

# 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



© Ivan Stevanovic/iStockphoto.

## CHAPTER CONTENTS

<b>8.1 DISCOVERING CLASSES</b>	376
<b>8.2 DESIGNING GOOD METHODS</b>	377
PT1	Consistency 381
ST1	Call by Value and Call by Reference 382
<b>8.3 PROBLEM SOLVING: PATTERNS FOR OBJECT DATA</b>	386
<b>8.4 STATIC VARIABLES AND METHODS</b>	391
PT2	Minimize the Use of Static Methods 393
CE1	Trying to Access Instance Variables in Static Methods 394
ST2	Alternative Forms of Instance and Static Variable Initialization 394
ST3	Static Imports 395
<b>8.5 PROBLEM SOLVING: SOLVE A SIMPLER PROBLEM FIRST</b>	395
<b>8.6 PACKAGES</b>	400
SYN	Package Specification 401
CE2	Confusing Dots 403
ST4	Package Access 403
HT1	Programming with Packages 404
<b>8.7 UNIT TEST FRAMEWORKS</b>	405
C&S	Personal Computing 407



© Ivan Stevanovic/iStockphoto.

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

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

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.

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.

### SELF CHECK



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: R8.3, R8.4, R8.5.

## 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**.



© Sergey Ivanov/Stockphoto.

*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.

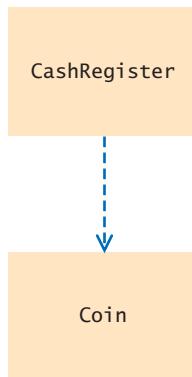
To visualize relationships between classes, such as dependence, programmers draw class diagrams. In this book, we use the UML (“**Unified Modeling Language**”) notation for objects and classes. UML is a notation for object-oriented analysis and design invented by Grady Booch, Ivar Jacobson, and James Rumbaugh, three leading researchers in object-oriented software 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  $\Rightarrow$ -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 $" + momSavings.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.

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
  
```

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`.

An immutable class has no mutator methods.

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

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.*



© manley099/iStockphoto.

#### 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 (accounts.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.*



AP Photo/Frank Franklin II.

### SELF CHECK



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: R8.6, R8.7, R8.11.

### 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.



Frank Rosenstein/Digital Vision/  
Getty Images, Inc.

*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 `transfer` method:

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

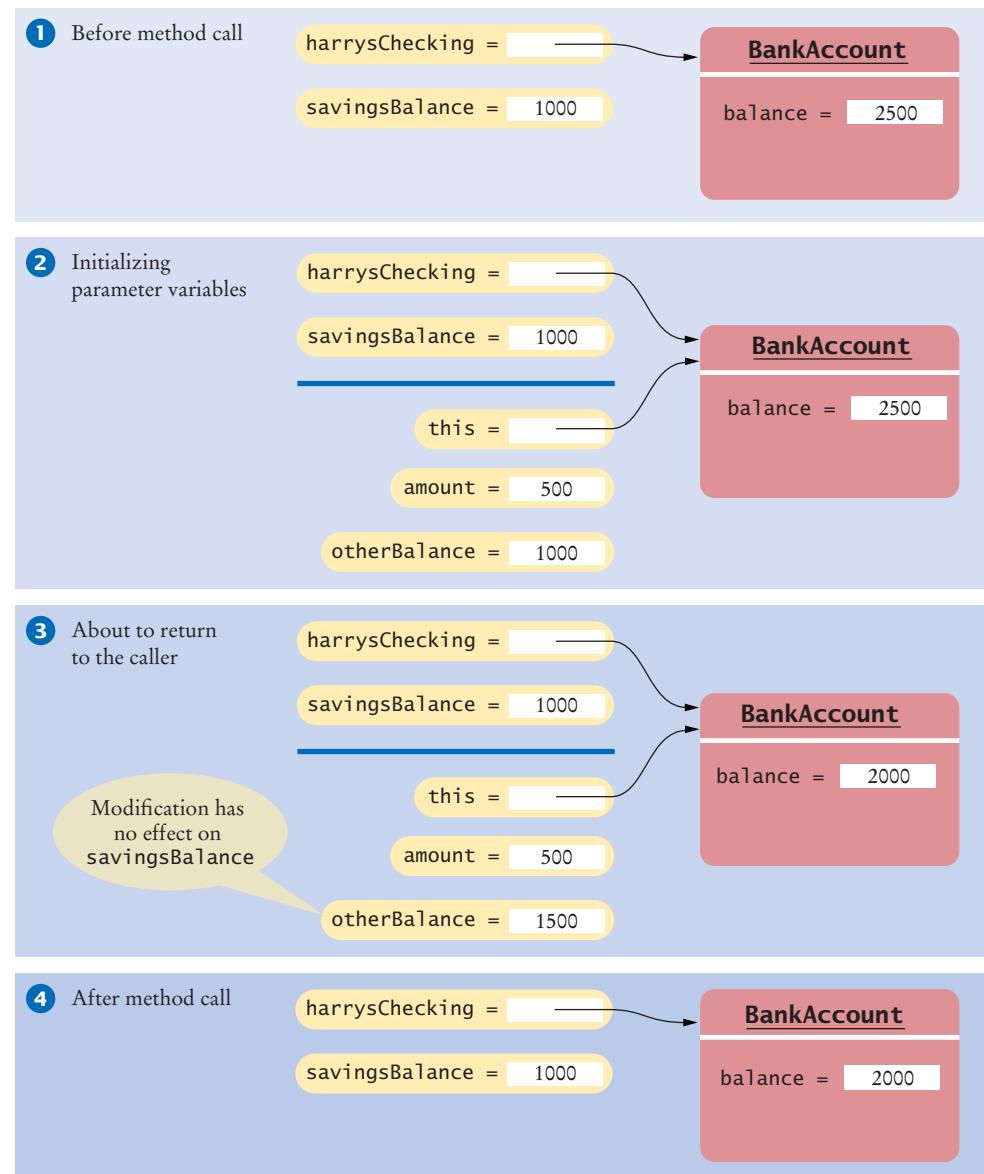
You might expect that after the call, the `savingsBalance` variable has been incremented to 1500. However, that is not the case. As the method starts, the parameter variable `otherBalance` is set to the same value as `savingsBalance` (see Figure 2). Then the `otherBalance` variable is set to a different 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, 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

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



**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.

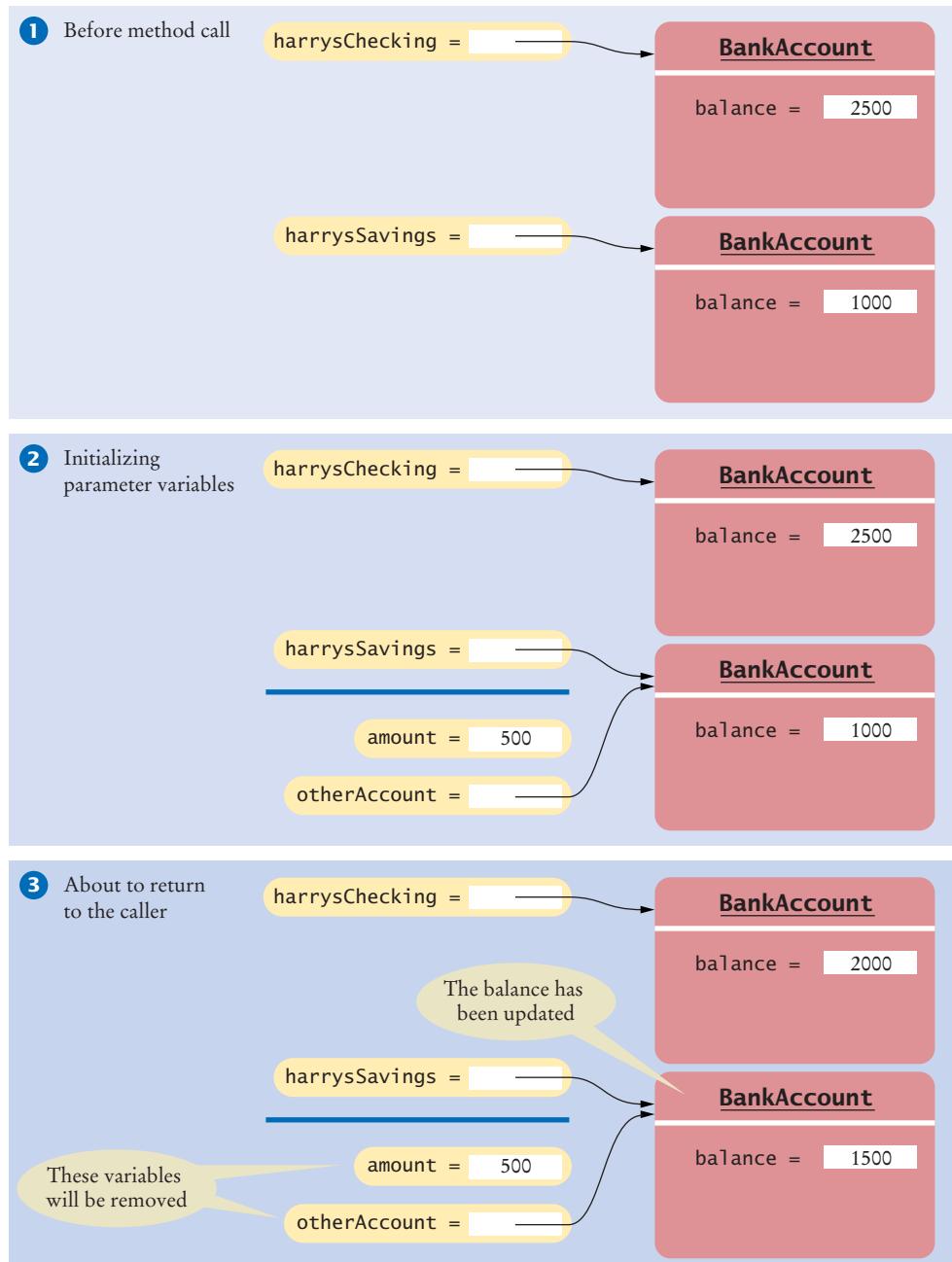
The confusion arises because a Java method can mutate an object when it receives an object reference as an argument (see Figure 3).

```
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); ❶
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 it 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:

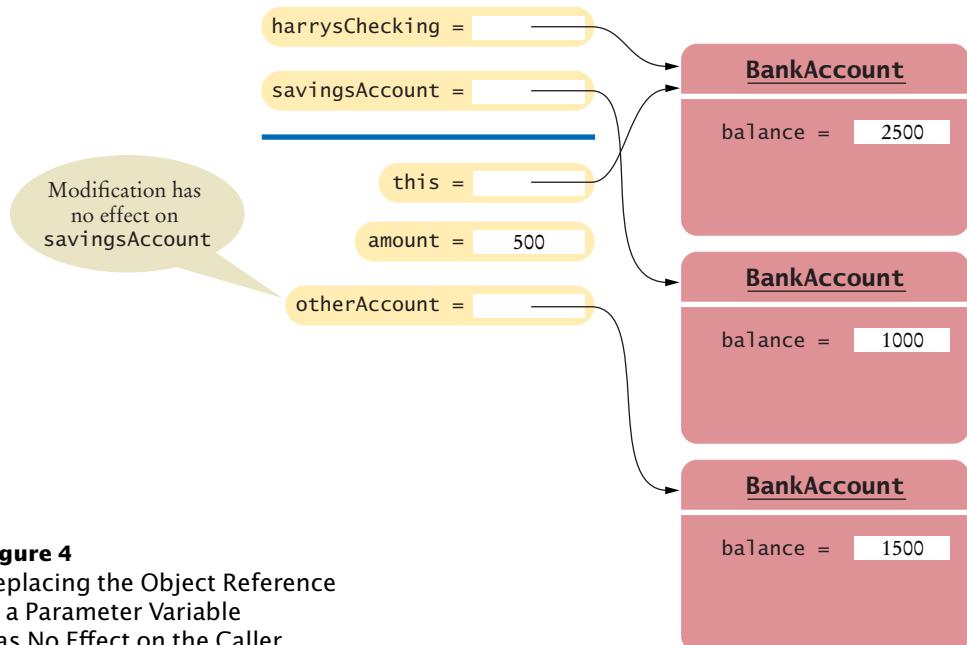
```
public class BankAccount
{
    ...
    public void transfer(double amount, BankAccount otherAccount)
    {
        balance = balance - amount;
        double newBalance = otherAccount.balance + amount;
        otherAccount = new BankAccount(newBalance); // Won't work
    }
}
```

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

```
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.

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



**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.

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

### 8.3.4 Managing Properties of an Object

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 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++; }
}
```



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

© John Alexander/Stockphoto.

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,

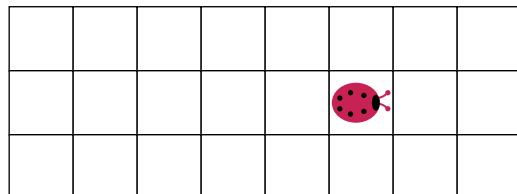
- A train drives along a track and keeps track of the distance from the terminus.
- A simulated bug living on a grid crawls from one grid location to the next, or makes 90 degree turns to the left or right.
- A cannonball is shot into the air, then descends as it is pulled by the gravitational force.

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

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

```
private double distanceFromTerminus;
```

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



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--; }
```

Exercise P8.10 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.

#### FULL CODE EXAMPLE

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download classes that use these patterns for object data.



#### SELF CHECK

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: E8.24, E8.25, E8.26.

## 8.4 Static Variables and Methods

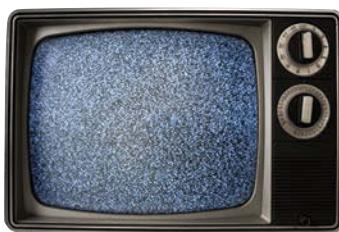
A static variable belongs to the class, not to any object of the class.

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.

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

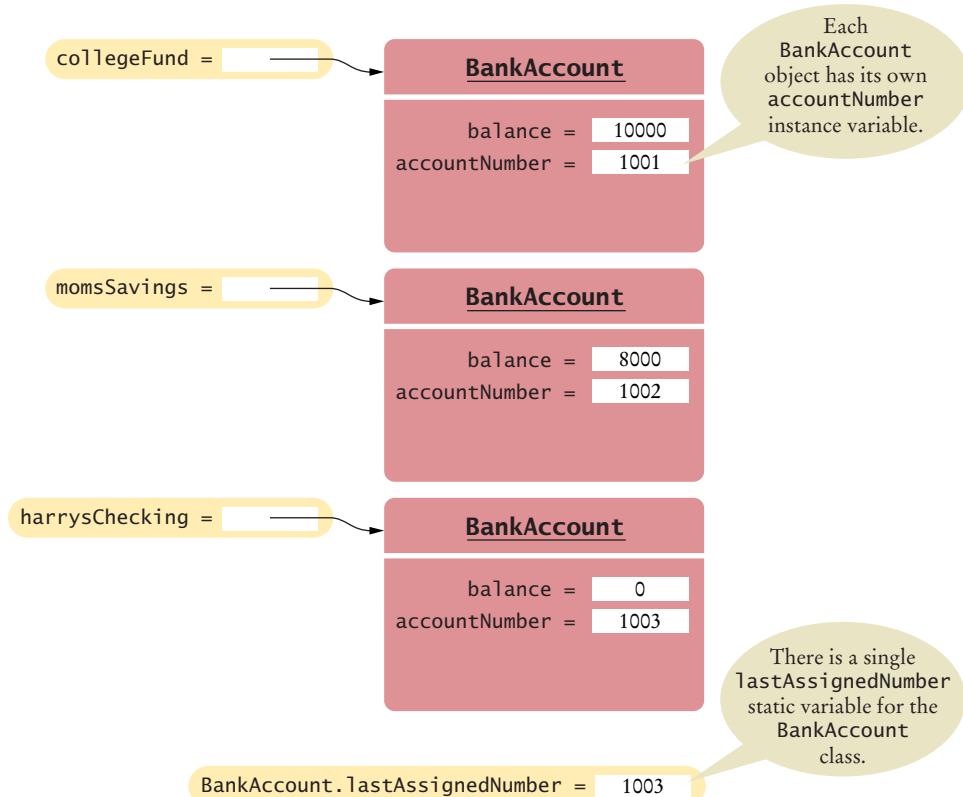
    public BankAccount()
    {
        lastAssignedNumber++;
        accountNumber = lastAssignedNumber;
    }
    ...
}
```

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.



© Diane Diederich/Stockphoto.

*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.*

**Figure 5** A Static Variable and Instance Variables

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`.

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.

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

```
public class Financial
{
```

A static method  
is not invoked on  
an object.

```

    /**
     * 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/bje06code](http://wiley.com/go/bje06code) to download a program with static methods and variables.

**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: R8.24, E8.6, E8.7.

**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 E8.8).

### 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



### 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.

### Special Topic 8.3



#### Static Imports

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.

## 8.5 Problem Solving: Solve a Simpler Problem First

When developing a solution to a complex problem, first solve a simpler task.

As you learn more about programming, the complexity of the tasks that you are asked to solve will increase. When you face a complex task, you should apply an important skill: simplifying the problem, and solving the simpler problem first.

This is a good strategy for several reasons. Usually, you learn something useful from solving the simpler task. Moreover, the complex problem can seem unsurmountable,

and you may find it difficult to know where to get started. When you are successful with a simpler problem first, you will be much more motivated to try the harder one.

It takes practice and a certain amount of courage to break down a problem into a sequence of simpler ones. The best way to learn this strategy is to practice it. When you work on your next assignment, ask yourself what is the absolutely simplest part of the task that is helpful for the end result, and start from there. With some experience, you will be able to design a plan that builds up a complete solution as a manageable sequence of intermediate steps.

Let us look at an example. You are asked to arrange pictures, lining them up along the top edges, separating them with small gaps, and starting a new row whenever you run out of room in the current row.



National Gallery of Art (see page C-1).

A Picture class is given to you. It has a constructor

```
public Picture(String filename)
```

and methods

```
public void move(int dx, int dy)
public Rectangle getBounds()
```

Make a plan consisting of a series of tasks, each a simple extension of the previous one, and ending with the original problem.

Instead of tackling the entire assignment at once, here is a plan that solves a series of simpler problems.

1. Draw one picture.



2. Draw two pictures next to each other.



3. Draw two pictures with a gap between them.



4. Draw all pictures in a long row.



5. Draw a row of pictures until you run out of room, then put one more picture in the next row.



Let's get started with this plan.

1. The purpose of the first step is to become familiar with the Picture class. As it turns out, the pictures are in files picture1.jpg ... picture20.jpg. Let's load the first one:

```
public class Gallery1
{
    public static void main(String[] args)
    {
        Picture pic = new Picture("picture1.jpg");
    }
}
```

That's enough to show the picture.



2. Now let's put the next picture after the first. We need to move it to the right-most *x*-coordinate of the preceding picture.



```
Picture pic = new Picture("picture1.jpg");
Picture pic2 = new Picture("picture2.jpg");
pic2.move(pic.getBounds().getMaxX(), 0);
```

3. The next step is to separate the two by a small gap when the second is moved:



```
final int GAP = 10;

Picture pic = new Picture("picture1.jpg");
Picture pic2 = new Picture("picture2.jpg");
double x = pic.getBounds().getMaxX() + GAP;
pic2.move(x, 0);
```

4. Now let's put all pictures in a row. Read the pictures in a loop, and then put each picture to the right of the one that preceded it. In the loop, you need to track two pictures: the one that is being read in, and the one that preceded it (see Section 6.7.6).



```
final int GAP = 10;
final int PICTURES = 20;

Picture pic = new Picture("picture1.jpg");
for (int i = 2; i <= PICTURES; i++)
{
    Picture previous = pic;
    pic = new Picture("picture" + i + ".jpg");
    double x = previous.getBounds().getMaxX() + GAP;
    pic.move(x, 0);
}
```

5. Of course, we don't want to have all pictures in a row. The right margin of a picture should not extend past MAX\_WIDTH.

```
double x = previous.getBounds().getMaxX() + GAP;
if (x + pic.getBounds().getWidth() < MAX_WIDTH)
{
    Place pic on current row.
}
else
{
    Place pic on next row.
}
```



If the image doesn't fit any more, then we need to put it on the next row, below all the pictures in the current row. We'll set a variable `maxY` to the maximum `y`-coordinate of all placed pictures, updating it whenever a new picture is placed:

```
maxY = Math.max(maxY, pic.getBounds().getMaxY());
```

The following statement places a picture on the next row:

```
pic.move(0, maxY + GAP);
```

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bjoe6code](http://wiley.com/go/bjoe6code) for the listings of all preliminary stages of the gallery program.

Now we have written complete programs for all preliminary stages. We know how to line up the pictures, how to separate them with gaps, how to find out when to start a new row, and where to start it.

With this knowledge, producing the final version is straightforward. Here is the program listing.

**section\_5/Gallery6.java**

```

1  public class Gallery6
2  {
3      public static void main(String[] args)
4      {
5          final int MAX_WIDTH = 720;
6          final int GAP = 10;
7          final int PICTURES = 20;
8
9          Picture pic = new Picture("picture1.jpg");
10         double maxY = 0;
11
12         for (int i = 2; i <= 20; i++)
13         {
14             maxY = Math.max(maxY, pic.getBounds().getMaxY());
15             Picture previous = pic;
16             pic = new Picture("picture" + i + ".jpg");
17             double x = previous.getBounds().getMaxX() + GAP;
18             if (x + pic.getBounds().getWidth() < MAX_WIDTH)
19             {
20                 pic.move(x, previous.getBounds().getY());
21             }
22             else
23             {
24                 pic.move(0, maxY + GAP);
25             }
26         }
27     }
28 }
```

**SELF CHECK**

- 21.** Suppose you are asked to find all words in which no letter is repeated from a list of words. What simpler problem could you try first?
- 22.** You need to write a program for DNA analysis that checks whether a substring of one string is contained in another string. What simpler problem can you solve first?
- 23.** You want to remove “red eyes” from images and are looking for red circles. What simpler problem can you start with?
- 24.** Consider the task of finding numbers in a string. For example, the string “In 1987, a typical personal computer cost \$3,000 and had 512 kilobytes of RAM.” has three numbers. Break this task down into a sequence of simpler tasks.

**Practice It**

Now you can try these exercises at the end of the chapter: R8.26, P8.5.

## 8.6 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

### 8.6.1 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 8.6.3 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 Java, related classes are grouped into packages.*

© Don Wilkie/Stockphoto.

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

## 8.6.2 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`.

## 8.6.3 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`.

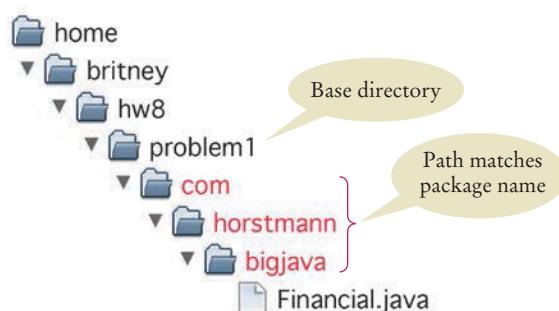
If you don't have your own domain name, you can still create a package name that has a high probability of being unique by writing your e-mail address backwards. For example, if Britney Walters has an e-mail address `walters@cs.sjsu.edu`, then she can use a package name `edu.sjsu.cs.walters` for her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

#### 8.6.4 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

**SELF CHECK**

- 25.** Which of the following are packages?
- java
  - java.lang
  - java.util
  - java.lang.Math
- 26.** Is a Java program without import statements limited to using the default and java.lang packages?
- 27.** 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: R8.28, E8.18, E8.19.

**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`).



© Don Wilkie/Stockphoto.

#### 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 sub-package 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 (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
```

## 8.7 Unit Test Frameworks

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.

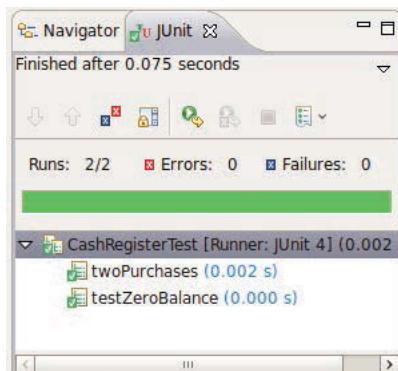
Unit testing frameworks were designed to quickly execute and evaluate test suites and to make it easy to incrementally add test cases. One of the most popular testing frameworks is JUnit. It is freely available at <http://junit.org>, and it is also built into a number of development environments, including 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.

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

**Figure 7**  
Unit Testing with JUnit



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 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 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.

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

### SELF CHECK



28. Provide a JUnit test class with one test case for the `Earthquake` class in Chapter 5.
29. 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: R8.30, E8.20, E8.21.



## Computing & Society 8.1 Personal Computing

VISICALC Screen: Reprint Courtesy of International Business Machines Corporation, © International Business Machines Corporation/IBM Corporation.

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.

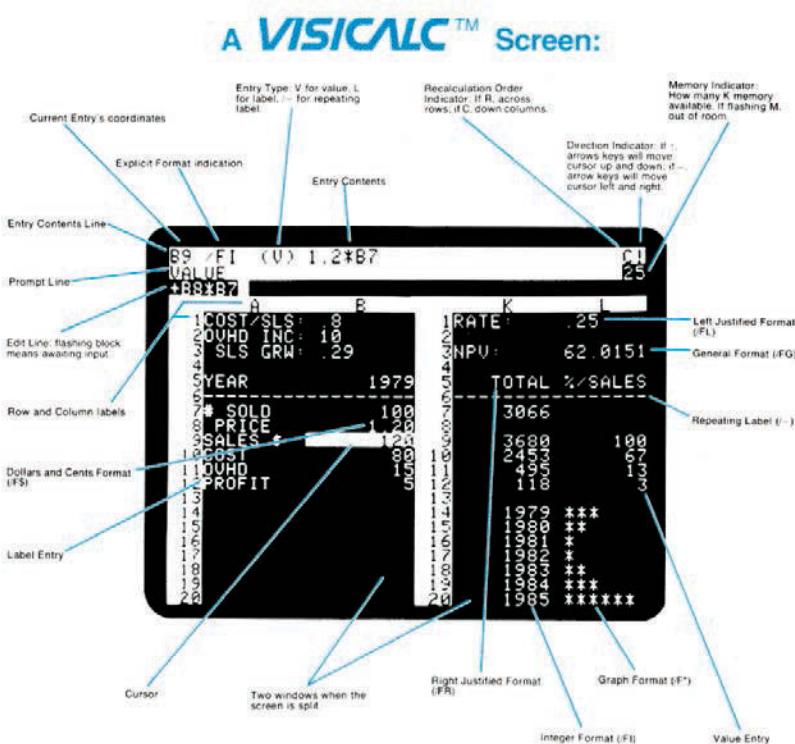
More importantly, it was a personal device. The managers were free to do the calculations that they wanted to do, not just the ones that the "high priests" in the data center provided.

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 smart phones, 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 install only those applications that the manufacturer publishes on the "app store". For example, Apple rejected MIT's iPad app for the educational language Scratch because it contained a virtual machine. 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?

The VisiCalc Spreadsheet Running on an Apple II



**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.

**Design programs that carry out complex tasks.**

- When developing a solution to a complex problem, first solve a simpler task.
- Make a plan consisting of a series of tasks, each a simple extension of the previous one, and ending with the original problem.

**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 EXERCISES**

- R8.1** Consider a car share system in which drivers pick up other riders, enabling them to make money during their commute while reducing traffic congestion. Riders wait at pickup points, are dropped off at their destinations, and pay for the distance traveled. Drivers get a monthly payment. An app lets drivers and riders enter their route and time. It notifies drivers and riders and handles billing. Find classes that would be useful for designing such a system.
- R8.2** Suppose you want to design a social network for internship projects at your university. Students can register their skills and availability. Project sponsors describe projects, required skills, expected work effort, and desired completion date. A search facility lets students find matching projects. Find classes that would be useful for designing such a system.
- R8.3** 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.
- R8.4** 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:
- Invoice
  - InvoicePrinter
  - PrintInvoice
  - InvoiceProgram

- R8.5** 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?
- R8.6** Look at the public interface of the `java.lang.System` class and discuss whether or not it is cohesive.
- R8.7** 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`.
- R8.8** 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`.
- R8.9** On which classes does the class `Integer` in the standard library depend?
- R8.10** On which classes does the class `Rectangle` in the standard library depend?
- R8.11** Classify the methods of the class `Scanner` that are used in this book as accessors and mutators.
- R8.12** Classify the methods of the class `Rectangle` as accessors and mutators.
- R8.13** Is the `Resistor` class in Exercise P8.12 a mutable or immutable class? Why?
- R8.14** Which of the following classes are immutable?
- `Rectangle`
  - `String`
  - `Random`
- R8.15** Which of the following classes are immutable?
- `PrintStream`
  - `Date`
  - `Integer`
- R8.16** 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?
- R8.17** 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;
    }
}

```

- R8.18** 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?

- R8.19** 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?

- R8.20** How can you write a method that swaps two floating-point numbers?

*Hint:* java.awt.Point.

- R8.21** 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;
}

```

- R8.22** 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.

