CHAPTER 8

DESIGNING CLASSES

CHAPTER GOALS

- To learn how to choose appropriate classes for a given problem
- To understand the concept of cohesion
- To minimize dependencies and side effects
- To learn how to find a data representation for a class
- To understand static methods and variables
- To learn about packages
- To learn about unit testing frameworks

CHAPTER CONTENTS

Good design should be both functional and attractive. When designing classes, each class should be dedicated to a particular purpose, and classes should work well together. In this chapter, you will learn how to discover classes, design good methods, and choose appropriate data representations. You will also learn how to design features that belong to the class as a whole, not individual objects, by using static methods and variables. You will see how to use packages to organize your classes. Finally, we introduce the JUnit testing framework that lets you verify the functionality of your classes.

8.1 Discovering Classes

You have used a good number of classes in the preceding chapters and probably designed a few classes yourself as part of your programming assignments. Designing a class can be a challenge—it is not always easy to tell how to start or whether the result is of good quality.

A class should represent a single concept from a problem domain, such as business, science, or mathematics.

What makes a good class? Most importantly, a class should *represent a single concept* from a problem domain. Some of the classes that you have seen represent concepts from mathematics:

- `Point`
- `Rectangle`
- `Ellipse`

Other classes are abstractions of real-life entities:

- `BankAccount`
- `CashRegister`

For these classes, the properties of a typical object are easy to understand. A `Rectangle` object has a width and height. Given a `BankAccount` object, you can deposit and withdraw money. Generally, concepts from a domain related to the program's purpose, such as science, business, or gaming, make good classes. The name for such a class should be a noun that describes the concept. In fact, a simple rule of thumb for getting started with class design is to look for nouns in the problem description.

One useful category of classes can be described as *actors*. Objects of an actor class carry out certain tasks for you. Examples of actors are the `Scanner` class of Chapter 4 and the `Random` class in Chapter 6. A `Scanner` object scans a stream for numbers and strings. A `Random` object generates random numbers. It is a good idea to choose class names for actors that end in "-er" or "-or". (A better name for the `Random` class might be `RandomNumberGenerator`.)

Very occasionally, a class has no objects, but it contains a collection of related static methods and constants. The `Math` class is an example. Such a class is called a *utility class*.

Finally, you have seen classes with only a `main` method. Their sole purpose is to start a program. From a design perspective, these are somewhat degenerate examples of classes.

What might not be a good class? If you can't tell from the class name what an object of the class is supposed to do, then you are probably not on the right track. For

example, your homework assignment might ask you to write a program that prints paychecks. Suppose you start by trying to design a class `PaycheckProgram`. What would an object of this class do? An object of this class would have to do everything that the homework needs to do. That doesn't simplify anything. A better class would be `Paycheck`. Then your program can manipulate one or more `Paycheck` objects.

Another common mistake is to turn a single operation into a class. For example, if your homework assignment is to compute a paycheck, you may consider writing a class `ComputePaycheck`. But can you visualize a "ComputePaycheck" object? The fact that "ComputePaycheck" isn't a noun tips you off that you are on the wrong track. On the other hand, a `Paycheck` class makes intuitive sense. The word "paycheck" is a noun. You can visualize a paycheck object. You can then think about useful methods of the `Paycheck` class, such as `computeTaxes`, that help you solve the assignment.

1. What is a simple rule of thumb for finding classes?
2. Your job is to write a program that plays chess. Might `ChessBoard` be an appropriate class? How about `MovePiece`?

Practice It Now you can try these exercises at the end of the chapter: R8.1, R8.2, R8.3.

8.2 Designing Good Methods

In the following sections, you will learn several useful criteria for analyzing and improving the public interface of a class.

8.2.1 Providing a Cohesive Public Interface

The public interface of a class is cohesive if all of its features are related to the concept that the class represents.

A class should represent a single concept. All interface features should be closely related to the single concept that the class represents. Such a public interface is said to be **cohesive**.

The members of a cohesive team have a common goal.

If you find that the public interface of a class refers to multiple concepts, then that is a good sign that it may be time to use separate classes instead. Consider, for example, the public interface of the `CashRegister` class in Chapter 4:

```
public class CashRegister
{
   public static final double QUARTER_VALUE = 0.25;
   public static final double DIME_VALUE = 0.1;
   public static final double NICKEL_VALUE = 0.05;
   . . .
   public void receivePayment(int dollars, int quarters,
         int dimes, int nickels, int pennies)
   . . .
}
```

There are really two concepts here: a cash register that holds coins and computes their total, and the values of individual coins. (For simplicity, we assume that the cash register only holds coins, not bills. Exercise E8.3 discusses a more general solution.)

It makes sense to have a separate `Coin` class and have coins responsible for knowing their values.

```
public class Coin
{
   . . .
   public Coin(double aValue, String aName) { . . . }
   public double getValue() { . . . }
   . . .
}
```

Then the `CashRegister` class can be simplified:

```
public class CashRegister
{
   . . .
   public void receivePayment(int coinCount, Coin coinType) { . . . }
   {
      payment = payment + coinCount * coinType.getValue();
   }
   . . .
}
```

Now the `CashRegister` class no longer needs to know anything about coin values. The same class can equally well handle euros or zorkmids!

This is clearly a better solution, because it separates the responsibilities of the cash register and the coins. The only reason we didn't follow this approach in Chapter 4 was to keep the `CashRegister` example simple.

8.2.2 Minimizing Dependencies

A class depends on another class if its methods use that class in any way.

Many methods need other classes in order to do their jobs. For example, the `receivePayment` method of the restructured `CashRegister` class now uses the `Coin` class. We say that the `CashRegister` class *depends on* the `Coin` class.

To visualize relationships between classes, such as dependence, programmers draw class diagrams. In this book, we use the UML ("**Unified Modeling Language**") notation for objects and classes. UML is a notation for object-oriented analysis and design invented by Grady Booch, Ivar Jacobson, and James Rumbaugh, three leading researchers in object-oriented softare development. (Appendix H has a summary of the UML notation used in this book.) The UML notation distinguishes between *object diagrams* and class diagrams. In an object diagram the class names are underlined; in a class diagram the class names are not underlined. In a class diagram, you denote dependency by a dashed line with a ≫-shaped open arrow tip that points to the dependent class. Figure 1 shows a class diagram indicating that the `CashRegister` class depends on the `Coin` class.

Note that the `Coin` class does *not* depend on the `CashRegister` class. All `Coin` methods can carry out their work without ever calling any method in the `CashRegister` class. Conceptually, coins have no idea that they are being collected in cash registers.

Here is an example of minimizing dependencies. Consider how we have always printed a bank balance:

```
System.out.println("The balance is now $" + momsSavings.getBalance());
```

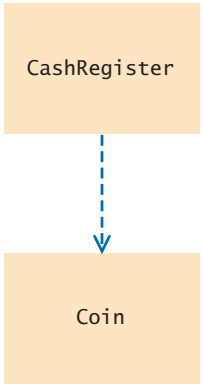


Figure 1
Dependency Relationship Between the `CashRegister` and `Coin` Classes

Why don't we simply have a `printBalance` method?

```
public void printBalance() // Not recommended
{
   System.out.println("The balance is now $" + balance);
}
```

The method depends on `System.out`. Not every computing environment has `System.out`. For example, an automatic teller machine doesn't display console messages. In other words, this design violates the rule of minimizing dependencies. The `printBalance` method couples the `BankAccount` class with the `System` and `PrintStream` classes.

It is best to place the code for producing output or consuming input in a separate class. That way, you decouple input/output from the actual work of your classes.

8.2.3 Separating Accessors and Mutators

A **mutator method** changes the state of an object. Conversely, an **accessor method** asks an object to compute a result, without changing the state.

An immutable class has no mutator methods.

Some classes have been designed to have only accessor methods and no mutator methods at all. Such classes are called **immutable**. An example is the `String` class. Once a string has been constructed, its content never changes. No method in the `String` class can modify the contents of a string. For example, the `toUpperCase` method does not change characters from the original string. Instead, it constructs a *new* string that contains the uppercase characters:

```
String name = "John Q. Public";
String uppercased = name.toUpperCase(); // name is not changed
```

References to objects of an immutable class can be safely shared.

An immutable class has a major advantage: It is safe to give out references to its objects freely. If no method can change the object's value, then no code can modify the object at an unexpected time.

Not every class should be immutable. Immutability makes most sense for classes that represent values, such as strings, dates, currency amounts, colors, and so on.

In mutable classes, it is still a good idea to cleanly separate accessors and mutators, in order to avoid accidental mutation. As a rule of thumb, a method that returns a value should not be a mutator. For example, one would not expect that calling `getBalance` on a `BankAccount` object would change the balance. (You would be pretty upset if your bank charged you a "balance inquiry fee".) If you follow this rule, then all mutators of your class have return type `void`.

Sometimes, this rule is bent a bit, and mutator methods return an informational value. For example, the `ArrayList` class has a `remove` method to remove an object.

```
ArrayList<String> names = ...;
boolean success = names.remove("Romeo");
```

That method returns `true` if the removal was successful; that is, if the list contained the object. Returning this value might be bad design if there was no other way to check whether an object exists in the list. However, there is such a method—the `contains` method. It is acceptable for a mutator to return a value if there is also an accessor that computes it.

The situation is less happy with the `Scanner` class. The `next` method is a mutator that returns a value. (The `next` method really is a mutator. If you call `next` twice in a row, it can return different results, so it must have mutated something inside the `Scanner` object.) Unfortunately, there is no accessor that returns the same value. This sometimes makes it awkward to use a `Scanner`. You must carefully hang on to the value that the `next` method returns because you have no second chance to ask for it. It would have been better if there was another method, say `peek`, that yields the next input without consuming it.

To check the temperature of the water in the bottle, you could take a sip, but that would be the equivalent of a mutator method.

8.2.4 Minimizing Side Effects

A side effect of a method is any externally observable data modification.

A **side effect** of a method is any kind of modification of data that is observable outside the method. Mutator methods have a side effect, namely the modification of the implicit parameter.

There is another kind of side effect that you should avoid. A method should generally not modify its parameter variables. Consider this example:

```
/**
   Computes the total balance of the given accounts.
   @param accounts a list of bank accounts
*/
   public double getTotalBalance(ArrayList<String> accounts)
   {
      double sum = 0;
      while (studentNames.size() > 0)
      {
         BankAccount account = accounts.remove(0); // Not recommended
         sum = sum + account.getBalance();
      }
      return sum;
   }
}
```

This method *removes* all names from the `accounts` parameter variable. After a call

```
double total = getTotalBalance(allAccounts);
```

`allAccounts` is empty! Such a side effect would not be what most programmers expect. It is better if the method visits the elements from the list without removing them.

When designing methods, minimize side effects.

Another example of a side effect is output. Consider again the `printBalance` method that we discussed in Section 8.2.2:

```
public void printBalance() // Not recommended
{
   System.out.println("The balance is now $" + balance);
}
```

This method mutates the `System.out` object, which is not a part of the `BankAccount` object. That is a side effect.

To avoid this side effect, keep most of your classes free from input and output operations, and concentrate input and output in one place, such as the main method of your program.

This taxi has an undesirable side effect, spraying bystanders with muddy water.

3. Why is the `CashRegister` class from Chapter 4 not cohesive?
4. Why does the `Coin` class not depend on the `CashRegister` class?
5. Why is it a good idea to minimize dependencies between classes?
6. Is the `substring` method of the `String` class an accessor or a mutator?
7. Is the `Rectangle` class immutable?
8. If a refers to a bank account, then the call `a.deposit(100)` modifies the bank account object. Is that a side effect?
9. Consider the `Student` class of Chapter 7. Suppose we add a method

```
void read(Scanner in)
{
   while (in.hasNextDouble())
   {
      addScore(in.nextDouble());
   }
}
```

Does this method have a side effect other than mutating the scores?

Practice It Now you can try these exercises at the end of the chapter: R8.4, R8.5, R8.9.

Programming Tip 8.1

Consistency

In this section you learned of two criteria for analyzing the quality of the public interface of a class. You should maximize cohesion and remove unnecessary dependencies. There is another criterion that we would like you to pay attention to—*consistency*. When you have a set of methods, follow a consistent scheme for their names and parameter variables. This is simply a sign of good craftsmanship.

Sadly, you can find any number of inconsistencies in the standard library. Here is an example: To show an input dialog box, you call

```
JOptionPane.showInputDialog(promptString)
```

To show a message dialog box, you call

```
JOptionPane.showMessageDialog(null, messageString)
```

What's the `null` argument? It turns out that the `showMessageDialog` method needs an argument to specify the parent window, or `null` if no parent window is required. But the `showInputDialog` method requires no parent window. Why the inconsistency? There is no reason. It would have been an easy matter to supply a `showMessageDialog` method that exactly mirrors the `showInputDialog` method.

Inconsistencies such as these are not fatal flaws, but they are an annoyance, particularly because they can be so easily avoided.

While it is possible to eat with mismatched silverware, consistency is more pleasant.

