their country. For example, an Internet user in China, searching for the Tiananmen Square massacre or air pollution in their hometown, may find nothing. Vietnam blocks access to Facebook, perhaps fearing that anti-government protesters might use it to organize themselves. The U.S. government has required publicly funded libraries and schools to install filters that block sexually-explicit and hate speech, and its security organizations have spied on the Internet usage of citizens.

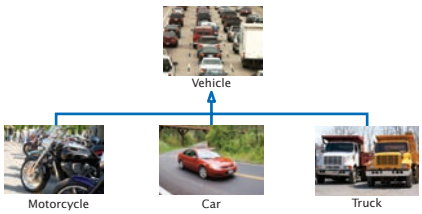
When the Internet is delivered by phone or TV cable companies, those companies sometimes slow down or block competing offerings. As this book is written, Americans discuss the merit of "net neutrality", the principle that carriers should give equal treatment to all data.

The Internet has become a powerful force for delivering information—both good and bad. It is our responsibility as citizens to demand of our government that we can control which information to access.

CHAPTER SUMMARY

Explain the notions of inheritance, superclass, and subclass.

- A subclass inherits data and behavior from a superclass.
- You can always use a subclass object in place of a superclass object.



Implement subclasses in Java.

- A subclass inherits all methods that it does not override.
- A subclass can override a superclass method by providing a new implementation.
- The `extends` reserved word indicates that a class inherits from a superclass.

Implement methods that override methods from a superclass.

- An overriding method can extend or replace the functionality of the superclass method.
- Use the reserved word `super` to call a superclass method.
- Unless specified otherwise, the subclass constructor calls the superclass constructor with no arguments.
- To call a superclass constructor, use the `super` reserved word in the first statement of the subclass constructor.
- The constructor of a subclass can pass arguments to a superclass constructor, using the reserved word `super`.

Use polymorphism for processing objects of related types.

- A subclass reference can be used when a superclass reference is expected.
- When the virtual machine calls an instance method, it locates the method of the implicit parameter's class. This is called dynamic method lookup.
- Polymorphism ("having multiple forms") allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.

Work with the `Object` class and its methods.

- Override the `toString` method to yield a string that describes the object's state.
- The `equals` method checks whether two objects have the same contents.
- If you know that an object belongs to a given class, use a cast to convert the type.
- The `instanceof` operator tests whether an object belongs to a particular type.

REVIEW EXERCISES

■ **R9.1** What are all the superclasses of the `JFrame` class? Consult the Java API documentation or Appendix D.

■ **R9.2** In Worked Example 9.1,

a. What are the subclasses of `Employee`?

b. What are the superclasses of `Manager`?

c. What are the super- and subclasses of `SalariedEmployee`?

d. Which classes override the `weeklyPay` method of the `Employee` class?

e. Which classes override the `setName` method of the `Employee` class?

f. What are the instance variables of an `HourlyEmployee` object?

■ **R9.3** Identify the superclass and subclass in each of the following pairs of classes.

a. `Employee`, `Manager`

b. `GraduateStudent`, `Student`

c. `Person`, `Student`

d. `Employee`, `Professor`

e. `BankAccount`, `CheckingAccount`

f. `Vehicle`, `Car`

g. `Vehicle`, `Minivan`

h. `Car`, `Minivan`

i. `Truck`, `Vehicle`

■ **R9.4** Consider a program for managing inventory in a small appliance store. Why isn't it useful to have a superclass `SmallAppliance` and subclasses `Toaster`, `CarVacuum`, `TravelIron`, and so on?

■ **R9.5** Which methods does the `ChoiceQuestion` class inherit from its superclass? Which methods does it override? Which methods does it add?

■ **R9.6** Which methods does the `SavingsAccount` class in How To 9.1 inherit from its superclass? Which methods does it override? Which methods does it add?

■ **R9.7** List the instance variables of a `CheckingAccount` object from How To 9.1.

■■ **R9.8** Suppose the class `Sub` extends the class `Sandwich`. Which of the following assignments are legal?

```
Sandwich x = new Sandwich();
Sub y = new Sub();
```

a. `x = y;`

b. `y = x;`

c. `y = new Sandwich();`

d. `x = new Sub();`

■ **R9.9** Draw an inheritance diagram that shows the inheritance relationships between these classes.

- `Person`
- `Employee`
- `Student`
- `Instructor`
- `Classroom`
- `Object`

■ **R9.10** In an object-oriented traffic simulation system, we have the classes listed below. Draw an inheritance diagram that shows the relationships between these classes.