- R8.23 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);
    }
}
```

- R8.24 Look at the methods in the `Integer` class. Which are static? Why?
- R8.25 Look at the methods in the `String` class (but ignore the ones that take an argument of type `char[]`). Which are static? Why?
- R8.26 Consider the task of *fully justifying* a paragraph of text to a target length, by putting as many words as possible on each line and evenly distributing extra spaces so that each line has the target length. Devise a plan for writing a program that reads a paragraph of text and prints it fully justified. Describe a sequence of progressively more complex intermediate programs, similar to the approach in Section 8.5.
- R8.27 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?
- R8.28 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.
- R8.29 What is the default package? Have you used it before this chapter in your programming?
- Testing R8.30 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 Reimplement the `Day` class of Worked Example 2.1 to be immutable. Change mutator methods to return new `Day` objects. Also change the demonstration program.

- E8.6 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.

- E8.7 Solve Exercise E8.6 by implementing classes `Cube`, `Sphere`, `Cylinder`, and `Cone`. Which approach is more object-oriented?
- E8.8 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.
- E8.9 Repeat Exercise E8.8, using multiple classes. Provide a `GradeBook` class that collects objects of type `Student`.

- E8.10 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?



**■■ E8.11** 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?

**■■ E8.12** 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`.

**■■ E8.13** Using the Picture class from Worked Example 6.2, write a method

```
public static Picture superimpose(Picture pic1, Picture pic2)
```

that superimposes two pictures, yielding a picture whose width and height are the larger of the widths and heights of `pic1` and `pic2`. In the area where both pictures have pixels, average the colors.

**■■ E8.14** Using the Picture class from Worked Example 6.2, write a method

```
public static Picture greenScreen(Picture pic1, Picture pic2)
```

that superimposes two pictures, yielding a picture whose width and height are the larger of the widths and heights of `pic1` and `pic2`. In the area where both pictures have pixels, use `pic1`, except when its pixels are green, in which case, you use `pic2`.

**■ E8.15** 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`.

**■■ E8.16** 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`.**■■ E8.17** 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.**■■ E8.18** 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.**■ E8.19** Place a `BankAccount` class in a package whose name is derived from your e-mail address, as described in Section 8.6. Keep the `BankAccountTester` class in the default package.**■■ Testing E8.20** 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 E8.21** Provide JUnit test class `TaxReturnTest` with three test methods that test different tax situations for the `TaxReturn` class in Chapter 5.

■ **Graphics E8.22** 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 E8.23** Repeat Exercise E8.22 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?

■ ■ **E8.24** 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.

■ ■ **E8.25** 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.

■ ■ **E8.26** 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() { . . . }
}
```

- \*\*\* P8.2** Improve the picture gallery program in Section 8.5 to fill the space more efficiently. Instead of lining up all pictures along the top edge, find the first available space where you can insert a picture (still respecting the gaps).



National Gallery of Art (see page C-1).

*Hint:* Solve a simpler problem first, lining up the pictures without gaps.



That is still not easy. You need to test whether a new picture fits. Put the bounding rectangles of all placed pictures in an `ArrayList` and implement a method

```
public static boolean intersects(Rectangle r, ArrayList<Rectangle> rectangles)
```

that checks whether `r` intersects any of the given rectangles. Use the `intersects` method in the `Rectangle` class.

Then you need to figure out where you can try putting the new picture. Try something simple first, and check the corner points of all existing rectangles. Try points with smaller  $y$ -coordinates first.

For a better fit, check all points whose  $x$ - and  $y$ -coordinates are the  $x$ - and  $y$ -coordinates of corner points, but not necessarily the same point.

Once that works, add the gaps between images.

- P8.3 Simulate a car sharing system in which car commuters pick up and drop off passengers at designated stations. Assume that there are 30 such stations, one at every mile along a route. At each station, randomly generate a number of cars and passengers, each of them with a desired target station.

Each driver picks up three random passengers whose destination is on the way to the car's destination, drops them off where requested, and picks up more if possible. A driver gets paid per passenger per mile. Run the simulation 1,000 times and report the average revenue per mile.

Use classes `Car`, `Passenger`, `Station`, and `Simulation` in your solution.

- P8.4 In Exercise P8.3, drivers picked up passengers at random. Try improving that scheme. Are drivers better off picking passengers that want to go as far as possible along their route? Is it worth looking at stations along the route to optimize the loading plan? Come up with a solution that increases average revenue per mile.

- P8.5 Tabular data are often saved in the CSV (comma-separated values) format. Each table row is stored in a line, and column entries are separated by commas. However, if an entry contains a comma or quotation marks, they enclosed in quotation marks, doubling any quotation marks of the entry. For example,

```
John Jacob Astor,1763,1848
"William Backhouse Astor, Jr.",1829,1892
"John Jacob ""Jakey"" Astor VI",1912,1992
```

Provide a class `Table` with methods

```
public void addLine(String line)
public String getEntry(int row, int column)
public int rows()
public int columns()
```

Solve this problem by producing progressively more complex intermediate versions of your class and a tester, similar to the approach in Section 8.5.

- P8.6 For faster sorting of letters, the U.S. Postal Service encourages companies that send large volumes of mail to use a bar code denoting the ZIP code (see Figure 8).

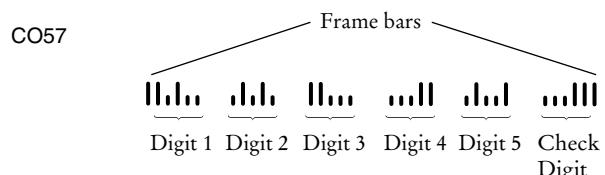
The encoding scheme for a five-digit ZIP code is shown in Figure 9. There are full-height frame bars on each side. The five encoded digits are followed by a check digit, which is computed as follows: Add up all digits, and choose the check digit to

\*\*\*\*\* \* ECRLOT \*\* CO57

CODE C671RTS2  
JOHN DOE  
1009 FRANKLIN BLVD  
SUNNYVALE CA 95014 – 5143



**Figure 8** A Postal Bar Code



**Figure 9** Encoding for Five-Digit Bar Codes

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.

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.)

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

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.

- **Business P8.7** 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.
- **Business P8.8** 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 P8.9** The Downtown Marketing Association wants to promote downtown shopping with a loyalty program similar to the one in Exercise P8.8. 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.



© ThreeJays/iStockphoto.

**■■■ Science P8.10** 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.

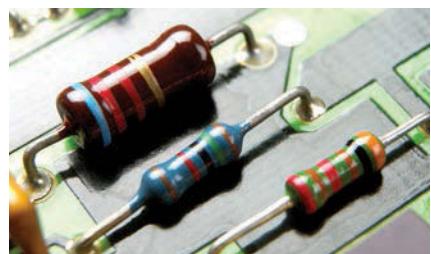
Supply the following methods:

- 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  $-9.81 \text{ m/s}^2$ ; 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  $\alpha$  and initial velocity  $v$ . (Compute the  $x$ -velocity as  $v \cos \alpha$  and the  $y$ -velocity as  $v \sin \alpha$ ; 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 P8.11** Continue Exercise P8.10, and draw the trajectory of the cannonball.

**■■ Science P8.12** The colored bands on the top-most resistor shown in the photo at right indicate a resistance of  $6.2 \text{ k}\Omega \pm 5$  percent. The resistor tolerance of  $\pm 5$  percent indicates the acceptable variation in the resistance. A  $6.2 \text{ k}\Omega \pm 5$  percent resistor could have a resistance as small as  $5.89 \text{ k}\Omega$  or as large as  $6.51 \text{ k}\Omega$ . We say that  $6.2 \text{ k}\Omega$  is the *nominal value* of the resistance and that the actual value of the resistance can be any value between  $5.89 \text{ k}\Omega$  and  $6.51 \text{ k}\Omega$ .



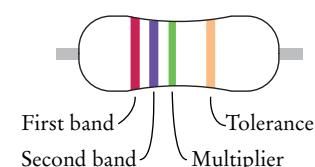
© Maria Toutoudaki/Stockphoto

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  $330 \Omega \pm 10$  percent resistors.

**■■ Science P8.13** In the `Resistor` class from Exercise P8.12, 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.



First band  
Second band  
Multiplier  
Tolerance

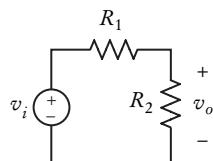
Color	Digit	Multiplier	Tolerance
Black	0	$\times 10^0$	—
Brown	1	$\times 10^1$	$\pm 1\%$
Red	2	$\times 10^2$	$\pm 2\%$
Orange	3	$\times 10^3$	—
Yellow	4	$\times 10^4$	—
Green	5	$\times 10^5$	$\pm 0.5\%$
Blue	6	$\times 10^6$	$\pm 0.25\%$
Violet	7	$\times 10^7$	$\pm 0.1\%$
Gray	8	$\times 10^8$	$\pm 0.05\%$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

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  $\pm 5$  percent, for a resistance of  $2,700 \text{ k}\Omega$ , plus or minus 5 percent.

**Science P8.14** 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 P8.12. 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 P8.12. 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 access 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. Of course, there is more than one way to simplify the problem. One way is to print the words in which the first letter is not repeated.
22. You could first write a program that prints all substrings of a given string.

23. You can look for a single red pixel, or a block of nine neighboring red pixels.
24. Here is one plan:
- Find the position of the first digit in a string.
  - Find the position of the first non-digit after a given position in a string.
  - Extract the first integer from a string (using the preceding two steps).
  - Print all integers from a string. (Use the first three steps, then repeat with the substring that starts after the extracted integer.)
25. (a) No; (b) Yes; (c) Yes; (d) No
26. No—you can use fully qualified names for all other classes, such as `java.util.Random` and `java.awt.Rectangle`.

27. `/home/me/cs101/hw1/problem1` or, on Windows, `c:\Users\Me\cs101\hw1\problem1`.

28. 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());
    }
}
```

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

# 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



© Jason Hosking/Photodisc/Getty Images, Inc.

## CHAPTER CONTENTS

### 9.1 INHERITANCE HIERARCHIES 424

**PT1** Use a Single Class for Variation in Values,  
Inheritance for Variation in Behavior 428

### 9.2 IMPLEMENTING SUBCLASSES 428

**SYN** Subclass Declaration 430

**CE1** Replicating Instance Variables from the  
Superclass 432

**CE2** Confusing Super- and Subclasses 432

### 9.3 OVERRIDING METHODS 433

**SYN** Calling a Superclass Method 433

**CE3** Accidental Overloading 437

**CE4** Forgetting to Use `super` When Invoking a  
Superclass Method 437

**ST1** Calling the Superclass Constructor 438

**SYN** Constructor with Superclass  
Initializer 438

### 9.4 POLYMORPHISM 439

**ST2** Dynamic Method Lookup and the  
Implicit Parameter 442

**ST3** Abstract Classes 443

**ST4** Final Methods and Classes 444

**ST5** Protected Access 444

**HT1** Developing an Inheritance Hierarchy 445

**WE1** Implementing an Employee Hierarchy for  
Payroll Processing

### 9.5 OBJECT: THE COSMIC SUPERCLASS 450

**SYN** The `instanceof` Operator 453

**CE5** Don't Use Type Tests 454

**ST6** Inheritance and the `toString` Method 455

**ST7** Inheritance and the `equals` Method 456

**C&S** Who Controls the Internet? 456



© Jason Hosking/Photodisc/Getty Images, Inc.

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.

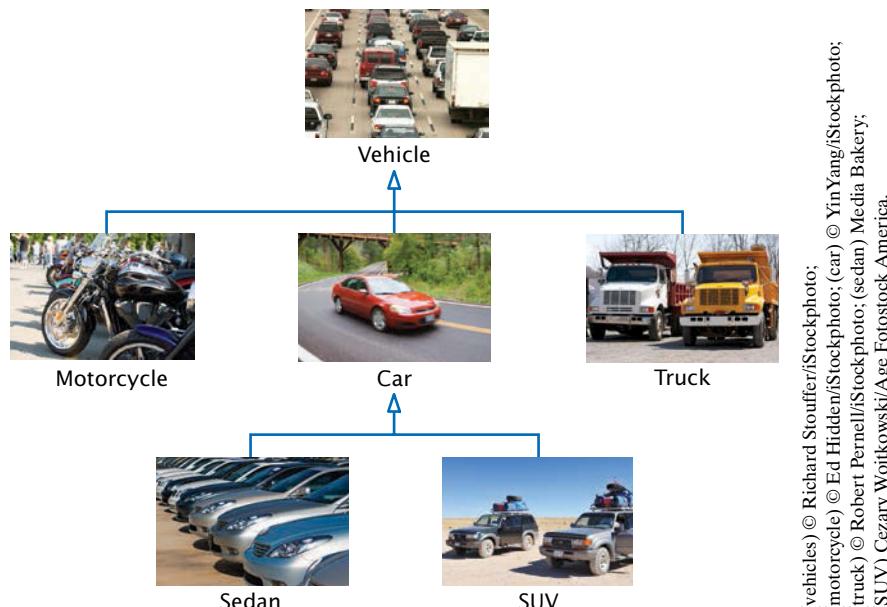
You can always use a subclass object in place of a superclass object.

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

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

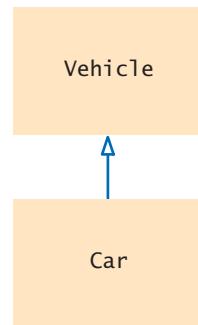
When you use inheritance in your programs, you can reuse code instead of duplicating it. This reuse comes in two forms. First, a subclass inherits the methods of the superclass. For example, if the Vehicle class has a drive method, then a subclass Car automatically inherits the method. It need not be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

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)
```

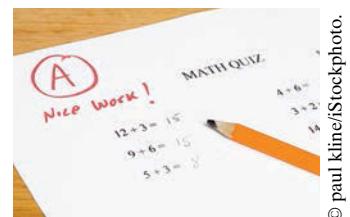
Because `Car` is a subclass of `Vehicle`, you can call that method with a `Car` object:

```
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).

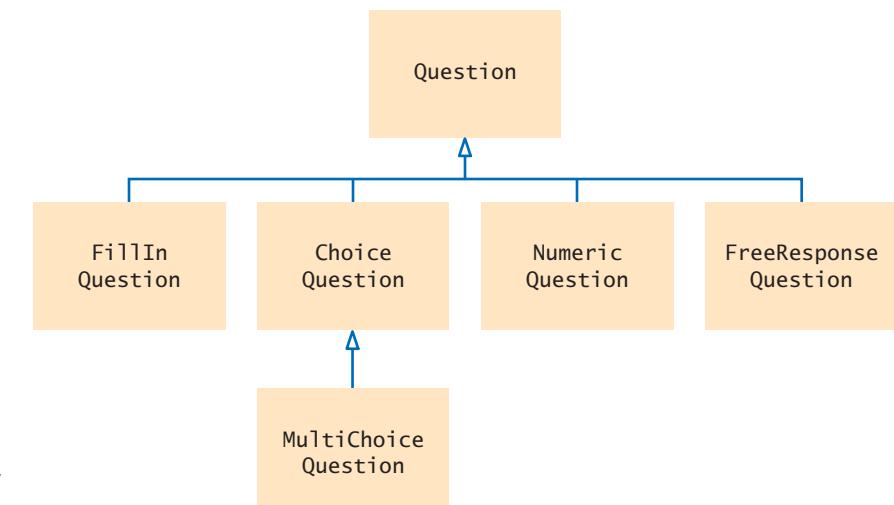
In this chapter, we will consider a simple hierarchy of classes. Most likely, you have taken computer-graded quizzes. A quiz consists of questions, and there are different kinds of questions:

- 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: R9.2, R9.8, R9.10.

## 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?

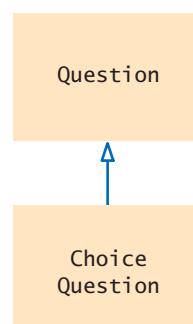
1. Australia
2. Canada
3. Denmark
4. United States

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).

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

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

A subclass inherits all methods that it does not override.



**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.*



Media Bakery.

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:

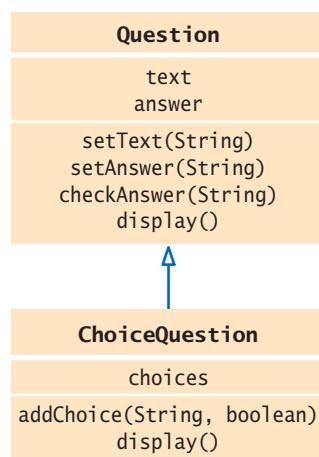
```
public class ChoiceQuestion extends Question
{
    // This instance variable is added to the subclass
    private ArrayList<String> choices;

    // 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.

Declare instance variables that are **added** to the subclass.

Declare methods that are **added** to the subclass.

Declare methods that the subclass **overrides**.

```
Subclass                      Superclass
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.

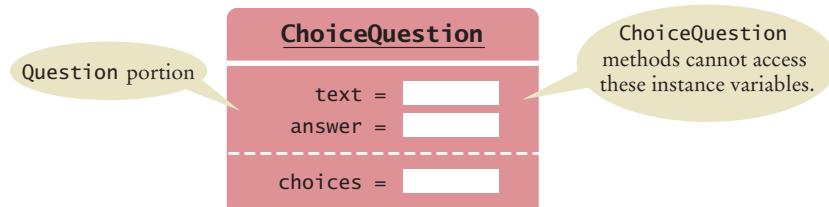
You can call the inherited methods on a subclass object:

```
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,

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

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

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

#### FULL CODE EXAMPLE

Go to [wiley.com/go/bje06code](http://wiley.com/go/bje06code) to download a program that shows a simple Car class extending a Vehicle class.



#### SELF CHECK



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: R9.4, E9.9, E9.13.

### Common Error 9.1



#### Replicating Instance Variables from the Superclass

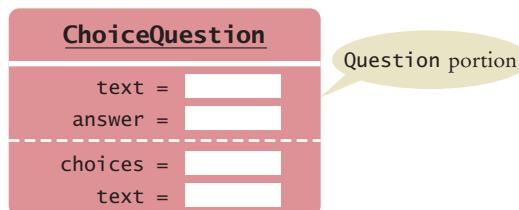
A subclass has no access to the private instance variables of the superclass.

```
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 437.

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
```



#### SELF CHECK

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: E9.4, E9.5, E9.14.

### 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.

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:

```
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.

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.

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`.

### Syntax 9.3

### Constructor with Superclass Initializer

**Syntax**

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

The superclass constructor is called first.

The constructor body can contain additional statements.

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

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`:

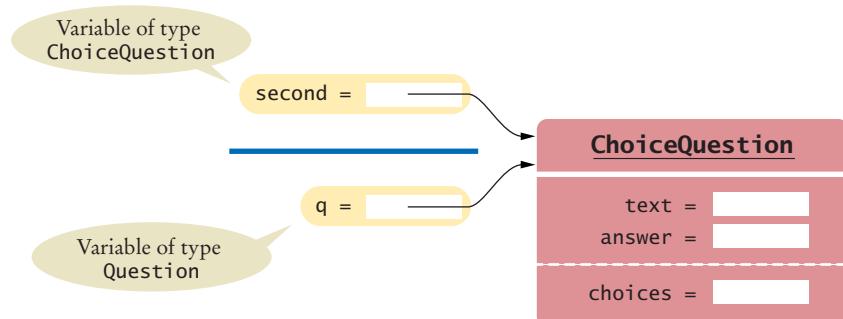
```
public static void presentQuestion(Question q)
{
    q.display();
    System.out.print("Your answer: ");
    Scanner in = new Scanner(System.in);
    String response = in.nextLine();
    System.out.println(q.checkAnswer(response));
}
```

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:

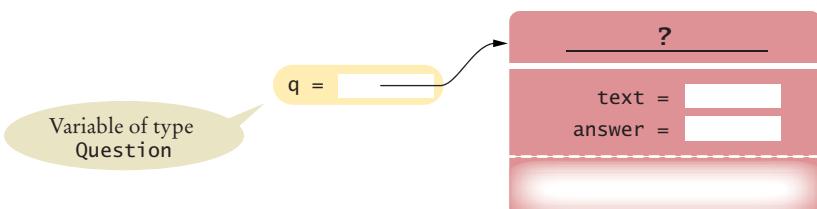
```
ChoiceQuestion second = new ChoiceQuestion();
...
presentQuestion(second); // OK to pass a ChoiceQuestion
```

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.*



© Alphophoto/iStockphoto.

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 441, you will see that the method called depends on the contents of the parameter variable `q`. In the first case, `q` refers to a `Question` object, so the `Question.display` method is called. But in the second case, `q` refers to a `ChoiceQuestion`, so the `ChoiceQuestion.display` method is called, showing the list of choices.

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

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

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

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.

## 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
```



#### SELF CHECK

16. Assuming `SavingsAccount` is a subclass of `BankAccount`, which of the following code fragments are valid in Java?
- `BankAccount account = new SavingsAccount();`
  - `SavingsAccount account2 = new BankAccount();`
  - `BankAccount account = null;`
  - `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: R9.7, E9.7, P9.17.

### 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

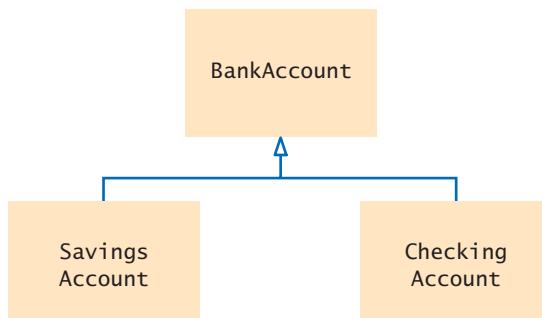
List the classes that are part of the hierarchy.

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/bjoe6code](http://wiley.com/go/bjoe6code) 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 [wiley.com/go/bjoe6examples](http://wiley.com/go/bjoe6examples) and download Worked Example 9.1.



Jose Luis Pelaez Inc./  
Getty Images, Inc.

## 9.5 Object: The Cosmic Superclass

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

- `toString`, which yields a string describing the object (Section 9.5.1).
- `equals`, which compares objects with each other (Section 9.5.2).
- `hashCode`, which yields a numerical code for storing the object in a set (see Special Topic 15.1).

### 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;
```

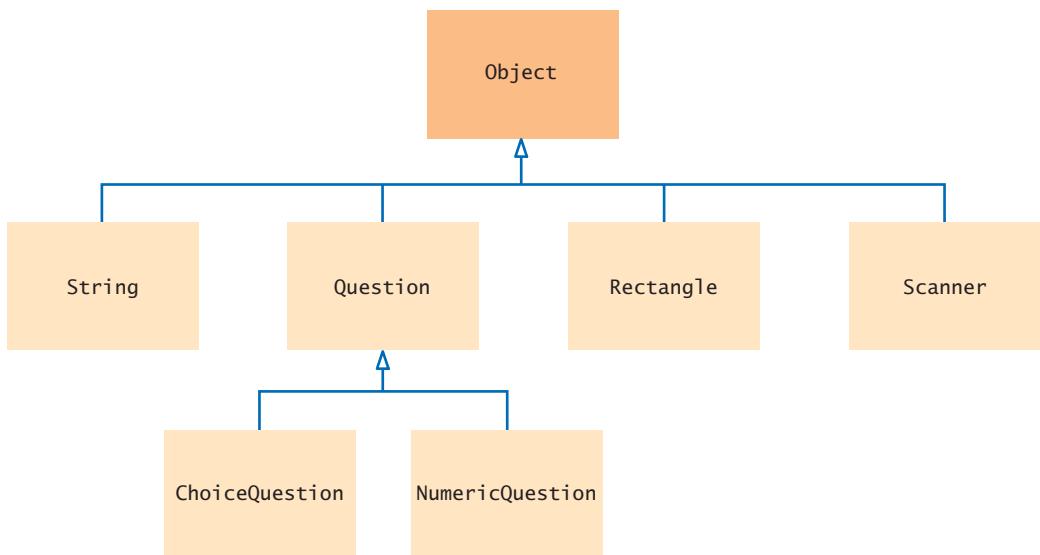
On one side of the `+` concatenation operator is a string, but on the other side is an object reference. The Java compiler automatically invokes the `toString` method to turn the object into a string. Then both strings are concatenated. In this case, the result is the string

```
"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 momSavings = new BankAccount(5000);
String s = momSavings.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.)

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

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

## 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
```

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)
    {
        . . .
    }
    . . .
}
```

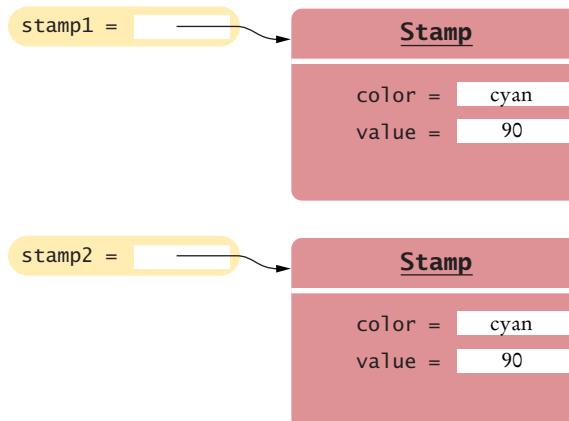
Now you have a slight problem. The `Object` class knows nothing about stamps, so it declares the `otherObject` parameter variable of the `equals` method to have the type `Object`. When overriding the method, you are not allowed to change the type of the parameter variable. Cast the parameter variable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

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

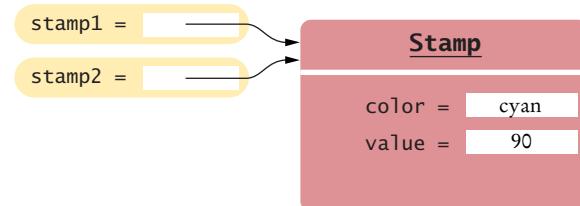


© Ken Brown/iStockphoto.

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



**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.

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.

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`.

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.

## Syntax 9.4

### The `instanceof` Operator

**Syntax**    `object instanceof TypeName`

If `anObject` is null,  
`instanceof` returns false.

Returns true if `anObject` can be cast to a `Question`.

```
if (anObject instanceof Question)
{
    Question q = (Question) anObject;
    ...
}
```

The object may belong to a subclass of `Question`.

You can invoke `Question` methods on this variable.

Two references to the same object.

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

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

**FULL CODE EXAMPLE**  
Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download a program that demonstrates the `toString` method and the `instanceof` operator.