Special Topic 8.1

Call by Value and Call by Reference

In Section 8.2.4, we recommended that you don't invoke a mutator method on a parameter variable. In this Special Topic, we discuss a related issue—what happens when you assign a new value to a parameter variable. Consider this method:

```
public class BankAccount
{
   . . .
   /**
      Transfers money from this account and tries to add it to a balance.
      @param amount the amount of money to transfer
      @param otherBalance balance to add the amount to
   */
   public void transfer(double amount, double otherBalance)  2
   {
      balance = balance - amount;
      otherBalance = otherBalance + amount;
         // Won't update the argument
   }  3
}
```

Now let's see what happens when we call the method:

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance);  1
System.out.println(savingsBalance);  4
```

You might expect that after the call, the `savingsBalance` variable has been incremented to 1500. However, that is not the case. As the method starts, the parameter variable `otherBalance` is set to the same value as `savingsBalance` (see Figure 2). Then the variable is set to a different value. That modification has no effect on `savingsBalance`, because `otherBalance` is a separate variable. When the method terminates, the `otherBalance` variable is removed, and `savingsBalance` isn't increased.

In Java, a method can never change the contents of a variable that is passed to a method.

In Java, parameter variables are initialized with the values of the argument expressions. When the method exits, the parameter variables are removed. Computer scientists refer to this call mechanism as "call by value".

For that reason, a Java method can never change the contents of a variable that is passed as an argument—the method manipulates a different variable.

Other programming languages such as C++ support a mechanism, called "call by reference", that can change the arguments of a method call. You will sometimes read in Java books

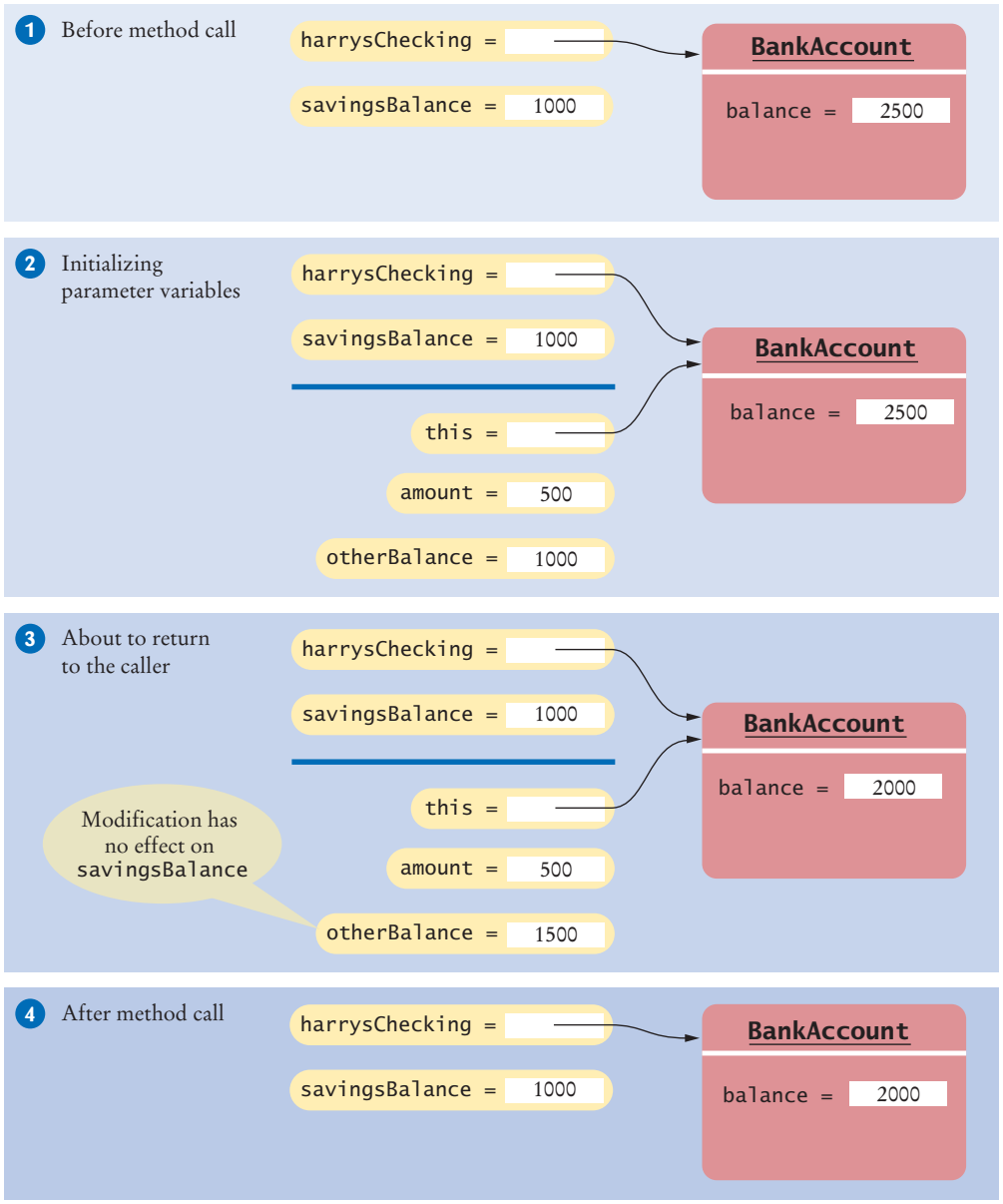


Figure 2 Modifying a Parameter Variable of a Primitive Type Has No Effect on Caller

that "numbers are passed by value, objects are passed by reference". That is technically not quite correct. In Java, objects themselves are never passed as arguments; instead, both numbers and *object references* are passed by value.

The confusion arises because a Java method can mutate an object when it receives an object reference as an argument (see Figure 3).

```
public class BankAccount
{
   . . .
   /**
      Transfers money from this account to another.
      @param amount the amount of money to transfer
      @param otherAccount account to add the amount to
```

```
   */
   public void transfer(double amount, BankAccount otherAccount) 2
   {
      balance = balance - amount;
      otherAccount.deposit(amount);
   } 3
}
```

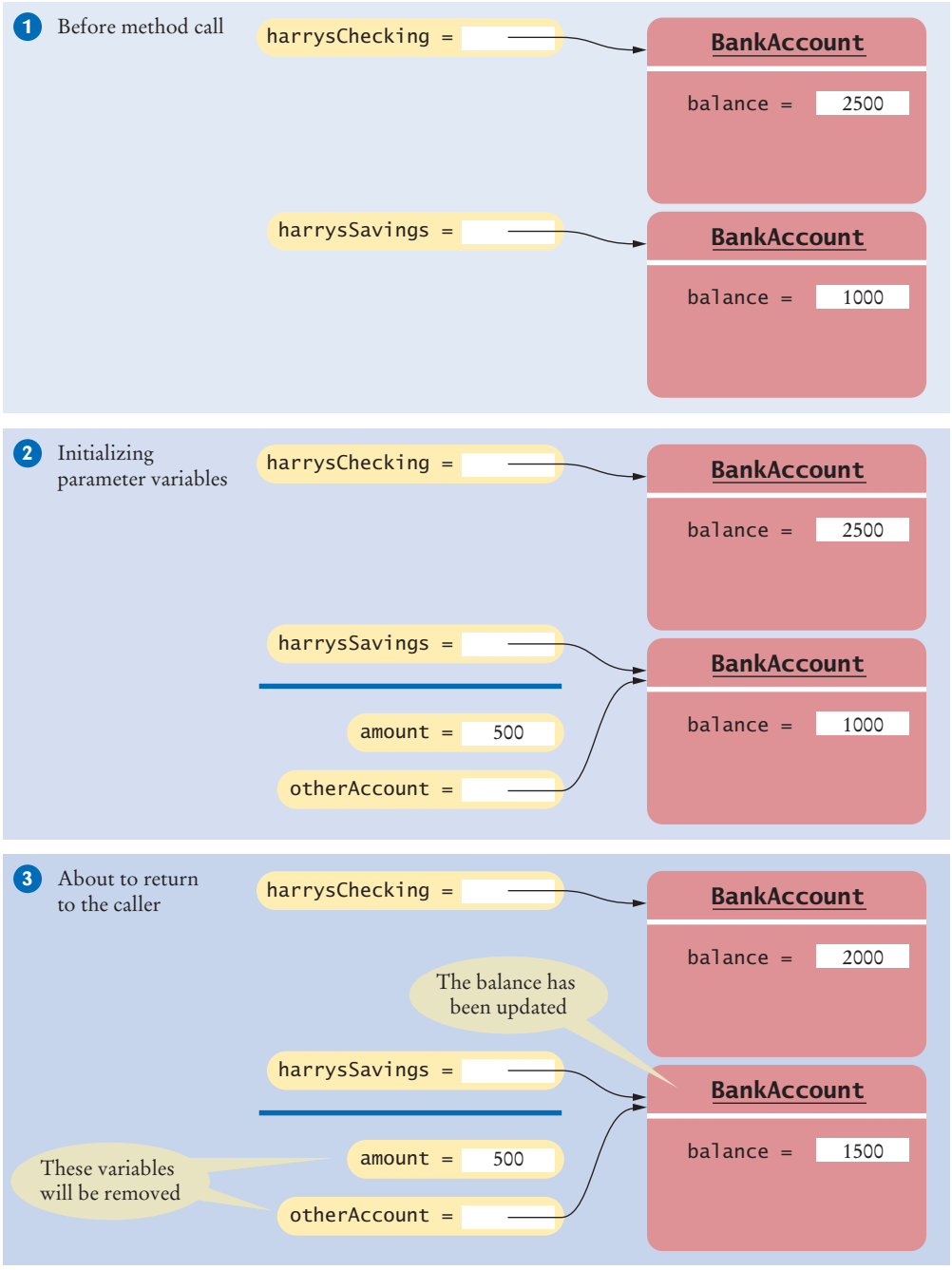


Figure 3 Methods Can Mutate Any Objects to Which They Hold References

Now we pass an object reference to the `transfer` method:

```
BankAccount harrysSavings = new BankAccount(1000);
harrysChecking.transfer(500, harrysSavings); 1
System.out.println(harrysSavings.getBalance());
```

This example works as expected. The parameter variable `otherAccount` contains a *copy* of the object reference `harrysSavings`. You saw in Section 2.8 what is means to make a copy of an object reference—you get another reference to the same object. Through that reference, the method is able to modify the object.

However, a method cannot *replace* an object reference that is passed as an argument. To appreciate this subtle difference, consider this method that tries to set the `otherAccount` parameter variable to a new object:

```
public class BankAccount
{
   . . .
   public void transfer(double amount, BankAccount otherAccount)
   {
      balance = balance - amount;
      double newBalance = otherAccount.balance + amount;
      otherAccount = new BankAccount(newBalance); // Won't work
   }
}
```

In this situation, we are not trying to change the state of the object to which the parameter variable `otherAccount` refers; instead, we are trying to replace the object with a different one (see Figure 4). Now the reference stored in parameter variable `otherAccount` is replaced with a reference to a new account. But if you call the method with

In Java, a method can change the state of an object reference argument, but it cannot replace the object reference with another.

```
harrysChecking.transfer(500, savingsAccount);
```

then that change does not affect the `savingsAccount` variable that is supplied in the call. This example demonstrates that objects are not passed by reference.

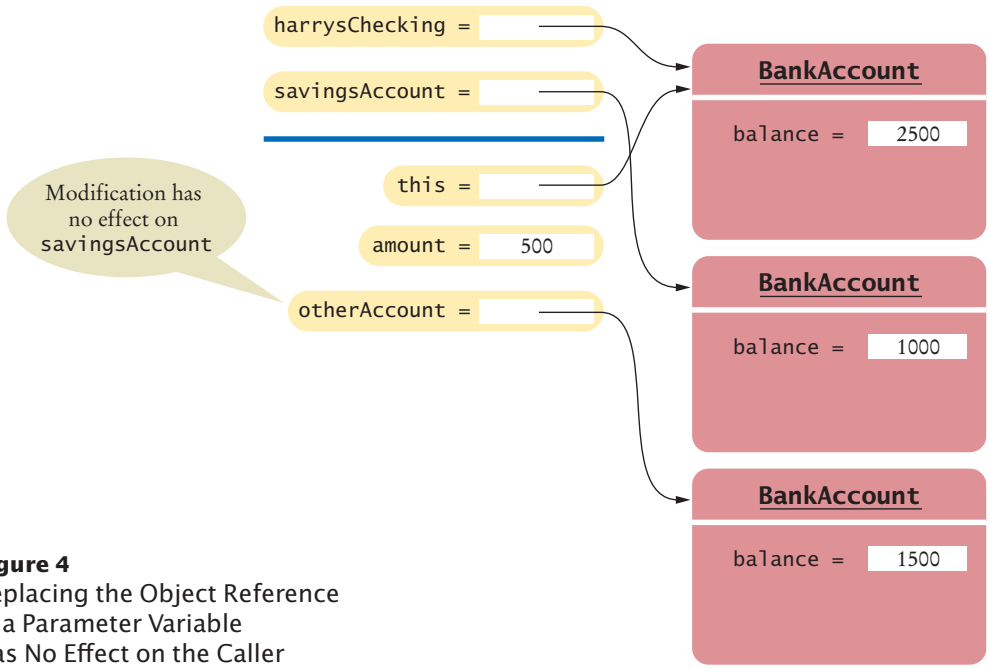


Figure 4
Replacing the Object Reference in a Parameter Variable Has No Effect on the Caller

To summarize:

- A Java method can't change the contents of any variable passed as an argument.
- A Java method can mutate an object when it receives a reference to it as an argument.

8.3 Problem Solving: Patterns for Object Data

When you design a class, you first consider the needs of the programmers who use the class. You provide the methods that the users of your class will call when they manipulate objects. When you implement the class, you need to come up with the instance variables for the class. It is not always obvious how to do this. Fortunately, there is a small set of recurring patterns that you can adapt when you design your own classes. We introduce these patterns in the following sections.

8.3.1 Keeping a Total

An instance variable for the total is updated in methods that increase or decrease the total amount.

Many classes need to keep track of a quantity that can go up or down as certain methods are called. Examples:

- A bank account has a balance that is increased by a deposit, decreased by a withdrawal.
- A cash register has a total that is increased when an item is added to the sale, cleared after the end of the sale.
- A car has gas in the tank, which is increased when fuel is added and decreased when the car drives.

In all of these cases, the implementation strategy is similar. Keep an instance variable that represents the current total. For example, for the cash register:

```
private double purchase;
```

Locate the methods that affect the total. There is usually a method to increase it by a given amount:

```
public void recordPurchase(double amount)
{
   purchase = purchase + amount;
}
```

Depending on the nature of the class, there may be a method that reduces or clears the total. In the case of the cash register, one can provide a `clear` method:

```
public void clear()
{
   purchase = 0;
}
```

There is usually a method that yields the current total. It is easy to implement:

```
public double getAmountDue()
{
   return purchase;
}
```

All classes that manage a total follow the same basic pattern. Find the methods that affect the total and provide the appropriate code for increasing or decreasing it. Find

the methods that report or use the total, and have those methods read the current total.

8.3.2 Counting Events

A counter that counts events is incremented in methods that correspond to the events.

You often need to count how many times certain events occur in the life of an object. For example:

- In a cash register, you may want to know how many items have been added in a sale.
- A bank account charges a fee for each transaction; you need to count them.

Keep a counter, such as

```
private int itemCount;
```

Increment the counter in those methods that correspond to the events that you want to count:

```
public void recordPurchase(double amount)
{
   purchase = purchase + amount;
   itemCount++;
}
```

You may need to clear the counter, for example at the end of a sale or a statement period:

```
public void clear()
{
   purchase = 0;
   itemCount = 0;
}
```

There may or may not be a method that reports the count to the class user. The count may only be used to compute a fee or an average. Find out which methods in your class make use of the count, and read the current value in those methods.

8.3.3 Collecting Values

An object can collect other objects in an array or array list.

Some objects collect numbers, strings, or other objects. For example, each multiple-choice question has a number of choices. A cash register may need to store all prices of the current sale.

Use an array list or an array to store the values. (An array list is usually simpler because you won't need to track the number of values.) For example,

```
public class Question
{
   private ArrayList<String> choices;
   . . .
}
```

A shopping cart object needs to manage a collection of items.

In the constructor, initialize the instance variable to an empty collection:

```
public Question()
{
   choices = new ArrayList<String>();
}
```

You need to supply some mechanism for adding values. It is common to provide a method for appending a value to the collection:

```
public void add(String option)
{
   choices.add(option);
}
```

The user of a `Question` object can call this method multiple times to add the choices.

8.3.4 Managing Properties of an Object

An object property can be accessed with a getter method and changed with a setter method.

A property is a value of an object that an object user can set and retrieve. For example, a `Student` object may have a name and an ID. Provide an instance variable to store the property's value and methods to get and set it.

