```
            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 == "Q")
         {
            done = true;
         }
      }
```

ONLINE EXAMPLE

The complete program with BankAccount, SavingsAccount, and CheckingAccount classes.

WORKED EXAMPLE 9.1

Implementing an Employee Hierarchy for Payroll Processing

This Worked Example shows how to implement payroll processing that works for different kinds of employees.

VIDEO EXAMPLE 9.1

Building a Discussion Board

In this Video Example, we will build a discussion board for students and instructors.

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 8). 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 15.1).

Available online in WileyPLUS and at `www.wiley.com/college/horstmann`.

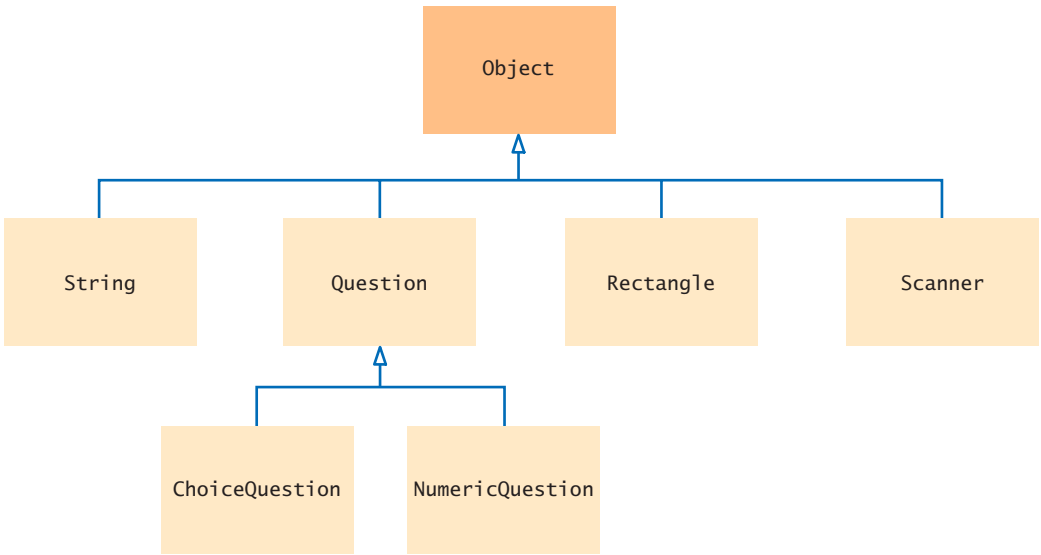


Figure 8 The `Object` Class Is the Superclass of Every Java Class

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]"
```

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 15.1 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 9
```

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 10
```

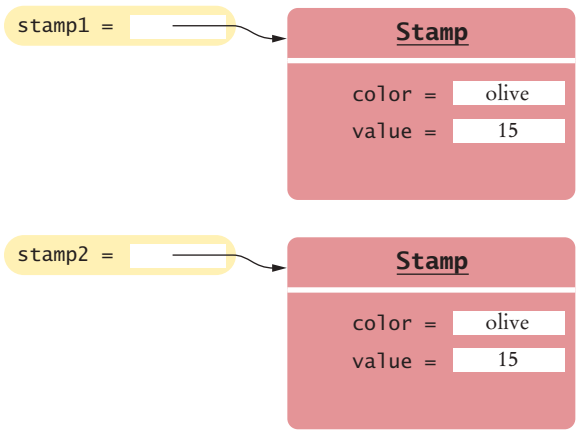


Figure 9 Two References to Equal Objects

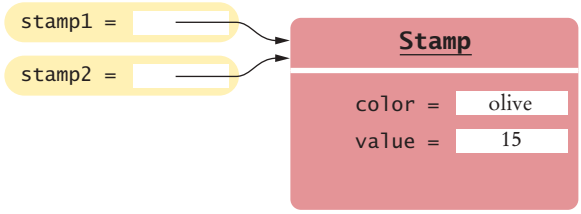


Figure 10 Two References to the Same Object

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

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

```
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
```

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

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

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;
```

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.

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

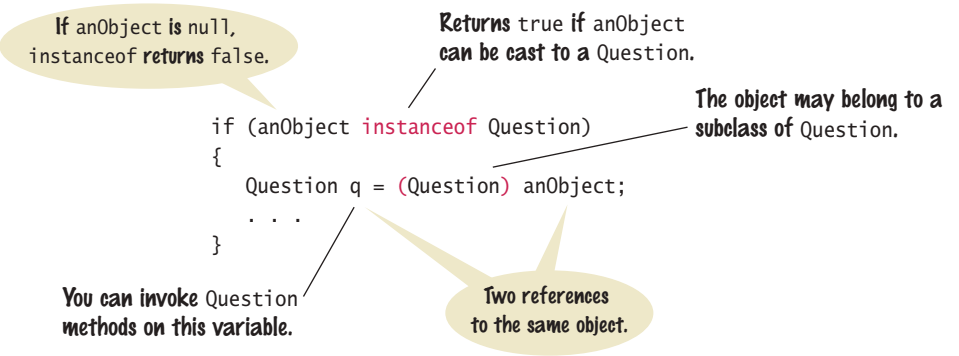
To protect against bad casts, you can use the `instanceof` operator. It tests whether an object belongs to a particular type. For example,

```
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`.

Syntax 9.3 The instanceof Operator

Syntax *object* `instanceof` *TypeName*



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 on page 446
{
   // 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();
```

ONLINE EXAMPLE

A program that demonstrates the `toString` method and the `instanceof` operator.

21. Why does the call

```
System.out.println(System.out);
```

produce a result such as `java.io.PrintStream@7a84e4`?

22. Will the following code fragment compile? Will it run? If not, what error is reported?

```
Object obj = "Hello";
System.out.println(obj.length());
```

23. Will the following code fragment compile? Will it run? If not, what error is reported?

```
Object obj = "Who was the inventor of Java?";
Question q = (Question) obj;
q.display();
```

24. Why don't we simply store all objects in variables of type `Object`?

25. Assuming that `x` is an object reference, what is the value of `x instanceof Object`?

Practice It Now you can try these exercises at the end of the chapter: P9.7, P9.8, P9.12.

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;
   }
}
```