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();
```

### SELF CHECK



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: E9.10, E9.11, E9.15.

### 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](http://www.gutenberg.org)). In 1989, Tim Berners-Lee, a computer scientist at CERN (the European organization for nuclear research) started work on hyperlinked documents, allowing users to browse by following links to related documents. 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 Andreessen, 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). Andreessen 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:

```

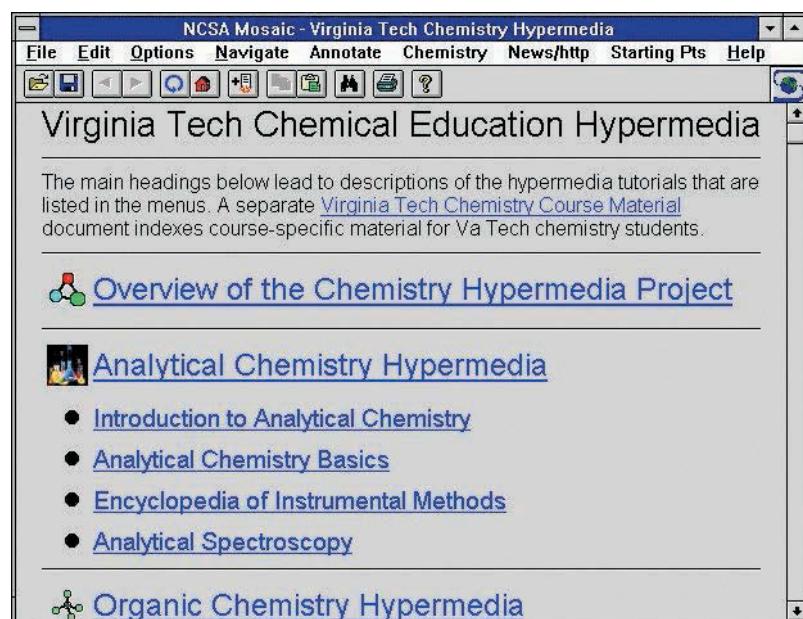
public CollectibleStamp extends Stamp
{
    private int year;
    ...
    public boolean equals(Object otherObject)
    {
        if (!super.equals(otherObject)) { return false; }
        CollectibleStamp other = (CollectibleStamp) otherObject;
        return year == other.year;
    }
}

```

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, and its security organizations have spied on the Internet usage of citizens.

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

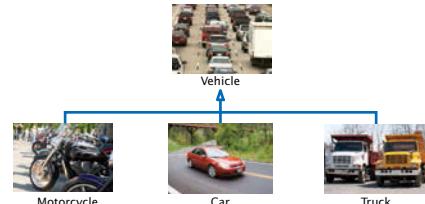
The Internet has become a powerful force for delivering information—both good and bad. It is our responsibility as citizens to demand of our government that we can control which information to access.



*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 EXERCISES

- **R9.1** In Worked Example 9.1,
  - a. What are the subclasses of `Employee`?
  - b. What are the superclasses of `Manager`?
  - c. What are the super- and subclasses of `SalariedEmployee`?
  - d. Which classes override the `weeklyPay` method of the `Employee` class?
  - e. Which classes override the `setName` method of the `Employee` class?
  - f. What are the instance variables of an `HourlyEmployee` object?
  
- **R9.2** Identify the superclass and subclass in each of the following pairs of classes.
 

a. <code>Employee</code> , <code>Manager</code> b. <code>GraduateStudent</code> , <code>Student</code> c. <code>Person</code> , <code>Student</code> d. <code>Employee</code> , <code>Professor</code> e. <code>BankAccount</code> , <code>CheckingAccount</code>	f. <code>Vehicle</code> , <code>Car</code> g. <code>Vehicle</code> , <code>Minivan</code> h. <code>Car</code> , <code>Minivan</code> i. <code>Truck</code> , <code>Vehicle</code>
---	--
  
- **R9.3** 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?
  
- **R9.4** Which methods does the `ChoiceQuestion` class inherit from its superclass? Which methods does it override? Which methods does it add?
  
- **R9.5** 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?
  
- **R9.6** List the instance variables of a `CheckingAccount` object from How To 9.1.
  
- ■ **R9.7** Suppose the class `Sub` extends the class `Sandwich`. Which of the following assignments are legal?
 

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

a. <code>x = y;</code> b. <code>y = x;</code>	c. <code>y = new Sandwich();</code> d. <code>x = new Sub();</code>
--	---
  
- **R9.8** Draw an inheritance diagram that shows the inheritance relationships between these classes.
 

<ul style="list-style-type: none"> <li>• Person</li> <li>• Employee</li> <li>• Student</li> </ul>	<ul style="list-style-type: none"> <li>• Instructor</li> <li>• Classroom</li> <li>• Object</li> </ul>
---	---
  
- **R9.9** In an object-oriented traffic simulation system, we have the classes listed below. Draw an inheritance diagram that shows the relationships between these classes.
 

<ul style="list-style-type: none"> <li>• Vehicle</li> <li>• Car</li> <li>• Truck</li> <li>• Sedan</li> <li>• Coupe</li> </ul>	<ul style="list-style-type: none"> <li>• PickupTruck</li> <li>• SportUtilityVehicle</li> <li>• Minivan</li> <li>• Bicycle</li> <li>• Motorcycle</li> </ul>
---	--

- **R9.10** What inheritance relationships would you establish among the following classes?

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

- **R9.11** How does a cast such as `(BankAccount) x` differ from a cast of number values such as `(int) x`?

- **R9.12** 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** Implement a subclass of `BankAccount` called `BasicAccount` whose `withdraw` method will not withdraw more money than is currently in the account.
- **E9.2** Implement a subclass of `BankAccount` called `BasicAccount` whose `withdraw` method charges a penalty of \$30 for each withdrawal that results in an overdraft.
- **E9.3** Reimplement the `CheckingAccount` class from How To 9.1 so that the first overdraft in any given month incurs a \$20 penalty, and any further overdrafts in the same month result in a \$30 penalty.
- **E9.4** Add a class `NumericQuestion` to the question hierarchy of Section 9.1. If the response and the expected answer differ by no more than 0.01, accept the response as correct.
- **E9.5** 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 \_\_\_\_\_
- **E9.6** 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".
- **E9.7** 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.
- **E9.8** 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.
- **E9.9** 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.
- **E9.10** Provide `toString` methods for the `Question` and `ChoiceQuestion` classes.
- **E9.11** Implement a superclass `Person`. Make two classes, `Student` and `Instructor`, that inherit from `Person`. A person has a name and a year of birth. A student has a major, and

an instructor has a salary. Write the class declarations, the constructors, and the methods `toString` for all classes. Supply a test program for these classes and methods.

- **E9.12** 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.
- **E9.13** 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.
- **E9.14** Repeat Exercise E9.13, but in the `BetterRectangle` constructor, invoke the superclass constructor.
- **E9.15** 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.
- **E9.16** Reimplement the `LabeledPoint` class of Exercise E9.15 by storing the location in a `java.awt.Point` object. Your `toString` method should invoke the `toString` method of the `Point` class.
- **Business E9.17** Change the `CheckingAccount` class in How To 9.1 so that a \$1 fee is levied for deposits or withdrawals in excess of three free monthly transactions. Place the code for computing the fee into a separate method that you call from the `deposit` and `withdraw` methods.

## PROGRAMMING PROJECTS

- **P9.1** Implement a class `Clock` whose `getHours` and `getMinutes` methods return the current time at your location. (Call `java.time.Instant.now().toString()` or, if you are not using Java 8, `new java.util.Date().toString()` and extract the time from that string.) Also provide a `getTime` method that returns a string with the hours and minutes by calling the `getHours` and `getMinutes` methods. Provide a subclass `WorldClock` whose constructor accepts a time offset. For example, if you live in California, a new `WorldClock(3)` should show the time in New York, three time zones ahead. Which methods did you override? (You should not override `getTime`.)
- **P9.2** Add an alarm feature to the `Clock` class of Exercise P9.1. When `setAlarm(hours, minutes)` is called, the clock stores the alarm. When you call `getTime`, and the alarm time has been reached or exceeded, return the time followed by the string "Alarm" (or, if you prefer, the string "\u23F0") and clear the alarm. What do you need to do to make the `setAlarm` method work for `WorldClock` objects?
- **Business P9.3** 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



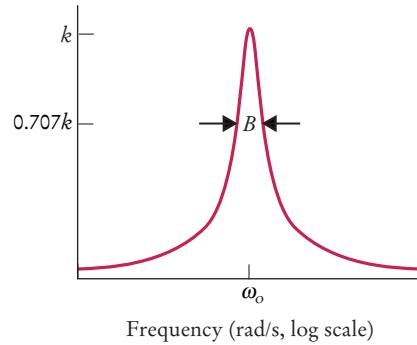
© Pali Rao/Stockphoto

with a mixture of appointments. Have the user enter a date and print out all appointments that occur on that date.

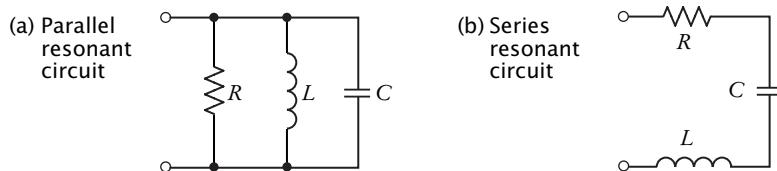
**■■ Business P9.4** Improve the appointment book program of Exercise P9.3. Give the user the option to add new appointments. The user must specify the type of the appointment, the description, and the date.

**■■■ Business P9.5** Improve the appointment book program of Exercises P9.3 and P9.4 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 P9.6** 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,  $\omega_0$ , the bandwidth,  $B$ , and the gain at the resonant frequency,  $k$ .



Two simple resonant circuits are shown in the figure below. The circuit in (a) is called a *parallel resonant circuit*. The circuit in (b) is called a *series resonant circuit*. Both resonant circuits consist of a resistor having resistance  $R$ , a capacitor having capacitance  $C$ , and an inductor having inductance  $L$ .



These circuits are designed by determining values of  $R$ ,  $C$ , and  $L$  that cause the resonant frequency response to be described by specified values of  $\omega_0$ ,  $B$ , and  $k$ . The design equations for the parallel resonant circuit are:

$$R = k, \quad C = \frac{1}{BR}, \text{ and} \quad L = \frac{1}{\omega_0^2 C}$$

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

$$R = \frac{1}{k}, \quad L = \frac{R}{B}, \text{ and} \quad C = \frac{1}{\omega_0^2 L}$$

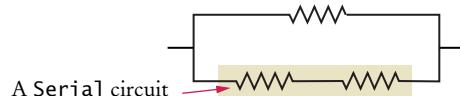
Write a Java program that represents `ResonantCircuit` as a superclass and represents `SeriesResonantCircuit` and `ParallelResonantCircuit` as subclasses. Give the superclass three private instance variables representing the parameters  $\omega_0$ ,  $B$ , and  $k$  of the resonant frequency response. 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  $\omega_0$ ,  $B$ , and  $k$ ) and the circuit (the values of  $R$ ,  $C$ , and  $L$ ).

All classes should provide appropriate constructors.

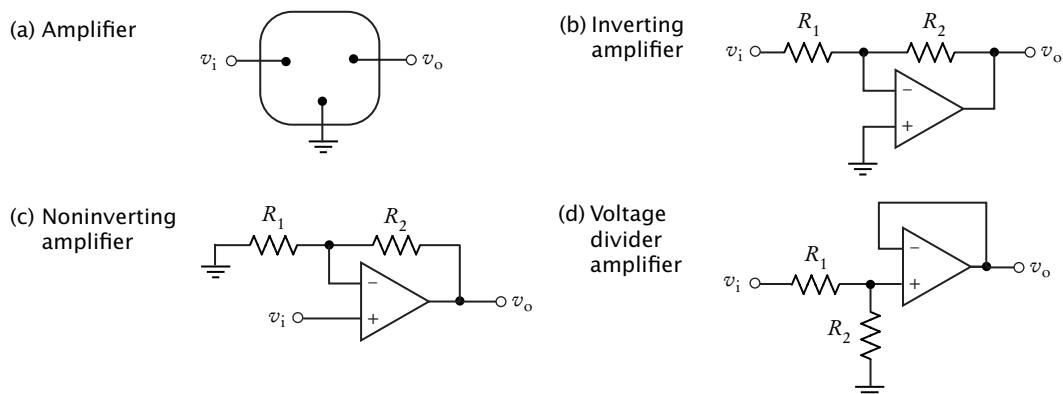
Supply a class that demonstrates that the subclasses all work properly.

- \*\*\* Science P9.7** 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 P9.8** 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.



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

amplifier and voltage divider amplifier are given by  $g = 1 + \frac{R_2}{R_1}$  and  $g = \frac{R_2}{R_1 + R_2}$ , respectively.

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

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

### 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.



## WORKED EXAMPLE 9.1



## Implementing an Employee Hierarchy for Payroll Processing

**Problem Statement** Your task is to implement payroll processing for different kinds of employees.

- Hourly employees get paid an hourly rate, but if they work more than 40 hours per week, the excess is paid at “time and a half”.
- Salaried employees get paid their salary, no matter how many hours they work.
- Managers are salaried employees who get paid a salary and a bonus.



Jose Luis Pelaez Inc./Getty Images, Inc.

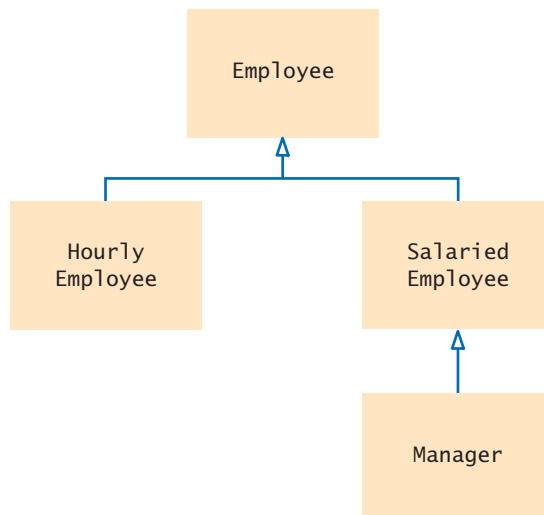
Your program should compute the pay for a collection of employees. For each employee, ask for the number of hours worked in a given week, then display the wages earned.

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

In our case, the problem description lists three classes: `HourlyEmployee`, `SalariedEmployee`, and `Manager`. We need a class that expresses the commonality among them: `Employee`.

**Step 2** Organize the classes into an inheritance hierarchy.

Here is the inheritance diagram for our classes:



**Step 3** Determine the common responsibilities of the classes.

In order to discover the common responsibilities, write pseudocode for processing the objects.

**For each employee**  
**Print the name of the employee.**  
**Read the number of hours worked.**  
**Compute the wages due for those hours.**

We conclude that the `Employee` superclass has these responsibilities:

**Get the name.**  
**Compute the wages due for a given number of hours.**

## WE2 Chapter 9 Inheritance

### Step 4 Decide which methods are overridden in subclasses.

In our example, there is no variation in getting the employee's name, but the salary is computed differently in each subclass, so `weeklyPay` will be overridden in each subclass.

```
/*
 * An employee with a name and a mechanism for computing weekly pay.
 */
public class Employee
{
    ...
    /**
     * Gets the name of this employee.
     * @return the name
     */
    public String getName() { . . . }

    /**
     * Computes the pay for one week of work.
     * @param hoursWorked the number of hours worked in the week
     * @return the pay for the given number of hours
     */
    public double weeklyPay(int hoursWorked) { . . . }
}
```

### Step 5 Declare the public interface of each class.

We will construct employees by supplying their name and salary information.

```
public class HourlyEmployee extends Employee
{
    ...
    /**
     * Constructs an hourly employee with a given name and weekly wage.
     */
    public HourlyEmployee(String name, double wage) { . . . }
    ...

    public class SalariedEmployee extends Employee
    {
        ...
        /**
         * Constructs a salaried employee with a given name and annual salary.
         */
        public SalariedEmployee(String name, double salary) { . . . }
        ...

    public class Manager extends SalariedEmployee
    {
        ...
        /**
         * Constructs a manager with a given name, annual salary, and weekly bonus.
         */
        public Manager(String name, double salary, double bonus) { . . . }
        ...
    }
```

These constructors need to set the name of the `Employee` object. We will add a method `setName` to the `Employee` class for this purpose:

```
public class Employee
{
    . . .
    public void setName(String employeeName) { . . . }
    . . .
}
```

Of course, each subclass needs a method for computing the weekly wages:

```
// This method overrides the superclass method
public double weeklyPay(int hoursWorked) { . . . }
```

In this simple example, no further methods are required.

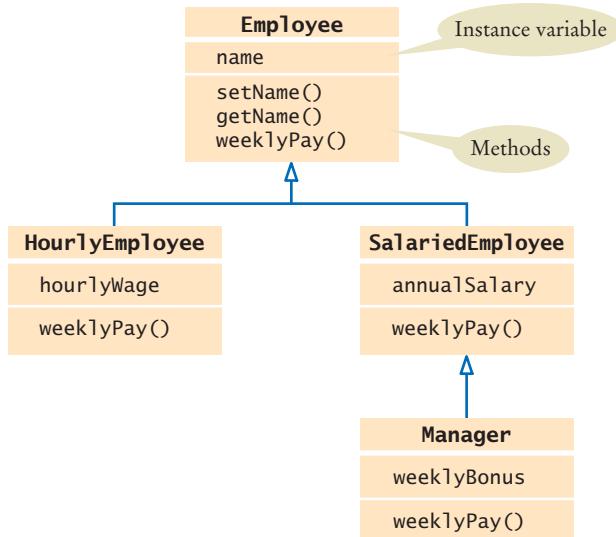
### Step 6

Identify instance variables.

All employees have a name. Therefore, the `Employee` class should have an instance variable `name`. (See the revised hierarchy below.)

What about the salaries? Hourly employees have an hourly wage, whereas salaried employees have an annual salary. While it would be possible to store these values in an instance variable of the superclass, it would not be a good idea. The resulting code, which would need to make sense of what that number means, would be complex and error-prone.

Instead, `HourlyEmployee` objects will store the hourly wage and `SalariedEmployee` objects will store the annual salary. Manager objects need to store the weekly bonus.



### Step 7

Implement constructors and methods.

In a subclass constructor, we need to remember to set the instance variables of the superclass:

```
public SalariedEmployee(String name, double salary)
{
    setName(name);
    annualSalary = salary;
}
```

Here we use a method. Special Topic 9.1 shows how to invoke a superclass constructor. We use that technique in the `Manager` constructor:

```
public Manager(String name, double salary, double bonus)
{
    super(name, salary)
    weeklyBonus = bonus;
```

## WE4 Chapter 9 Inheritance

```
}
```

The weekly pay needs to be computed as specified in the problem description:

```
public class HourlyEmployee extends Employee
{
    . .
    public double weeklyPay(int hoursWorked)
    {
        double pay = hoursWorked * hourlyWage;
        if (hours_worked > 40)
        {
            // Add overtime
            pay = pay + ((hoursWorked - 40) * 0.5) * hourlyWage;
        }
        return pay;
    }
}

public class SalariedEmployee extends Employee
{
    . .
    public double weeklyPay(int hoursWorked)
    {
        final int WEEKS_PER_YEAR = 52;
        return annualSalary / WEEKS_PER_YEAR;
    }
}
```

In the case of the Manager, we need to call the version from the SalariedEmployee superclass:

```
public class Manager extends Employee
{
    . .
    public double weeklyPay(int hours)
    {
        return super.weeklyPay(hours) + weeklyBonus;
    }
}
```

**Step 8** Construct objects of different subclasses and process them.

In our sample program, we populate an array of employees and compute the weekly salaries:

```
Employee[] staff = new Employee[3];
staff[0] = new HourlyEmployee("Morgan, Harry", 30);
staff[1] = new SalariedEmployee("Lin, Sally", 52000);
staff[2] = new Manager("Smith, Mary", 104000, 50);

Scanner in = new Scanner(System.in);
for (Employee e : staff)
{
    System.out.print("Hours worked by " + e.getName() + ": ");
    int hours = in.nextInt();
    System.out.println("Salary: " + e.weeklyPay(hours));
}
```

The complete code for this program is contained in the ch09/worked\_example\_1 directory of your source code.

# INTERFACES

## CHAPTER GOALS

To be able to declare and use interface types

To appreciate how interfaces can be used to decouple classes

To learn how to implement helper classes as inner classes

To implement event listeners in graphical applications

## CHAPTER CONTENTS

### 10.1 USING INTERFACES FOR ALGORITHM REUSE 466

**SYN** Declaring an Interface 468

**SYN** Implementing an Interface 469

**CE1** Forgetting to Declare Implementing Methods as Public 472

**CE2** Trying to Instantiate an Interface 472

**ST1** Constants in Interfaces 473

**J81** Static Methods in Interfaces 473

**J82** Default Methods 473

**J83** Conflicting Default Methods 474

### 10.2 WORKING WITH INTERFACE TYPES 475

**WE1** Investigating Number Sequences 

### 10.3 THE COMPARABLE INTERFACE 477

**PT1** Comparing Integers and Floating-Point Numbers 478

**ST2** The clone Method and the Cloneable Interface 479

### 10.4 USING INTERFACES FOR CALLBACKS 482

**J84** Lambda Expressions 485

**ST3** Generic Interface Types 486



© supermimicry/iStockphoto.

### 10.5 INNER CLASSES 487

**ST4** Anonymous Classes 488

### 10.6 MOCK OBJECTS 489

### 10.7 EVENT HANDLING 490

**CE3** Modifying Parameter Types in the Implementing Method 495

**CE4** Trying to Call Listener Methods 495

**J85** Lambda Expressions for Event Handling 496

### 10.8 BUILDING APPLICATIONS WITH BUTTONS 496

**CE5** Forgetting to Attach a Listener 498

**PT2** Don't Use a Container as a Listener 499

### 10.9 PROCESSING TIMER EVENTS 499

**CE6** Forgetting to Repaint 502

### 10.10 MOUSE EVENTS 502

**ST5** Keyboard Events 506

**ST6** Event Adapters 506

**C&S** Open Source and Free Software 507



© supermimicry/iStockphoto.

A mixer rotates any tools that will attach to its motor's shaft. In other words, a single motor can be used with multiple tools. We want to be able to *reuse* software components in the same way. In this chapter, you will learn an important strategy for separating the reusable part of a computation from the parts that vary in each reuse scenario. The reusable part invokes methods of an *interface*, not caring how the methods are implemented—just as the mixer doesn't care about the shape of the attachment. In a program, the reusable code is combined with a class that implements the interface methods. To produce a different application, you plug in another class that implements the same interface.

## 10.1 Using Interfaces for Algorithm Reuse

When you provide a service, you want to make it available to the largest possible set of clients. A restaurant serves people, and in Java, one might model this with a method

```
public void serve(Person client)
```

But what if the restaurant is willing to serve other creatures too? Then it makes sense to define a new type with exactly the methods that need to be invoked as the service processes an object. Such a type is called an *interface type*.

For example, a *Customer* interface type might have methods `eat` and `pay`. We can then redeclare the service as

```
public void serve(Customer client)
```

If the `Person` and `Cat` classes conform to the interface, then you can pass objects of those classes to the `serve` method.

As a more practical example, you will study the `Comparable` interface type of the Java library. It has a method `compareTo` that determines which of two objects should come first in sorted order. It is then possible to implement a sorting service that accepts collections of many different classes. All that matters is that the classes conform to the `Comparable` interface. The sorting service doesn't care about anything other than the `compareTo` method, which it uses to arrange the objects in order.

In the following sections, you will learn how to discover when an interface type is useful, which methods it should require, how to define the interface, and how to define classes that conform to it.



© Oxana Oleynichenko/iStockphoto.

*This restaurant is willing to serve anyone who conforms to the Customer interface with eat and pay methods.*

### 10.1.1 Discovering an Interface Type

In this section, we will look at a service that computes averages, and we want to make it as general as possible. Let's start with one implementation of the service that computes the average balance of an array of bank accounts:

```
public static double average(BankAccount[] objects)
{
    double sum = 0;
    for (BankAccount obj : objects)
    {
        sum = sum + obj.getBalance();
    }
    if (objects.length > 0) { return sum / objects.length; }
    else { return 0; }
}
```

Now suppose we want to compute an average of other objects. We have to write that method again. Here it is for Country objects:

```
public static double average(Country[] objects)
{
    double sum = 0;
    for (Country obj : objects)
    {
        sum = sum + obj.getArea();
    }
    if (objects.length > 0) { return sum / objects.length; }
    else { return 0; }
}
```

Clearly, the algorithm for computing the average is the same in all cases, but the details of measurement differ. We would like to provide a *single* method that provides this service.

But there is a problem. Each class has a different name for the method that returns the value that is being averaged. In the BankAccount class, we call `getBalance`. In the Country class, we call `getArea`.

Suppose that the various classes agree on a method `getMeasure` that obtains the measure to be used in the data analysis. For bank accounts, `getMeasure` returns the balance. For countries, `getMeasure` returns the area, and so on.

Then we can implement a single method that computes

```
sum = sum + obj.getMeasure();
```

But agreeing on the name of the method is only half the solution. In Java, we also must declare the type of the variable `obj`. Of course, you can't write

```
BankAccount or Country or . . . obj; // No
```

We need to invent a new type that describes any class whose objects can be measured. You will see how to do that in the next section.

## 10.1.2 Declaring an Interface Type

A Java interface type declares the methods that can be applied to a variable of that type.

In Java, an **interface type** is used to specify required operations. The declaration is similar to the declaration of a class. You list the methods that the interface requires. However, you need not supply an implementation for the methods. For example, here is the declaration of an interface type that we call `Measurable`:

```
public interface Measurable
{
    double getMeasure();
}
```

The `Measurable` interface type requires a single method, `getMeasure`. In general, an interface type can require multiple methods.

## Syntax 10.1 Declaring an Interface

```
Syntax public interface InterfaceName
{
    method headers
}
```

The methods of an interface  
are automatically public.

```
public interface Measurable
{
    double getMeasure();
}
```

No implementation is provided.

An interface type is similar to a class, but there are several important differences:

- An interface type does not have instance variables.
- Methods in an interface must be *abstract* (that is, without an implementation) or as of Java 8, *static*, or *default* methods (see Java 8 Note 10.1 and Java 8 Note 10.2).
- All methods in an interface type are automatically public.
- An interface type has no constructor. Interfaces are not classes, and you cannot construct objects of an interface type.

Now that we have a type that denotes measurability, we can implement a reusable average method:

```
public static double average(Measurable[] objects)
{
    double sum = 0;
    for (Measurable obj : objects)
    {
        sum = sum + obj.getMeasure();
    }
    if (objects.length > 0) { return sum / objects.length; }
    else { return 0; }
}
```

This method is useful for objects of any class that conforms to the `Measurable` type. In the next section, you will see what a class must do to make its objects measurable.

Note that the `Measurable` interface is not a type in the standard library—it was created specifically for this book, to provide a very simple example for studying the interface concept.



*This standmixer provides the “rotation” service to any attachment that conforms to a common interface. Similarly, the average method at the end of this section works with any class that implements a common interface.*

### 10.1.3 Implementing an Interface Type

Use the `implements` reserved word to indicate that a class implements an interface type.

The average method of the preceding section can process objects of any class that implements the `Measurable` interface. A class **implements an interface** type if it declares the interface in an `implements` clause, like this:

```
public class BankAccount implements Measurable
```

The class should then implement the abstract method or methods that the interface requires:

```
public class BankAccount implements Measurable
{
    ...
    public double getMeasure()
    {
        return balance;
    }
}
```

Note that the class must declare the method as `public`, whereas the interface need not—all methods in an interface are `public`.

Once the `BankAccount` class implements the `Measurable` interface type, `BankAccount` objects are instances of the `Measurable` type:

```
Measurable obj = new BankAccount(); // OK
```

A variable of type `Measurable` holds a reference to an object of some class that implements the `Measurable` interface.

Similarly, it is an easy matter to modify the `Country` class to implement the `Measurable` interface:

```
public class Country implements Measurable
{
    ...
    public double getMeasure()
    {
        return area;
    }
}
```

### Syntax 10.2 Implementing an Interface

**Syntax**

```
public class ClassName implements InterfaceName, InterfaceName, . . .  
{  
    instance variables  
    methods  
}
```

```
public class BankAccount implements Measurable  
{  
    ...
    public double getMeasure()  
    {  
        return balance;  
    }
    ...
}
```

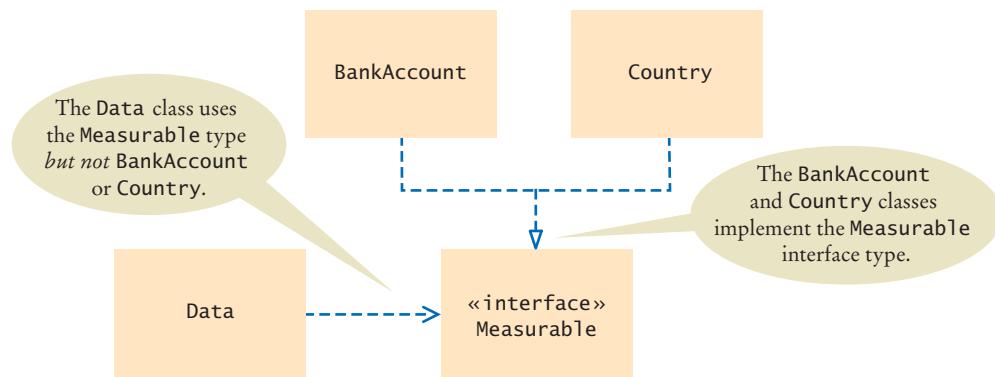
List all interface types that this class implements.

BankAccount instance variables

Other BankAccount methods

This method provides the implementation for the abstract method declared in the interface.

**Figure 1**  
UML Diagram of the Data Class and the Classes that Implement the Measurable Interface



Use interface types to make code more reusable.

The program at the end of this section uses a single average method (placed in class Data) to compute the average of bank accounts and the average of countries.

This is a typical usage for interface types. By inventing the Measurable interface type, we have made the average method reusable.

Figure 1 shows the relationships between the Data class, the Measurable interface, and the classes that implement the interface. Note that the Data class depends only on the Measurable interface. It is decoupled from the BankAccount and Country classes.

In the UML notation, interfaces are tagged with an indicator «interface». A dotted arrow with a triangular tip (--->) denotes the *implements* relationship between a class and an interface. You have to look carefully at the arrow tips—a dotted line with an open arrow tip (--->) denotes the *uses* relationship or dependency.

### section\_1/Data.java

```

1  public class Data
2  {
3      /**
4      * Computes the average of the measures of the given objects.
5      * @param objects an array of Measurable objects
6      * @return the average of the measures
7      */
8      public static double average(Measurable[] objects)
9      {
10         double sum = 0;
11         for (Measurable obj : objects)
12         {
13             sum = sum + obj.getMeasure();
14         }
15         if (objects.length > 0) { return sum / objects.length; }
16         else { return 0; }
17     }
18 }

```

### section\_1/MeasurableTester.java

```

1  /**
2   * This program demonstrates the measurable BankAccount and Country classes.
3   */
4  public class MeasurableTester
5  {
6      public static void main(String[] args)
7      {

```

```

8   // Calling the average method with an array of BankAccount objects
9   Measurable[] accounts = new Measurable[3];
10  accounts[0] = new BankAccount(0);
11  accounts[1] = new BankAccount(10000);
12  accounts[2] = new BankAccount(2000);
13
14  double averageBalance = Data.average(accounts);
15  System.out.println("Average balance: " + averageBalance);
16  System.out.println("Expected: 4000");
17
18  // Calling the average method with an array of Country objects
19  Measurable[] countries = new Measurable[3];
20  countries[0] = new Country("Uruguay", 176220);
21  countries[1] = new Country("Thailand", 513120);
22  countries[2] = new Country("Belgium", 30510);
23
24  double averageArea = Data.average(countries);
25  System.out.println("Average area: " + averageArea);
26  System.out.println("Expected: 239950");
27 }
28 }
```

### Program Run

```

Average balance: 4000
Expected: 4000
Average area: 239950
Expected: 239950
```

#### 10.1.4 Comparing Interfaces and Inheritance

In Chapter 9, you saw how to use inheritance to model hierarchies of related classes, such as different kinds of quiz questions. Multiple-choice questions and fill-in questions are questions with specific characteristics.

Interfaces model a somewhat different relationship. Consider for example the `BankAccount` and `Country` classes in the preceding section. Both implement the `Measurable` interface type, but otherwise they have nothing in common. Being measurable is just one aspect of what it means to be a bank account or country. It is useful to model this common aspect, because it enables other programmers to write tools that exploit the commonality, such as the method for computing averages.

A class can implement more than one interface, for example

```
public class Country implements Measurable, Named
```

Here, `Named` is a different interface:

```
public interface Named
{
    String getName();
}
```

In contrast, a class can only extend (inherit from) a single superclass.

An interface describes the behavior that an implementation should supply. Prior to Java 8, an interface could not provide any implementation. Now, it is possible to supply a *default* implementation, and the distinction between interfaces and abstract classes (see Special Topic 9.3) has become more subtle. The significant difference that remains is that an interface type has *no state* (that is, no instance variables).

Generally, you will develop interfaces when you have code that processes objects of different classes in a common way. For example, a drawing program might have different objects that can be drawn, such as lines, images, text, and so on. In this situation, a `Drawable` interface with a `draw` method will be useful. Another example is a traffic simulation that models the movement of people, cars, dogs, balls, and so on. In this example, you might create a `Moveable` interface with methods `move` and `getPosition`.

**SELF CHECK**

1. Suppose you want to use the `average` method to find the average salary of an array of `Employee` objects. What condition must the `Employee` class fulfill?
2. Why can't the `average` method have a parameter variable of type `Object[]`?
3. Why can't you use the `average` method to find the average length of `String` objects?
4. What is wrong with this code?