```
public class Student
{
   private String name;
   . . .
   public String getName() { return name; }
   public void setName(String newName) { name = newName; }
   . . .
}
```

It is common to add error checking to the setter method. For example, we may want to reject a blank name:

```
public void setName(String newName)
{
   if (newName.length() > 0) { name = newName; }
}
```

Some properties should not change after they have been set in the constructor. For example, a student's ID may be fixed (unlike the student's name, which may change). In that case, don't supply a setter method.

```
public class Student
{
   private int id;
   . . .
   public Student(int anId) { id = anId; }
   public String getId() { return id; }
   // No setId method
   . . .
}
```

8.3.5 Modeling Objects with Distinct States

Some objects have behavior that varies depending on what has happened in the past. For example, a `Fish` object may look for food when it is hungry and ignore food after it has eaten. Such an object would need to remember whether it has recently eaten.

If a fish is in a hungry state, its behavior changes.

If your object can have one of several states that affect the behavior, supply an instance variable for the current state.

Supply an instance variable that models the state, together with some constants for the state values:

```
public class Fish
{
   private int hungry;

   public static final int NOT_HUNGRY = 0;
   public static final int SOMEWHAT_HUNGRY = 1;
   public static final int VERY_HUNGRY = 2;
   . . .
}
```

(Alternatively, you can use an enumeration—see Special Topic 5.4.)

Determine which methods change the state. In this example, a fish that has just eaten won't be hungry. But as the fish moves, it will get hungrier:

```
public void eat()
{
   hungry = NOT_HUNGRY;
   . . .
}

public void move()
{
   . . .
   if (hungry < VERY_HUNGRY) { hungry++; }
}
```

Finally, determine where the state affects behavior. A fish that is very hungry will want to look for food first:

```
public void move()
{
   if (hungry == VERY_HUNGRY)
   {
      Look for food.
   }
   . . .
}
```

8.3.6 Describing the Position of an Object

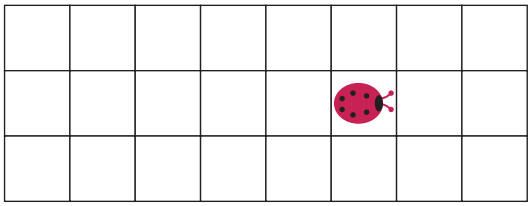
To model a moving object, you need to store and update its position.

Some objects move around during their lifetime, and they remember their current position. For example,

- A train drives along a track and keeps track of the distance from the terminus.
- A simulated bug living on a grid crawls from one grid location to the next, or makes 90 degree turns to the left or right.
- A cannonball is shot into the air, then descends as it is pulled by the gravitational force.

Such objects need to store their position. Depending on the nature of their movement, they may also need to store their orientation or velocity.

A bug in a grid needs to store its row, column, and direction.



If the object moves along a line, you can represent the position as a distance from a fixed point:

```
private double distanceFromTerminus;
```

If the object moves in a grid, remember its current location and direction in the grid:

```
private int row;
private int column;
private int direction; // 0 = North, 1 = East, 2 = South, 3 = West
```

When you model a physical object such as a cannonball, you need to track both the position and the velocity, possibly in two or three dimensions. Here we model a cannonball that is shot upward into the air:

```
private double zPosition;
private double zVelocity;
```

There will be methods that update the position. In the simplest case, you may be told by how much the object moves:

```
public void move(double distanceMoved)
{
   distanceFromTerminus = distanceFromTerminus + distanceMoved;
}
```

If the movement happens in a grid, you need to update the row or column, depending on the current orientation.

```
public void moveOneUnit()
{
   if (direction == NORTH) { row--; }
   else if (direction == EAST) { column++; }
   else if (direction == SOUTH) { row++; }
   else if (direction == WEST) { column--; }
}
```

FULL CODE EXAMPLE

Go to `wiley.com/go/javacode` to download classes that use these patterns for object data.

Exercise P8.6 shows you how to update the position of a physical object with known velocity.

Whenever you have a moving object, keep in mind that your program will simulate the actual movement in some way. Find out the rules of that simulation, such as movement along a line or in a grid with integer coordinates. Those rules determine how to represent the current position. Then locate the methods that move the object, and update the positions according to the rules of the simulation.

10. Suppose we want to count the number of transactions in a bank account in a statement period, and we add a counter to the `BankAccount` class:

```
public class BankAccount
{
   private int transactionCount;
   . . .
```

```
}
```

In which methods does this counter need to be updated?

11. In How To 3.1, the `CashRegister` class does not have a `getTotalPurchase` method. Instead, you have to call `receivePayment` and then `giveChange`. Which recommendation of Section 8.2.4 does this design violate? What is a better alternative?
12. In the example in Section 8.3.3, why is the `add` method required? That is, why can't the user of a `Question` object just call the `add` method of the `ArrayList<String>` class?
13. Suppose we want to enhance the `CashRegister` class in How To 3.1 to track the prices of all purchased items for printing a receipt. Which instance variable should you provide? Which methods should you modify?
14. Consider an `Employee` class with properties for tax ID number and salary. Which of these properties should have only a getter method, and which should have getter and setter methods?
15. Suppose the `setName` method in Section 8.3.4 is changed so that it returns true if the new name is set, false if not. Is this a good idea?
16. Look at the `direction` instance variable in the bug example in Section 8.3.6. This is an example of which pattern?

Practice It Now you can try these exercises at the end of the chapter: E8.21, E8.22, E8.23.

8.4 Static Variables and Methods

A static variable belongs to the class, not to any object of the class.

Sometimes, a value properly belongs to a class, not to any object of the class. You use a **static variable** for this purpose. Here is a typical example: We want to assign bank account numbers sequentially. That is, we want the bank account constructor to construct the first account with number 1001, the next with number 1002, and so on. To solve this problem, we need to have a single value of `lastAssignedNumber` that is a property of the *class*, not any object of the class. Such a variable is called a static variable because you declare it using the `static` reserved word.

The reserved word `static` *is a holdover from the C++ language. Its use in Java has no relationship to the normal use of the term.*

```
public class BankAccount
{
   private double balance;
   private int accountNumber;
   private static int lastAssignedNumber = 1000;

   public BankAccount()
   {
      lastAssignedNumber++;
      accountNumber = lastAssignedNumber;
   }
   . . .
}
```

Every BankAccount object has its own balance and accountNumber instance variables, but all objects share a single copy of the lastAssignedNumber variable (see Figure 5). That variable is stored in a separate location, outside any BankAccount objects.

Like instance variables, static variables should always be declared as private to ensure that methods of other classes do not change their values. However, static *constants* may be either private or public.

For example, the BankAccount class can define a public constant value, such as

```
public class BankAccount
{
   public static final double OVERDRAFT_FEE = 29.95;
   . . .
}
```

Methods from any class can refer to such a constant as BankAccount.OVERDRAFT_FEE.

A static method is not invoked on an object.

Sometimes a class defines methods that are not invoked on an object. Such a method is called a **static method**. A typical example of a static method is the sqrt method in the Math class. Because numbers aren't objects, you can't invoke methods on them. For example, if x is a number, then the call x.sqrt() is not legal in Java. Therefore, the Math class provides a static method that is invoked as Math.sqrt(x). No object of the Math class is constructed. The Math qualifier simply tells the compiler where to find the sqrt method.

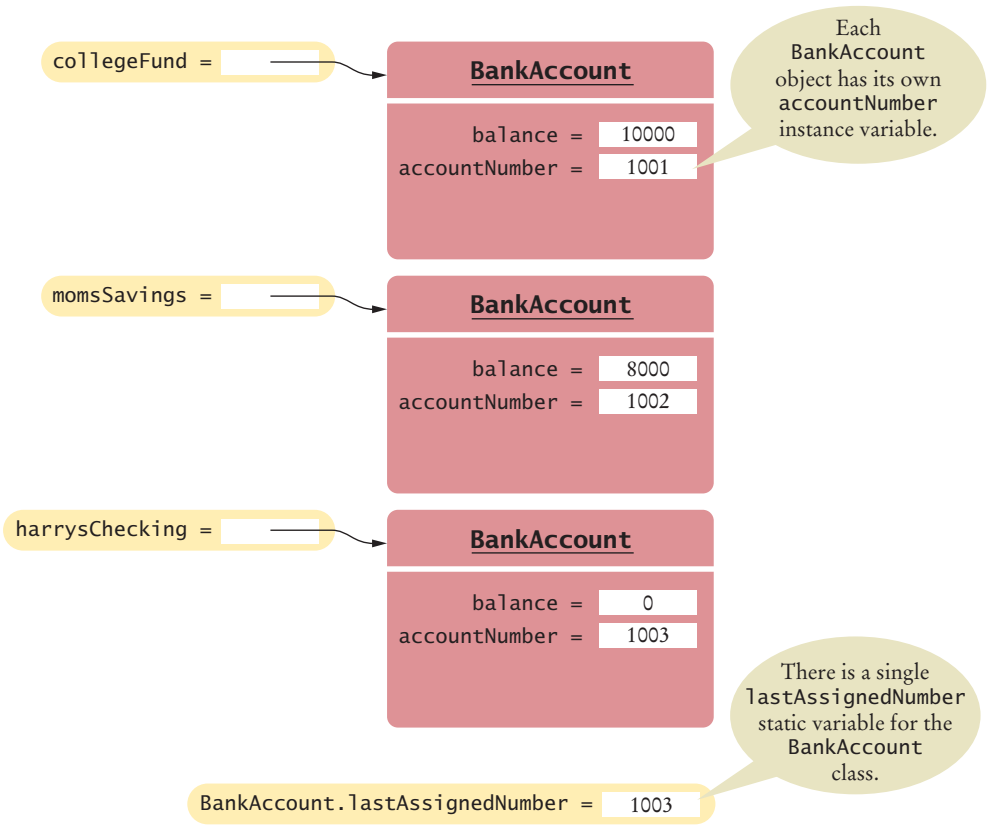


Figure 5 A Static Variable and Instance Variables

You can define your own static methods for use in other classes. Here is an example:

```
public class Financial
{
   /**
      Computes a percentage of an amount.
      @param percentage the percentage to apply
      @param amount the amount to which the percentage is applied
      @return the requested percentage of the amount
   */
   public static double percentOf(double percentage, double amount)
   {
      return (percentage / 100) * amount;
   }
}
```

FULL CODE EXAMPLE

Go to wiley.com/go/javacode to download a program with static methods and variables.

When calling this method, supply the name of the class containing it:

```
double tax = Financial.percentOf(taxRate, total);
```

In object-oriented programming, static methods are not very common. Nevertheless, the `main` method is always static. When the program starts, there aren't any objects. Therefore, the first method of a program must be a static method.

SELF CHECK

17. Name two static variables of the `System` class.
18. Name a static constant of the `Math` class.
19. The following method computes the average of an array of numbers:

    ```
    public static double average(double[] values)
    ```

    Why should it not be defined as an instance method?
20. Harry tells you that he has found a great way to avoid those pesky objects: Put all code into a single class and declare all methods and variables `static`. Then `main` can call the other static methods, and all of them can access the static variables. Will Harry's plan work? Is it a good idea?

Practice It Now you can try these exercises at the end of the chapter: R8.22, E8.5, E8.6.

Programming Tip 8.2

Minimize the Use of Static Methods

It is possible to solve programming problems by using classes with only static methods. In fact, before object-oriented programming was invented, that approach was quite common. However, it usually leads to a design that is not object-oriented and makes it hard to evolve a program.

Consider the task of How To 7.1. A program reads scores for a student and prints the final score, which is obtained by dropping the lowest one. We solved the problem by implementing a `Student` class that stores student scores. Of course, we could have simply written a program with a few static methods:

```
public class ScoreAnalyzer
{
   public static double[] readInputs() { . . . }
   public static double sum(double[] values) { . . . }
   public static double minimum(double[] values) { . . . }
```

```
   public static double finalScore(double[] values)
   {
      if (values.length == 0) { return 0; }
      else if (values.length == 1) { return values[0]; }
      else { return sum(values) - minimum(values); }
   }

   public static void main(String[] args)
   {
      System.out.println(finalScore(readInputs()));
   }
}
```

That solution is fine if one's sole objective is to solve a simple homework problem. But suppose you need to modify the program so that it deals with multiple students. An object-oriented program can evolve the `Student` class to store grades for many students. In contrast, adding more functionality to static methods gets messy quickly (see Exercise E8.7).

Common Error 8.1

Trying to Access Instance Variables in Static Methods

A static method does not operate on an object. In other words, it has no implicit parameter, and you cannot directly access any instance variables. For example, the following code is wrong:

```
public class SavingsAccount
{
   private double balance;
   private double interestRate;

   public static double interest(double amount)
   {
      return (interestRate / 100) * amount;
      // ERROR: Static method accesses instance variable
   }
}
```

Because different savings accounts can have different interest rates, the `interest` method should not be a static method.

Special Topic 8.2

Static Imports

Starting with Java version 5.0, there is a variant of the `import` directive that lets you use static methods and variables without class prefixes. For example,

```
import static java.lang.System.*;
import static java.lang.Math.*;

public class RootTester
{
   public static void main(String[] args)
   {
      double r = sqrt(PI); // Instead of Math.sqrt(Math.PI)
      out.println(r);      // Instead of System.out
   }
```

```
}
```

Static imports can make programs easier to read, particularly if they use many mathematical functions.

Special Topic 8.3

Alternative Forms of Instance and Static Variable Initialization

As you have seen, instance variables are initialized with a default value (`0`, `false`, or `null`, depending on their type). You can then set them to any desired value in a constructor, and that is the style that we prefer in this book.

However, there are two other mechanisms to specify an initial value. Just as with local variables, you can specify initialization values for instance variables. For example,