- `Vehicle`
- `Car`
- `Truck`
- `Sedan`
- `Coupe`
- `PickupTruck`
- `SportUtilityVehicle`
- `Minivan`
- `Bicycle`
- `Motorcycle`

■ **R9.11** What inheritance relationships would you establish among the following classes?

- `Student`
- `Professor`
- `TeachingAssistant`
- `Employee`
- `Secretary`
- `DepartmentChair`
- `Janitor`
- `SeminarSpeaker`
- `Person`
- `Course`
- `Seminar`
- `Lecture`
- `ComputerLab`

■■ **R9.12** How does a cast such as `(BankAccount) x` differ from a cast of number values such as `(int) x`?

■■■ **R9.13** Which of these conditions returns `true`? Check the Java documentation for the inheritance patterns. Recall that `System.out` is an object of the `PrintStream` class.

a. `System.out instanceof PrintStream`

b. `System.out instanceof OutputStream`

c. `System.out instanceof LogStream`
d. `System.out instanceof Object`
e. `System.out instanceof String`
f. `System.out instanceof Writer`

PRACTICE EXERCISES

■■ **E9.1** Suppose the class `Employee` is declared as follows:

```
public class Employee
{
   private String name;
   private double baseSalary;

   public void setName(String newName) { . . . }
   public void setBaseSalary(double newSalary) { . . . }
   public String getName() { . . . }
   public double getSalary() { . . . }
}
```

Declare a class `Manager` that inherits from the class `Employee` and adds an instance variable `bonus` for storing a salary bonus. Implement the constructors and methods, and supply a test program.

■ **E9.2** Implement a subclass of `BankAccount` called `BasicAccount` whose `withdraw` method will not withdraw more money than is currently in the account.

■ **E9.3** Implement a subclass of `BankAccount` called `BasicAccount` whose `withdraw` method charges a penalty of $30 for each withdrawal that results in an overdraft.

■■ **E9.4** Reimplement the `CheckingAccount` class from How To 9.1 so that the first overdraft in any given month incurs a $20 penalty, and any further overdrafts in the same month result in a $30 penalty.

■■ **E9.5** Add a class `NumericQuestion` to the question hierarchy of Section 9.1. If the response and the expected answer differ by no more than 0.01, accept the response as correct.

■■ **E9.6** Add a class `FillInQuestion` to the question hierarchy of Section 9.1. Such a question is constructed with a string that contains the answer, surrounded by _ _, for example, `"The inventor of Java was _James Gosling_"`. The question should be displayed as

```
The inventor of Java was _____
```

■ **E9.7** Modify the `checkAnswer` method of the `Question` class so that it does not take into account different spaces or upper/lowercase characters. For example, the response `"JAMES gosling"` should match an answer of `"James Gosling"`.

■■ **E9.8** Add a class `AnyCorrectChoiceQuestion` to the question hierarchy of Section 9.1 that allows multiple correct choices. The respondent should provide any one of the correct choices. The answer string should contain all of the correct choices, separated by spaces. Provide instructions in the question text.

■■ **E9.9** Add a class `MultiChoiceQuestion` to the question hierarchy of Section 9.1 that allows multiple correct choices. The respondent should provide all correct choices, separated by spaces. Provide instructions in the question text.

■■ **E9.10** Add a method `addText` to the `Question` superclass and provide a different implementation of `ChoiceQuestion` that calls `addText` rather than storing an array list of choices.

■ **E9.11** Provide `toString` methods for the `Question` and `ChoiceQuestion` classes.

■■ **E9.12** Implement a superclass `Person`. Make two classes, `Student` and `Instructor`, that inherit from `Person`. A person has a name and a year of birth. A student has a major, and an instructor has a salary. Write the class declarations, the constructors, and the methods `toString` for all classes. Supply a test program for these classes and methods.

■■ **E9.13** Make a class `Employee` with a name and salary. Make a class `Manager` inherit from `Employee`. Add an instance variable, named `department`, of type `String`. Supply a method `toString` that prints the manager's name, department, and salary. Make a class `Executive` inherit from `Manager`. Supply appropriate `toString` methods for all classes. Supply a test program that tests these classes and methods.

■■ **E9.14** The `java.awt.Rectangle` class of the standard Java library does not supply a method to compute the area or perimeter of a rectangle. Provide a subclass `BetterRectangle` of the `Rectangle` class that has `getPerimeter` and `getArea` methods. *Do not add any instance variables*. In the constructor, call the `setLocation` and `setSize` methods of the `Rectangle` class. Provide a program that tests the methods that you supplied.