```
Measurable meas = new Measurable();
System.out.println(meas.getMeasure());
```

5. What is wrong with this code?
- ```
Measurable meas = new Country("Uruguay", 176220);
System.out.println(meas.getName());
```

**Practice It** Now you can try these exercises at the end of the chapter: E10.1, E10.2, E10.3.

**Common Error 10.1****Forgetting to Declare Implementing Methods as Public**

The methods in an interface are not declared as `public`, because they are public by default. However, the methods in a class are not public by default—their default access level is “package” access, which we discussed in Chapter 8. It is a common error to forget the `public` reserved word when declaring a method from an interface:

```
public class BankAccount implements Measurable
{
    .
    .
    double getMeasure() // Oops—should be public
    {
        return balance;
    }
}
```

Then the compiler complains that the method has a weaker access level, namely package access instead of public access. The remedy is to declare the method as `public`.

**Common Error 10.2****Trying to Instantiate an Interface**

You can declare variables whose type is an interface, for example:

```
Measurable meas;
```

However, you can *never* construct an object of an interface type:

```
Measurable meas = new Measurable(); // Error
```

Interfaces aren't classes. There are no objects whose types are interfaces. If a variable of an interface type refers to an object, then the object must belong to some class—a class that implements the interface:

```
Measurable meas = new BankAccount(); // OK
```

**Special Topic 10.1****Constants in Interfaces**

Interfaces cannot have instance variables, but it is legal to specify **constants**. When declaring a constant in an interface, you can (and should) omit the reserved words `public static final`, because all variables in an interface are automatically `public static final`. For example,

```
public interface Named
{
    String NO_NAME = "(NONE)";
    ...
}
```

Now the constant `Named.NO_NAME` can be used to denote the absence of a name.

It is not very common to have constants in interface types. In particular, you should avoid multiple related constants (such as `int NORTH = 1`, `int NORTHEAST = 2`, and so on). In that case, use an enumerated type instead (see Special Topic 5.4).

**Java 8 Note 10.1****FULL CODE EXAMPLE**

Go to [wiley.com/go/bje06code](http://wiley.com/go/bje06code) to see an example of a static method in an interface.

**Static Methods in Interfaces**

Before Java 8, all methods in an interface had to be abstract. Java 8 allows static methods in interfaces that work exactly like static methods in classes. A static method of an interface does not operate on objects, and its purpose should relate to the interface that contains it.

For example, it would be perfectly sensible to place the average method from Section 10.1 into the `Measurable` interface:

```
public interface Measurable
{
    double getMeasure(); // An abstract method
    static double average(Measurable[] objects) // A static method
    {
        . . . // Same implementation as in Section 10.1
    }
}
```

To call this method, provide the name of the interface and the method name:

```
double meanArea = Measurable.average(countries);
```

**Java 8 Note 10.2****Default Methods**

A **default method** is a non-static method in an interface that has an implementation. A class that implements the method either inherits the default behavior or overrides it. By providing default methods in an interface, it is less work to define a class that implements an interface.

For example, the `Measurable` interface can declare `getMeasure` as a default method:

```
public interface Measurable
{
    default double getMeasure() { return 0; }
}
```

If a class implements the interface and doesn't provide a `getMeasure` method, then it inherits this default method.

This particular example isn't all that useful. One doesn't normally want each object to have measure zero. Here is a more interesting example, in which a default method calls another interface method:

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download an example of a default method in an interface.

```
public interface Measurable
{
    double getMeasure(); // An abstract method
    default boolean smallerThan(Measurable other)
    {
        return getMeasure() < other.getMeasure();
    }
}
```

The `smallerThan` method tests whether an object has a smaller measure than another, which is useful for arranging objects by increasing measure.

A class that implements the `Measurable` interface only needs to implement `getMeasure`, and it automatically inherits the `smallerThan` method. This can be a very useful mechanism. For example, the `Comparator` interface that is described in Special Topic 14.5 has one abstract method but more than a dozen default methods.

**Java 8 Note 10.3****8****FULL CODE EXAMPLE**

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download a program that demonstrates conflicting methods.

**Conflicting Default Methods**

It is possible (although quite rare) for a class to inherit conflicting default methods from two interfaces, or to inherit a default method that conflicts with one of its methods. There are two rules that deal with this possibility:

1. *Classes win.* When a class extends another class and implements an interface, both of which define the same method, the subclass inherits the superclass method and ignores the default method.
2. *Interfaces clash.* When a class implements two interfaces with the same default method, then the class must override the method and implement it.

To understand these rules, consider this example:

```
public class Person
{
    public String name() { return firstName() + " " + lastName(); }
    . .
}

public interface Named
{
    default String name() { return "(NONE)"; }
}

public class User extends Person implements Named
{
    // Inherits Person.name()
    . .
}
```

In this situation, the method defined in the superclass wins over the method from the interface. However, the situation is different if `Person` is an interface:

```
public interface Person
{
    default String name() { return firstName() + " " + lastName(); }
    . .
}
```

Suppose a class implements both interfaces:

```
public class User implements Person, Named { . . . }
```

This class *must* override the `name` method, in a way that is appropriate for the context. The details depend on why the designer of the `User` class chose to implement the `Named` interface. For example, suppose there is a method that checks a `Named[]` array for duplicates, and the program wants to call it to ensure that account names are unique. In that case, `User.name` should return the account name.

## 10.2 Working with Interface Types

In the preceding section, you saw how to implement a simple service that accepted an interface. As you saw, you were able to pass objects of different classes to the service, and the service was able to invoke a method of the interface. In the following sections, you will learn the rules for working with interface types in Java.

### 10.2.1 Converting from Classes to Interfaces

Have a close look at the call

```
double averageBalance = Data.average(accounts);
```

from the program of the preceding section. Here, `accounts` is an array of `BankAccount` objects. However, the `average` method expects an array whose element type is `Measurable`:

```
public double average(Measurable[] objects)
```

It is legal to convert from the `BankAccount` type to the `Measurable` type. In general, you can convert from a class type to the type of any interface that the class implements. For example,

```
BankAccount account = new BankAccount(1000);
Measurable meas = account; // OK
```

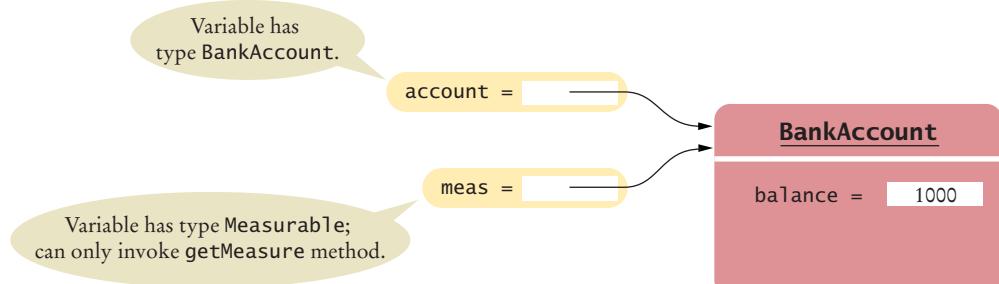
Alternatively, a `Measurable` variable can refer to an object of the `Country` class of the preceding section because that class also implements the `Measurable` interface.

```
Country uruguay = new Country("Uruguay", 176220);
Measurable meas = uruguay; // Also OK
```

However, the `Rectangle` class from the standard library doesn't implement the `Measurable` interface. Therefore, the following assignment is an error:

```
Measurable meas = new Rectangle(5, 10, 20, 30); // Error
```

You can convert from a class type to an interface type, provided the class implements the interface.



**Figure 2**  
Variables of Class and Interface Types

## 10.2.2 Invoking Methods on Interface Variables

Now suppose that the variable `meas` has been initialized with a reference to an object of some class that implements the `Measurable` interface. You don't know to which class that object belongs. But you do know that the class implements the methods of the interface type, and you can invoke them:

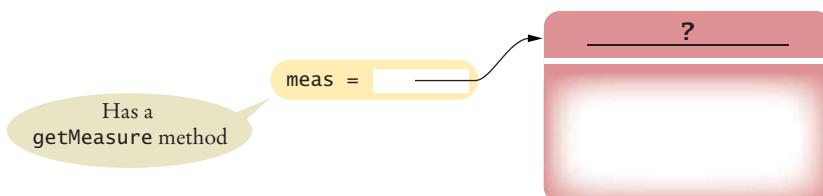
```
double result = meas.getMeasure();
```

Now let's think through the call to the `getMeasure` method more carefully. Which `getMeasure` method is called? The `BankAccount` and `Country` classes provide two different implementations of that method. How does the correct method get called if the caller doesn't even know the exact class to which `meas` belongs?

This is again polymorphism in action. (See Section 9.4 for a discussion of polymorphism.) The Java virtual machine locates the correct method by first looking at the class of the actual object, and then calling the method with the given name in that class. That is, if `meas` refers to a `BankAccount` object, then the `BankAccount.getMeasure` method is called. If `meas` refers to a `Country` object, then the `Country.getMeasure` method is called.

Method calls on an interface reference are polymorphic. The appropriate method is determined at run time.

**Figure 3**  
An Interface Reference  
Can Refer to an Object  
of Any Class that  
Implements the Interface



## 10.2.3 Casting from Interfaces to Classes

Occasionally, it happens that you store an object in an interface reference and you need to convert its type back. Consider this method that returns the object with the larger measure:

```
public static Measurable larger(Measurable obj1, Measurable obj2)
{
    if (obj1.getMeasure() > obj2.getMeasure())
    {
        return obj1;
    }
    else
    {
        return obj2;
    }
}
```

The `larger` method returns the object with the larger measure, *as a Measurable reference*. It has no choice—it does not know the exact type of the object. Let's use the method:

```
Country uruguay = new Country("Uruguay", 176220);
Country thailand = new Country("Thailand", 513120);
Measurable max = larger(uruguay, thailand);
```

Now what can you do with the `max` reference? You know it refers to a `Country` object, but the compiler doesn't. For example, you cannot call the `getName` method:

```
String countryName = max.getName(); // Error
```



If a Person object is actually a Superhero, you need a cast before you can apply any Superhero methods.

You need a cast to convert from an interface type to a class type.

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bje06code](http://wiley.com/go/bje06code) to download a demonstration of conversions between class and interface types.

**SELF CHECK**

That call is an error, because the `Measurable` type has no `getName` method.

However, as long as you are absolutely sure that `max` refers to a `Country` object, you can use the `cast` notation to convert its type back:

```
Country maxCountry = (Country) max;
String name = maxCountry.getName();
```

If you are wrong, and the object doesn't actually refer to a country, a run-time exception will occur.

6. Can you use a cast (`BankAccount`) `meas` to convert a `Measurable` variable `meas` to a `BankAccount` reference?
7. If both `BankAccount` and `Country` implement the `Measurable` interface, can a `Country` reference be converted to a `BankAccount` reference?
8. Why is it impossible to construct a `Measurable` object?
9. Why can you nevertheless declare a variable whose type is `Measurable`?
10. What does this code fragment print? Why is this an example of polymorphism?

```
Measurable[] data = { new BankAccount(10000), new Country("Belgium", 30510) };
System.out.println(average(data));
```

**Practice It** Now you can try these exercises at the end of the chapter: R10.3, R10.4, R10.5.

**WORKED EXAMPLE 10.1****Investigating Number Sequences**

Learn how to use a `Sequence` interface to investigate properties of arbitrary number sequences. Go to [wiley.com/go/bje06examples](http://wiley.com/go/bje06examples) and download Worked Example 10.1.



© Norekbo/  
iStockphoto

## 10.3 The Comparable Interface

Implement the `Comparable` interface so that objects of your class can be compared, for example, in a `sort` method.

In the preceding sections, we defined the `Measurable` interface and provided an `average` method that works with any classes implementing that interface. In this section, you will learn about the `Comparable` interface of the standard Java library.

The `Measurable` interface is used for measuring a single object. The `Comparable` interface is more complex because comparisons involve two objects. The interface declares a `compareTo` method. The call

```
a.compareTo(b)
```

must return a negative number if `a` should come before `b`, zero if `a` and `b` are the same, and a positive number if `b` should come before `a`.



*The `compareTo` method checks whether another object is larger or smaller.*

© Janis Dreosti/Stockphoto

The Comparable interface has a single method:

```
public interface Comparable
{
    int compareTo(Object otherObject);
}
```

For example, the BankAccount class can implement Comparable like this:

```
public class BankAccount implements Comparable
{
    ...
    public int compareTo(Object otherObject)
    {
        BankAccount other = (BankAccount) otherObject;
        if (balance < other.balance) { return -1; }
        if (balance > other.balance) { return 1; }
        return 0;
    }
    ...
}
```

This compareTo method compares bank accounts by their balance. Note that the compareTo method has a parameter variable of type Object. To turn it into a BankAccount reference, we use a cast:

```
BankAccount other = (BankAccount) otherObject;
```

Once the BankAccount class implements the Comparable interface, you can sort an array of bank accounts with the Arrays.sort method:

```
BankAccount[] accounts = new BankAccount[3];
accounts[0] = new BankAccount(10000);
accounts[1] = new BankAccount(0);
accounts[2] = new BankAccount(2000);
Arrays.sort(accounts);
```

The accounts array is now sorted by increasing balance.

### SELF CHECK



11. How can you sort an array of Country objects by increasing area?
12. Can you use the Arrays.sort method to sort an array of String objects? Check the API documentation for the String class.
13. Can you use the Arrays.sort method to sort an array of Rectangle objects? Check the API documentation for the Rectangle class.
14. Write a method max that finds the larger of any two Comparable objects.
15. Write a call to the method of Self Check 14 that computes the larger of two bank accounts, then prints its balance.

### Practice It

Now you can try these exercises at the end of the chapter: E10.10, E10.30.

### Programming Tip 10.1



### Comparing Integers and Floating-Point Numbers

When you implement a comparison method, you need to return a negative integer to indicate that the first object should come before the other, zero if they are equal, or a positive integer otherwise. You have seen how to implement this decision with three branches. When you compare *nonnegative* integers, there is a simpler way: subtract the integers:

```
public class Person implements Comparable
{
```

```

private int id; // Must be ≥ 0
...
public int compareTo(Object otherObject)
{
    Person other = (Person) otherObject;
    return id - other.id;
}
}

```

The difference is negative if `id < other.id`, zero if the values are the same, and positive otherwise.

This trick doesn't work if the integers can be negative because the difference can overflow (see Exercise R10.1). However, the `Integer.compare` method always works:

```
return Integer.compare(id, other.id); // Safe for negative integers
```

You cannot compare floating-point values by subtraction (see Exercise R10.2). Instead, use the `Double.compare` method:

```

public class BankAccount implements Comparable
{
    ...
    public int compareTo(Object otherObject)
    {
        BankAccount other = (BankAccount) otherObject;
        return Double.compare(balance, other.balance);
    }
}

```

## Special Topic 10.2



### The `clone` Method and the `Cloneable` Interface

You know that copying an object reference simply gives you two references to the same object:

```

BankAccount account = new BankAccount(1000);
BankAccount account2 = account;
account2.deposit(500);
// Now both account and account2 refer to a bank account with a balance of 1500

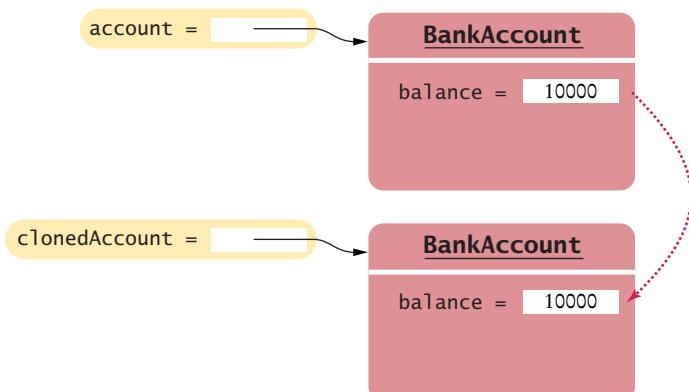
```

What can you do if you actually want to make a copy of an object? That is the purpose of the `clone` method. The `clone` method must return a *new* object that has an identical state to the existing object (see Figure 4).

Here is how to call it:

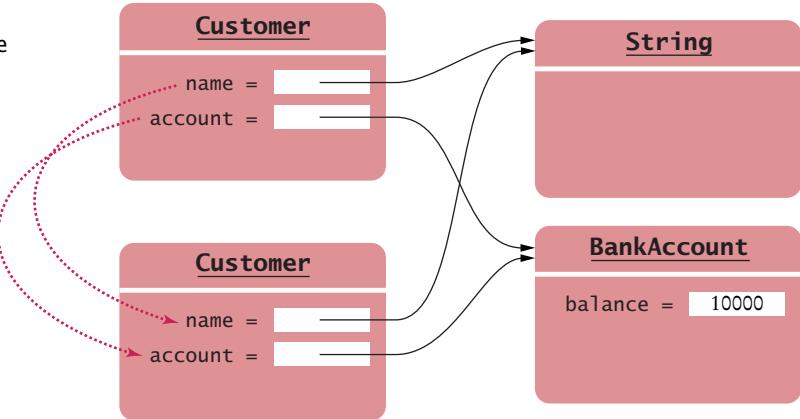
```
BankAccount clonedAccount = (BankAccount) account.clone();
```

The return type of the `clone` method is the class `Object`. When you call the method, you must use a cast to inform the compiler that `account.clone()` really returns a `BankAccount` object.



**Figure 4**  
Cloning Objects

**Figure 5**  
The `Object.clone` Method Makes a Shallow Copy

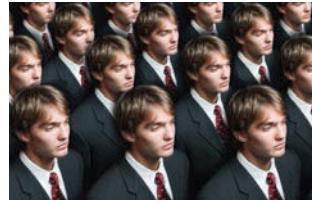


The `Object.clone` method is the starting point for the `clone` methods in your own classes. It creates a new object of the same type as the original object. It also automatically copies the instance variables from the original object to the cloned object. Here is a first attempt to implement the `clone` method for the `BankAccount` class:

```

public class BankAccount
{
    ...
    public Object clone()
    {
        // Not complete
        Object clonedAccount = super.clone();
        return clonedAccount;
    }
}

```



*The `clone` method makes an identical copy of an object.*

© Alex Gumerov/iStockphoto.

However, this `Object.clone` method must be used with care. It only shifts the problem of cloning by one level; it does not completely solve it. Specifically, if an object contains a reference to another object, then the `Object.clone` method makes a copy of that object reference, not a clone of that object. Figure 5 shows how the `Object.clone` method works with a `Customer` object that has references to a `String` object and a `BankAccount` object. As you can see, the `Object.clone` method copies the references to the cloned `Customer` object and does not clone the objects to which they refer. Such a copy is called a **shallow copy**.

There is a reason why the `Object.clone` method does not systematically clone all sub-objects. In some situations, it is unnecessary. For example, if an object contains a reference to a string, there is no harm in copying the string reference, because Java string objects can never change their contents. The `Object.clone` method does the right thing if an object contains only numbers, Boolean values, and strings. But it must be used with caution when an object contains references to mutable objects.

For that reason, there are two safeguards built into the `Object.clone` method to ensure that it is not used accidentally. First, the method is declared `protected` (see Special Topic 9.5). This prevents you from accidentally calling `x.clone()` if the class to which `x` belongs hasn't declared `clone` to be `public`.

As a second precaution, `Object.clone` checks that the object being cloned implements the `Cloneable` interface. If not, it throws an exception. The `Object.clone` method looks like this:

```

public class Object
{
    protected Object clone() throws CloneNotSupportedException
    {
        if (this instanceof Cloneable)
        {

```

```
// Copy the instance variables  
    . . .  
}  
else  
{  
    throw new CloneNotSupportedException();  
}  
}  
}
```

Unfortunately, all that safeguarding means that the legitimate callers of `Object.clone()` pay a price—they must catch that exception (see Chapter 11) *even if their class implements Cloneable*.

```
public class BankAccount implements Cloneable
{
    . . .
    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch (CloneNotSupportedException e)
        {
            // Can't happen because we implement Cloneable but we still must catch it.
            return null;
        }
    }
}
```

If an object contains a reference to another mutable object, then you must call `clone` for that reference. For example, suppose the `Customer` class has an instance variable of class `BankAccount`. You can implement `Customer.clone` as follows:

```
public class Customer implements Cloneable
{
    private String name;
    private BankAccount account;
    ...
    public Object clone()
    {
        try
        {
            Customer cloned = (Customer) super.clone();
            cloned.account = (BankAccount) account.clone();
            return cloned;
        }
        catch(CloneNotSupportedException e)
        {
            // Can't happen because we implement Cloneable
            return null;
        }
    }
}
```

In general, implementing the `clone` method requires these steps:

- Make the class implement the `Cloneable` interface type.
  - In the `clone` method, call `super.clone()`. Catch the `CloneNotSupportedException` if the superclass is `Object`.
  - Clone any mutable instance variables.

## 10.4 Using Interfaces for Callbacks

© Dan Herrick/Stockphoto.



*A callback object waits to be called. The algorithm that has the callback object only calls it when it needs to have the information that the callback can provide.*

**A callback is a mechanism for specifying code that is executed at a later time.**

In this section, we introduce the notion of a **callback**, show how it leads to a more flexible average method, and study how a callback can be implemented in Java by using interface types.

To understand why a further improvement to the average method is desirable, consider these limitations of the `Measurable` interface:

- You can add the `Measurable` interface only to classes under your control. If you want to process a set of `Rectangle` objects, you cannot make the `Rectangle` class implement another interface—it is a library class, which you cannot change.
- You can measure an object in only one way. If you want to analyze a set of cars both by speed and price, you are stuck.

Therefore, let's rethink the average method. The method measures objects, requiring them to be of type `Measurable`. The responsibility for measuring lies with the objects themselves. That is the cause for the limitations.

It would be better if we could give the average method the data to be averaged, and separately a method for measuring the objects. When collecting rectangles, we might give it a method for computing the area of a rectangle. When collecting cars, we might give it a method for getting the car's price.

Such a method is called a **callback**. A callback is a mechanism for bundling up a block of code so that it can be invoked at a later time.

In some programming languages, it is possible to specify callbacks directly, as blocks of code or names of methods. But Java is an object-oriented programming language. Therefore, you turn callbacks into objects. This process starts by declaring an interface for the callback:

```
public interface Measurer
{
    double measure(Object anObject);
}
```

The `measure` method measures an object and returns its measurement. Here we use the fact that all objects can be converted to the type `Object`.

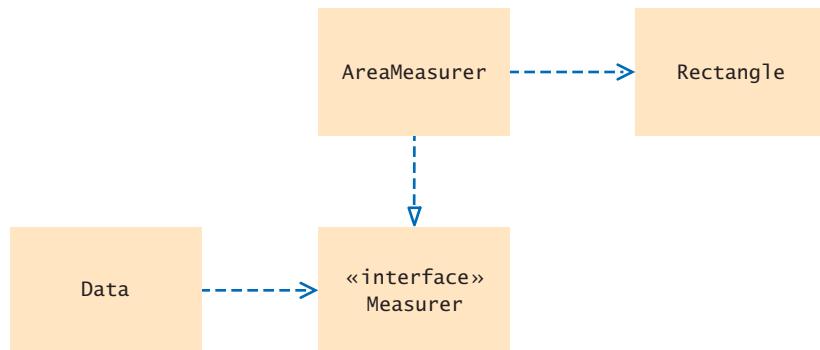
The code that makes the call to the callback receives an object of a class that implements this interface. In our case, the improved average method receives a `Measurer` object.

```
public static double average(Object[] objects, Measurer meas)
{
    double sum = 0;
    for (Object obj : objects)
    {
        sum = sum + meas.measure(obj);
    }
    if (objects.length > 0) { return sum / objects.length; }
    else { return 0; }
}
```

The `average` method simply makes a callback to the `measure` method whenever it needs to measure any object.

Finally, a specific callback is obtained by implementing the `Measurer` interface. For example, here is how you can measure rectangles by area. Provide a class

**Figure 6**  
UML Diagram of the Data Class and the Measurer Interface



```

public class AreaMeasurer implements Measurer
{
    public double measure(Object anObject)
    {
        Rectangle aRectangle = (Rectangle) anObject;
        double area = aRectangle.getWidth() * aRectangle.getHeight();
        return area;
    }
}
  
```

Note that the `measure` method has a parameter variable of type `Object`, even though this particular measurer just wants to measure rectangles. The method parameter types must match those of the `measure` method in the `Measurer` interface. Therefore, the `anObject` parameter variable is cast to the `Rectangle` type:

```
Rectangle aRectangle = (Rectangle) anObject;
```

What can you do with an `AreaMeasurer`? You need it to compute the average area of rectangles. Construct an object of the `AreaMeasurer` class and pass it to the `average` method:

```

Measurer areaMeas = new AreaMeasurer();
Rectangle[] rects
= { new Rectangle(5, 10, 20, 30), new Rectangle(10, 20, 30, 40) };
double averageArea = average(rects, areaMeas);
  
```

The `average` method will ask the `AreaMeasurer` object to measure the rectangles.

Figure 6 shows the UML diagram of the classes and interfaces of this solution. As in Figure 1, the `Data` class (which holds the `average` method) is decoupled from the class whose objects it processes (`Rectangle`). However, unlike in Figure 1, the `Rectangle` class is no longer coupled with another class. Instead, to process rectangles, you provide a small “helper” class `AreaMeasurer`. This helper class has only one purpose: to tell the `average` method how to measure its objects.

Here is the complete program:

#### section\_4/Measurer.java

```

1  /**
2   * Describes any class whose objects can measure other objects.
3   */
4  public interface Measurer
5  {
6      /**
7       * Computes the measure of an object.
  
```

```

8      @param anObject the object to be measured
9      @return the measure
10     */
11     double measure(Object anObject);
12 }

```

### section\_4/AreaMeasurer.java

```

1 import java.awt.Rectangle;
2 /**
3  * Objects of this class measure rectangles by area.
4  */
5 public class AreaMeasurer implements Measurer
6 {
7     public double measure(Object anObject)
8     {
9         Rectangle aRectangle = (Rectangle) anObject;
10        double area = aRectangle.getWidth() * aRectangle.getHeight();
11        return area;
12    }
13 }
14

```

### section\_4/Data.java

```

1 public class Data
2 {
3     /**
4      * Computes the average of the measures of the given objects.
5      * @param objects an array of objects
6      * @param meas the measurer for the objects
7      * @return the average of the measures
8      */
9     public static double average(Object[] objects, Measurer meas)
10    {
11        double sum = 0;
12        for (Object obj : objects)
13        {
14            sum = sum + meas.measure(obj);
15        }
16        if (objects.length > 0) { return sum / objects.length; }
17        else { return 0; }
18    }
19 }

```

### section\_4/MeasurerTester.java

```

1 import java.awt.Rectangle;
2 /**
3  * This program demonstrates the use of a Measurer.
4  */
5 public class MeasurerTester
6 {
7     public static void main(String[] args)
8     {
9         Measurer areaMeas = new AreaMeasurer();
10        Rectangle[] rects = new Rectangle[]
11        {
12
13

```

```

14     new Rectangle(5, 10, 20, 30),
15     new Rectangle(10, 20, 30, 40),
16     new Rectangle(20, 30, 5, 15)
17   };
18
19   double averageArea = Data.average(rects, areaMeas);
20   System.out.println("Average area: " + averageArea);
21   System.out.println("Expected: 625");
22 }

```

### Program Run