```
public class Coin
{
   private double value = 1;
   private String name = "Dollar";
   . . .
}
```

These default values are used for *every* object that is being constructed.

There is also another, much less common, syntax. You can place one or more *initialization blocks* inside the class declaration. All statements in that block are executed whenever an object is being constructed. Here is an example:

```
public class Coin
{
   private double value;
   private String name;
   {
      value = 1;
      name = "Dollar";
   }
   . . .
}
```

For static variables, you use a static initialization block:

```
public class BankAccount
{
   private static int lastAssignedNumber;
   static
   {
      lastAssignedNumber = 1000;
   }
   . . .
}
```

All statements in the static initialization block are executed once when the class is loaded. Initialization blocks are rarely used in practice.

When an object is constructed, the initializers and initialization blocks are executed in the order in which they appear. Then the code in the constructor is executed. Because the rules for the alternative initialization mechanisms are somewhat complex, we recommend that you simply use constructors to do the job of construction.

8.5 Packages

A package is a set of related classes.

A Java program consists of a collection of classes. So far, most of your programs have consisted of a small number of classes. As programs get larger, however, simply distributing the classes over multiple files isn't enough. An additional structuring mechanism is needed.

In Java, packages provide this structuring mechanism. A Java **package** is a set of related classes. For example, the Java library consists of several hundred packages, some of which are listed in Table 1.

Table 1 Important Packages in the Java Library

Package	Purpose	Sample Class
`java.lang`	Language support	`Math`
`java.util`	Utilities	`Random`
`java.io`	Input and output	`PrintStream`
`java.awt`	Abstract Windowing Toolkit	`Color`
`java.applet`	Applets	`Applet`
`java.net`	Networking	`Socket`
`java.sql`	Database access through Structured Query Language	`ResultSet`
`javax.swing`	Swing user interface	`JButton`
`org.w3c.dom`	Document Object Model for XML documents	`Document`

8.5.1 Organizing Related Classes into Packages

To put one of your classes in a package, you must place a line

```
package packageName;
```

as the first instruction in the source file containing the class. A package name consists of one or more identifiers separated by periods. (See Section 8.5.3 for tips on constructing package names.)

For example, let's put the `Financial` class introduced in this chapter into a package named `com.horstmann.bigjava`. The `Financial.java` file must start as follows:

```
package com.horstmann.bigjava;
public class Financial
{
   . . .
}
```

In addition to the named packages (such as `java.util` or `com.horstmann.bigjava`), there is a special package, called the *default package*, which has no name. If you did not

In Java, related classes are grouped into packages.

include any `package` statement at the top of your source file, its classes are placed in the default package.

8.5.2 Importing Packages

If you want to use a class from a package, you can refer to it by its full name (package name plus class name). For example, `java.util.Scanner` refers to the `Scanner` class in the `java.util` package:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

The `import` directive lets you refer to a class of a package by its class name, without the package prefix.

Naturally, that is somewhat inconvenient. For that reason, you usually import a name with an `import` statement:

```
import java.util.Scanner;
```

Then you can refer to the class as `Scanner` without the package prefix.

You can import *all classes* of a package with an `import` statement that ends in `.*`. For example, you can use the statement

```
import java.util.*;
```

to import all classes from the `java.util` package. That statement lets you refer to classes like `Scanner` or `Random` without a `java.util` prefix.

However, you never need to import the classes in the `java.lang` package explicitly. That is the package containing the most basic Java classes, such as `Math` and `Object`. These classes are always available to you. In effect, an automatic `import java.lang.*;` statement has been placed into every source file.

Finally, you don't need to import other classes in the same package. For example, when you implement the class `homework1.Tester`, you don't need to import the class `homework1.Bank`. The compiler will find the `Bank` class without an `import` statement because it is located in the same package, `homework1`.

8.5.3 Package Names

Placing related classes into a package is clearly a convenient mechanism to organize classes. However, there is a more important reason for packages: to avoid **name clashes**. In a large project, it is inevitable that two people will come up with the same name for the same concept. This even happens in the standard Java class library (which has now grown to thousands of classes). There is a class `Timer` in the `java.util`

Syntax 8.1 Package Specification

Syntax `package` *packageName*;

```
package com.horstmann.bigjava;
```

The classes in this file belong to this package.

A good choice for a package name is a domain name in reverse.

package and another class called `Timer` in the `javax.swing` package. You can still tell the Java compiler exactly which `Timer` class you need, simply by referring to them as `java.util.Timer` and `javax.swing.Timer`.

Of course, for the package-naming convention to work, there must be some way to ensure that package names are unique. It wouldn't be good if the car maker BMW placed all its Java code into the package `bmw`, and some other programmer (perhaps Britney M. Walters) had the same bright idea. To avoid this problem, the inventors of Java recommend that you use a package-naming scheme that takes advantage of the uniqueness of Internet domain names.

Use a domain name in reverse to construct an unambiguous package name.

For example, I have a domain name `horstmann.com`, and there is nobody else on the planet with the same domain name. (I was lucky that the domain name `horstmann.com` had not been taken by anyone else when I applied. If your name is Walters, you will sadly find that someone else beat you to `walters.com`.) To get a package name, turn the domain name around to produce a package name prefix, such as `com.horstmann`.

If you don't have your own domain name, you can still create a package name that has a high probability of being unique by writing your e-mail address backwards. For example, if Britney Walters has an e-mail address `walters@cs.sjsu.edu`, then she can use a package name `edu.sjsu.cs.walters` for her own classes.

Some instructors will want you to place each of your assignments into a separate package, such as `homework1`, `homework2`, and so on. The reason is again to avoid name collision. You can have two classes, `homework1.Bank` and `homework2.Bank`, with slightly different properties.

8.5.4 Packages and Source Files

The path of a class file must match its package name.

A source file must be located in a subdirectory that matches the package name. The parts of the name between periods represent successively nested directories. For example, the source files for classes in the package `com.horstmann.bigjava` would be placed in a subdirectory `com/horstmann/bigjava`. You place the subdirectory inside the *base directory* holding your program's files. For example, if you do your homework assignment in a directory `/home/britney/hw8/problem1`, then you can place the class files for the `com.horstmann.bigjava` package into the directory `/home/britney/hw8/problem1/com/horstmann/bigjava`, as shown in Figure 6. (Here, we are using UNIX-style file names. Under Windows, you might use `c:\Users\Britney\hw8\problem1\com\horstmann\bigjava`.)

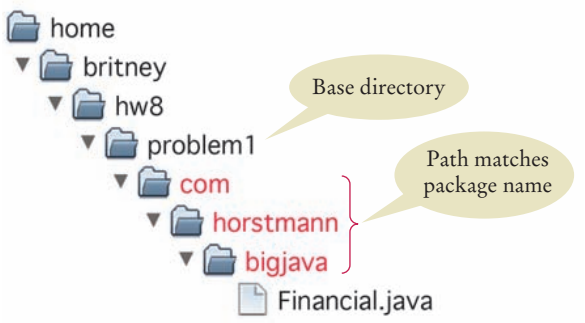


Figure 6 Base Directories and Subdirectories for Packages

21. Which of the following are packages?

a. `java`

b. `java.lang`

c. `java.util`

d. `java.lang.Math`

22. Is a Java program without `import` statements limited to using the default and `java.lang` packages?

23. Suppose your homework assignments are located in the directory `/home/me/cs101` (`c:\Users\Me\cs101` on Windows). Your instructor tells you to place your homework into packages. In which directory do you place the class `hw1.problem1.TicTacToeTester`?

Practice It Now you can try these exercises at the end of the chapter: R8.25, E8.15, E8.16.

Common Error 8.2

Confusing Dots

In Java, the dot symbol (.) is used as a separator in the following situations:

- Between package names (`java.util`)
- Between package and class names (`homework1.Bank`)
- Between class and inner class names (`Ellipse2D.Double`)
- Between class and instance variable names (`Math.PI`)
- Between objects and methods (`account.getBalance()`)

When you see a long chain of dot-separated names, it can be a challenge to find out which part is the package name, which part is the class name, which part is an instance variable name, and which part is a method name. Consider

```
java.lang.System.out.println(x);
```

Because `println` is followed by an opening parenthesis, it must be a method name. Therefore, `out` must be either an object or a class with a static `println` method. (Of course, we know that `out` is an object reference of type `PrintStream`.) Again, it is not at all clear, without context, whether `System` is another object, with a public variable `out`, or a class with a `static` variable.

Judging from the number of pages that the Java language specification devotes to this issue, even the compiler has trouble interpreting these dot-separated sequences of strings.

To avoid problems, it is helpful to adopt a strict coding style. If class names always start with an uppercase letter, and variable, method, and package names always start with a lowercase letter, then confusion can be avoided.

Special Topic 8.4

Package Access

If a class, instance variable, or method has no `public` or `private` modifier, then all methods of classes in the same package can access the feature. For example, if a class is declared as `public`, then all other classes in all packages can use it. But if a class is declared without an access modifier, then only the other classes in the *same* package can use it. Package access is a reasonable default for classes, but it is extremely unfortunate for instance variables.

An instance variable or method that is not declared as `public` or `private` can be accessed by all classes in the same package, which is usually not desirable.

It is a common error to *forget* the reserved word `private`, thereby opening up a potential security hole. For example, at the time of this writing, the `Window` class in the `java.awt` package contained the following declaration:

```
public class Window extends Container
{
   String warningString;
   . . .
}
```

There actually was no good reason to grant package access to the `warningString` instance variable—no other class accesses it.

Package access for instance variables is rarely useful and always a potential security risk. Most instance variables are given package access by accident because the programmer simply forgot the `private` reserved word. It is a good idea to get into the habit of scanning your instance variable declarations for missing `private` modifiers.

HOW TO 8.1

Programming with Packages

This How To explains in detail how to place your programs into packages.

Problem Statement Place each homework assignment into a separate package. That way, you can have classes with the same name but different implementations in separate packages (such as `homework1.problem1.Bank` and `homework1.problem2.Bank`).

Step 1 Come up with a package name.

Your instructor may give you a package name to use, such as `homework1.problem2`. Or, perhaps you want to use a package name that is unique to you. Start with your e-mail address, written backwards. For example, `walters@cs.sjsu.edu` becomes `edu.sjsu.cs.walters`. Then add a subpackage that describes your project, such as `edu.sjsu.cs.walters.cs1project`.

Step 2 Pick a *base directory*.

The base directory is the directory that contains the directories for your various packages, for example, `/home/britney` or `c:\Users\Britney`.

Step 3 Make a subdirectory from the base directory that matches your package name.

The subdirectory must be contained in your base directory. Each segment must match a segment of the package name. For example,

```
mkdir -p /home/britney/homework1/problem2 (in UNIX)
```

or

```
mkdir /s c:\Users\Britney\homework1\problem2 (in Windows)
```

Step 4 Place your source files into the package subdirectory.

For example, if your homework consists of the files `Tester.java` and `Bank.java`, then you place them into

```
/home/britney/homework1/problem2/Tester.java
/home/britney/homework1/problem2/Bank.java
```

or

```
c:\Users\Britney\homework1\problem2\Tester.java
c:\Users\Britney\homework1\problem2\Bank.java
```

Step 5 Use the `package` statement in each source file.

The first noncomment line of each file must be a package statement that lists the name of the package, such as

```
package homework1.problem2;
```

Step 6 Compile your source files from the *base directory*.

Change to the base directory (from Step 2) to compile your files. For example,

```
cd /home/britney
javac homework1/problem2/Tester.java
```

or

```
c:
cd \Users\Britney
javac homework1\problem2\Tester.java
```

Note that the Java compiler needs the *source file name and not the class name.* That is, you need to supply file separators (`/` on UNIX, `\` on Windows) and a file extension (`.java`).

Step 7 Run your program from the *base directory*.

Unlike the Java compiler, the Java interpreter needs the *class name (and not a file name) of the class containing the* `main` *method*. That is, use periods as package separators, and don't use a file extension. For example,