■■■ **E9.15** Repeat Exercise •• E9.14, but in the `BetterRectangle` constructor, invoke the superclass constructor.

■■ **E9.16** A labeled point has *x*- and *y*-coordinates and a string label. Provide a class `LabeledPoint` with a constructor `LabeledPoint(int x, int y, String label)` and a `toString` method that displays `x`, `y`, and the label.

■■ **E9.17** Reimplement the `LabeledPoint` class of Exercise •• E9.16 by storing the location in a `java.awt.Point` object. Your `toString` method should invoke the `toString` method of the `Point` class.

■■ **Business E9.18** Change the `CheckingAccount` class in How To 9.1 so that a $1 fee is levied for deposits or withdrawals in excess of three free monthly transactions. Place the code for computing the fee into a separate method that you call from the `deposit` and `withdraw` methods.

PROGRAMMING PROJECTS

■■ **P9.1** Implement a class `ChessPiece` with method `setPosition(String coordinates)`. The coordinate string identifies the row and column in chess notation, such as `"d8"` for the initial position of the black queen. Also provide a method `ArrayList<String> canMoveTo()` that enumerates the valid moves from the current position. Provide subclasses `Pawn`, `Knight`, `Bishop`, `Rook`, `Queen`, and `King`.

■■ **P9.2** Implement a class `Clock` whose `getHours` and `getMinutes` methods return the current time at your location. (Call `java.time.LocalTime.now().toString()` and extract the time from that string.) Also provide a `getTime` method that returns a string with the hours and minutes by calling the `getHours` and `getMinutes` methods. Provide a subclass `WorldClock` whose constructor accepts a time offset. For example, if you live in California,

a `new WorldClock(3)` should show the time in New York, three time zones ahead. Which methods did you override? (You should not override `getTime`.)

■■ **P9.3** Add an alarm feature to the `Clock` class of Exercise •• P9.2. When `setAlarm(hours, minutes)` is called, the clock stores the alarm. When you call `getTime`, and the alarm time has been reached or exceeded, return the time followed by the string `"Alarm"` (or, if you prefer, the string `"\u23F0"`) and clear the alarm. What do you need to do to make the `setAlarm` method work for `WorldClock` objects?

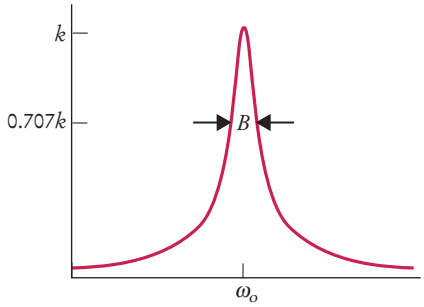
■■ **Business P9.4** Implement a superclass `Appointment` and subclasses `Onetime`, `Daily`, and `Monthly`. An appointment has a description (for example, "see the dentist") and occurs on one or more dates. Write a method `occursOn(int year, int month, int day)` that checks whether the appointment occurs on that date. For example, for a monthly appointment, you must check whether the day of the month matches. Then fill an array of `Appointment` objects with a mixture of appointments. Have the user enter a date and print out all appointments that occur on that date.

© Pali Rao/iStockphoto.

■■ **Business P9.5** Improve the appointment book program of Exercise •• Business P9.4. Give the user the option to add new appointments. The user must specify the type of the appointment, the description, and the date.

■■■ **Business P9.6** Improve the appointment book program of Exercises •• Business P9.4 and •• Business P9.5 by letting the user save the appointment data to a file and reload the data from a file. The saving part is straightforward: Make a method `save`. Save the type, description, and date to a file. The loading part is not so easy. First determine the type of the appointment to be loaded, create an object of that type, and then call a `load` method to load the data.

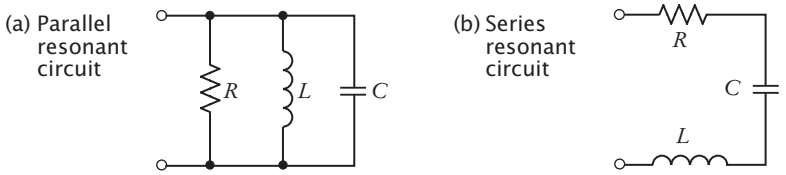
■■ **Science P9.7** Resonant circuits are used to select a signal (e.g., a radio station or TV channel) from among other competing signals. Resonant circuits are characterized by the frequency response shown in the figure below. The resonant frequency response is completely described by three parameters: the resonant frequency, ω_o, the bandwidth, B, and the gain at the resonant frequency, k.