```
Average area: 625
Expected: 625
```

### SELF CHECK



16. Suppose you want to use the `average` method of Section 10.1 to find the average length of `String` objects. Why can't this work?
17. How can you use the `average` method of this section to find the average length of `String` objects?
18. Why does the `measure` method of the `Measurer` interface have one more argument than the `getMeasure` method of the `Measurable` interface?
19. Write a method `max` with three arguments that finds the larger of any two objects, using a `Measurer` to compare them.
20. Write a call to the method of Self Check 19 that computes the larger of two rectangles, then prints its width and height.

### Practice It

Now you can try these exercises at the end of the chapter: R10.9, E10.8, E10.9.

### Java 8 Note 10.4



### Lambda Expressions

In the preceding section, you saw how to use interfaces for specifying variations in behavior. The `average` method needs to measure each object, and it does so by calling the `measure` method of the supplied `Measurer` object.

Unfortunately, the caller of the `average` method has to do a fair amount of work; namely, to define a class that implements the `Measurer` interface and to construct an object of that class. Java 8 has a convenient shortcut for these steps, provided that the interface has a *single abstract method*. Such an interface is called a **functional interface** because its purpose is to define a single function. The `Measurer` interface is an example of a functional interface.

To specify that single function, you can use a **lambda expression**, an expression that defines the parameters and return value of a method in a compact notation. Here is an example:

```
(Object obj) -> ((BankAccount) obj).getBalance()
```

This expression defines a function that, given an object, casts it to a `BankAccount` and returns the balance.

(The term “lambda expression” comes from a mathematical notation that uses the Greek letter lambda ( $\lambda$ ) instead of the `->` symbol. In other programming languages, such an expression is called a *function expression*.)

A lambda expression cannot stand alone. It needs to be assigned to a variable whose type is a functional interface:

```
Measurer accountMeas = (Object obj) -> ((BankAccount) obj).getBalance();
```

Now the following actions occur:

1. A class is defined that implements the functional interface. The single abstract method is defined by the lambda expression.
2. An object of that class is constructed.
3. The variable is assigned a reference to that object.

You can also pass a lambda expression to a method. Then the parameter variable of the method is initialized with the constructed object. For example, consider the call

```
double averageBalance = average(accounts,
    (Object obj) -> ((BankAccount) obj).getBalance());
```

In the same way as before, an object is constructed that belongs to a class implementing `Measurer`. The object is used to initialize the parameter variable `meas` of the `average` method. Recall that the parameter variable has type `Measurer`:

```
public static double average(Object[] objects, Measurer meas)
{
    . .
    sum = sum + meas.measure(obj);
    . .
}
```

The `average` method calls the `measure` method on `meas`, which in turn executes the body of the lambda expression.

In its simplest form, a lambda expression contains a list of parameters and the expression that is being computed from the parameters. If more work needs to be done, you can write a method body in the usual way, enclosed in braces and with a `return` statement:

```
Measurer areaMeas = (Object obj) ->
{
    Rectangle r = (Rectangle) obj;
    return r.getWidth() * r.getHeight();
};
```

Conceptually, lambda expressions are easiest to understand as a convenient notation for callbacks. Consider any method that needs to call some code that varies from one call to the next. This can be achieved as follows:

1. The implementor of the method defines an interface that describes the purpose of the code to be executed. That interface has a single method.
2. The method receives a parameter of that interface, and calls the single method of the interface whenever the code that can vary needs to be called.
3. The caller of the method provides a lambda expression whose body is the code that should be called in this invocation.

You will see additional examples of using lambda expressions for event handlers (Section 10.5) and comparators (Section 14.8). Chapter 19 uses lambda expressions extensively for processing complex data. In that chapter, we will study lambda expressions in greater depth.

### Special Topic 10.3



### Generic Interface Types

In Section 10.3, you saw how to use the “raw” version of the `Comparable` interface type. In fact, the `Comparable` interface is a parameterized type, similar to the `ArrayList` type:

```
public interface Comparable<T>
{
    int compareTo(T other)
}
```

The type parameter specifies the type of the objects that this class is willing to accept for comparison. Usually, this type is the same as the class type itself. For example, the `BankAccount` class would implement `Comparable<BankAccount>`, like this:

```
public class BankAccount implements Comparable<BankAccount>
{
    . . .
    public int compareTo(BankAccount other)
    {
        return Double.compare(balance, other.balance);
    }
}
```

The type parameter has a significant advantage: You need not use a cast to convert an `Object` parameter variable into the desired type.

Similarly, the `Measurer` interface can be improved by making it into a generic type:

```
public interface Measurer<T>
{
    double measure(T anObject);
}
```

The type parameter specifies the type of the parameter of the `measure` method. Again, you avoid the cast from `Object` when implementing the interface:

```
public class AreaMeasurer implements Measurer<Rectangle>
{
    public double measure(Rectangle anObject)
    {
        double area = anObject.getWidth() * anObject.getHeight();
        return area;
    }
}
```

See Chapter 18 for an in-depth discussion of implementing and using generic classes.

## 10.5 Inner Classes

© angelhell/iStockphoto.



An inner class is a class that is declared inside another class.

The `AreaMeasurer` class of the preceding section is a very trivial class. We need this class only because the `average` method needs an object of some class that implements the `Measurer` interface. When you have a class that serves a very limited purpose, such as this one, you can declare the class inside the method that needs it:

```
public class MeasurerTester
{
    public static void main(String[] args)
    {
        class AreaMeasurer implements Measurer
        {
            . . .
        }

        Measurer areaMeas = new AreaMeasurer();
        double averageArea = Data.average(rects, areaMeas);

        . . .
    }
}
```

An inner class is declared inside another class.

Inner classes are commonly used for utility classes that should not be visible elsewhere in a program.

A class that is declared inside another class, such as the `AreaMeasurer` class in this example, is called an **inner class**. This arrangement signals to the reader of your program that the `AreaMeasurer` class is not interesting beyond the scope of this method. Because an inner class inside a method is not a publicly accessible feature, you don't need to document it as thoroughly.

You can also declare an inner class inside an enclosing class, but outside of its methods. Then the inner class is available to all methods of the enclosing class.

```
public class MeasurerTester
{
    class AreaMeasurer implements Measurer
    {
        . .
    }

    public static void main(String[] args)
    {
        .
        Measurer areaMeas = new AreaMeasurer();
        double averageArea = Data.average(rects, areaMeas);
        .
    }
}
```

When you compile the source files for a program that uses inner classes, have a look at the class files in your program directory—you will find that the inner classes are stored in files with curious names, such as `MeasurerTester$1AreaMeasurer.class`. The exact names aren't important. The point is that the compiler turns an inner class into a regular class file.

### SELF CHECK



21. Why would you use an inner class instead of a regular class?
22. When would you place an inner class inside a class but outside any methods?
23. How many class files are produced when you compile the `MeasurerTester` program from this section?

**Practice It** Now you can try these exercises at the end of the chapter: E10.11, E10.13.

### Special Topic 10.4



### Anonymous Classes

An entity is *anonymous* if it does not have a name. In a program, something that is only used once doesn't usually need a name. For example, you can replace

```
Country belgium = new Country("Belgium", 30510);
countries.add(belgium);
```

with

```
countries.add(new Country("Belgium", 30510));
```

if the country is not used elsewhere in the same method. The object `new Country("Belgium", 30510)` is an **anonymous object**. Programmers like anonymous objects, because they don't have to go through the trouble of coming up with a name. If you have struggled with the decision whether to call a coin `c`, `dime`, or `aCoin`, you'll understand this sentiment.

Inner classes often give rise to a similar situation. After a single object of the `AreaMeasurer` has been constructed, the class is never used again. In Java, it is possible to declare an **anonymous class** if all you ever need is a single object of the class.

```

public static void main(String[] args)
{
    // Construct an object of an anonymous class
    Measurer m = new Measurer()
        // Class declaration starts here
    {
        public double measure(Object anObject)
        {
            Rectangle aRectangle = (Rectangle) anObject;
            return aRectangle.getWidth() * aRectangle.getHeight();
        }
    };
}

double result = Data.average(rectangles, m);
.
.
}

```

This means: Construct an object of a class that implements the `Measurer` interface by declaring the `measure` method as specified. This style was popular before lambda expressions were added to Java 8. Nowadays, it is simpler to use a lambda expression. We do not use anonymous classes in this book.

## 10.6 Mock Objects

A mock object provides the same services as another object, but in a simplified manner.

When you work on a program that consists of multiple classes, you often want to test some of the classes before the entire program has been completed. A very effective technique for this purpose is the use of mock objects. A **mock object** provides the same services as another object, but in a simplified manner.

Consider a grade book application that manages quiz scores for students. This calls for a class `GradeBook` with methods such as

```

public void addScore(int studentId, double score)
public double getAverageScore(int studentId)
public void save(String filename)

```

Now consider the class `GradingProgram` that manipulates a `GradeBook` object. That class calls the methods of the `GradeBook` class. We would like to test the `GradingProgram` class without having a fully functional `GradeBook` class.

To make this work, declare an interface type with the same methods that the `GradeBook` class provides. A common convention is to use the letter `I` as the prefix for such an interface:

```

public interface IGradeBook
{
    void addScore(int studentId, double score);
    double getAverageScore(int studentId);
    void save(String filename);
    .
}

```

The `GradingProgram` class should *only* use this interface, never the `GradeBook` class. Of course, the `GradeBook` class will implement this interface, but as already mentioned, it may not be ready for some time.

In the meantime, provide a mock implementation that makes some simplifying assumptions. Saving is not actually necessary for testing the user interface. We can also temporarily restrict it to the case of a single student.

© Don Nichols/iStockphoto.



*If you just want to practice arranging the Christmas decorations, you don't need a real tree. Similarly, you can use mock objects to test parts of your computer program.*

Both the mock class and the actual class implement the same interface.

```
public class MockGradeBook implements IGradeBook
{
    private ArrayList<Double> scores;

    public MockGradeBook() { scores = new ArrayList<Double>(); }

    public void addScore(int studentId, double score)
    {
        // Ignore studentId
        scores.add(score);
    }
    public double getAverageScore(int studentId)
    {
        double total = 0;
        for (double x : scores) { total = total + x; }
        return total / scores.size();
    }
    public void save(String filename)
    {
        // Do nothing
    }
    .
}
```

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download a program that demonstrates the use of mock objects for testing.


**SELF CHECK**


24. Why is it necessary that the real class and the mock class implement the same interface type?
25. Why is the technique of mock objects particularly effective when the GradeBook and GradingProgram class are developed by two programmers?

**Practice It** Now you can try these exercises at the end of the chapter: P10.19, P10.20.

## 10.7 Event Handling

This and the following sections continue the book's graphics track. You will learn how interfaces are used when programming graphical user interfaces.

In the applications that you have written so far, user input was under control of the *program*. The program asked the user for input in a specific order. For example, a program might ask the user to supply first a name, then a dollar amount. But the programs that you use every day on your computer don't work like that. In a program with a graphical user interface, the *user* is in control. The user can use both the mouse and the keyboard and can manipulate many parts of the user interface in any desired order. For example, the user can enter information into text fields, pull down menus, click buttons, and drag scroll bars in any order. The program must react to the user commands in whatever order they arrive. Having to deal with many possible inputs in random order is quite a bit harder than simply forcing the user to supply input in a fixed order.

User-interface events include key presses, mouse moves, button clicks, menu selections, and so on.

In the following sections, you will learn how to write Java programs that can react to user-interface events, such as menu selections and mouse clicks. The Java windowing toolkit has a sophisticated mechanism that allows a program to specify the events in which it is interested and which objects to notify when one of these events occurs.

### 10.7.1 Listening to Events

© Serey Tryapitsyn/iStockphoto.



*In an event-driven user interface, the program receives an event whenever the user manipulates an input component.*

An event listener belongs to a class that is provided by the application programmer. Its methods describe the actions to be taken when an event occurs.

Event sources report on events. When an event occurs, the event source notifies all event listeners.

Whenever the user of a graphical program types characters or uses the mouse anywhere inside one of the windows of the program, the Java windowing toolkit sends a notification to the program that an **event** has occurred. The windowing toolkit generates huge numbers of events. For example, whenever the mouse moves a tiny interval over a window, a “mouse move” event is generated. Whenever the mouse button is clicked, “mouse pressed” and “mouse released” events are generated. In addition, higher-level events are generated when a user selects a menu item or button.

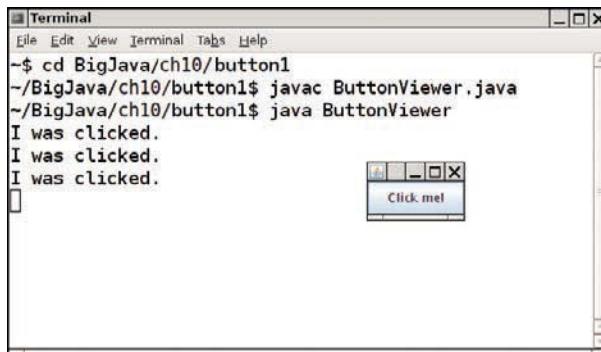
Most programs don’t want to be flooded by irrelevant events. For example, consider what happens when selecting a menu item with the mouse. The mouse moves over the menu item, then the mouse button is pressed, and finally the mouse button is released. Rather than receiving all these mouse events, a program can indicate that it only cares about menu selections, not about the underlying mouse events. However, if the mouse input is used for drawing shapes on a virtual canvas, it is necessary to closely track mouse events.

Every program must indicate which events it needs to receive. It does that by installing **event listener** objects. An event listener object belongs to a class that you provide. The methods of your event listener classes contain the instructions that you want to have executed when the events occur.

To install a listener, you need to know the **event source**. The event source is the user-interface component that generates a particular event. You add an event listener object to the appropriate event sources. Whenever the event occurs, the event source calls the appropriate methods of all attached event listeners.

This sounds somewhat abstract, so let’s run through an extremely simple program that prints a message whenever a button is clicked (see Figure 7). Button listeners must belong to a class that implements the `ActionListener` interface:

```
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```



**Figure 7** Implementing an Action Listener

This particular interface has a single method, `actionPerformed`. It is your job to supply a class whose `actionPerformed` method contains the instructions that you want executed whenever the button is clicked.

Here is a very simple example of such a listener class:

### section\_7\_1/ClickListener.java

```

1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3
4 /**
5  * An action listener that prints a message.
6 */
7 public class ClickListener implements ActionListener
8 {
9     public void actionPerformed(ActionEvent event)
10    {
11        System.out.println("I was clicked.");
12    }
13 }
```

We ignore the values of the `event` parameter variable of the `actionPerformed` method—it contains additional details about the event, such as the time at which it occurred.

Once the listener class has been declared, we need to construct an object of the class and add it to the button:

```
ActionListener listener = new ClickListener();
button.addActionListener(listener);
```

Whenever the button is clicked, it calls

```
listener.actionPerformed(event);
```

As a result, the message is printed.

You can think of the `actionPerformed` method as another example of a callback, similar to the `measure` method of the `Measurer` class. The windowing toolkit calls the `actionPerformed` method whenever the button is pressed, whereas the `Data` class calls the `measure` method whenever it needs to measure an object.

The `ButtonViewer` class, shown below, constructs a frame with a button and adds a `ClickListener` to the button. You can test this program out by opening a console window, starting the `ButtonViewer` program from that console window, clicking the button, and watching the messages in the console window.

### section\_7\_1/ButtonViewer.java

```

1 import java.awt.event.ActionListener;
2 import javax.swing.JButton;
3 import javax.swing.JFrame;
4
5 /**
6  * This program demonstrates how to install an action listener.
7 */
8 public class ButtonViewer
9 {
10     private static final int FRAME_WIDTH = 100;
11     private static final int FRAME_HEIGHT = 60;
12
13     public static void main(String[] args)
14     {
```

Use `JButton` components for buttons. Attach an `ActionListener` to each button.

```
15  JFrame frame = new JFrame();
16  JButton button = new JButton("Click me!");
17  frame.add(button);
18
19  ActionListener listener = new ClickListener();
20  button.addActionListener(listener);
21
22  frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
23  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24  frame.setVisible(true);
25 }
26 }
```

## 10.7.2 Using Inner Classes for Listeners

In the preceding section, you saw how the code to be executed when a button is clicked is placed into a listener class. It is common to implement listener classes as inner classes like this:

```
 JButton button = new JButton("...");

// This inner class is declared in the same method as the button variable
class MyListener implements ActionListener
{
    ...
};

ActionListener listener = new MyListener();
button.addActionListener(listener);
```

There are two advantages to making a listener class into an inner class. First, listener classes tend to be very short. You can put the inner class close to where it is needed, without cluttering up the remainder of the project. Moreover, inner classes have a very attractive feature: Their methods can access instance variables and methods of the surrounding class.

This feature is particularly useful when implementing event handlers. It allows the inner class to access variables without having to receive them as constructor or method arguments.

Let's look at an example. Suppose we want to add interest to a bank account whenever a button is clicked.

Methods of an inner class can access variables from the surrounding class.

```
 JButton button = new JButton("Add Interest");
final BankAccount account = new BankAccount(INITIAL_BALANCE);

// This inner class is declared in the same method as the account and button variables.
class AddInterestListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        // The listener method accesses the account variable from the surrounding block
        double interest = account.getBalance() * INTEREST_RATE / 100;
        account.deposit(interest);
    }
};

ActionListener listener = new AddInterestListener();
button.addActionListener(listener);
```

Local variables that are accessed by an inner class method must not change after they have been initialized.

There is a technical wrinkle. In versions of Java before Java 8, an inner class can access a surrounding local variable only if the variable is declared as `final`. As of Java 8, the variable must only be *effectively final*. Such a variable must behave like a `final` variable (that is, stay unchanged after it has been initialized), but it need not be declared with the `final` modifier. In our example, the `account` variable always refers to the same bank account, so it is legal to access it from the inner class. For the benefit of users of Java 7 or earlier, we declare it as `final`.

An inner class can also access *instance* variables of the surrounding class, again with a restriction. The instance variable must belong to the object that constructed the inner class object. If the inner class object was created inside a static method, it can only access static variables.

Here is the source code for the program:

### section\_7\_2/InvestmentViewer1.java

```
1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3 import javax.swing.JButton;
4 import javax.swing.JFrame;
5
6 /**
7     This program demonstrates how an action listener can access
8     a variable from a surrounding block.
9 */
10 public class InvestmentViewer1
11 {
12     private static final int FRAME_WIDTH = 120;
13     private static final int FRAME_HEIGHT = 60;
14
15     private static final double INTEREST_RATE = 10;
16     private static final double INITIAL_BALANCE = 1000;
17
18     public static void main(String[] args)
19     {
20         JFrame frame = new JFrame();
21
22         // The button to trigger the calculation
23         JButton button = new JButton("Add Interest");
24         frame.add(button);
25
26         // The application adds interest to this bank account
27         final BankAccount account = new BankAccount(INITIAL_BALANCE);
28
29         class AddInterestListener implements ActionListener
30         {
31             public void actionPerformed(ActionEvent event)
32             {
33                 // The listener method accesses the account variable
34                 // from the surrounding block
35                 double interest = account.getBalance() * INTEREST_RATE / 100;
36                 account.deposit(interest);
37                 System.out.println("balance: " + account.getBalance());
38             }
39         }
40
41         ActionListener listener = new AddInterestListener();
42         button.addActionListener(listener);
43 }
```

```

44     frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
45     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
46     frame.setVisible(true);
47 }
48
}

```

**Program Run**

```

balance: 1100.0
balance: 1210.0
balance: 1331.0
balance: 1464.1

```

**SELF CHECK**

- 26.** Which objects are the event source and the event listener in the `ButtonViewer` program?
- 27.** Why is it legal to assign a `ClickListener` object to a variable of type `ActionListener`?
- 28.** When do you call the `actionPerformed` method?
- 29.** Why would an inner class method want to access a variable from a surrounding scope?
- 30.** If an inner class accesses a local variable from a surrounding scope, what special rule applies?

**Practice It**

Now you can try these exercises at the end of the chapter: R10.16, R10.22, E10.17.

**Common Error 10.3****Modifying Parameter Types in the Implementing Method**

When you implement an interface, you must declare each method *exactly* as it is specified in the interface. Accidentally making small changes to the parameter types is a common error. Here is the classic example:

```

class MyListener implements ActionListener
{
    public void actionPerformed() // Oops . . . forgot ActionEvent parameter variable
    {
        . . .
    }
}

```

As far as the compiler is concerned, this class fails to provide the method

```
public void actionPerformed(ActionEvent event)
```

You have to read the error message carefully and pay attention to the parameter and return types to find your error.

**Common Error 10.4****Trying to Call Listener Methods**

Some students try to call the listener methods themselves:

```
ActionEvent event = new ActionEvent(. . .); // Don't do this
listener.actionPerformed(event);
```

You should not call the listener. The Java user interface calls it when the program user has clicked a button.

Java 8 Note 10.5



### Lambda Expressions for Event Handling

Java 8 Note 10.4 showed you how to use lambda expressions for instances of classes that implement “functional” interfaces; that is, interfaces with a single abstract method. This includes event handlers such as `ActionListener` objects.

For example, instead of declaring a `ClickListener` class and adding an instance as a listener to a button, you can simply add the listener as follows:

```
button.addActionListener(  
    (ActionEvent event) -> System.out.println("I was clicked."));
```

## 10.8 Building Applications with Buttons

In this section, you will learn how to structure a graphical application that contains buttons. We will put a button to work in our simple investment viewer program. Whenever the button is clicked, interest is added to a bank account, and the new balance is displayed (see Figure 8).



**Figure 8** An Application with a Button

First, we construct an object of the `JButton` class, passing the button label to the constructor, like this:

```
JButton button = new JButton("Add Interest");
```

We also need a user-interface component that displays a message, namely the current bank balance. Such a component is called a *label*. You pass the initial message string to the `JLabel` constructor, like this:

```
JLabel label = new JLabel("balance: " + account.getBalance());
```

The frame of our application contains both the button and the label. However, we cannot simply add both components directly to the frame—they would be placed on top of each other. The solution is to put them into a `panel`, a container for other user-interface components, and then add the panel to the frame:

```
JPanel panel = new JPanel();  
panel.add(button);  
panel.add(label);  
frame.add(panel);
```

Now we are ready for the hard part—the event listener that handles button clicks. As in the preceding section, it is necessary to provide a class that implements the `ActionListener` interface, and to place the button action into the `actionPerformed` method. Our listener class adds interest to the account and displays the new balance:

```
class AddInterestListener implements ActionListener  
{
```

Use a `JPanel` container to group multiple user-interface components together.

Specify button click actions through classes that implement the `ActionListener` interface.

```

public void actionPerformed(ActionEvent event)
{
    double interest = account.getBalance() * INTEREST_RATE / 100;
    account.deposit(interest);
    label.setText("balance: " + account.getBalance());
}
}

```

© Eduard Andras/iStockphoto.



*Whenever a button is pressed, the actionPerformed method is called on all listeners.*

There is just a minor technicality. The `actionPerformed` method manipulates the `account` and `label` variables. These are local variables of the `main` method of the investment viewer program, not instance variables of the `AddInterestListener` class. In versions prior to Java 8, the `account` and `label` variables need to be declared as `final` so that the `actionPerformed` method can access them.

Let's put the pieces together:

```

public static void main(String[] args)
{
    .
    .
    JButton button = new JButton("Add Interest");
    final BankAccount account = new BankAccount(INITIAL_BALANCE);
    final JLabel label = new JLabel("balance: " + account.getBalance());

    class AddInterestListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            double interest = account.getBalance() * INTEREST_RATE / 100;
            account.deposit(interest);
            label.setText("balance: " + account.getBalance());
        }
    }

    ActionListener listener = new AddInterestListener();
    button.addActionListener(listener);
    .
}

```

With a bit of practice, you will learn to glance at this code and translate it into plain English: “When the button is clicked, add interest and set the label text.”

Here is the complete program. It demonstrates how to add multiple components to a frame, by using a panel, and how to implement listeners as inner classes.

### section\_8/InvestmentViewer2.java

```

1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3 import javax.swing.JButton;
4 import javax.swing.JFrame;
5 import javax.swing.JLabel;
6 import javax.swing.JPanel;
7
8 /**
9  * This program displays the growth of an investment.
10 */
11 public class InvestmentViewer2
12 {
13     private static final int FRAME_WIDTH = 400;
14     private static final int FRAME_HEIGHT = 100;
15

```

```

16  private static final double INTEREST_RATE = 10;
17  private static final double INITIAL_BALANCE = 1000;
18
19  public static void main(String[] args)
20  {
21      JFrame frame = new JFrame();
22
23      // The button to trigger the calculation
24      JButton button = new JButton("Add Interest");
25
26      // The application adds interest to this bank account
27      final BankAccount account = new BankAccount(INITIAL_BALANCE);
28
29      // The label for displaying the results
30      final JLabel label = new JLabel("balance: " + account.getBalance());
31
32      // The panel that holds the user-interface components
33      JPanel panel = new JPanel();
34      panel.add(button);
35      panel.add(label);
36      frame.add(panel);
37
38      class AddInterestListener implements ActionListener
39      {
40          public void actionPerformed(ActionEvent event)
41          {
42              double interest = account.getBalance() * INTEREST_RATE / 100;
43              account.deposit(interest);
44              label.setText("balance: " + account.getBalance());
45          }
46      }
47
48      ActionListener listener = new AddInterestListener();
49      button.addActionListener(listener);
50
51      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
52      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
53      frame.setVisible(true);
54  }
55 }
```



31. How do you place the "balance: . . ." message to the left of the "Add Interest" button?
32. Why was it not necessary to declare the `button` variable as `final`?

**Practice It** Now you can try these exercises at the end of the chapter: E10.18, E10.19, E10.20.

### Common Error 10.5



#### Forgetting to Attach a Listener

If you run your program and find that your buttons seem to be dead, double-check that you attached the button listener. The same holds for other user-interface components. It is a surprisingly common error to program the listener class and the event handler action without actually attaching the listener to the event source.

## Programming Tip 10.2

**Don't Use a Container as a Listener**

In this book, we use inner classes for event listeners. That approach works for many different event types. Once you master the technique, you don't have to think about it anymore. Many development environments automatically generate code with inner classes, so it is a good idea to be familiar with them.

However, some programmers bypass the event listener classes and instead turn a container (such as a panel or frame) into a listener. Here is a typical example. The `actionPerformed` method is added to the viewer class. That is, the viewer implements the `ActionListener` interface.

```
public class InvestmentViewer
    implements ActionListener // This approach is not recommended
{
    public InvestmentViewer()
    {
        JButton button = new JButton("Add Interest");
        button.addActionListener(this);
        ...
    }

    public void actionPerformed(ActionEvent event)
    {
        ...
    }
    ...
}
```

Now the `actionPerformed` method is a part of the `InvestmentViewer` class rather than part of a separate listener class. The listener is installed as this.

This technique has two major flaws. First, it separates the button declaration from the button action. Also, it doesn't *scale* well. If the viewer class contains two buttons that each generate action events, then the `actionPerformed` method must investigate the event source, which leads to code that is tedious and error-prone.

## 10.9 Processing Timer Events

In this section we will study timer events and show how you can use them to implement simple animations.

The `Timer` class in the `javax.swing` package generates a sequence of action events, spaced at even time intervals. (You can think of a timer as an invisible button that is automatically clicked.) This is useful whenever you want to have an object updated at regular intervals. For example, in an animation, you may want to update a scene ten times per second and redisplay the image to give the illusion of movement.

When you use a timer, you specify the frequency of the events and an object of a class that implements the `ActionListener` interface. Place whatever action you want to occur inside the `actionPerformed` method. Finally, start the timer.

```
class MyListener implements ActionListener
{
```

A timer generates timer events at fixed intervals.



*A Swing timer notifies a listener with each "tick".*

© jeff giniewicz/Stockphoto

```
public void actionPerformed(ActionEvent event)
{
    Action that is executed at each timer event.
}
}

MyListener listener = new MyListener();
Timer t = new Timer(interval, listener);
t.start();
```

Then the timer calls the `actionPerformed` method of the `listener` object every `interval` milliseconds.

Our sample program will display a moving rectangle. We first supply a `RectangleComponent` class with a `moveRectangleBy` method that moves the rectangle by a given amount.

### section\_9/RectangleComponent.java

```
1 import java.awt.Graphics;
2 import java.awt.Graphics2D;
3 import java.awt.Rectangle;
4 import javax.swing.JComponent;
5
6 /**
7     This component displays a rectangle that can be moved.
8 */
9 public class RectangleComponent extends JComponent
10 {
11     private static final int BOX_X = 100;
12     private static final int BOX_Y = 100;
13     private static final int BOX_WIDTH = 20;
14     private static final int BOX_HEIGHT = 30;
15
16     private Rectangle box;
17
18     public RectangleComponent()
19     {
20         // The rectangle that the paintComponent method draws
21         box = new Rectangle(BOX_X, BOX_Y, BOX_WIDTH, BOX_HEIGHT);
22     }
23
24     public void paintComponent(Graphics g)
25     {
26         Graphics2D g2 = (Graphics2D) g;
27         g2.draw(box);
28     }
29
30     /**
31      Moves the rectangle by a given amount.
32      @param dx the amount to move in the x-direction
33      @param dy the amount to move in the y-direction
34     */
35     public void moveRectangleBy(int dx, int dy)
36     {
37         box.translate(dx, dy);
38         repaint();
39     }
40 }
```

The repaint method causes a component to repaint itself. Call repaint whenever you modify the shapes that the paintComponent method draws.

Note the call to repaint in the moveRectangleBy method. This call is necessary to ensure that the component is repainted after the state of the rectangle object has been changed. Keep in mind that the component object does not contain the pixels that show the drawing. The component merely contains a Rectangle object, which itself contains four coordinate values. Calling translate updates the rectangle coordinate values. The call to repaint forces a call to the paintComponent method. The paintComponent method redraws the component, causing the rectangle to appear at the updated location.