```
cd /home/britney
java homework1.problem2.Tester
```

or

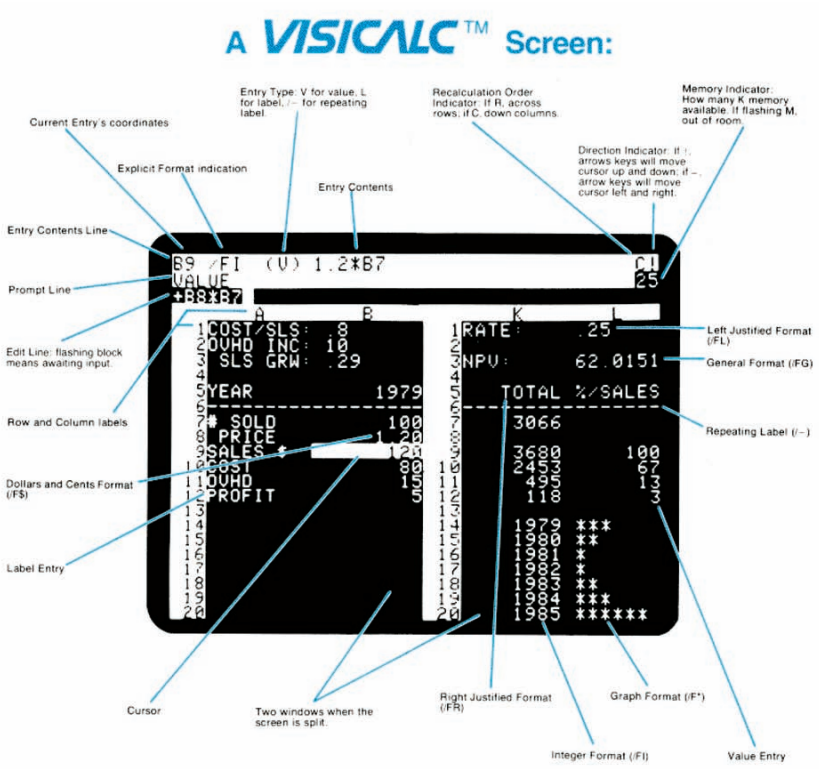
```
c:
cd \Users\Britney
java homework1.problem2.Tester
```

Computing & Society 8.1 Personal Computing

In 1971, Marcian E. "Ted" Hoff, an engineer at Intel Corporation, was working on a chip for a manufacturer of electronic calculators. He realized that it would be a better idea to develop a *general-purpose* chip that could be *programmed* to interface with the keys and display of a calculator, rather than to do yet another custom design. Thus, the *microprocessor* was born. At the time, its primary application was as a controller for calculators, washing machines, and the like. It took years for the computer industry to notice that a genuine central processing unit was now available as a single chip.

Hobbyists were the first to catch on. In 1974 the first computer *kit*, the Altair 8800, was available from MITS Electronics for about $350. The kit consisted of the microprocessor, a circuit board, a very small amount of memory, toggle switches, and a row of display lights. Purchasers had to solder and assemble it, then program it in machine language through the toggle switches. It was not a big hit.

The first big hit was the Apple II. It was a real computer with a keyboard, a monitor, and a floppy disk drive. When it was first released, users had a $3,000 machine that could play Space Invaders, run a primitive bookkeeping program, or let users program it in BASIC. The original Apple II did not even support lowercase letters, making it worthless for word processing. The breakthrough came in 1979 with a new spreadsheet program, VisiCalc. In a spreadsheet, you enter financial data and their relationships into a grid of rows and columns (see the figure). Then you modify some of the data and watch in real time how the others change. For example, you can see how changing the mix of widgets in a manufacturing plant might affect estimated costs and profits. Corporate managers snapped up VisiCalc and the computer that was needed to run it. For them, the computer was a spreadsheet machine. More importantly, it was a personal device. The managers were free to do the calculations that they wanted to do, not just the ones that the "high priests" in the data center provided.



The Visicalc Spreadsheet Running on an Apple II

Personal computers have been with us ever since, and countless users have tinkered with their hardware and software, sometimes establishing highly successful companies or creating free software for millions of users. This "freedom to tinker" is an important part of personal computing. On a personal device, you should be able to install the software that you want to install to make you more productive or creative, even if that's not the same software that most people use. You should be able to add peripheral equipment of your choice. For the first thirty years of personal computing, this freedom was largely taken for granted.

We are now entering an era where smartphones, tablets, and smart TV sets are replacing functions that were traditionally fulfilled by personal computers. While it is amazing to carry more computing power in your cell phone than in the best personal computers of the 1990s, it is disturbing that we lose a degree of personal control. With some phone or tablet brands, you can only install those applications that the manufacturer publishes on the "app store". For example, Apple does not allow children to learn the Scratch language on the iPad. You'd think it would be in Apple's interest to encourage the next generation to be enthusiastic about programming, but they have a general policy of denying programmability on "their" devices, in order to thwart competitive environments such as Flash or Java.

When you select a device for making phone calls or watching movies, it is worth asking who is in control. Are you purchasing a personal device that you can use in any way you choose, or are you being tethered to a flow of data that is controlled by somebody else?

8.6 Unit Test Frameworks

Unit test frameworks simplify the task of writing classes that contain many test cases.

Up to now, we have used a very simple approach to testing. We provided tester classes whose `main` method computes values and prints actual and expected values. However, that approach has limitations. The `main` method gets messy if it contains many tests. And if an exception occurs during one of the tests, the remaining tests are not executed.

Unit testing frameworks were designed to quickly execute and evaluate test suites and to make it easy to incrementally add test cases. One of the most popular testing frameworks is JUnit. It is freely available at `http://junit.org`, and it is also built into a number of development environments, including BlueJ and Eclipse. Here we describe JUnit 4, the most current version of the library as this book is written.

When you use JUnit, you design a companion test class for each class that you develop. You provide a method for each test case that you want to have executed. You use "annotations" to mark the test methods. An annotation is an advanced Java feature that places a marker into the code that is interpreted by another tool. In the case of JUnit, the `@Test` annotation is used to mark test methods.

In each test case, you make some computations and then compute some condition that you believe to be true. You then pass the result to a method that communicates a test result to the framework, most commonly the `assertEquals` method. The `assertEquals` method takes as arguments the expected and actual values and, for floating-point numbers, a tolerance value.

It is also customary (but not required) that the name of the test class ends in `Test`, such as `CashRegisterTest`. Here is a typical example:

```
import org.junit.Test;
import org.junit.Assert;

public class CashRegisterTest
{
   @Test public void twoPurchases()
   {
      CashRegister register = new CashRegister();
      register.recordPurchase(0.75);
      register.recordPurchase(1.50);
      register.receivePayment(2, 0, 5, 0, 0);
      double expected = 0.25;
      Assert.assertEquals(expected, register.giveChange(), EPSILON);
   }
   // More test cases
   . . .
}
```

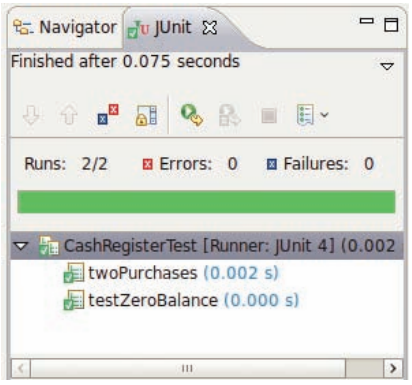
If all test cases pass, the JUnit tool shows a green bar (see Figure 7). If any of the test cases fail, the JUnit tool shows a red bar and an error message.

Your test class can also have other methods (whose names should not be annotated with `@Test`). These methods typically carry out steps that you want to share among test methods.

The JUnit philosophy is to run all tests whenever you change your code.

The JUnit philosophy is simple. Whenever you implement a class, also make a companion test class. You design the tests as you design the program, one test method at a time. The test cases just keep accumulating in the test class. Whenever you have detected an actual failure, add a test case that flushes it out, so that you can be sure

Figure 7
Unit Testing with JUnit



that you won't introduce that particular bug again. Whenever you modify your class, simply run the tests again.

If all tests pass, the user interface shows a green bar and you can relax. Otherwise, there is a red bar, but that's also good. It is much easier to fix a bug in isolation than inside a complex program.

SELF CHECK

24. Provide a JUnit test class with one test case for the `Earthquake` class in Chapter 5.

25. What is the significance of the `EPSILON` argument in the `assertEquals` method?

Practice It Now you can try these exercises at the end of the chapter: R8.27, E8.17, E8.18.

CHAPTER SUMMARY

Find classes that are appropriate for solving a programming problem.

- A class should represent a single concept from a problem domain, such as business, science, or mathematics.

Design methods that are cohesive, consistent, and minimize side effects.

- The public interface of a class is cohesive if all of its features are related to the concept that the class represents.
- A class depends on another class if its methods use that class in any way.
- An immutable class has no mutator methods.
- References to objects of an immutable class can be safely shared.

- A side effect of a method is any externally observable data modification.
- When designing methods, minimize side effects.
- In Java, a method can never change the contents of a variable that is passed to a method.
- In Java, a method can change the state of an object reference argument, but it cannot replace the object reference with another.

Use patterns to design the data representation of an object.

- An instance variable for the total is updated in methods that increase or decrease the total amount.
- A counter that counts events is incremented in methods that correspond to the events.
- An object can collect other objects in an array or array list.
- An object property can be accessed" with a getter method and changed with a setter method.
- If your object can have one of several states that affect the behavior, supply an instance variable for the current state.
- To model a moving object, you need to store and update its position.

Understand the behavior of static variables and static methods.

- A static variable belongs to the class, not to any object of the class.
- A static method is not invoked on an object.

Use packages to organize sets of related classes.

- A package is a set of related classes.
- The `import` directive lets you refer to a class of a package by its class name, without the package prefix.
- Use a domain name in reverse to construct an unambiguous package name.
- The path of a class file must match its package name.
- An instance variable or method that is not declared as `public` or `private` can be accessed by all classes in the same package, which is usually not desirable.

Use JUnit for writing unit tests.

- Unit test frameworks simplify the task of writing classes that contain many test cases.
- The JUnit philosophy is to run all tests whenever you change your code.

REVIEW QUESTIONS

R8.1 Your task is to write a program that simulates a vending machine. Users select a product and provide payment. If the payment is sufficient to cover the purchase price of the product, the product is dispensed and change is given. Otherwise, the payment is returned to the user. Name an appropriate class for implementing this program. Name two classes that would not be appropriate and explain why.

R8.2 Your task is to write a program that reads a customer's name and address, followed by a sequence of purchased items and their prices, and prints an invoice.

Discuss which of the following would be good classes for implementing this program:

a. `Invoice`

b. `InvoicePrinter`

c. `PrintInvoice`

d. `InvoiceProgram`

■■■ **R8.3** Your task is to write a program that computes paychecks. Employees are paid an hourly rate for each hour worked; however, if they worked more than 40 hours per week, they are paid at 150 percent of the regular rate for those overtime hours. Name an actor class that would be appropriate for implementing this program. Then name a class that isn't an actor class that would be an appropriate alternative. How does the choice between these alternatives affect the program structure?

■■ **R8.4** Look at the public interface of the `java.lang.System` class and discuss whether or not it is cohesive.

■■ **R8.5** Suppose an `Invoice` object contains descriptions of the products ordered, and the billing and shipping addresses of the customer. Draw a UML diagram showing the dependencies between the classes `Invoice`, `Address`, `Customer`, and `Product`.

■■ **R8.6** Suppose a vending machine contains products, and users insert coins into the vending machine to purchase products. Draw a UML diagram showing the dependencies between the classes `VendingMachine`, `Coin`, and `Product`.

■■ **R8.7** On which classes does the class `Integer` in the standard library depend?

■■ **R8.8** On which classes does the class `Rectangle` in the standard library depend?

■ **R8.9** Classify the methods of the class `Scanner` that are used in this book as accessors and mutators.

■ **R8.10** Classify the methods of the class `Rectangle` as accessors and mutators.

■ **R8.11** Is the `Resistor` class in Exercise P8.8 a mutable or immutable class? Why?

■ **R8.12** Which of the following classes are immutable?

a. `Rectangle`

b. `String`

c. `Random`

■ **R8.13** Which of the following classes are immutable?

a. `PrintStream`

b. `Date`

c. `Integer`

R8.14 Consider a method

```
public class DataSet
{
   /**
      Reads all numbers from a scanner and adds them to this data set.
      @param in a Scanner
   */
   public void read(Scanner in) { . . . }
   . . .
```

```
}
```

Describe the side effects of the `read` method. Which of them are not recommended, according to Section 8.2.4? Which redesign eliminates the unwanted side effect? What is the effect of the redesign on coupling?

■■ **R8.15** What side effect, if any, do the following three methods have?

```
public class Coin
{
   . . .
   public void print()
   {
      System.out.println(name + " " + value);
   }

   public void print(PrintStream stream)
   {
      stream.println(name + " " + value);
   }

   public String toString()
   {
      return name + " " + value;
   }
}
```

■■■ **R8.16** Ideally, a method should have no side effects. Can you write a program in which no method has a side effect? Would such a program be useful?

■■ **R8.17** Consider the following method that is intended to swap the values of two integers:

```
public static void falseSwap(int a, int b)
{
   int temp = a;
   a = b;
   b = temp;
}

public static void main(String[] args)
{
   int x = 3;
   int y = 4;
   falseSwap(x, y);
   System.out.println(x + " " + y);
}
```

Why doesn't the method swap the contents of `x` and `y`?

■■■ **R8.18** How can you write a method that swaps two floating-point numbers?
Hint: `java.awt.Point`.

■■ **R8.19** Draw a memory diagram that shows why the following method can't swap two `BankAccount` objects:

```
public static void falseSwap(BankAccount a, BankAccount b)
{
   BankAccount temp = a;
   a = b;
   b = temp;
}
```

■ **R8.20** Consider an enhancement of the `Die` class of Chapter 6 with a static variable

```
public class Die
{
   private int sides;
   private static Random generator = new Random();
   public Die(int s) { . . . }
   public int cast() { . . . }
}
```

Draw a memory diagram that shows three dice:

```
Die d4 = new Die(4);
Die d6 = new Die(6);
Die d8 = new Die(8);
```

Be sure to indicate the values of the `sides` and `generator` variables.

■ **R8.21** Try compiling the following program. Explain the error message that you get.

```
public class Print13
{
   public void print(int x)
   {
      System.out.println(x);
   }

   public static void main(String[] args)
   {
      int n = 13;
      print(n);
   }
}
```

■ **R8.22** Look at the methods in the `Integer` class. Which are static? Why?

■■ **R8.23** Look at the methods in the `String` class (but ignore the ones that take an argument of type `char[]`). Which are static? Why?

■■ **R8.24** The `in` and `out` variables of the `System` class are public static variables of the `System` class. Is that good design? If not, how could you improve on it?

■■ **R8.25** Every Java program can be rewritten to avoid `import` statements. Explain how, and rewrite `RectangleComponent.java` from Section 2.9.3 to avoid `import` statements.

■ **R8.26** What is the default package? Have you used it before this chapter in your programming?

■■ **Testing R8.27** What does JUnit do when a test method throws an exception? Try it out and report your findings.

PRACTICE EXERCISES

■■ **E8.1** Implement the `Coin` class described in Section 8.2. Modify the `CashRegister` class so that coins can be added to the cash register, by supplying a method

```
void receivePayment(int coinCount, Coin coinType)
```

The caller needs to invoke this method multiple times, once for each type of coin that is present in the payment.

■■ **E8.2** Modify the `giveChange` method of the `CashRegister` class so that it returns the number of coins of a particular type to return:

```
int giveChange(Coin coinType)
```

The caller needs to invoke this method for each coin type, in decreasing value.

■ **E8.3** Real cash registers can handle both bills and coins. Design a single class that expresses the commonality of these concepts. Redesign the `CashRegister` class and provide a method for entering payments that are described by your class. Your primary challenge is to come up with a good name for this class.

E8.4 Reimplement the `BankAccount` class so that it is immutable. The `deposit` and `withdraw` methods need to return new `BankAccount` objects with the appropriate balance.

■■ **E8.5** Write static methods

- `public static double cubeVolume(double h)`
- `public static double cubeSurface(double h)`
- `public static double sphereVolume(double r)`
- `public static double sphereSurface(double r)`
- `public static double cylinderVolume(double r, double h)`
- `public static double cylinderSurface(double r, double h)`
- `public static double coneVolume(double r, double h)`
- `public static double coneSurface(double r, double h)`

that compute the volume and surface area of a cube with height `h`, sphere with radius `r`, a cylinder with circular base with radius `r` and height `h`, and a cone with circular base with radius `r` and height `h`. Place them into a class `Geometry`. Then write a program that prompts the user for the values of `r` and `h`, calls the six methods, and prints the results.

■■ **E8.6** Solve Exercise E8.5 by implementing classes `Cube`, `Sphere`, `Cylinder`, and `Cone`. Which approach is more object-oriented?

■■■ **E8.7** Modify the application of How To 7.1 so that it can deal with multiple students. First, ask the user for all student names. Then read in the scores for all quizzes, prompting for the score of each student. Finally, print the names of all students and their final scores. Use a single class and only static methods.

■■■ **E8.8** Repeat Exercise E8.7, using multiple classes. Provide a `GradeBook` class that collects objects of type `Student`.

■■■ **E8.9** Write methods