Frequency (rad/s, log scale)

Two simple resonant circuits are shown in the figure below. The circuit in (a) is called a *parallel resonant circuit*. The circuit in (b) is called a *series resonant circuit*.

Both resonant circuits consist of a resistor having resistance R, a capacitor having capacitance C, and an inductor having inductance L.



These circuits are designed by determining values of R, C, and L that cause the resonant frequency response to be described by specified values of ω_o, B, and k. The design equations for the parallel resonant circuit are:

$$R = k, \quad C = \frac{1}{BR}, \text{ and } \quad L = \frac{1}{\omega_o^2 C}$$

Similarly, the design equations for the series resonant circuit are:

$$R = \frac{1}{k}, \quad L = \frac{R}{B}, \text{ and } \quad C = \frac{1}{\omega_o^2 L}$$

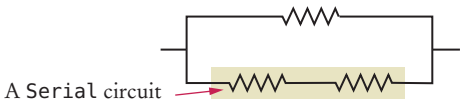
Write a Java program that represents `ResonantCircuit` as a superclass and represents `SeriesResonantCircuit` and `ParallelResonantCircuit` as subclasses. Give the superclass three private instance variables representing the parameters ω_o, B, and k of the resonant frequency response. The superclass should provide public instance methods to get and set each of these variables. The superclass should also provide a `display` method that prints a description of the resonant frequency response.

Each subclass should provide a method that designs the corresponding resonant circuit. The subclasses should also override the `display` method of the superclass to print descriptions of both the frequency response (the values of ω_o, B, and k) and the circuit (the values of R, C, and L).

All classes should provide appropriate constructors.

Supply a class that demonstrates that the subclasses all work properly.

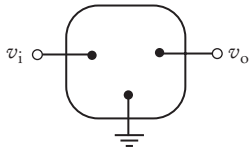
■■■ **Science P9.8** In this problem, you will model a circuit consisting of an arbitrary configuration of resistors. Provide a superclass `Circuit` with a instance method `getResistance`. Provide a subclass `Resistor` representing a single resistor. Provide subclasses `Serial` and `Parallel`, each of which contains an `ArrayList<Circuit>`. A `Serial` circuit models a series of circuits, each of which can be a single resistor or another circuit. Similarly, a `Parallel` circuit models a set of circuits in parallel. For example, the following circuit is a `Parallel` circuit containing a single resistor and one `Serial` circuit:



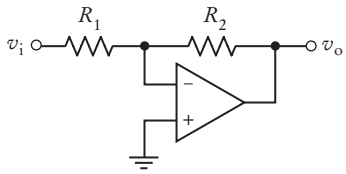
Use Ohm's law to compute the combined resistance.

■■ **Science P9.9** Part (a) of the figure below shows a symbolic representation of an electric circuit called an *amplifier*. The input to the amplifier is the voltage v_i and the output is the voltage v_o. The output of an amplifier is proportional to the input. The constant of proportionality is called the "gain" of the amplifier.

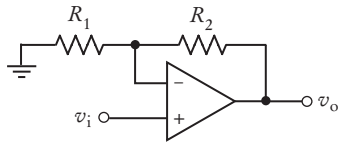
(a) Amplifier



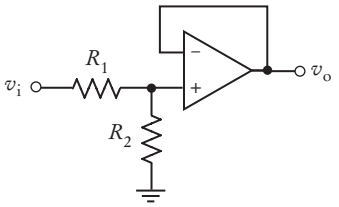
(b) Inverting amplifier



(c) Noninverting amplifier



(d) Voltage divider amplifier



Parts (b), (c), and (d) show schematics of three specific types of amplifier: the *inverting amplifier*, *noninverting amplifier*, and *voltage divider amplifier*. Each of these three amplifiers consists of two resistors and an op amp. The value of the gain of each amplifier depends on the values of its resistances. In particular, the gain, g, of the inverting amplifier is given by $g = -\frac{R_2}{R_1}$. Similarly the gains of the noninverting amplifier and voltage divider amplifier are given by $g = 1 + \frac{R_2}{R_1}$ and $g = \frac{R_2}{R_1 + R_2}$, respectively.

Write a Java program that represents the amplifier as a superclass and represents the inverting, noninverting, and voltage divider amplifiers as subclasses. Give the superclass a `getGain` method and a `getDescription` method that returns a string identifying the amplifier. Each subclass should have a constructor with two arguments, the resistances of the amplifier. The subclasses need to override the `getGain` and `getDescription` methods of the superclass.

Supply a class that demonstrates that the subclasses all work properly for sample values of the resistances.