The actionPerformed method of the timer listener simply calls component.moveBy(1, 1). This moves the rectangle one pixel down and to the right. Because the actionPerformed method is called many times per second, the rectangle appears to move smoothly across the frame.

### section\_9/RectangleFrame.java

```

1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3 import javax.swing.JFrame;
4 import javax.swing.Timer;
5
6 /**
7  * This frame contains a moving rectangle.
8 */
9 public class RectangleFrame extends JFrame
10 {
11     private static final int FRAME_WIDTH = 300;
12     private static final int FRAME_HEIGHT = 400;
13
14     private RectangleComponent scene;
15
16     class TimerListener implements ActionListener
17     {
18         public void actionPerformed(ActionEvent event)
19         {
20             scene.moveRectangleBy(1, 1);
21         }
22     }
23
24     public RectangleFrame()
25     {
26         scene = new RectangleComponent();
27         add(scene);
28
29         setSize(FRAME_WIDTH, FRAME_HEIGHT);
30
31         ActionListener listener = new TimerListener();
32
33         final int DELAY = 100; // Milliseconds between timer ticks
34         Timer t = new Timer(DELAY, listener);
35         t.start();
36     }
37 }
```

### section\_9/RectangleViewer.java

```

1 import javax.swing.JFrame;
2
3 /**
```

```

4   This program moves the rectangle.
5  */
6 public class RectangleViewer
7 {
8     public static void main(String[] args)
9     {
10    JFrame frame = new RectangleFrame();
11    frame.setTitle("An animated rectangle");
12    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13    frame.setVisible(true);
14  }
15 }

```

### SELF CHECK



33. Why does a timer require a listener object?  
 34. What would happen if you omitted the call to repaint in the moveBy method?

### Practice It

Now you can try these exercises at the end of the chapter: E10.27, E10.28.

### Common Error 10.6



### Forgetting to Repaint

You have to be careful when your event handlers change the data in a painted component. When you make a change to the data, the component is not automatically painted with the new data. You must call the repaint method of the component, either in the event handler or in the component's mutator methods. Your component's paintComponent method will then be invoked with an appropriate Graphics object. Note that you should not call the paintComponent method directly.

This is a concern only for your own painted components. When you make a change to a standard Swing component such as a JLabel, the component is automatically repainted.

## 10.10 Mouse Events

Use a mouse listener to capture mouse events.

If you write programs that show drawings, and you want users to manipulate the drawings with a mouse, then you need to process mouse events. Mouse events are more complex than button clicks or timer ticks.

A mouse listener must implement the `MouseListener` interface, which contains the following five methods:

```

public interface MouseListener
{
    void mousePressed(MouseEvent event);
        // Called when a mouse button has been pressed on a component
    void mouseReleased(MouseEvent event);
        // Called when a mouse button has been released on a component
    void mouseClicked(MouseEvent event);
        // Called when the mouse has been clicked on a component
    void mouseEntered(MouseEvent event);
        // Called when the mouse enters a component
    void mouseExited(MouseEvent event);
        // Called when the mouse exits a component
}

```

The `mousePressed` and `mouseReleased` methods are called whenever a mouse button is pressed or released. If a button is pressed and released in quick succession, and the mouse has not moved, then the `mouseClicked` method is called as well. The `mouseEntered` and `mouseExited` methods can be used to paint a user-interface component in a special way whenever the mouse is pointing inside it.

The most commonly used method is `mousePressed`. Users generally expect that their actions are processed as soon as the mouse button is pressed.

You add a mouse listener to a component by calling the `addMouseListener` method:

```
public class MyMouseListener implements MouseListener
{
    // Implements five methods
}

MouseListener listener = new MyMouseListener();
component.addMouseListener(listener);
```

In our sample program, a user clicks on a component containing a rectangle. Whenever the mouse button is pressed, the rectangle is moved to the mouse location. We first enhance the `RectangleComponent` class and add a `moveRectangleTo` method to move the rectangle to a new position.

### section\_10/RectangleComponent2.java

```
1 import java.awt.Graphics;
2 import java.awt.Graphics2D;
3 import java.awt.Rectangle;
4 import javax.swing.JComponent;
5
6 /**
7     This component displays a rectangle that can be moved.
8 */
9 public class RectangleComponent2 extends JComponent
10 {
11     private static final int BOX_X = 100;
12     private static final int BOX_Y = 100;
13     private static final int BOX_WIDTH = 20;
14     private static final int BOX_HEIGHT = 30;
15
16     private Rectangle box;
17
18     public RectangleComponent2()
19     {
20         // The rectangle that the paintComponent method draws
21         box = new Rectangle(BOX_X, BOX_Y, BOX_WIDTH, BOX_HEIGHT);
22     }
23
24     public void paintComponent(Graphics g)
25     {
26         Graphics2D g2 = (Graphics2D) g;
27         g2.draw(box);
```



© James Brey/Stockphoto.

*In Swing, a mouse event isn't a gathering of rodents; it's notification of a mouse click by the program user.*

```

28     }
29
30     /**
31      Moves the rectangle to the given location.
32      @param x the x-position of the new location
33      @param y the y-position of the new location
34     */
35     public void moveRectangleTo(int x, int y)
36     {
37         box.setLocation(x, y);
38         repaint();
39     }
40 }
```

Note the call to `repaint` in the `moveRectangleTo` method. As explained in the preceding section, this call causes the component to repaint itself and show the rectangle in the new position.

Now, add a mouse listener to the component. Whenever the mouse is pressed, the listener moves the rectangle to the mouse location.

```

class MousePressListener implements MouseListener
{
    public void mousePressed(MouseEvent event)
    {
        int x = event.getX();
        int y = event.getY();
        component.moveRectangleTo(x, y);
    }

    // Do-nothing methods
    public void mouseReleased(MouseEvent event) {}
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}
    public void mouseExited(MouseEvent event) {}
}
```

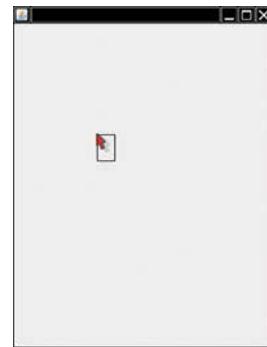
It often happens that a particular listener specifies actions only for one or two of the listener methods. Nevertheless, all five methods of the interface must be implemented. The unused methods are simply implemented as do-nothing methods.

Go ahead and run the `RectangleViewer2` program. Whenever you click the mouse inside the frame, the top-left corner of the rectangle moves to the mouse pointer (see Figure 9).

### section\_10/RectangleFrame2.java

```

1  import java.awt.event.MouseListener;
2  import java.awt.event.MouseEvent;
3  import javax.swing.JFrame;
4
5  /**
6   * This frame contains a moving rectangle.
7   */
8  public class RectangleFrame2 extends JFrame
9  {
10     private static final int FRAME_WIDTH = 300;
11     private static final int FRAME_HEIGHT = 400;
```



**Figure 9**  
Clicking the Mouse  
Moves the Rectangle

```

12     private RectangleComponent2 scene;
13
14     class MousePressListener implements MouseListener
15     {
16         public void mousePressed(MouseEvent event)
17         {
18             int x = event.getX();
19             int y = event.getY();
20             scene.moveRectangleTo(x, y);
21         }
22
23         // Do-nothing methods
24         public void mouseReleased(MouseEvent event) {}
25         public void mouseClicked(MouseEvent event) {}
26         public void mouseEntered(MouseEvent event) {}
27         public void mouseExited(MouseEvent event) {}
28     }
29
30
31     public RectangleFrame2()
32     {
33         scene = new RectangleComponent2();
34         add(scene);
35
36         MouseListener listener = new MousePressListener();
37         scene.addMouseListener(listener);
38
39         setSize(FRAME_WIDTH, FRAME_HEIGHT);
40     }
41 }
```

### section\_10/RectangleViewer2.java

```

1  import javax.swing.JFrame;
2
3  /**
4   * This program displays a rectangle that can be moved with the mouse.
5   */
6  public class RectangleViewer2
7  {
8      public static void main(String[] args)
9      {
10         JFrame frame = new RectangleFrame2();
11         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12         frame.setVisible(true);
13     }
14 }
```



35. Why was the `moveRectangleBy` method in the `RectangleComponent` replaced with a `moveRectangleTo` method?
36. Why must the `MousePressListener` class supply five methods?

**Practice It** Now you can try these exercises at the end of the chapter: R10.19, E10.29.

## Special Topic 10.5

**Keyboard Events**

If you program a game, you may want to process key-strokes, such as the arrow keys. Add a key listener to the component on which you draw the game scene. The KeyListener interface has three methods. As with a mouse listener, you are most interested in key press events, and you can leave the other two methods empty. Your key listener class should look like this:

```
class MyKeyListener implements KeyListener
{
    public void keyPressed(KeyEvent event)
    {
        String key = KeyStroke.getKeyStrokeForEvent(event).toString();
        key = key.replace("pressed ", "");
        Process key.
    }

    // Do-nothing methods
    public void keyReleased(KeyEvent event) {}
    public void keyTyped(KeyEvent event) {}
}
```



*Whenever the program user presses a key, a key event is generated.*

© shironosov/iStockphoto.com

## FULL CODE EXAMPLE

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download a program that uses the arrow keys to move a rectangle.

## Special Topic 10.6

**Event Adapters**

In the preceding section you saw how to install a mouse listener into a mouse event source and how the listener methods are called when an event occurs. Usually, a program is not interested in all listener notifications. For example, a program may only be interested in mouse clicks and may not care that these mouse clicks are composed of “mouse pressed” and “mouse released” events. Of course, the program could supply a listener that implements all those methods in which it has no interest as “do-nothing” methods, for example:

```
class MouseClickListener implements MouseListener
{
    public void mouseClicked(MouseEvent event)
    {
        Mouse click action.
    }

    // Four do-nothing methods
    public void mouseEntered(MouseEvent event) {}
    public void mouseExited(MouseEvent event) {}
    public void mousePressed(MouseEvent event) {}
    public void mouseReleased(MouseEvent event) {}
```

```
}
```

To avoid this labor, some friendly soul has created a `MouseAdapter` class that implements the `MouseListener` interface such that all methods do nothing. You can *extend* that class, inheriting the do-nothing methods and overriding the methods that you care about, like this:

```
class MouseClickListener extends MouseAdapter
{
    public void mouseClicked(MouseEvent event)
    {
        Mouse click action.
    }
}
```

There is also a `KeyAdapter` class that implements the `KeyListener` interface with three do-nothing methods.



## Computing & Society 10.1 Open Source and Free Software

Most companies that produce software regard the source code as a trade secret. After all, if customers or competitors had access to the source code, they could study it and create similar programs without paying the original vendor. For the same reason, customers dislike secret source code. If a company goes out of business or decides to discontinue support for a computer program, its users are left stranded. They are unable to fix bugs or adapt the program to a new operating system. Fortunately, many software packages are distributed as "open source software", giving its users the right to see, modify, and redistribute the source code of a program.

Having access to source code is not sufficient to ensure that software serves the needs of its users. Some companies have created software that spies on users or restricts access to previously purchased books, music, or videos. If that software runs on a server or in an embedded device, the user cannot change its behavior. In the article <http://www.gnu.org/philosophy/free-software-even-more-important.en.html>, Richard Stallman, a famous computer scientist and winner of a MacArthur "genius" grant, describes the "free software movement" that champions the right of users to control what their software does. This is an ethical position that goes beyond using open source for reasons of convenience or cost savings.

Stallman is the originator of the GNU project (<http://gnu.org/gnu/the-gnu-project.html>) that has produced an entirely free version of a UNIX-compatible operating system: the GNU operating system. All programs of the GNU project are licensed under the GNU General Public License (GNU GPL). The license allows you to make as many copies as you wish, make any modifications to the source, and redistribute the original and modified programs, charging nothing at all or whatever the market will bear. In return, you must agree that your modifications also fall under the license. You must give out the source code to any changes that you distribute, and anyone else can distribute them under the same conditions. The GNU GPL forms a social contract. Users of the software enjoy the freedom to use and modify the software, and in return they are obligated to share any improvements that they make available.

Some commercial software vendors have attacked the GPL as "viral" and "undermining the commercial software sector". Other companies have a more nuanced strategy, producing free or open source software, but charging for support or proprietary extensions. For example, the Java Development Kit is available under the GPL, but companies that need security updates for old versions or other support must pay Oracle.

Open source software sometimes lacks the polish of commercial software because many of the programmers are

volunteers who are interested in solving their own problems, not in making a product that is easy to use by everyone. Open source software has been particularly successful in areas that are of interest to programmers, such as the Linux kernel, Web servers, and programming tools.

The open source software community can be very competitive and creative. It is quite common to see several competing projects that take ideas from each other, all rapidly becoming more capable. Having many programmers involved, all reading the source code, often means that bugs tend to get squashed quickly. Eric Raymond describes open source development in his famous article "The Cathedral and the Bazaar" (<http://catb.org/~esr/writings/cathedralbazaar/cathedral-bazaar/index.html>). He writes "Given enough eyeballs, all bugs are shallow".



Courtesy of Richard Stallman.

*Richard Stallman, a pioneer of the free software movement*

**CHAPTER SUMMARY****Use interfaces for making a service available to multiple classes.**

- A Java interface type declares the methods that can be applied to a variable of that type.
- Use the `implements` reserved word to indicate that a class implements an interface type.
- Use interface types to make code more reusable.

**Describe how to convert between class and interface types.**

- You can convert from a class type to an interface type, provided the class implements the interface.
- Method calls on an interface reference are polymorphic. The appropriate method is determined at run time.
- You need a cast to convert from an interface type to a class type.

**Use the Comparable interface from the Java library.**

- Implement the `Comparable` interface so that objects of your class can be compared, for example, in a sort method.

**Describe how to use interface types for providing callbacks.**

- A callback is a mechanism for specifying code that is executed at a later time.

**Use inner classes to limit the scope of a utility class.**

- An inner class is declared inside another class.
- Inner classes are commonly used for utility classes that should not be visible elsewhere in a program.

**Use mock objects for supplying test versions of classes.**

- A mock object provides the same services as another object, but in a simplified manner.
- Both the mock class and the actual class implement the same interface.

**Implement event listeners to react to events in user-interface programming.**

- User-interface events include key presses, mouse moves, button clicks, menu selections, and so on.
- An event listener belongs to a class that is provided by the application programmer. Its methods describe the actions to be taken when an event occurs.
- Event sources report on events. When an event occurs, the event source notifies all event listeners.
- Use `JButton` components for buttons. Attach an `ActionListener` to each button.
- Methods of an inner class can access local and instance variables from the surrounding scope.
- Local variables that are accessed by an inner class method must not change after they have been initialized.

**Build graphical applications that use buttons.**

- Use a JPanel container to group multiple user-interface components together.
- Specify button click actions through classes that implement the ActionListener interface.

**Use a timer for drawing animations.**

- A timer generates timer events at fixed intervals.
- The repaint method causes a component to repaint itself. Call repaint whenever you modify the shapes that the paintComponent method draws.

**Write programs that process mouse events.**

- Use a mouse listener to capture mouse events.

**STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER**

|                                            |                                            |                                         |
|--------------------------------------------|--------------------------------------------|-----------------------------------------|
| <code>java.awt.Component</code>            | <code>java.awt.event.KeyListener</code>    | <code>java.lang.Double</code>           |
| <code>    addKeyListener</code>            | <code>    keyPressed</code>                | <code>    java.lang.Integer</code>      |
| <code>    addMouseListener</code>          | <code>    keyReleased</code>               | <code>    compare</code>                |
| <code>    repaint</code>                   | <code>    keyTyped</code>                  | <code>javax.swing.AbstractButton</code> |
| <code>    setFocusable</code>              | <code>java.awt.event.MouseEvent</code>     | <code>    addActionListener</code>      |
| <code>java.awt.Container</code>            | <code>    getX</code>                      | <code>    javax.swing.JButton</code>    |
| <code>    add</code>                       | <code>    getY</code>                      | <code>    javax.swing.JLabel</code>     |
| <code>java.awt.Dimension</code>            | <code>java.awt.event.MouseListener</code>  | <code>    javax.swing.JPanel</code>     |
| <code>java.awt.Rectangle</code>            | <code>    mouseClicked</code>              | <code>    javax.swing.KeyStroke</code>  |
| <code>     setLocation</code>              | <code>    mouseEntered</code>              | <code>    getKeyStrokeForEvent</code>   |
| <code>java.awt.event.ActionListener</code> | <code>    mouseExited</code>               | <code>    javax.swing.Timer</code>      |
| <code>     actionPerformed</code>          | <code>    mousePressed</code>              | <code>    start</code>                  |
| <code>java.awt.event.KeyEvent</code>       | <code>    mouseReleased</code>             | <code>    stop</code>                   |
|                                            | <code>java.lang.Comparable&lt;T&gt;</code> |                                         |
|                                            | <code>    compareTo</code>                 |                                         |

**REVIEW EXERCISES**

- ■ R10.1** Suppose an int value `a` is two billion and `b` is `-a`. What is the result of `a - b`? Of `b - a`? What is the result of `Integer.compare(a, b)`? Of `Integer.compare(b - a)`?
- ■ R10.2** Suppose a double value `a` is 0.6 and `b` is 0.3. What is the result of `(int)(a - b)`? Of `(int)(b - a)`? What is the result of `Double.compare(a, b)`? Of `Double.compare(b - a)`?
- R10.3** Suppose `C` is a class that implements the interfaces `I` and `J`. Which of the following assignments require a cast?

```
C c = . . .;
I i = . . .;
J j = . . .;

a. c = i;
b. j = c;
c. i = j;
```

- **R10.4** Suppose `C` is a class that implements the interfaces `I` and `J`, and suppose `i` is declared as: `I i = new C();`

Which of the following statements will throw an exception?

- a.** `C c = (C) i;`
- b.** `J j = (J) i;`
- c.** `i = (I) null;`

- **R10.5** Suppose the class `Sandwich` implements the `Edible` interface, and you are given the variable declarations

```
Sandwich sub = new Sandwich();
Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
Edible e = null;
```

Which of the following assignment statements are legal?

- |                                                    |                                                   |
|----------------------------------------------------|---------------------------------------------------|
| <b>a.</b> <code>e = sub;</code>                    | <b>e.</b> <code>e = cerealBox;</code>             |
| <b>b.</b> <code>sub = e;</code>                    | <b>f.</b> <code>e = (Edible) cerealBox;</code>    |
| <b>c.</b> <code>sub = (Sandwich) e;</code>         | <b>g.</b> <code>e = (Rectangle) cerealBox;</code> |
| <b>d.</b> <code>sub = (Sandwich) cerealBox;</code> | <b>h.</b> <code>e = (Rectangle) null;</code>      |

- ■ **R10.6** The classes `Rectangle2D.Double`, `Ellipse2D.Double`, and `Line2D.Double` implement the `Shape` interface. The `Graphics2D` class depends on the `Shape` interface but not on the rectangle, ellipse, and line classes. Draw a UML diagram denoting these facts.

- ■ **R10.7** Suppose `r` contains a reference to a new `Rectangle(5, 10, 20, 30)`. Which of the following assignments is legal? (Look inside the API documentation to check which interfaces the `Rectangle` class implements.)

- |                                              |                                            |
|----------------------------------------------|--------------------------------------------|
| <b>a.</b> <code>Rectangle a = r;</code>      | <b>e.</b> <code>Measurable e = r;</code>   |
| <b>b.</b> <code>Shape b = r;</code>          | <b>f.</b> <code>Serializable f = r;</code> |
| <b>c.</b> <code>String c = r;</code>         | <b>g.</b> <code>Object g = r;</code>       |
| <b>d.</b> <code>ActionListener d = r;</code> |                                            |

- ■ **R10.8** Classes such as `Rectangle2D.Double`, `Ellipse2D.Double`, and `Line2D.Double` implement the `Shape` interface. The `Shape` interface has a method

```
Shape getBounds()
```

that returns a rectangle completely enclosing the shape. Consider the method call:

```
Shape s = . . . ;
Rectangle r = s.getBounds();
```

Explain why this is an example of polymorphism.

- ■ **R10.9** Suppose you need to process an array of employees to find the average salary. Discuss what you need to do to use the `Data.average` method in Section 10.1 (which processes `Measurable` objects). What do you need to do to use the second implementation (in Section 10.4)? Which is easier?

- **R10.10** What happens if you try to use an array of `String` objects with the `Data.average` method in Section 10.1?

- ■ **R10.11** How can you use the `Data.average` method in Section 10.4 if you want to compute the average length of the strings?

- ■ **R10.12** What happens if you pass an array of strings and an `AreaMeasurer` to the `Data.average` method of Section 10.4?

- R10.13** Consider this top-level and inner class. Which variables can the f method access?

```
public class T
{
    private int t;

    public void m(final int x, int y)
    {
        int a;
        final int b;

        class C implements I
        {
            public void f()
            {
                . . .
            }

            final int c;
            . . .
        }
    }
}
```

- R10.14** What happens when an inner class tries to access a local variable that assumes more than one value? Try it out and explain your findings.

- Graphics R10.15** How would you reorganize the InvestmentViewer1 program if you needed to make AddInterestListener into a top-level class (that is, not an inner class)?

- Graphics R10.16** What is an event object? An event source? An event listener?
- Graphics R10.17** From a programmer's perspective, what is the most important difference between the user interfaces of a console application and a graphical application?
- Graphics R10.18** What is the difference between an ActionEvent and a MouseEvent?
- Graphics R10.19** Why does the ActionListener interface have only one method, whereas the MouseListener has five methods?
- Graphics R10.20** Can a class be an event source for multiple event types? If so, give an example.
- Graphics R10.21** What information does an action event object carry? What additional information does a mouse event object carry?
- Graphics R10.22** Why are we using inner classes for event listeners? If Java did not have inner classes, could we still implement event listeners? How?
- Graphics R10.23** What is the difference between the paintComponent and repaint methods?
- Graphics R10.24** What is the difference between a frame and a panel?

## PRACTICE EXERCISES

- E10.1** Add a method

```
public static Measurable max(Measurable[] objects)
to the Data class that returns the object with the largest measure.
```

- **E10.2** Implement a class `Quiz` that implements the `Measurable` interface. A quiz has a score and a letter grade (such as `B+`). Use the `Data` class of Exercise E10.1 to process an array of quizzes. Display the average score and the quiz with the highest score (both letter grade and score).
- **E10.3** A person has a name and a height in centimeters. Use the `Data` class of Exercise E10.1 to process an array of `Person` objects. Display the average height and the name of the tallest person.
- **E10.4** Add static methods `largest` and `smallest` to the `Measurable` interface. The methods should return the object with the largest or smallest measure from an array of `Measurable` objects.
- **E10.5** In the `Sequence` interface of Worked Example 10.1, add static methods that yield `Sequence` instances:

```
static Sequence multiplesOf(int n)
static Sequence powersOf(int n)
```

For example, `Sequence.powersOf(2)` should return the same sequence as the `SquareSequence` class in the worked example.

-  ■ **E10.6** In Worked Example 10.1, add a default method  
`default int[] values(int n)`  
 that yields an array of the first `n` values of the sequence.
- **E10.7** In Worked Example 10.1, make the `process` method a default method of the `Sequence` interface.
- **E10.8** Add a method to the `Data` class that returns the object with the largest measure, as measured by the supplied measurer:

```
public static Object max(Object[] objects, Measurer m)
```

- **E10.9** Using a different `Measurer` object, process a set of `Rectangle` objects to find the rectangle with the largest perimeter.
- **E10.10** Modify the `Coin` class from Chapter 8 to have it implement the `Comparable` interface.
- **E10.11** Repeat Exercise E10.9, making the `Measurer` into an inner class inside the `main` method.
- **E10.12** Repeat Exercise E10.9, making the `Measurer` an inner class outside the `main` method.
- **E10.13** Implement a class `Bag` that stores items represented as strings. Items can be repeated. Supply methods for adding an item, and for counting how many times an item has been added:

```
public void add(String itemName)
public int count(String itemName)
```

Your `Bag` class should store the data in an `ArrayList<Item>`, where `Item` is an inner class with two instance variables: the name of the item and the quantity.

- **E10.14** Implement a class `Grid` that stores measurements in a rectangular grid. The grid has a given number of rows and columns, and a description string can be added for any grid location. Supply the following constructor and methods:

```
public Grid(int numRows, int numColumns)
public void add(int row, int column, String description)
public String getDescription(int row, int column)
public ArrayList<Location> getDescribedLocations()
```

Here, `Location` is an inner class that encapsulates the row and the column of a grid location.

- **E10.15** Reimplement Exercise E10.14 where the grid is unbounded. The constructor has no arguments, and the row and column parameter variables of the `add` and `getDescription` methods can be arbitrary integers.
- **Graphics E10.16** Write a method `randomShape` that randomly generates objects implementing the `Shape` interface in the Java library API: some mixture of rectangles, ellipses, and lines, with random positions. Call it ten times and draw all of them.
- **Graphics E10.17** Enhance the `ButtonViewer` program so that it prints a message “I was clicked *n* times!” whenever the button is clicked. The value *n* should be incremented with each click.
- **Graphics E10.18** Enhance the `ButtonViewer` program so that it has two buttons, each of which prints a message “I was clicked *n* times!” whenever the button is clicked. Each button should have a separate click count.
- **Graphics E10.19** Enhance the `ButtonViewer` program so that it has two buttons labeled A and B, each of which prints a message “Button *x* was clicked!”, where *x* is A or B.
- **Graphics E10.20** Implement a `ButtonViewer` program as in Exercise E10.19, using only a single listener class.
- **Graphics E10.21** Enhance the `ButtonViewer` program so that it prints the time at which the button was clicked.
- **Graphics E10.22** Implement the `AddInterestListener` in the `InvestmentViewer1` program as a regular class (that is, not an inner class). *Hint:* Store a reference to the bank account. Add a constructor to the listener class that sets the reference.
- **Graphics E10.23** Implement the `AddInterestListener` in the `InvestmentViewer2` program as a regular class (that is, not an inner class). *Hint:* Store references to the bank account and the label in the listener. Add a constructor to the listener class that sets the references.
- **E10.24** Reimplement the program in Section 10.7.2, specifying the listener with a lambda expression (see Java 8 Note 10.4).
- **E10.25** Reimplement the program in Section 10.8, specifying the listener with a lambda expression (see Java 8 Note 10.4).
- **E10.26** Reimplement the program in Section 10.9, specifying the listener with a lambda expression (see Java 8 Note 10.4).
- **Graphics E10.27** Write a program that uses a timer to print the current time once a second. *Hint:* The following code prints the current time:
 

```
Date now = new Date();
System.out.println(now);
```

 The `Date` class is in the `java.util` package.
- **Graphics E10.28** Change the `RectangleComponent` for the animation program in Section 10.9 so that the rectangle bounces off the edges of the component rather than moving outside.
- **Graphics E10.29** Change the `RectangleComponent` for the mouse listener program in Section 10.10 so that a new rectangle is added to the component whenever the mouse is clicked. *Hint:* Keep an `ArrayList<Rectangle>` and draw all rectangles in the `paintComponent` method.

- **E10.30** Supply a class `Person` that implements the `Comparable` interface. Compare persons by their names. Ask the user to input ten names and generate ten `Person` objects. Using the `compareTo` method, determine and print the first and last person among them.

## PROGRAMMING PROJECTS

- ■ **P10.1** Modify the `display` method of the `LastDigitDistribution` class of Worked Example 10.1 so that it produces a histogram, like this:

```
0: ****
1: *****
2: *****
```

Scale the bars so that widest one has length 40.

- ■ **P10.2** Write a class `PrimeSequence` that implements the `Sequence` interface of Worked Example 10.1, and produces the sequence of prime numbers.
- **P10.3** Add a method `hasNext` to the `Sequence` interface of Worked Example 10.1 that returns `false` if the sequence has no more values. Implement a class `MySequence` producing a sequence of real data of your choice, such as populations of cities or countries, temperatures, or stock prices. Obtain the data from the Internet and reformat the values so that they are placed into an array. Return one value at a time in the `next` method, until you reach the end of the data. Your `SequenceDemo` class should display the distribution of the last digits of all sequence values.
- **P10.4** Provide a class `FirstDigitDistribution` that works just like the `LastDigitDistribution` class of Worked Example 10.1, except that it counts the distribution of the first digit of each value. (It is a well-known fact that the first digits of random values are *not* uniformly distributed. This fact has been used to detect accounting fraud, when sequences of transaction amounts had an unnatural distribution of their first digits.)

- ■ **P10.5** Declare an interface `Filter` as follows:

```
public interface Filter { boolean accept(Object x); }
```

Modify the implementation of the `Data` class in Section 10.4 to use both a `Measurer` and a `Filter` object. Only objects that the filter accepts should be processed. Demonstrate your modification by processing a collection of bank accounts, filtering out all accounts with balances less than \$1,000.

-  ■ ■ **P10.6** Solve Exercise P10.5, using a lambda expression for the filter.
-  ■ ■ **P10.7** In Exercise P10.5, add a method to the `Filter` interface that counts how many objects are accepted by the filter:

```
static int count(Object[] values, Filter condition)
```

-  ■ ■ **P10.8** In Exercise P10.5, add a method to the `Filter` interface that retains all objects accepted by the filter and removes the others:

```
static void retainAll(Object[] values, Filter condition)
```

-  ■ ■ **P10.9** In Exercise P10.5, add a method `default boolean reject(Object x)` to the `Filter` interface that returns true for all objects that this filter doesn't accept.
-  ■ ■ **P10.10** In Exercise P10.5, add a method `default Filter invert()` to the `Filter` interface that yields a filter accepting exactly the objects that this filter rejects.