```
public static double perimeter(Ellipse2D.Double e);
public static double area(Ellipse2D.Double e);
```

that compute the area and the perimeter of the ellipse `e`. Add these methods to a class `Geometry`. The challenging part of this assignment is to find and implement an accurate formula for the perimeter. Why does it make sense to use a static method in this case?

■■ **E8.10** Write methods

```
public static double angle(Point2D.Double p, Point2D.Double q)
public static double slope(Point2D.Double p, Point2D.Double q)
```

that compute the angle between the x-axis and the line joining two points, measured in degrees, and the slope of that line. Add the methods to the class `Geometry`. Supply suitable preconditions. Why does it make sense to use a static method in this case?

■■■ **E8.11** Write methods

```
public static boolean isInside(Point2D.Double p, Ellipse2D.Double e)
public static boolean isOnBoundary(Point2D.Double p, Ellipse2D.Double e)
```

that test whether a point is inside or on the boundary of an ellipse. Add the methods to the class `Geometry`.

■ **E8.12** Write a method

```
public static int readInt(
      Scanner in, String prompt, String error, int min, int max)
```

that displays the prompt string, reads an integer, and tests whether it is between the minimum and maximum. If not, print an error message and repeat reading the input. Add the method to a class `Input`.

■■ **E8.13** Consider the following algorithm for computing x^n for an integer n. If `n < 0`, x^n is $1/x^{-n}$. If n is positive and even, then $x^n = (x^{n/2})^2$. If n is positive and odd, then $x^n = x^{n-1} \times x$. Implement a static method `double intPower(double x, int n)` that uses this algorithm. Add it to a class called `Numeric`.

■■ **E8.14** Improve the `Die` class of Chapter 6. Turn the `generator` variable into a static variable so that all needles share a single random number generator.

■■ **E8.15** Implement `Coin` and `CashRegister` classes as described in Exercise E8.1. Place the classes into a package called `money`. Keep the `CashRegisterTester` class in the default package.

■ **E8.16** Place a `BankAccount` class in a package whose name is derived from your e-mail address, as described in Section 8.5. Keep the `BankAccountTester` class in the default package.

■■ **Testing E8.17** Provide a JUnit test class `StudentTest` with three test methods, each of which tests a different method of the `Student` class in How To 7.1.

■■ **Testing E8.18** Provide JUnit test class `TaxReturnTest` with three test methods that test different tax situations for the `TaxReturn` class in Chapter 5.

■ **Graphics E8.19** Write methods

- `public static void drawH(Graphics2D g2, Point2D.Double p);`
- `public static void drawE(Graphics2D g2, Point2D.Double p);`
- `public static void drawL(Graphics2D g2, Point2D.Double p);`
- `public static void drawO(Graphics2D g2, Point2D.Double p);`

that show the letters H, E, L, O in the graphics window, where the point `p` is the top-left corner of the letter. Then call the methods to draw the words "HELLO" and "HOLE" on the graphics display. Draw lines and ellipses. Do not use the `drawString` method. Do not use `System.out`.

■■ **Graphics E8.20** Repeat Exercise E8.19 by designing classes `LetterH`, `LetterE`, `LetterL`, and `LetterO`, each with a constructor that takes a `Point2D.Double` parameter (the top-left corner) and a method `draw(Graphics2D g2)`. Which solution is more object-oriented?

E8.21 Add a method `ArrayList<Double> getStatement()` to the `BankAccount` class that returns a list of all deposits and withdrawals as positive or negative values. Also add a method `void clearStatement()` that resets the statement.

E8.22 Implement a class `LoginForm` that simulates a login form that you find on many web pages. Supply methods

```
public void input(String text)
public void click(String button)
public boolean loggedIn()
```

The first input is the user name, the second input is the password. The `click` method can be called with arguments `"Submit"` and `"Reset"`. Once a user has been successfully logged in, by supplying the user name, password, and clicking on the submit button, the `loggedIn` method returns true and further input has no effect. When a user tries to log in with an invalid user name and password, the form is reset.

Supply a constructor with the expected user name and password.

E8.23 Implement a class `Robot` that simulates a robot wandering on an infinite plane. The robot is located at a point with integer coordinates and faces north, east, south, or west. Supply methods

```
public void turnLeft()
public void turnRight()
public void move()
public Point getLocation()
public String getDirection()
```

The `turnLeft` and `turnRight` methods change the direction but not the location. The `move` method moves the robot by one unit in the direction it is facing. The `getDirection` method returns a string `"N"`, `"E"`, `"S"`, or `"W"`.

PROGRAMMING PROJECTS

■■■ P8.1 Declare a class `ComboLock` that works like the combination lock in a gym locker, as shown here. The lock is constructed with a combination—three numbers between 0 and 39. The `reset` method resets the dial so that it points to 0. The `turnLeft` and `turnRight` methods turn the dial by a given number of ticks to the left or right. The `open` method attempts to open the lock. The lock opens if the user first turned it right to the first number in the combination, then left to the second, and then right to the third.

```
public class ComboLock
{
   . . .
   public ComboLock(int secret1, int secret2, int secret3) { . . . }
   public void reset() { . . . }
   public void turnLeft(int ticks) { . . . }
   public void turnRight(int ticks) { . . . }
   public boolean open() { . . . }
}
```

■■■ Business P8.2 Implement a program that prints paychecks for a group of student assistants. Deduct federal and Social Security taxes. (You may want to use the tax computation used

in Chapter 5. Find out about Social Security taxes on the Internet.) Your program should prompt for the names, hourly wages, and hours worked of each student.

■■■ **P8.3** For faster sorting of letters, the United States Postal Service encourages companies that send large volumes of mail to use a bar code denoting the ZIP code (see Figure 8).

The encoding scheme for a five-digit ZIP code is shown in Figure 8. There are full-height frame bars on each side. The five encoded digits are followed by a check digit, which is computed as follows: Add up all digits, and choose the check digit to make the sum a multiple of 10. For example, the sum of the digits in the ZIP code 95014 is 19, so the check digit is 1 to make the sum equal to 20.

Digit	Weight				
	7	4	2	1	0
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	0
7	1	0	0	0	0
8	1	0	0	1	1
9	1	0	1	0	0
0	1	1	0	0	0

Each digit of the ZIP code, and the check digit, is encoded according to the table at right, where 0 denotes a half bar and 1 a full bar. Note that they represent all combinations of two full and three half bars. The digit can be computed easily from the bar code using the column weights 7, 4, 2, 1, 0. For example, 01100 is

$$0 \times 7 + 1 \times 4 + 1 \times 2 + 0 \times 1 + 0 \times 0 = 6$$

The only exception is 0, which would yield 11 according to the weight formula.

Write a program that asks the user for a ZIP code and prints the bar code. Use : for half bars, | for full bars. For example, 95014 becomes

```
||:|:::|:|:||:::::::||:|::|:::|||
```

(Alternatively, write a graphical application that draws real bars.)

Your program should also be able to carry out the opposite conversion: Translate bars into their ZIP code, reporting any errors in the input format or a mismatch of the digits.

Figure 8 A Postal Bar Code

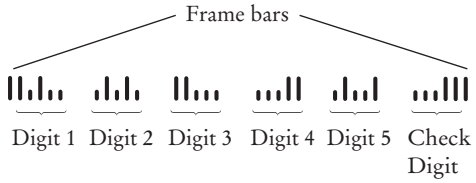


Figure 9 Encoding for Five-Digit Bar Codes

■■ **Business P8.4** Design a `Customer` class to handle a customer loyalty marketing campaign. After accumulating $100 in purchases, the customer receives a $10 discount on the next purchase.

Provide methods

- `void makePurchase(double amount)`
- `boolean discountReached()`

Provide a test program and test a scenario in which a customer has earned a discount and then made over $90, but less than $100 in purchases. This should not result in a second discount. Then add another purchase that results in the second discount.

■■■ Business P8.5 The Downtown Marketing Association wants to promote downtown shopping with a loyalty program similar to the one in Exercise P8.4. Shops are identified by a number between 1 and 20. Add a new parameter variable to the `makePurchase` method that indicates the shop. The discount is awarded if a customer makes purchases in at least three different shops, spending a total of $100 or more.

■■■ Science P8.6 Design a class `Cannonball` to model a cannonball that is fired into the air. A ball has

- An x- and a y-position.
- An x- and a y-velocity.

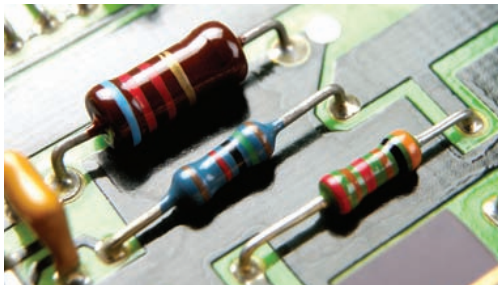
Supply the following methods:

- A constructor with an x-position (the y-position is initially `0`)
- A method `move(double deltaSec)` that moves the ball to the next position First compute the distance traveled in `deltaSec` seconds, using the current velocities, then update the x- and y-positions; then update the y-velocity by taking into account the gravitational acceleration of –9.81 m/s^2; the x-velocity is unchanged.
- A method `Point getLocation()` that gets the current location of the cannonball, rounded to integer coordinates
- A method `ArrayList<Point> shoot(double alpha, double v, double deltaSec)` whose arguments are the angle α and initial velocity v (Compute the x-velocity as v cos α and the y-velocity as v sin α; then keep calling `move` with the given time interval until the y-position is `0`; return an array list of locations after each call to move.

Use this class in a program that prompts the user for the starting angle and the initial velocity. Then call `shoot` and print the locations.

■ Graphics P8.7 Continue Exercise P8.6, and draw the trajectory of the cannonball.

■■ Science P8.8 The colored bands on the top-most resistor shown in the photo at right indicate a resistance of 6.2 kΩ ±5 percent. The resistor tolerance of ±5 percent indicates the acceptable variation in the resistance. A 6.2 kΩ ±5 percent resistor could have a resistance as small as 5.89 kΩ or as large as 6.51 kΩ. We say that 6.2 kΩ is the *nominal value* of the

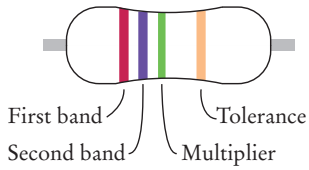


resistance and that the actual value of the resistance can be any value between 5.89 kΩ and 6.51 kΩ.

Write a program that represents a resistor as a class. Provide a single constructor that accepts values for the nominal resistance and tolerance and then determines the actual value randomly. The class should provide public methods to get the nominal resistance, tolerance, and the actual resistance.

Write a `main` method for the program that demonstrates that the class works properly by displaying actual resistances for ten 330 Ω ±10 percent resistors.

■■ Science P8.9 In the `Resistor` class from Exercise P8.8, supply a method that returns a description of the "color bands" for the resistance and tolerance. A resistor has four color bands:



- The first band is the first significant digit of the resistance value.
- The second band is the second significant digit of the resistance value.
- The third band is the decimal multiplier.
- The fourth band indicates the tolerance.

Color	Digit	Multiplier	Tolerance
Black	0	$\times 10^0$	—
Brown	1	$\times 10^1$	±1%
Red	2	$\times 10^2$	±2%
Orange	3	$\times 10^3$	—
Yellow	4	$\times 10^4$	—
Green	5	$\times 10^5$	±0.5%
Blue	6	$\times 10^6$	±0.25%
Violet	7	$\times 10^7$	±0.1%
Gray	8	$\times 10^8$	±0.05%
White	9	$\times 10^9$	—
Gold	—	$\times 10^{-1}$	±5%
Silver	—	$\times 10^{-2}$	±10%
None	—	—	±20%

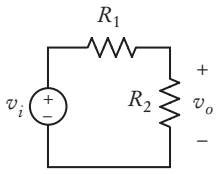
For example (using the values from the table as a key), a resistor with red, violet, green, and gold bands (left to right) will have 2 as the first digit, 7 as the second digit, a multiplier of 10^5, and a tolerance of ±5 percent, for a resistance of 2,700 kΩ, plus or minus 5 percent.

■■ Science P8.10 The figure below shows a frequently used electric circuit called a "voltage divider". The input to the circuit is the voltage v_i. The output is the voltage v_o. The output of

a voltage divider is proportional to the input, and the constant of proportionality is called the "gain" of the circuit. The voltage divider is represented by the equation

$$G = \frac{v_o}{v_i} = \frac{R_2}{R_1 + R_2}$$

where G is the gain and R_1 and R_2 are the resistances of the two resistors that comprise the voltage divider.



Manufacturing variations cause the actual resistance values to deviate from the nominal values, as described in Exercise P8.8. In turn, variations in the resistance values cause variations in the values of the gain of the voltage divider. We calculate the *nominal value of the gain* using the nominal resistance values and the *actual value of the gain* using actual resistance values.

Write a program that contains two classes, `VoltageDivider` and `Resistor`. The `Resistor` class is described in Exercise P8.8. The `VoltageDivider` class should have two instance variables that are objects of the `Resistor` class. Provide a single constructor that accepts two `Resistor` objects, nominal values for their resistances, and the resistor tolerance. The class should provide public methods to get the nominal and actual values of the voltage divider's gain.

Write a `main` method for the program that demonstrates that the class works properly by displaying nominal and actual gain for ten voltage dividers each consisting of 5 percent resistors having nominal values $R_1 = 250\ \Omega$ and $R_2 = 750\ \Omega$.

ANSWERS TO SELF-CHECK QUESTIONS

1. Look for nouns in the problem description.
2. Yes (`ChessBoard`) and no (`MovePiece`).
3. Some of its features deal with payments, others with coin values.
4. None of the coin operations require the `CashRegister` class.
5. If a class doesn't depend on another, it is not affected by interface changes in the other class.
6. It is an accessor—calling `substring` doesn't modify the string on which the method is invoked. In fact, all methods of the `String` class are accessors.
7. No—`translate` is a mutator method.
8. It is a side effect; this kind of side effect is common in object-oriented programming.
9. Yes—the method affects the state of the `Scanner` argument.
10. It needs to be incremented in the `deposit` and `withdraw` methods. There also needs to be some method to reset it after the end of a statement period.
11. The `giveChange` method is a mutator that returns a value that cannot be determined any other way. Here is a better design. The `receivePayment` method could decrease the `purchase` instance variable. Then the program user would call `receivePayment`, determine the change by calling `getAmountDue`, and call the `clear` method to reset the cash register for the next sale.
12. The `ArrayList<String>` instance variable is private, and the class users cannot acccess it.

13. You need to supply an instance variable that can hold the prices for all purchased items. This could be an `ArrayList<Double>` or `ArrayList<String>`, or it could simply be a `String` to which you append lines. The instance variable needs to be updated in the `recordPurchase` method. You also need a method that returns the receipt.
14. The tax ID of an employee does not change, and no setter method should be supplied. The salary of an employee can change, and both getter and setter methods should be supplied.
15. Section 8.2.3 suggests that a setter should return `void`, or perhaps a convenience value that the user can also determine in some other way. In this situation, the caller could check whether `newName` is blank, so the change is fine.
16. It is an example of the "state pattern" described in Section 8.3.5. The direction is a state that changes when the bug turns, and it affects how the bug moves.
17. `System.in` and `System.out`.
18. `Math.PI`
19. The method needs no data of any object. The only required input is the `values` argument.
20. Yes, it works. Static methods can access static variables of the same class. But it is a terrible idea. As your programming tasks get more complex, you will want to use objects and classes to organize your programs.
21. (a) No; (b) Yes; (c) Yes; (d) No
22. No—you can use fully qualified names for all other classes, such as `java.util.Random` and `java.awt.Rectangle`.
23. `/home/me/cs101/hw1/problem1` or, on Windows, `c:\Users\Me\cs101\hw1\problem1`.
24. Here is one possible answer.

```
public class EarthquakeTest
{
   @Test public void testLevel4()
   {
      Earthquake quake = new Earthquake(4);
      Assert.assertEquals(
         "Felt by many people, no destruction",
         quake.getDescription());
    }
}
```

25. It is a tolerance threshold for comparing floating-point numbers. We want the equality test to pass if there is a small roundoff error.

CHAPTER 9

INHERITANCE

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.

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

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.

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

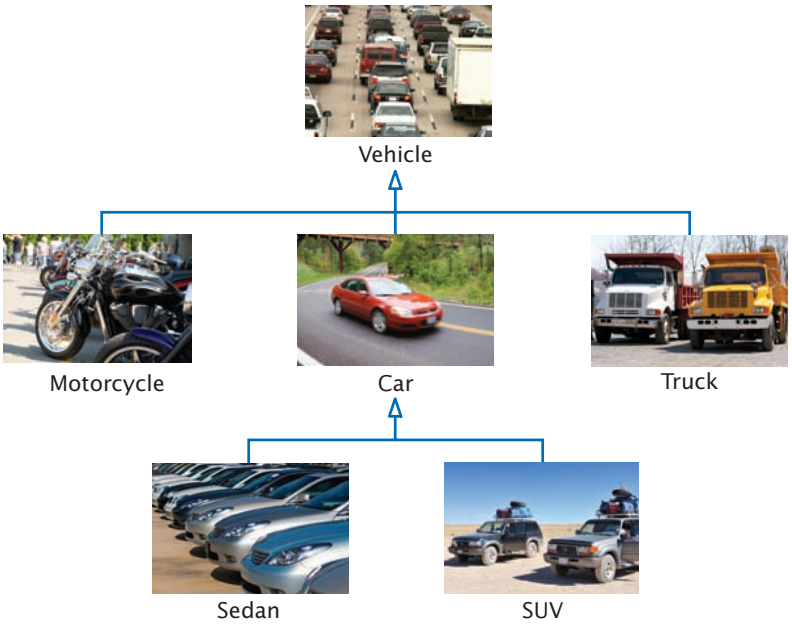
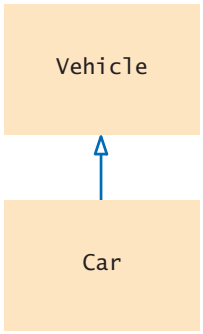


Figure 1 An Inheritance Hierarchy of Vehicle Classes

Figure 2
An Inheritance Diagram



that 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)
```

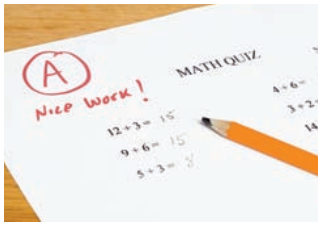
Because `Car` is a subclass of `Vehicle`, you can call that method with a `Car` object:

```
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



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

Figure 3 shows an inheritance hierarchy for these question types.

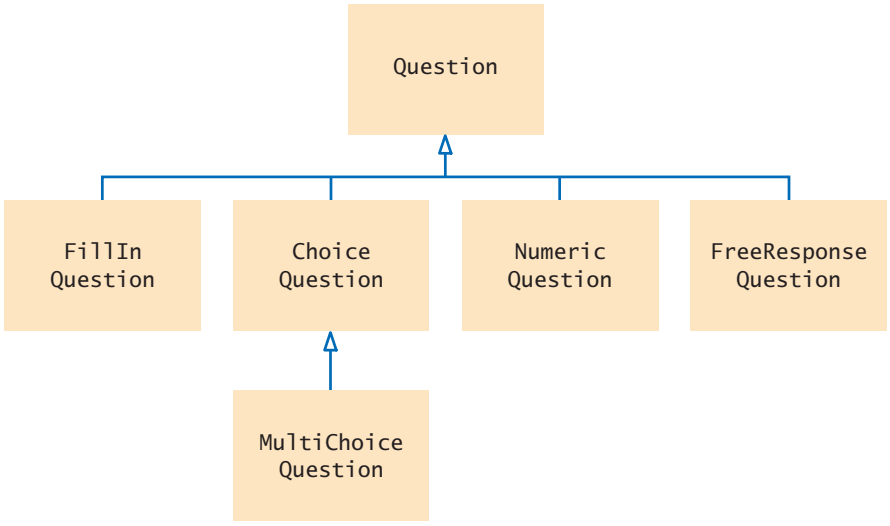


Figure 3
Inheritance Hierarchy of Question Types

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.

section_1/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:

section_1/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
```

1. Consider classes `Manager` and `Employee`. Which should be the superclass and which should be the subclass?
2. What are the inheritance relationships between classes `BankAccount`, `CheckingAccount`, and `SavingsAccount`?
3. What are all the superclasses of the `JFrame` class? Consult the Java API documentation or Appendix D.
4. Consider the method `doSomething(Car c)`. List all vehicle classes from Figure 1 whose objects *cannot* be passed to this method.
5. Should a class `Quiz` inherit from the class `Question`? Why or why not?

Practice It Now you can try these exercises at the end of the chapter: R9.1, R9.7, R9.9.

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).

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.

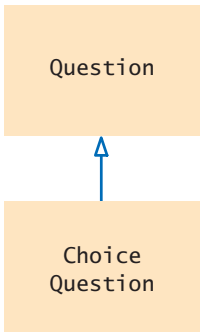


Figure 4
The `ChoiceQuestion` Class is a Subclass of the `Question` Class

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.

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.

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.

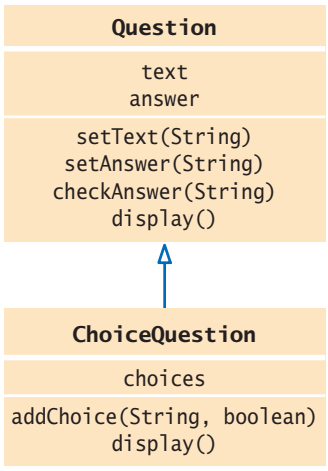


Figure 5
The `ChoiceQuestion` Class Adds an Instance Variable and a Method, and Overrides a Method

Syntax 9.1 Subclass Declaration

Syntax

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

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.

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

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

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

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.

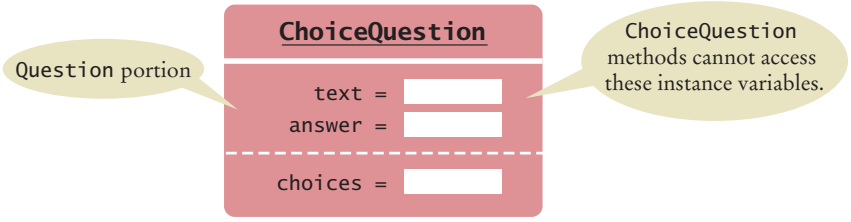


Figure 6
Data Layout of a Subclass Object

For example,

```
question.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);
   }
}
```

FULL CODE EXAMPLE

Go to `wiley.com/go/javacode` to download a program that shows a simple `Car` class extending a `Vehicle` class.

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);
```

6. Suppose `q` is an object of the class `Question` and `cq` an object of the class `ChoiceQuestion`. Which of the following calls are legal?
   a. `q.setAnswer(response)`
   b. `cq.setAnswer(response)`
   c. `q.addChoice(choice, true)`
   d. `cq.addChoice(choice, true)`
7. 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. Omit constructors and methods.
8. Which instance variables does the `Manager` class from Self Check 7 have?
9. In the `Manager` class, provide the method header (but not the implementation) for a method that overrides the `getSalary` method from the class `Employee`.

10. Which methods does the `Manager` class from Self Check 9 inherit?

Practice It Now you can try these exercises at the end of the chapter: R9.3, E9.6, E9.10.

Common Error 9.1

Replicating Instance Variables from the Superclass

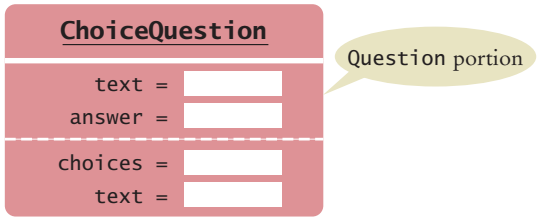
A subclass has no access to the private instance variables of the superclass.

```
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.

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 page 435.

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
   . . .
}
```

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.

section_3/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   }
```

section_3/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
```

11. What is wrong with the following implementation of the `display` method?

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

12. What is wrong with the following implementation of the `display` method?

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

13. Look again at the implementation of the `addChoice` method that calls the `setAnswer` method of the superclass. Why don't you need to call `super.setAnswer`?

14. In the `Manager` class of Self Check 7, override the `getName` method so that managers have a `*` before their name (such as `*Lin, Sally`).

15. In the `Manager` class of Self Check 9, override the `getSalary` method so that it returns the sum of the salary and the bonus.

Practice It Now you can try these exercises at the end of the chapter: E9.1, E9.2, E9.11.

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.

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.

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.

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).

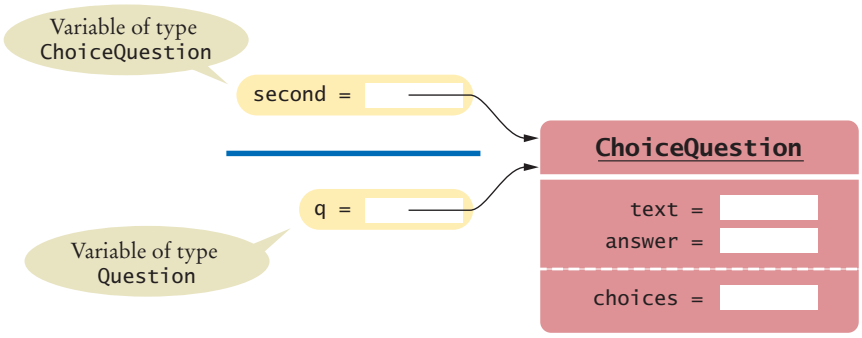


Figure 7
Variables of Different Types Referring to the Same Object

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

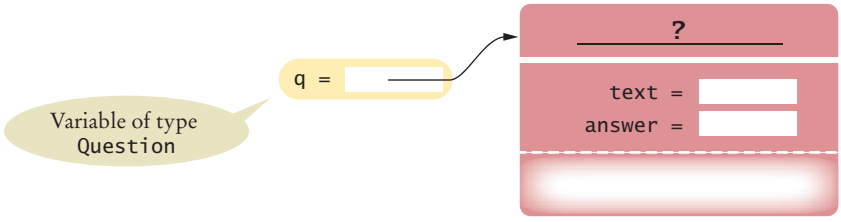


Figure 8
A `Question` Reference Can Refer to an Object of Any Subclass of `Question`

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

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?
```

Which `display` method is called? If you look at the program output on page 439, 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.

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

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**.