**■■ P10.11** Consider an interface

```
public interface NumberFormatter
{
    String format(int n);
}
```

Provide four classes that implement this interface. A `DefaultFormatter` formats an integer in the usual way. A `DecimalSeparatorFormatter` formats an integer with decimal separators; for example, one million as 1,000,000. An `AccountingFormatter` formats negative numbers with parentheses; for example, -1 as (1). A `BaseFormatter` formats the number in base  $n$ , where  $n$  is any number between 2 and 36 that is provided in the constructor.

**■■ P10.12** Write a method that takes an array of integers and a `NumberFormatter` object (from Exercise P10.11) and prints each number on a separate line, formatted with the given formatter. The numbers should be right aligned.**■■ P10.13** The `System.out.printf` method has predefined formats for printing integers, floating-point numbers, and other data types. But it is also extensible. If you use the `S` format, you can print any class that implements the `Formattable` interface. That interface has a single method:

```
void formatTo(Formatter formatter, int flags, int width, int precision)
```

In this exercise, you should make the `BankAccount` class implement the `Formattable` interface. Ignore the flags and precision and simply format the bank balance, using the given width. In order to achieve this task, you need to get an `Appendable` reference like this:

```
Appendable a = formatter.out();
```

`Appendable` is another interface with a method

```
void append(CharSequence sequence)
```

`CharSequence` is yet another interface that is implemented by (among others) the `String` class. Construct a string by first converting the bank balance into a string and then padding it with spaces so that it has the desired width. Pass that string to the `append` method.

**■■■ P10.14** Enhance the `formatTo` method of Exercise P10.13 by taking into account the precision.**■■■ Graphics P10.15** Write a program that displays a scrolling message in a panel. Use a timer for the scrolling effect. In the timer's action listener, move the starting position of the message and repaint. When the message has left the window, reset the starting position to the other corner. Provide a user interface to customize the message text, font, foreground and background colors, and the scrolling speed and direction.**■■■ Graphics P10.16** Write a program that allows the user to specify a triangle with three mouse presses. After the first mouse press, draw a small dot. After the second mouse press, draw a line joining the first two points. After the third mouse press, draw the entire triangle. The fourth mouse press erases the old triangle and starts a new one.**■■■ Graphics P10.17** Implement a program that allows two players to play tic-tac-toe. Draw the game grid and an indication of whose turn it is (X or O).

© Kathy Muller/Stockphoto

Upon the next click, check that the mouse click falls into an empty location, fill the location with the mark of the current player, and give the other player a turn. If the game is won, indicate the winner. Also supply a button for starting over.

- **Graphics P10.18** Write a program that lets users design bar charts with a mouse. When the user clicks inside a bar, the next mouse click extends the length of the bar to the  $x$ -coordinate of the mouse click. (If it is at or near 0, the bar is removed.) When the user clicks below the last bar, a new bar is added whose length is the  $x$ -coordinate of the mouse click.
- ■ **Testing P10.19** Consider the task of writing a program that plays tic-tac-toe against a human opponent. A user interface `TicTacToeUI` reads the user's moves and displays the computer's moves and the board. A class `TicTacToeStrategy` determines the next move that the computer makes. A class `TicTacToeBoard` represents the current state of the board. Complete all classes except for the strategy class. Instead, use a mock class that simply picks the first available empty square.
- ■ **Testing P10.20** Consider the task of translating a plain-text book from Project Gutenberg (<http://gutenberg.org>) to HTML. For example, here is the start of the first chapter of Tolstoy's Anna Karenina:

### Chapter 1

Happy families are all alike; every unhappy family is unhappy in its own way.

Everything was in confusion in the Oblonskys' house. The wife had discovered that the husband was carrying on an intrigue with a French girl, who had been a governess in their family, and she had announced to her husband that she could not go on living in the same house with him . . .

The equivalent HTML is:

```
<h1>Chapter 1</h1>
<p>Happy families are all alike; every unhappy
family is unhappy in its own way.</p>
<p>Everything was in confusion in the
Oblonskys' house. The wife had discovered
that the husband was carrying on an intrigue
with a French girl, who had been a governess in
their family, and she had announced to her
husband that she could not go on living in the
same house with him ...</p>
```

The HTML conversion can be carried out in two steps. First, the plain text is assembled into *segments*, blocks of text of the same kind (heading, paragraph, and so on). Then each segment is converted, by surrounding it with the HTML tags and converting special characters.

Fetching the text from the Internet and breaking it into segments is a challenging task. Provide an interface and a mock implementation. Combine it with a class that uses the mock implementation to finish the formatting task.

- **Graphics P10.21** Write a program that demonstrates the growth of a roach population. Start with two roaches and double the number of roaches with each button click.
- **Graphics P10.22** Write a program that animates a car so that it moves across a frame.
- **Graphics P10.23** Write a program that animates two cars moving across a frame in opposite directions (but at different heights so that they don't collide.)

Plain Text	HTML
“ ”	&ldquo; (left) or &rdquo; (right)
‘ ’	&lsquo; (left) or &rsquo; (right)
—	&emdash;
<	&lt;
>	&gt;
&	&amp;

- Graphics P10.24** Write a program that prompts the user to enter the *x*- and *y*-positions of the center and a radius, using `JOptionPane` dialogs. When the user clicks a “Draw” button, prompt for the inputs and draw a circle with that center and radius in a component.
- Graphics P10.25** Write a program that allows the user to specify a circle by clicking on the center and then typing the radius in a `JOptionPane`. Note that you don’t need a “Draw” button.
- Graphics P10.26** Write a program that allows the user to specify a circle with two mouse presses, the first one on the center and the second on a point on the periphery. *Hint:* In the mouse press handler, you must keep track of whether you already received the center point in a previous mouse press.
- Graphics P10.27** Design an interface `MoveableShape` that can be used as a generic mechanism for animating a shape. A moveable shape must have two methods: `move` and `draw`. Write a generic `AnimationPanel` that paints and moves any `MoveableShape` (or array list of `MoveableShape` objects). Supply moveable rectangle and car shapes.
- P10.28** Your task is to design a general program for managing board games with two players. Your program should be flexible enough to handle games such as tic-tac-toe, chess, or the Game of Nim of Programming Project 6.5.  
Design an interface `Game` that describes a board game. Think about what your program needs to do. It asks the first player to input a move—a string in a game-specific format, such as `Be3` in chess. Your program knows nothing about specific games, so the `Game` interface must have a method such as
- ```
boolean isValidMove(String move)
```
- Once the move is found to be valid, it needs to be executed—the interface needs another method `executeMove`. Next, your program needs to check whether the game is over. If not, the other player’s move is processed. You should also provide some mechanism for displaying the current state of the board.
- Design the `Game` interface and provide two implementations of your choice—such as `Nim` and `Chess` (or `TicTacToe` if you are less ambitious). Your `GamePlayer` class should manage a `Game` reference without knowing which game is played, and process the moves from both players. Supply two programs that differ only in the initialization of the `Game` reference.

## ANSWERS TO SELF-CHECK QUESTIONS

1. It must implement the `Measurable` interface, and its `getMeasure` method must return the salary.
2. The `Object` class doesn’t have a `getMeasure` method.
3. You cannot modify the `String` class to implement `Measurable`—`String` is a library class. See Section 10.4 for a solution.
4. `Measurable` is not a class. You cannot construct objects of type `Measurable`.
5. The variable `meas` is of type `Measurable`, and that type has no `getName` method.
6. Only if `meas` actually refers to a `BankAccount` object.
7. No—a `Country` reference can be converted to a `Measurable` reference, but if you attempt to cast that reference to a `BankAccount`, an exception occurs.
8. `Measurable` is an interface. Interfaces have no instance variables.
9. That variable never refers to a `Measurable` object. It refers to an object of some class—a class that implements the `Measurable` interface.

10. The code fragment prints 20255. The average method calls `getMeasure` on each object in the array. In the first call, the object is a `BankAccount`. In the second call, the object is a `Country`. A different `getMeasure` method is called in each case. The first call returns the account balance, the second one the area, which are then averaged.

11. Have the `Country` class implement the `Comparable` interface, as shown below, and call `Arrays.sort`.

```
public class Country implements Comparable {
    ...
    public int compareTo(Object otherObject) {
        Country other = (Country) otherObject;
        if (area < other.area) { return -1; }
        if (area > other.area) { return 1; }
        return 0;
    }
}
```

12. Yes, you can, because `String` implements the `Comparable` interface type.

13. No. The `Rectangle` class does not implement the `Comparable` interface.

14. `public static Comparable max(Comparable a, Comparable b)`  
`{`  
 `if (a.compareTo(b) > 0) { return a; }`  
 `else { return b; }`  
`}`

15. `BankAccount larger`  
`= (BankAccount) max(first, second);`  
`System.out.println(larger.getBalance());`

Note that the result must be cast from `Comparable` to `BankAccount` so that you can invoke the `getBalance` method.

16. The `String` class doesn't implement the `Measurable` interface.

17. Implement a class `StringMeasurer` that implements the `Measurer` interface.

18. A `Measurer` measures an object, whereas `getMeasure` measures "itself", that is, the implicit parameter.

19. `public static Object max(Object a, Object b, Measurer m)`  
`{`  
 `if (m.getMeasure(a) > m.getMeasure(b))`  
 `{`  
 `return a;`  
 `}`  
 `else { return b; }`  
`}`

20. `Rectangle larger = (Rectangle) max(first, second, areaMeas);`  
`System.out.println(larger.getWidth() + " by "`  
`+ larger.getHeight());`

Note that the result of `max` must be cast from `Object` to `Rectangle` so that you can invoke the `getWidth` and `getHeight` methods.

21. Inner classes are convenient for insignificant classes. Also, their methods can access local and instance variables from the surrounding scope.

22. When the inner class is needed by more than one method of the classes.

23. Four: one for the outer class, one for the inner class, and two for the `Data` and `Measurer` classes.

24. You want to implement the `GradingProgram` class in terms of that interface so that it doesn't have to change when you switch between the mock class and the actual class.

25. Because the developer of `GradingProgram` doesn't have to wait for the `GradeBook` class to be complete.

26. The `button` object is the event source. The `listener` object is the event listener.

27. The `ClickListener` class implements the `ActionListener` interface.

28. You don't. It is called whenever the button is clicked.

29. Direct access is simpler than the alternative—passing the variable as an argument to a constructor or method.

30. The local variable must not change. Prior to Java 8, it must be declared as `final`.

31. First add `label` to the `panel`, then add `button`.

32. The `actionPerformed` method does not access that variable.

33. The timer needs to call some method whenever the time interval expires. It calls the `actionPerformed` method of the listener object.

34. The moved rectangles won't be painted, and the rectangle will appear to be stationary until the frame is repainted for an external reason.

35. Because you know the current mouse position, not the amount by which the mouse has moved.

36. It implements the `MouseListener` interface, which has five methods.



## WORKED EXAMPLE 10.1

## Investigating Number Sequences



In this Worked Example, we investigate properties of number sequences. A number sequence can be a sequence of measurements, prices, random values, or mathematical values (such as the sequence of prime numbers). There are many interesting properties that can be investigated. For example, you can look for hidden patterns or test whether a sequence is truly random.

**Problem Statement** Investigate how the last digit of each value is distributed. For a given sequence of values, produce a chart such as

```
0: 105
1: 94
2: 81
3: 112
4: 89
5: 103
6: 103
7: 100
8: 108
9: 105
```



© Norelbo/iStockphoto.

In order to produce arbitrary sequences, we declare an interface type with a single method:

```
public interface Sequence
{
    int next();
}
```

The `LastDigitDistribution` class analyzes sequences. It keeps an array of ten counters. Its `process` method receives a `Sequence` object and the number of values to process and updates the counters:

```
public void process(Sequence seq, int valuesToProcess)
{
    for (int i = 1; i <= valuesToProcess; i++)
    {
        int value = seq.next();
        int lastDigit = value % 10;
        counters[lastDigit]++;
    }
}
```

Note that this method has no knowledge of how the sequence values are produced.

To analyze a specific sequence, you provide a class that implements the `Sequence` interface. Here are two examples: the sequence of perfect squares (0 1 4 9 16 25 ...) and a sequence of random integers.

```
public class SquareSequence implements Sequence
{
    private int n;

    public int next()
    {
        n++;
        return n * n;
    }
}
```

## WE2 Chapter 10 Interfaces

```
}

public class RandomSequence implements Sequence
{
    public int next()
    {
        return (int) (Integer.MAX_VALUE * Math.random());
    }
}
```

The following class demonstrates the analysis process. Note the pattern of the last digits of the sequence of perfect squares.

```
public class SequenceDemo
{
    public static void main(String[] args)
    {
        LastDigitDistribution dist1 = new LastDigitDistribution();
        dist1.process(new SquareSequence(), 1000);
        dist1.display();
        System.out.println();

        LastDigitDistribution dist2 = new LastDigitDistribution();
        dist2.process(new RandomSequence(), 1000);
        dist2.display();
    }
}
```

### Program Run

```
0: 100
1: 200
2: 0
3: 0
4: 200
5: 100
6: 200
7: 0
8: 0
9: 200

0: 105
1: 94
2: 81
3: 112
4: 89
5: 103
6: 103
7: 100
8: 108
9: 105
```

The complete program is contained in the `ch10/worked_example_1` directory of the book's companion code.

---

# INPUT/OUTPUT AND EXCEPTION HANDLING

## CHAPTER GOALS

- To read and write text files
- To process command line arguments
- To throw and catch exceptions
- To implement programs that propagate checked exceptions

## CHAPTER CONTENTS

### 11.1 READING AND WRITING

#### TEXT FILES 520

- CE1** Backslashes in File Names 523
- CE2** Constructing a Scanner with a String 523
- ST1** Reading Web Pages 523
- ST2** File Dialog Boxes 523
- ST3** Character Encodings 524

### 11.2 TEXT INPUT AND OUTPUT 525

- ST4** Regular Expressions 532
- ST5** Reading an Entire File 533

### 11.3 COMMAND LINE ARGUMENTS 533

- HT1** Processing Text Files 536
- WE1** Analyzing Baby Names 
- C&S** Encryption Algorithms 539



James King-Holmes/Bletchley Park Trust/Photo Researchers, Inc.

### 11.4 EXCEPTION HANDLING 540

- SYN** Throwing an Exception 540
- SYN** Catching Exceptions 542
- SYN** The throws Clause 545
- SYN** The try-with-resources Statement 545
- PT1** Throw Early, Catch Late 548
- PT2** Do Not Squelch Exceptions 548
- PT3** Do Throw Specific Exceptions 548
- ST6** Assertions 549
- ST7** The try/finally Statement 549

### 11.5 APPLICATION: HANDLING INPUT ERRORS 549

- C&S** The Ariane Rocket Incident 554



James King-Holmes/Bletchley Park Trust/Photo Researchers, Inc.

In this chapter, you will learn how to read and write files—a very useful skill for processing real world data. As an application, you will learn how to encrypt data. (The Enigma machine shown at left is an encryption device used by Germany in World War II. Pioneering British computer scientists broke the code and were able to intercept encoded messages, which was a significant help in winning the war.) The remainder of this chapter tells you how your programs can report and recover from problems, such as missing files or malformed content, using the exception-handling mechanism of the Java language.

## 11.1 Reading and Writing Text Files

Use the Scanner class for reading text files.

We begin this chapter by discussing the common task of reading and writing files that contain text. Examples of text files include not only files that are created with a simple text editor, such as Windows Notepad, but also Java source code and HTML files.

In Java, the most convenient mechanism for reading text is to use the `Scanner` class. You already know how to use a `Scanner` for reading console input. To read input from a disk file, the `Scanner` class relies on another class, `File`, which describes disk files and directories. (The `File` class has many methods that we do not discuss in this book; for example, methods that delete or rename a file.)

To begin, construct a `File` object with the name of the input file:

```
File inputFile = new File("input.txt");
```

Then use the `File` object to construct a `Scanner` object:

```
Scanner in = new Scanner(inputFile);
```

This `Scanner` object reads text from the file `input.txt`. You can use the `Scanner` methods (such as `nextInt`, `nextDouble`, and `next`) to read data from the input file.

For example, you can use the following loop to process numbers in the input file:

```
while (in.hasNextDouble())
{
    double value = in.nextDouble();
    Process value.
}
```

When writing text files, use the `PrintWriter` class and the `print`/`println`/`printf` methods.

To write output to a file, you construct a `PrintWriter` object with the desired file name, for example

```
PrintWriter out = new PrintWriter("output.txt");
```

If the output file already exists, it is emptied before the new data are written into it. If the file doesn't exist, an empty file is created.

The `PrintWriter` class is an enhancement of the `PrintStream` class that you already know—`System.out` is a `PrintStream` object. You can use the familiar `print`, `println`, and `printf` methods with any `PrintWriter` object:

```
out.println("Hello, World!");
out.printf("Total: %.2f\n", total);
```

Close all files  
when you are done  
processing them.

When you are done processing a file, be sure to *close* the Scanner or PrintWriter:

```
in.close();
out.close();
```

If your program exits without closing the PrintWriter, some of the output may not be written to the disk file.

The following program puts these concepts to work. It reads a file containing numbers and writes the numbers, lined up in a column and followed by their total, to another file.

For example, if the input file has the contents

```
32 54 67.5 29 35 80
115 44.5 100 65
```

then the output file is

```
32.00
54.00
67.50
29.00
35.00
80.00
115.00
44.50
100.00
65.00
Total: 622.00
```

There is one additional issue that we need to tackle. If the input file for a Scanner doesn't exist, a `FileNotFoundException` occurs when the `Scanner` object is constructed. The compiler insists that we specify what the program should do when that happens. Similarly, the `PrintWriter` constructor generates this exception if it cannot open the file for writing. (This can happen if the name is illegal or the user does not have the authority to create a file in the given location.) In our sample program, we want to terminate the `main` method if the exception occurs. To achieve this, we label the `main` method with a `throws` declaration:

```
public static void main(String[] args) throws FileNotFoundException
```

You will see in Section 11.4 how to deal with exceptions in a more professional way.

The `File`, `PrintWriter`, and `FileNotFoundException` classes are contained in the `java.io` package.

### section\_1/Total.java

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.io.PrintWriter;
4 import java.util.Scanner;
5
6 /**
7  * This program reads a file with numbers, and writes the numbers to another
8  * file, lined up in a column and followed by their total.
9 */
10 public class Total
11 {
12     public static void main(String[] args) throws FileNotFoundException
13     {
```

```

14 // Prompt for the input and output file names
15
16 Scanner console = new Scanner(System.in);
17 System.out.print("Input file: ");
18 String inputFileName = console.nextLine();
19 System.out.print("Output file: ");
20 String outputFileName = console.nextLine();
21
22 // Construct the Scanner and PrintWriter objects for reading and writing
23
24 File inputFile = new File(inputFileName);
25 Scanner in = new Scanner(inputFile);
26 PrintWriter out = new PrintWriter(outputFileName);
27
28 // Read the input and write the output
29
30 double total = 0;
31
32 while (in.hasNextDouble())
33 {
34     double value = in.nextDouble();
35     out.printf("%15.2f\n", value);
36     total = total + value;
37 }
38
39 out.printf("Total: %.2f\n", total);
40
41 in.close();
42 out.close();
43 }
44 }
```

**SELF CHECK**

- What happens when you supply the same name for the input and output files to the `Total` program? Try it out if you are not sure.
- What happens when you supply the name of a nonexistent input file to the `Total` program? Try it out if you are not sure.
- Suppose you wanted to add the total to an existing file instead of writing a new file. Self Check 1 indicates that you cannot simply do this by specifying the same file for input and output. How can you achieve this task? Provide the pseudo-code for the solution.
- How do you modify the `Total` program so that it shows the average, not the total, of the inputs?
- How can you modify the `Total` program so that it writes the values in two columns, like this:

|        |        |
|--------|--------|
| 32.00  | 54.00  |
| 67.50  | 29.00  |
| 35.00  | 80.00  |
| 115.00 | 44.50  |
| 100.00 | 65.00  |
| Total: | 622.00 |

**Practice It** Now you can try these exercises at the end of the chapter: R11.1, R11.2, E11.1.

**Common Error 11.1****Backslashes in File Names**

When you specify a file name as a string literal, and the name contains backslash characters (as in a Windows file name), you must supply each backslash twice:

```
File inputFile = new File("c:\\homework\\input.dat");
```

A single backslash inside a quoted string is an **escape character** that is combined with the following character to form a special meaning, such as `\n` for a newline character. The `\\" combination denotes a single backslash.`

When a user supplies a file name to a program, however, the user should not type the backslash twice.

**Common Error 11.2****Constructing a Scanner with a String**

When you construct a `PrintWriter` with a string, it writes to a file:

```
PrintWriter out = new PrintWriter("output.txt");
```

However, this does *not* work for a `Scanner`. The statement

```
Scanner in = new Scanner("input.txt"); // Error?
```

does *not* open a file. Instead, it simply reads through the string: `in.next()` returns the string "input.txt". (This is occasionally useful—see Section 11.2.5.)

You must simply remember to use `File` objects in the `Scanner` constructor:

```
Scanner in = new Scanner(new File("input.txt")); // OK
```

**Special Topic 11.1****Reading Web Pages**

You can read the contents of a web page with this sequence of commands:

```
String address = "http://horstmann.com/index.html";
URL pageLocation = new URL(address);
Scanner in = new Scanner(pageLocation.openStream());
```

Now simply read the contents of the web page with the `Scanner` in the usual way. The `URL` constructor and the `openStream` method can throw an `IOException`, so you need to tag the `main` method with `throws IOException`. (See Section 11.4.3 for more information on the `throws` clause.)

The `URL` class is contained in the `java.net` package.

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download a program that reads data from a web page.

**Special Topic 11.2****File Dialog Boxes**

In a program with a graphical user interface, you will want to use a file dialog box (such as the one shown in the figure below) whenever the users of your program need to pick a file. The `JFileChooser` class implements a file dialog box for the Swing user-interface toolkit.

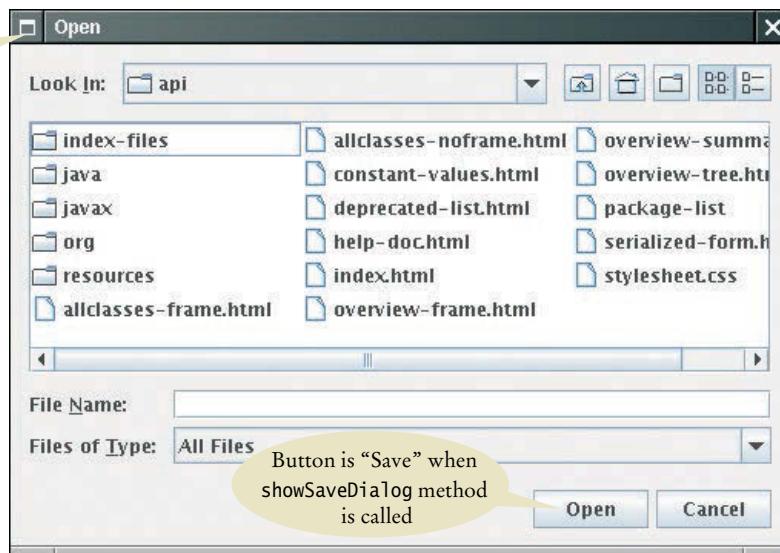
The `JFileChooser` class has many options to fine-tune the display of the dialog box, but in its most basic form it is quite simple: Construct a file chooser object; then call the `showOpenDialog` or `showSaveDialog` method. Both methods show the same dialog box, but the button for selecting a file is labeled "Open" or "Save", depending on which method you call.

For better placement of the dialog box on the screen, you can specify the user-interface component over which to pop up the dialog box. If you don't care where the dialog box pops up, you can simply pass `null`. The `showOpenDialog` and `showSaveDialog` methods return either `JFileChooser.APPROVE_OPTION`, if the user has chosen a file, or `JFileChooser.CANCEL_OPTION`, if the user canceled the selection. If a file was chosen, then you call the `getSelectedFile` method to obtain a `File` object that describes the file. Here is a complete example:

```
JFileChooser chooser = new JFileChooser();
Scanner in = null;
if (chooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION)
{
    File selectedFile = chooser.getSelectedFile();
    in = new Scanner(selectedFile);
    ...
}
```

**FULL CODE EXAMPLE**  
Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download a program that demonstrates how to use a file chooser.

Call with  
`showOpenDialog`  
method



A `JFileChooser` Dialog Box

### Special Topic 11.3



### Character Encodings

A **character** (such as the letter A, the digit 0, the accented character é, the Greek letter π, the symbol £, or the Chinese character 中) is encoded as a sequence of bytes. Each byte is a value between 0 and 255.

Unfortunately, the encoding is not uniform. In 1963, ASCII (the American Standard Code for Information Interchange) defined an encoding for 128 characters, which you can find in Appendix A. ASCII encodes all upper- and lowercase Latin letters and digits, as well as common symbols such as + \* %, as values between 0 and 127. For example, the code for the letter A is 65.

As different populations felt the need to encode their own alphabets, they designed their own codes. Many of them built upon ASCII, using the values in the range from 128 to 255 for their own language. For example, in Spain, the letter é was encoded as 233. But in Greece, the code 233 denoted the letter ι (a lowercase iota). As you can imagine, if a Spanish tourist named José sent an e-mail to a Greek hotel, this created a problem.

To resolve this issue, the design of **Unicode** was begun in 1987. As described in Computing & Society 4.2, each character in the world is given a unique integer value. However, there are still multiple encodings of those integers in binary. The most popular encoding is called UTF-8. It encodes each character as a sequence of one to four bytes. For example, an A is still 65, as in ASCII, but an é is 195 169. The details of the encoding don't matter, as long as you specify that you want UTF-8 when you read and write a file.

As this book goes to print, the Windows and Macintosh operating systems have not yet made the switch to UTF-8. Java picks up the character encoding from the operating system. Unless you specifically request otherwise, the Scanner and PrintWriter classes will read and write files in that encoding. That's fine if your files contain only ASCII characters, or if the creator and the recipient use the same encoding. But if you need to process files with accented characters, Chinese characters, or special symbols, you should specifically request the UTF-8 encoding. Construct a scanner with

```
Scanner in = new Scanner(file, "UTF-8");
```

and a print writer with

```
PrintWriter out = new PrintWriter(file, "UTF-8");
```

You may wonder why Java can't just figure out the character encoding. However, consider the string José. In UTF-8, that's 74 111 115 195 169. The first three bytes, for Jos, are in the ASCII range and pose no problem. But the next two bytes, 195 169, could be é in UTF-8 or Ä in the traditional Spanish encoding. The Scanner object doesn't understand Spanish, and it can't decide which encoding to choose.

Therefore, you should always specify the UTF-8 encoding when you exchange files with users from other parts of the world.

## 11.2 Text Input and Output

In the following sections, you will learn how to process text with complex contents, and you will learn how to cope with challenges that often occur with real data.

### 11.2.1 Reading Words

The next method reads a string that is delimited by white space.

The next method of the Scanner class reads the next string. Consider the loop

```
while (in.hasNext())
{
    String input = in.next();
    System.out.println(input);
}
```

If the user provides the input:

```
Mary had a little lamb
```

this loop prints each word on a separate line:

```
Mary
had
a
little
lamb
```

However, the words can contain punctuation marks and other symbols. The next method returns any sequence of characters that is not white space. **White space**

includes spaces, tab characters, and the newline characters that separate lines. For example, the following strings are considered “words” by the `next` method:

```
snow.  
1729  
C++
```

(Note the period after `snow`—it is considered a part of the word because it is not white space.)

Here is precisely what happens when the `next` method is executed. Input characters that are white space are *consumed*—that is, removed from the input. However, they do not become part of the word. The first character that is not white space becomes the first character of the word. More characters are added until either another white space character occurs, or the end of the input file has been reached. However, if the end of the input file is reached before any character was added to the word, a “no such element exception” occurs.

Sometimes, you want to read just the words and discard anything that isn’t a letter. You achieve this task by calling the `useDelimiter` method on your `Scanner` object:

```
Scanner in = new Scanner(. . .);  
in.useDelimiter("[^A-Za-z]+");
```

Here, we set the character pattern that separates words to “any sequence of characters other than letters”. (See Special Topic 11.4.) With this setting, punctuation and numbers are not included in the words returned by the `next` method.

### 11.2.2 Reading Characters

Sometimes, you want to read a file one character at a time. You will see an example in Section 11.3 where we encrypt the characters of a file. You achieve this task by calling the `useDelimiter` method on your `Scanner` object with an empty string:

```
Scanner in = new Scanner(. . .);  
in.useDelimiter("");
```

Now each call to `next` returns a string consisting of a single character. Here is how you can process the characters:

```
while (in.hasNext())  
{  
    char ch = in.next().charAt(0);  
    Process ch.  
}
```

### 11.2.3 Classifying Characters

The `Character` class has methods for classifying characters.

When you read a character, or when you analyze the characters in a word or line, you often want to know what kind of character it is. The `Character` class declares several useful methods for this purpose. Each of them has an argument of type `char` and returns a boolean value (see Table 1).

For example, the call

```
Character.isDigit(ch)
```

returns `true` if `ch` is a digit ('0' . . . '9' or a digit in another writing system—see Computing & Society 4.2), `false` otherwise.

**Table 1 Character Testing Methods**

| Method       | Examples of Accepted Characters |
|--------------|---------------------------------|
| isDigit      | 0, 1, 2                         |
| isLetter     | A, B, C, a, b, c                |
| isUpperCase  | A, B, C                         |
| isLowerCase  | a, b, c                         |
| isWhiteSpace | space, newline, tab             |

## 11.2.4 Reading Lines

The `nextLine` method reads an entire line.

When each line of a file is a data record, it is often best to read entire lines with the `nextLine` method:

```
String line = in.nextLine();
```

The next input line (without the newline character) is placed into the string `line`. You can then take the line apart for further processing.

The `hasNextLine` method returns `true` if there is at least one more line in the input, `false` when all lines have been read. To ensure that there is another line to process, call the `hasNextLine` method before calling `nextLine`.

Here is a typical example of processing lines in a file. A file with population data from the CIA Fact Book site (<https://www.cia.gov/library/publications/the-world-factbook/index.html>) contains lines such as the following:

```
China 1330044605
India 1147995898
United States 303824646
. . .
```

Because some country names have more than one word, it would be tedious to read this file using the `next` method. For example, after reading `United`, how would your program know that it needs to read another word before reading the population count?

Instead, read each input line into a string:

```
while (in.hasNextLine())
{
    String line = nextLine();
    Process line.
}
```

Use the `isDigit` and `isWhiteSpace` methods in Table 1 to find out where the name ends and the number starts.

Locate the first digit:

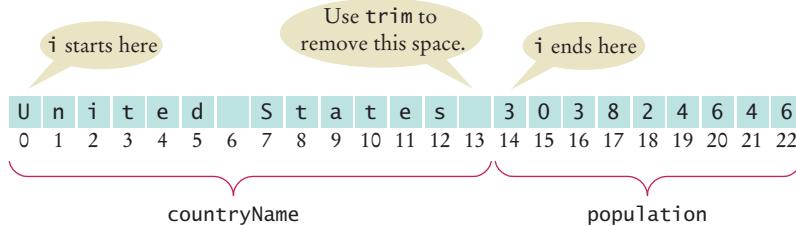
```
int i = 0;
while (!Character.isDigit(line.charAt(i))) { i++; }
```

Then extract the country name and population:

```
String countryName = line.substring(0, i);
String population = line.substring(i);
```

However, the country name contains one or more spaces at the end. Use the `trim` method to remove them:

```
countryName = countryName.trim();
```



The `trim` method returns the string with all white space at the beginning and end removed.

There is one additional problem. The population is stored in a string, not a number. In Section 11.2.6, you will see how to convert the string to a number.

## 11.2.5 Scanning a String

In the preceding section, you saw how to break a string into parts by looking at individual characters. Another approach is occasionally easier. You can use a Scanner object to read the characters from a string:

```
Scanner lineScanner = new Scanner(line);
```

Then you can use `lineScanner` like any other `Scanner` object, reading words and numbers:

```
String countryName = lineScanner.next(); // Read first word
// Add more words to countryName until number encountered
while (!lineScanner.hasNextInt())
{
    countryName = countryName + " " + lineScanner.next();
}
int populationValue = lineScanner.nextInt();
```

## 11.2.6 Converting Strings to Numbers

Sometimes you have a string that contains a number, such as the population string in Section 11.2.4. For example, suppose that the string is the character sequence "303824646". To get the integer value 303824646, you use the `Integer.parseInt` method:

```
int populationValue = Integer.parseInt(population);
// populationValue is the integer 303824646
```

To convert a string containing floating-point digits to its floating-point value, use the `Double.parseDouble` method. For example, suppose `input` is the string "3.95".