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

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 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.

section_4/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
```

16. Assuming `SavingsAccount` is a subclass of `BankAccount`, which of the following code fragments are valid in Java?

a. `BankAccount account = new SavingsAccount();`

b. `SavingsAccount account2 = new BankAccount();`

c. `BankAccount account = null;`

d. `SavingsAccount account2 = account;`

17. If `account` is a variable of type `BankAccount` that holds a non-`null` reference, what do you know about the object to which `account` refers?

18. Declare an array `quiz` that can hold a mixture of `Question` and `ChoiceQuestion` objects.
19. Consider the code fragment

```
ChoiceQuestion cq = . . .; // A non-null value
cq.display();
```

    Which actual method is being called?
20. Is the method call `Math.sqrt(2)` resolved through dynamic method lookup?

Practice It Now you can try these exercises at the end of the chapter: R9.6, E9.4, E9.14.

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.

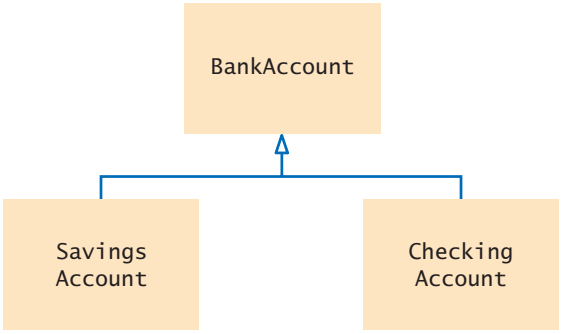
In the "Month end" command, accumulate interest or clear the transaction counter, depending on the type of the bank account. Then print the balance of all accounts.

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

FULL CODE EXAMPLE

Go to wiley.com/go/javacode to download 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. Go to `www.wiley.com/go/javaexamples` and download Worked Example 9.1.

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 15.1).

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"
```

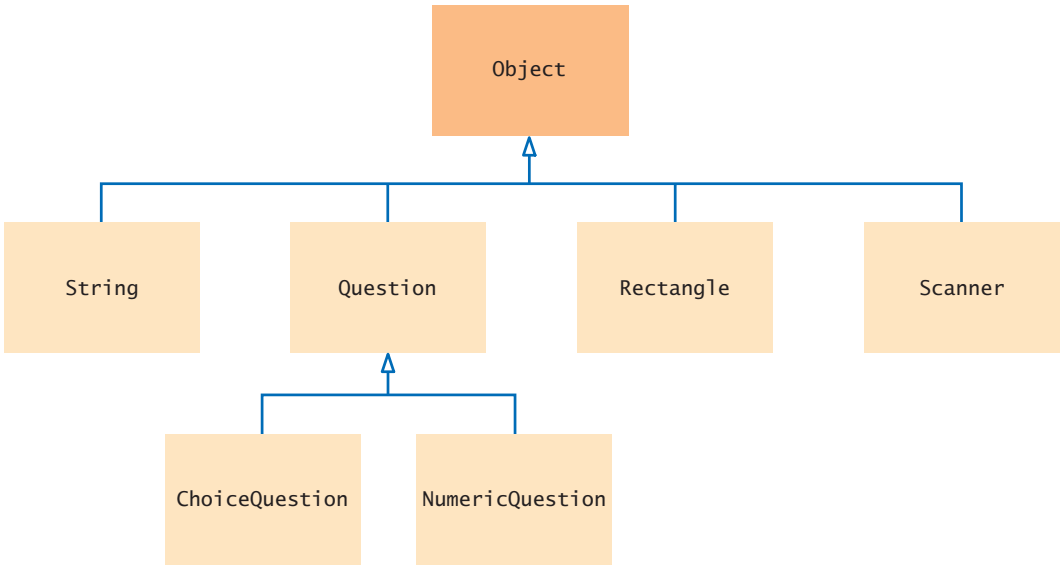


Figure 9 The `Object` Class Is the Superclass of Every Java Class

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 10
```

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
```

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

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;
```

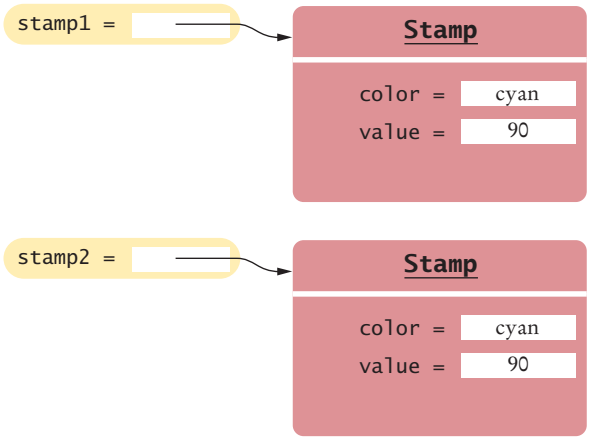


Figure 10 Two References to Equal Objects

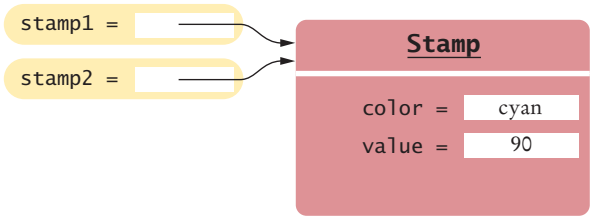


Figure 11 Two References to the Same Object

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.

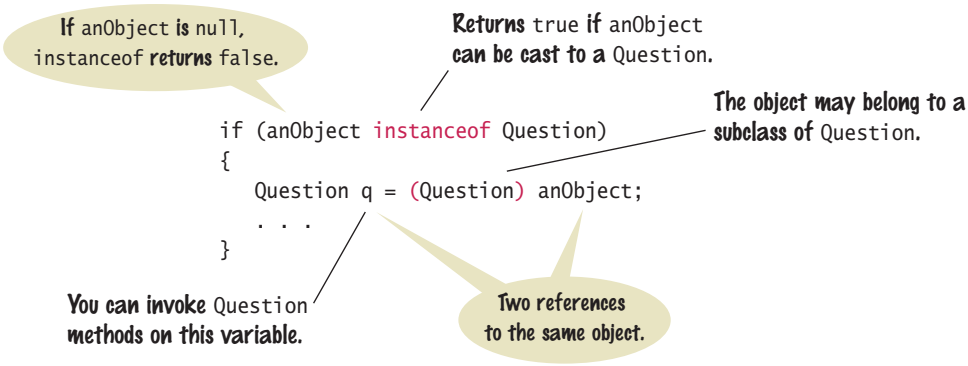
To protect against bad casts, you can use the `instanceof` operator. It tests whether an object belongs to a particular type. For example,

```
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 9.4 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;
}
```

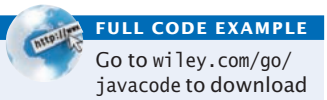
Note that `instanceof` is *not* a method. It is an operator, just like `+` or `<`. However, it does not operate on numbers. To the left is an object, and to the right a type name.

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

```
if (q instanceof ChoiceQuestion) // Don't do this—see Common Error 9.5
{
   // 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();
```



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: E9.7, E9.8, E9.12.

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)
{
```

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 (`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. 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

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

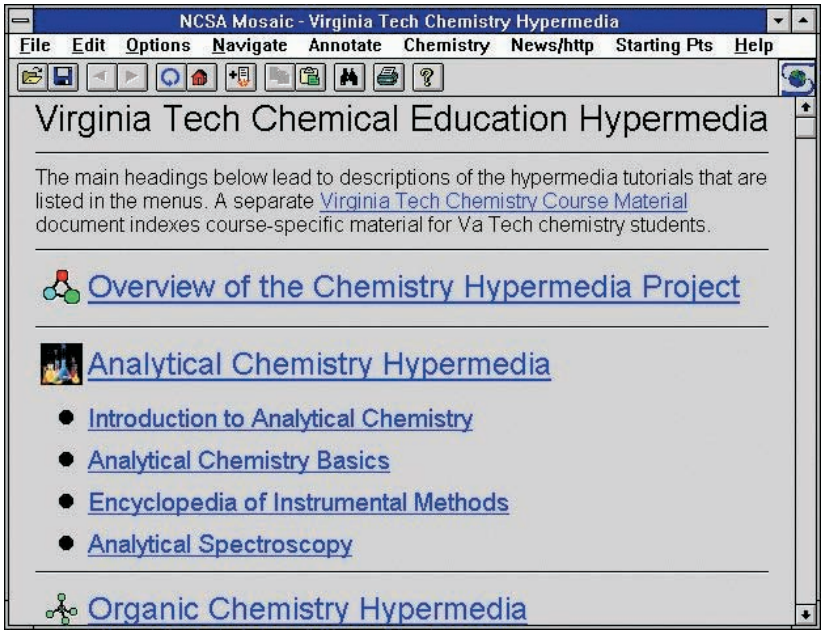
When you implement `equals` in a subclass, you should first call `equals` in the superclass to check whether the superclass instance variables match. Here is an example:

```
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;
   }
}
```

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.

When the Internet is delivered by phone or TV cable companies, those companies sometimes interfere with competing Internet offerings. Cell phone companies refused to carry Voice-over-IP services, and cable companies slowed down movie streaming.

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.

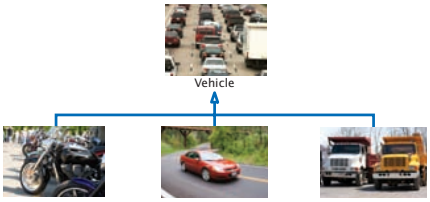


The NCSA Mosaic Browser

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 QUESTIONS

■ **R9.1** 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`

■ **R9.2** 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?

■ **R9.3** Which methods does the `ChoiceQuestion` class inherit from its superclass? Which methods does it override? Which methods does it add?

■ **R9.4** 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?

■ **R9.5** List the instance variables of a `CheckingAccount` object from How To 9.1.

■■ **R9.6** 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();`

■ **R9.7** Draw an inheritance diagram that shows the inheritance relationships between these classes.

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

■ **R9.8** 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`

■ **R9.9** What inheritance relationships would you establish among the following classes?

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

■■ **R9.10** How does a cast such as `(BankAccount) x` differ from a cast of number values such as `(int) x`?

■■■ **R9.11** 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** 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, then accept the response as correct.

■■ **E9.2** 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 _____
```

■ **E9.3** 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"`.

■■ **E9.4** 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.

■■ **E9.5** 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.

■■ **E9.6** 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.

■ **E9.7** Provide `toString` methods for the `Question` and `ChoiceQuestion` classes.

■■ **E9.8** 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 that tests these classes and methods.

■■ **E9.9** 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.

■■ **E9.10** 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.

■■■ **E9.11** Repeat Exercise E9.10, but in the `BetterRectangle` constructor, invoke the superclass constructor.

■■ **E9.12** 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.

■■ **E9.13** Reimplement the `LabeledPoint` class of Exercise E9.12 by storing the location in a `java.awt.Point` object. Your `toString` method should invoke the `toString` method of the `Point` class.

■■ **Business E9.14** 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.

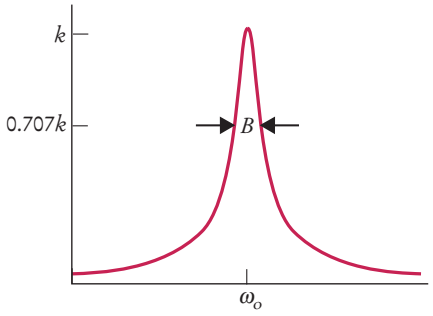
PROGRAMMING PROJECTS

■■ **Business P9.1** Implement a superclass `Appointment` and subclasses `Onetime`, `Daily`, and `Monthly`. An appointment has a description (for example, "see the dentist") and a date. 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.

■■ **Business P9.2** Improve the appointment book program of Exercise P9.1. Give the user the option to add new appointments. The user must specify the type of the appointment, the description, and the date.

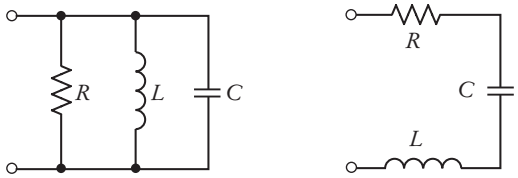
■■■ **Business P9.3** Improve the appointment book program of Exercise P9.1 and P9.2 by letting the user save the appointment data to a file and reload the data from a file. The saving part is straightforward: Make a method `save`. Save the type, description, and date to a file. The loading part is not 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 P9.4 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.



(a) Parallel resonant circuit (b) Series resonant circuit

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 the `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.

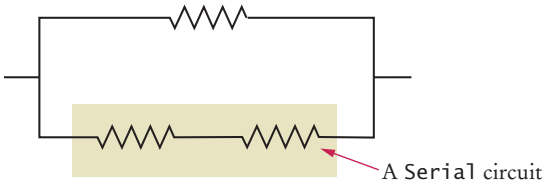
Each subclass should provide a method that designs the corresponding resonant circuit. The subclasses should also override the `display` method of the superclass to

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.

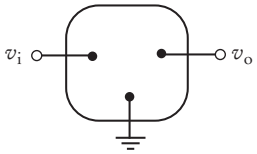
Supply a class that demonstrates that the subclasses all work properly.

■■■ **Science P9.5** 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:

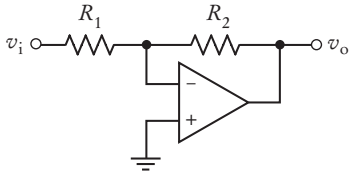


Use Ohm's law to compute the combined resistance.

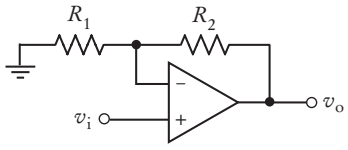
■■ **Science P9.6** Part (a) of the figure below shows a symbolic representation of an electric circuit called an *amplifier*. The input to the amplifier is the voltage v_i and the output is the voltage v_o. The output of an amplifier is proportional to the input. The constant of proportionality is called the "gain" of the amplifier.



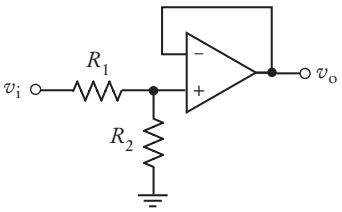
(a) Amplifier



(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 subclass two methods, `getGain` 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.

ANSWERS TO SELF-CHECK QUESTIONS

1. Because every manager is an employee but not the other way around, the `Manager` class is more specialized. It is the subclass, and `Employee` is the superclass.
2. `CheckingAccount` and `SavingsAccount` both inherit from the more general class `BankAccount`.
3. The classes `Frame`, `Window`, and `Component` in the `java.awt` package, and the class `Object` in the `java.lang` package.
4. Vehicle, truck, motorcycle
5. It shouldn't. A quiz isn't a question; it *has* questions.
6. a, b, d
7.
```
public class Manager extends Employee
{
   private double bonus;
   // Constructors and methods omitted
}
```
8. `name`, `baseSalary`, and `bonus`
9.
```
public class Manager extends Employee
{
   . . .
   public double getSalary() { . . . }
}
```
10. `getName`, `setName`, `setBaseSalary`
11. The method is not allowed to access the instance variable `text` from the superclass.
12. The type of the `this` reference is `ChoiceQuestion`. Therefore, the `display` method of `ChoiceQuestion` is selected, and the method calls itself.
13. Because there is no ambiguity. The subclass doesn't have a `setAnswer` method.
14.
```
public String getName()
{
   return "*" + super.getName();
}
```
15.
```
public double getSalary()
{
   return super.getSalary() + bonus;
}
```
16. a only.
17. It belongs to the class `BankAccount` or one of its subclasses.
18. `Question[] quiz = new Question[SIZE];`
19. You cannot tell from the fragment—`cq` may be initialized with an object of a subclass of `ChoiceQuestion`. The `display` method of whatever object `cq` references is invoked.
20. No. This is a static method of the `Math` class. There is no implicit parameter object that could be used to dynamically look up a method.
21. Because the implementor of the `PrintStream` class did not supply a `toString` method.
22. The second line will not compile. The class `Object` does not have a method `length`.
23. The code will compile, but the second line will throw a class cast exception because `Question` is not a superclass of `String`.
24. There are only a few methods that can be invoked on variables of type `Object`.
25. The value is `false` if `x` is `null` and `true` otherwise.