```
double price = Double.parseDouble(input);
// price is the floating-point number 3.95
```

You need to be careful when calling the `Integer.parseInt` and `Double.parseDouble` methods. The argument must be a string containing the digits of an integer, without any additional characters. Not even spaces are allowed! In our situation, we happen to

If a string contains the digits of a number, you use the `Integer.parseInt` or `Double.parseDouble` method to obtain the number value.

know that there won't be any spaces at the beginning of the string, but there might be some at the end. Therefore, we use the `trim` method:

```
int populationValue = Integer.parseInt(population.trim());
```

How To 11.1 on page 536 continues this example.

## 11.2.7 Avoiding Errors When Reading Numbers

You have used the `nextInt` and `nextDouble` methods of the `Scanner` class many times, but here we will have a look at what happens in “abnormal” situations. Suppose you call

```
int value = in.nextInt();
```

The `nextInt` method recognizes numbers such as 3 or -21. However, if the input is not a properly formatted number, an “input mismatch exception” occurs. For example, consider an input containing the characters

White space is consumed and the word `21st` is read. However, this word is not a properly formatted number, causing an input mismatch exception in the `nextInt` method.

If there is no input at all when you call `nextInt` or `nextDouble`, a “no such element exception” occurs. To avoid exceptions, use the `hasNextInt` method to screen the input when reading an integer. For example,

```
if (in.hasNextInt())
{
    int value = in.nextInt();
    . .
}
```

Similarly, you should call the `hasNextDouble` method before calling `nextDouble`.

## 11.2.8 Mixing Number, Word, and Line Input

The `nextInt`, `nextDouble`, and `next` methods *do not* consume the white space that follows the number or word. This can be a problem if you alternate between calling `nextInt`/`nextDouble`/`next` and `nextLine`. Suppose a file contains country names and population values in this format:

```
China
1330044605
India
1147995898
United States
303824646
```

Now suppose you read the file with these instructions:

```
while (in.hasNextLine())
{
    String countryName = in.nextLine();
    int population = in.nextInt();
    Process the country name and population.
}
```

Initially, the input contains

```
C h i n a \n 1 3 3 0 0 4 4 6 0 5 \n I n d i a \n
```

After the first call to the `nextLine` method, the input contains

```
1 3 3 0 0 4 4 6 0 5 \n I n d i a \n
```

After the call to `nextInt`, the input contains

```
\n I n d i a \n
```

Note that the `nextInt` call did *not* consume the newline character. Therefore, the second call to `nextLine` reads an empty string!

The remedy is to add a call to `nextLine` after reading the population value:

```
String countryName = in.nextLine();
int population = in.nextInt();
in.nextLine(); // Consume the newline
```

The call to `nextLine` consumes any remaining white space *and* the newline character.

### 11.2.9 Formatting Output

When you write numbers or strings, you often want to control how they appear. For example, dollar amounts are usually formatted with two significant digits, such as

```
Cookies: 3.20
```

You know from Section 4.3.2 how to achieve this output with the `printf` method. In this section, we discuss additional options of the `printf` method.

Suppose you need to print a table of items and prices, each stored in an array, such as

```
Cookies: 3.20
Linguine: 2.95
Clams: 17.29
```

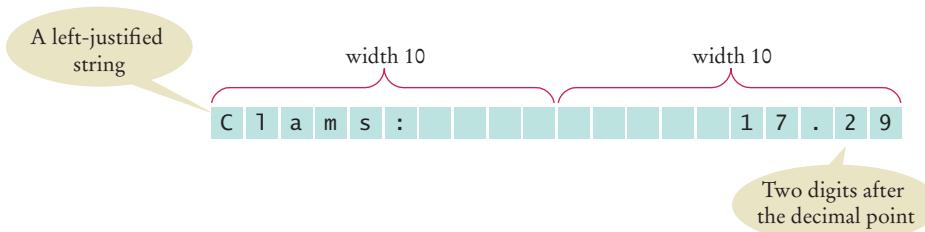
Note that the item strings line up to the left, whereas the numbers line up to the right. By default, the `printf` method lines up values to the right.

To specify left alignment, you add a hyphen (-) before the field width:

```
System.out.printf("%-10s%10.2f", items[i] + ":", prices[i]);
```

Here, we have two format specifiers.

- `%-10s` formats a left-justified string. The string `items[i] + ":"` is padded with spaces so it becomes ten characters wide. The - indicates that the string is placed on the left, followed by sufficient spaces to reach a width of 10.
- `%10.2f` formats a floating-point number, also in a field that is ten characters wide. However, the spaces appear to the left and the value to the right.



**Table 2** Format Flags

| Flag | Meaning                                 | Example                 |
|------|---|-------------------------|
| -    | Left alignment                          | 1.23 followed by spaces |
| 0    | Show leading zeroes                     | 001.23                  |
| +    | Show a plus sign for positive numbers   | +1.23                   |
| (    | Enclose negative numbers in parentheses | (1.23)                  |
| ,    | Show decimal separators                 | 12,300                  |
| ^    | Convert letters to uppercase            | 1.23E+1                 |

A construct such as %-10s or %10.2f is called a *format specifier*: it describes how a value should be formatted.

A format specifier has the following structure:

- The first character is a %.
- Next, there are optional “flags” that modify the format, such as - to indicate left alignment. See Table 2 for the most common format flags.
- Next is the field width, the total number of characters in the field (including the spaces used for padding), followed by an optional precision for floating-point numbers.
- The format specifier ends with the *format type*, such as f for floating-point values or s for strings. There are quite a few format types—Table 3 shows the most important ones.

**Table 3** Format Types

| Code | Type   | Example |
|------|--|---------|
| d    | Decimal integer  | 123     |
| f    | Fixed floating-point   | 12.30   |
| e    | Exponential floating-point   | 1.23e+1 |
| g    | General floating-point<br>(exponential notation is used for very large or very small values) | 12.3    |
| s    | String   | Tax:    |

**SELF CHECK**

6. Suppose the input contains the characters Hello, World!. What are the values of word and input after this code fragment?

```
String word = in.next();
String input = in.nextLine();
```

7. Suppose the input contains the characters 995.0 Fred. What are the values of number and input after this code fragment?
- ```
int number = 0;
if (in.hasNextInt()) { number = in.nextInt(); }
String input = in.next();
```
8. Suppose the input contains the characters 6E6 \$6,995.00. What are the values of x1 and x2 after this code fragment?
- ```
double x1 = in.nextDouble();
double x2 = in.nextDouble();
```
9. Your input file contains a sequence of numbers, but sometimes a value is not available and is marked as N/A. How can you read the numbers and skip over the markers?
10. How can you remove spaces from the country name in Section 11.2.4 without using the trim method?

**Practice It** Now you can try these exercises at the end of the chapter: E11.4, E11.6, E11.7.

### Special Topic 11.4



### Regular Expressions

A **regular expression** describes a character pattern. For example, numbers have a simple form. They contain one or more digits. The regular expression describing numbers is [0-9]+. The set [0-9] denotes any digit between 0 and 9, and the + means “one or more”.

The search commands of professional programming editors understand regular expressions. Moreover, several utility programs use regular expressions to locate matching text. A commonly used program that uses regular expressions is **grep** (which stands for “global regular expression print”). You can run grep from a command line or from inside some compilation environments. Grep is part of the UNIX operating system, and versions are available for Windows. It needs a regular expression and one or more files to search. When grep runs, it displays a set of lines that match the regular expression.

Suppose you want to find all magic numbers (see Programming Tip 4.1) in a file.

```
grep [0-9]+ Homework.java
```

lists all lines in the file Homework.java that contain sequences of digits. That isn’t terribly useful; lines with variable names x1 will be listed. OK, you want sequences of digits that do *not* immediately follow letters:

```
grep [^A-Za-z][0-9]+ Homework.java
```

The set [^A-Za-z] denotes any characters that are *not* in the ranges A to Z and a to z. This works much better, and it shows only lines that contain actual numbers.

The `useDelimiter` method of the `Scanner` class accepts a regular expression to describe delimiters—the blocks of text that separate words. As already mentioned, if you set the delimiter pattern to [^A-Za-z]+, a delimiter is a sequence of one or more characters that are not letters.

There are two useful methods of the `String` class that use regular expressions. The `split` method splits a string into an array of strings, with the delimiter specified as a regular expression. For example,

```
String[] tokens = line.split("\\s+");
```

splits input along white space. The `replaceAll` method yields a string in which all matches of a regular expression are replaced with a string. For example, `word.replaceAll("[aeiou]", "")` is the word with all vowels removed.

For more information on regular expressions, consult one of the many tutorials on the Internet by pointing your search engine to “regular expression tutorial”.

## Special Topic 11.5

**Reading an Entire File**

In the preceding section, you saw how to read lines, words, and characters from a file. Alternatively, you can read the entire file into a list of lines, or into a single string. Use the `Files` and `Paths` classes, like this:

```
String filename = . . .;
List<String> lines = Files.readAllLines(Paths.get(filename));
String content = new String(Files.readAllBytes(Paths.get(filename)));
```

The `Files` class has many other uses—see Chapter 21.

## 11.3 Command Line Arguments

Depending on the operating system and Java development environment used, there are different methods of starting a program—for example, by selecting “Run” in the compilation environment, by clicking on an icon, or by typing the name of the program at the prompt in a command shell window. The latter method is called “invoking the program from the command line”. When you use this method, you must of course type the name of the program, but you can also type in additional information that the program can use. These additional strings are called **command line arguments**. For example, if you start a program with the command line

```
java ProgramClass -v input.dat
```

then the program receives two command line arguments: the strings “`-v`” and “`input.dat`”. It is entirely up to the program what to do with these strings. It is customary to interpret strings starting with a hyphen (-) as program options.

Should you support command line arguments for your programs, or should you prompt users, perhaps with a graphical user interface? For a casual and infrequent user, an interactive user interface is much better. The user interface guides the user along and makes it possible to navigate the application without much knowledge. But for a frequent user, a command line interface has a major advantage: it is easy to automate. If you need to process hundreds of files every day, you could spend all your time typing file names into file chooser dialog boxes. However, by using batch files or shell scripts (a feature of your computer’s operating system), you can automatically call a program many times with different command line arguments.

Your program receives its command line arguments in the `args` parameter of the `main` method:

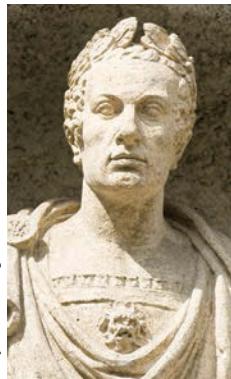
```
public static void main(String[] args)
```

In our example, `args` is an array of length 2, containing the strings

```
args[0]:  "-v"
args[1]:  "input.dat"
```

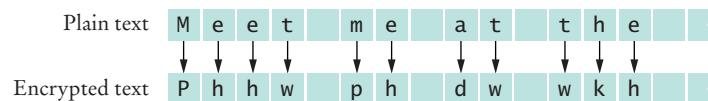
Let us write a program that *encrypts* a file—that is, scrambles it so that it is unreadable except to those who know the decryption method. Ignoring 2,000 years of progress in the field of encryption, we will use a method familiar to Julius Caesar, replacing A with a D, B with an E, and so on (see Figure 1).

Programs that start from the command line receive the command line arguments in the `main` method.



The emperor Julius Caesar used a simple scheme to encrypt messages.

**Figure 1**  
Caesar Cipher



The program takes the following command line arguments:

- An optional -d flag to indicate decryption instead of encryption
- The input file name
- The output file name

For example,

```
java CaesarCipher input.txt encrypt.txt
```

encrypts the file `input.txt` and places the result into `encrypt.txt`.

```
java CaesarCipher -d encrypt.txt output.txt
```

decrypts the file `encrypt.txt` and places the result into `output.txt`.

### section\_3/CaesarCipher.java

```

1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.io.PrintWriter;
4 import java.util.Scanner;
5
6 /**
7      This program encrypts a file using the Caesar cipher.
8 */
9 public class CaesarCipher
10 {
11     public static void main(String[] args) throws FileNotFoundException
12     {
13         final int DEFAULT_KEY = 3;
14         int key = DEFAULT_KEY;
15         String inFile = "";
16         String outFile = "";
17         int files = 0; // Number of command line arguments that are files
18
19         for (int i = 0; i < args.length; i++)
20         {
21             String arg = args[i];
22             if (arg.charAt(0) == '-')
23             {
24                 // It is a command line option
25
26                 char option = arg.charAt(1);
27                 if (option == 'd') { key = -key; }
28                 else { usage(); return; }
29             }
30             else
31             {
32                 // It is a file name
33
34                 files++;
35                 if (files == 1) { inFile = arg; }
36                 else if (files == 2) { outFile = arg; }
37             }
38         }
    }
```

```

39     if (files != 2) { usage(); return; }
40
41     Scanner in = new Scanner(new File(inFile));
42     in.useDelimiter(""); // Process individual characters
43     PrintWriter out = new PrintWriter(outFile);
44
45     while (in.hasNext())
46     {
47         char from = in.next().charAt(0);
48         char to = encrypt(from, key);
49         out.print(to);
50     }
51     in.close();
52     out.close();
53 }
54
55 /**
56  * Encrypts upper- and lowercase characters by shifting them
57  * according to a key.
58  * @param ch the letter to be encrypted
59  * @param key the encryption key
60  * @return the encrypted letter
61 */
62 public static char encrypt(char ch, int key)
63 {
64     int base = 0;
65     if ('A' <= ch && ch <= 'Z') { base = 'A'; }
66     else if ('a' <= ch && ch <= 'z') { base = 'a'; }
67     else { return ch; } // Not a letter
68     int offset = ch - base + key;
69     final int LETTERS = 26; // Number of letters in the Roman alphabet
70     if (offset > LETTERS) { offset = offset - LETTERS; }
71     else if (offset < 0) { offset = offset + LETTERS; }
72     return (char) (base + offset);
73 }
74
75 /**
76  * Prints a message describing proper usage.
77 */
78 public static void usage()
79 {
80     System.out.println("Usage: java CaesarCipher [-d] infile outfile");
81 }
82 }

```

**SELF CHECK**

11. If the program is invoked with `java CaesarCipher -d file1.txt`, what are the elements of args?
12. Trace the program when it is invoked as in Self Check 11.
13. Will the program run correctly if the program is invoked with `java CaesarCipher file1.txt file2.txt -d`? If so, why? If not, why not?
14. Encrypt CAESAR using the Caesar cipher.
15. How can you modify the program so that the user can specify an encryption key other than 3 with a -k option, for example  
`java CaesarCipher -k15 input.txt output.txt`

**Practice It**

Now you can try these exercises at the end of the chapter: R11.5, E11.10, E11.11.

**HOW TO 11.1****Processing Text Files**

Processing text files that contain real data can be surprisingly challenging. This How To gives you step-by-step guidance using world population data.

**Problem Statement** Read two country data files, `worldpop.txt` and `worldarea.txt` (supplied with your source code). Both files contain the same countries in the same order. Write a file `world_pop_density.txt` that contains country names and population densities (people per square km), with the country names aligned left and the numbers aligned right:

|                |        |
|----------------|--------|
| Afghanistan    | 50.56  |
| Akrotiri       | 127.64 |
| Albania        | 125.91 |
| Algeria        | 14.18  |
| American Samoa | 288.92 |
| .              | .      |



© Oksana Perkins/Stockphoto.

*Singapore is one of the most densely populated countries in the world.*

**Step 1** Understand the processing task.

As always, you need to have a clear understanding of the task before designing a solution. Can you carry out the task by hand (perhaps with smaller input files)? If not, get more information about the problem.

One important consideration is whether you can process the data as it becomes available, or whether you need to store it first. For example, if you are asked to write out sorted data, you first need to collect all input, perhaps by placing it in an array list. However, it is often possible to process the data “on the go”, without storing it.

In our example, we can read each file one line at a time and compute the density for each line because our input files store the population and area data in the same order.

The following pseudocode describes our processing task.

```

While there are more lines to be read
    Read a line from each file.
    Extract the country name.
    population = number following the country name in the line from the first file
    area = number following the country name in the line from the second file
    If area != 0
        density = population / area
    Print country name and density.

```

**Step 2** Determine which files you need to read and write.

This should be clear from the problem. In our example, there are two input files, the population data and the area data, and one output file.

**Step 3** Choose a mechanism for obtaining the file names.

There are three options:

- Hard-coding the file names (such as `"worldpop.txt"`).
- Asking the user:

```

Scanner in = new Scanner(System.in);
System.out.print("Enter filename: ");
String inFile = in.nextLine();

```

- Using command-line arguments for the file names.

In our example, we use hard-coded file names for simplicity.

**Step 4** Choose between line, word, and character-based input.

As a rule of thumb, read lines if the input data is grouped by lines. That is the case with tabular data, such as in our example, or when you need to report line numbers.

When gathering data that can be distributed over several lines, then it makes more sense to read words. Keep in mind that you lose all white space when you read words.

Reading characters is mostly useful for tasks that require access to individual characters. Examples include analyzing character frequencies, changing tabs to spaces, or encryption.

**Step 5** With line-oriented input, extract the required data.

It is simple to read a line of input with the `nextLine` method. Then you need to get the data out of that line. You can extract substrings, as described in Section 11.2.4.

Typically, you will use methods such as `Character.isWhitespace` and `Character.isDigit` to find the boundaries of substrings.

If you need any of the substrings as numbers, you must convert them, using `Integer.parseInt` or `Double.parseDouble`.

**Step 6** Use classes and methods to factor out common tasks.

Processing input files usually has repetitive tasks, such as skipping over white space or extracting numbers from strings. It really pays off to isolate these tedious operations from the remainder of the code.

In our example, we have a task that occurs twice: splitting an input line into the country name and the value that follows. We implement a simple `CountryValue` class for this purpose, using the technique described in Section 11.2.4.

Here is the complete source code:

**how\_to\_1/CountryValue.java**

```

1  /**
2   * Describes a value that is associated with a country.
3  */
4  public class CountryValue
5  {
6      private String country;
7      private double value;
8
9      /**
10     * Constructs a CountryValue from an input line.
11     * @param line a line containing a country name, followed by a value
12     */
13    public CountryValue(String line)
14    {
15        int i = 0; // Locate the start of the first digit
16        while (!Character.isDigit(line.charAt(i))) { i++; }
17        int j = i - 1; // Locate the end of the preceding word
18        while (Character.isWhitespace(line.charAt(j))) { j--; }
19        country = line.substring(0, j + 1); // Extract the country name
20        value = Double.parseDouble(line.substring(i).trim()); // Extract the value
21    }
22
23    /**
24     * Gets the country name.
25     * @return the country name
26     */
27    public String getCountry() { return country; }
28

```

```

29     /**
30      Gets the associated value.
31      @return the value associated with the country
32     */
33     public double getValue() { return value; }
34 }
```

### how\_to\_1/PopulationDensity.java

```

1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.io.PrintWriter;
4 import java.util.Scanner;
5
6 public class PopulationDensity
7 {
8     public static void main(String[] args) throws FileNotFoundException
9     {
10        // Open input files
11        Scanner in1 = new Scanner(new File("worldpop.txt"));
12        Scanner in2 = new Scanner(new File("worldarea.txt"));
13
14        // Open output file
15        PrintWriter out = new PrintWriter("world_pop_density.txt");
16
17        // Read lines from each file
18        while (in1.hasNextLine() && in2.hasNextLine())
19        {
20            CountryValue population = new CountryValue(in1.nextLine());
21            CountryValue area = new CountryValue(in2.nextLine());
22
23            // Compute and print the population density
24            double density = 0;
25            if (area.getValue() != 0) // Protect against division by zero
26            {
27                density = population.getValue() / area.getValue();
28            }
29            out.printf("%-40s%15.2f\n", population.getCountry(), density);
30        }
31
32        in1.close();
33        in2.close();
34        out.close();
35    }
36 }
```



### WORKED EXAMPLE 11.1

#### Analyzing Baby Names



Learn how to use data from the Social Security Administration to analyze the most popular baby names. Go to [wiley.com/go/bjoe6examples](http://wiley.com/go/bjoe6examples) and download Worked Example 11.1.



© Nancy Ross/  
iStockphoto.



## Computing & Society 11.1 Encryption Algorithms

This chapter's exercise section gives a few algorithms for encrypting text. Don't actually use any of those methods to send secret messages to your lover. Any skilled cryptographer can *break* these schemes in a very short time—that is, reconstruct the original text without knowing the secret keyword.

In 1978, Ron Rivest, Adi Shamir, and Leonard Adleman introduced an encryption method that is much more powerful. The method is called *RSA encryption*, after the last names of its inventors. The exact scheme is too complicated to present here, but it is not actually difficult to follow. You can find the details in <http://theory.lcs.mit.edu/~rivest/rsapaper.pdf>.

RSA is a remarkable encryption method. There are two keys: a public key and a private key (see the figure). You can print the public key on your business card (or in your e-mail signature block) and give it to anyone. Then anyone can send you messages that only you can decrypt. Even though everyone else knows the public key, and even if they intercept all the messages coming to you, they cannot break the scheme and actually read the messages. In 1994, hundreds of researchers, collaborating over the Internet, cracked an RSA message encrypted with a 129-digit key. Messages encrypted with a key of 230 digits or more are expected to be secure.

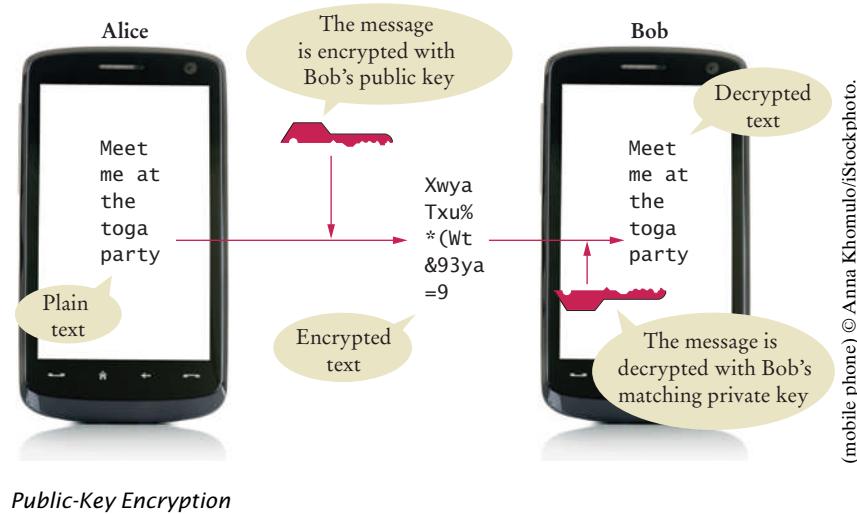
The inventors of the algorithm obtained a *patent* for it. A patent is a deal that society makes with an inventor. For a period of 20 years, the inventor has an exclusive right to its commercialization, may collect royalties from others wishing to manufacture the invention, and may even stop competitors from using it altogether. In return, the inventor must publish the invention, so that others may learn from it, and must relinquish all claim to it after the monopoly period ends. The presumption is that in the absence

of patent law, inventors would be reluctant to go through the trouble of inventing, or they would try to cloak their techniques to prevent others from copying their devices.

There has been some controversy about the RSA patent. Had there not been patent protection, would the inventors have published the method anyway, thereby giving the benefit to society without the cost of the 20-year monopoly? In this case, the answer is probably yes. The inventors were academic researchers who live on salaries rather than sales receipts and are usually rewarded for their discoveries by a boost in their reputation and careers. Would their followers have been as active in discovering (and patenting) improvements? There is no way of knowing, of course. Is an algorithm even patentable, or is it a mathematical fact that belongs to nobody? The patent office did take the latter attitude for a long time. The RSA inventors and many others described their inventions in terms of imaginary electronic devices, rather than algorithms, to circumvent

that restriction. Nowadays, the patent office will award software patents.

There is another interesting aspect to the RSA story. A programmer, Phil Zimmermann, developed a program called PGP (for *Pretty Good Privacy*) that is based on RSA. Anyone can use the program to encrypt messages, and decryption is not feasible even with the most powerful computers. You can get a copy of a free PGP implementation from the GNU project (<http://www.gnupg.org>). The existence of strong encryption methods bothers the United States government to no end. Criminals and foreign agents can send communications that the police and intelligence agencies cannot decipher. The government considered charging Zimmermann with breaching a law that forbids the unauthorized export of munitions, arguing that he should have known that his program would appear on the Internet. There have been serious proposals to make it illegal for private citizens to use these encryption methods, or to keep the keys secret from law enforcement.



(mobile phone) © Anna Khomulo/Stockphoto.

## 11.4 Exception Handling

There are two aspects to dealing with program errors: *detection* and *handling*. For example, the Scanner constructor can detect an attempt to read from a non-existent file. However, it cannot handle that error. A satisfactory way of handling the error might be to terminate the program, or to ask the user for another file name. The Scanner class cannot choose between these alternatives. It needs to report the error to another part of the program.

In Java, *exception handling* provides a flexible mechanism for passing control from the point of error detection to a handler that can deal with the error. In the following sections, we will look into the details of this mechanism.

### 11.4.1 Throwing Exceptions

To signal an exceptional condition, use the `throw` statement to throw an exception object.

#### FULL CODE EXAMPLE

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download a program that demonstrates throwing an exception.

When you detect an error condition, your job is really easy. You just *throw* an appropriate exception object, and you are done. For example, suppose someone tries to withdraw too much money from a bank account.

```
if (amount > balance)
{
    // Now what?
}
```

First look for an appropriate exception class. The Java library provides many classes to signal all sorts of exceptional conditions. Figure 2 shows the most useful ones. (The classes are arranged as a tree-shaped inheritance hierarchy, with more specialized classes at the bottom of the tree.)

Look around for an exception type that might describe your situation. How about the `ArithmaticException`? Is it an arithmetic error to have a negative balance? No—Java can deal with negative numbers. Is the amount to be withdrawn illegal? Indeed it is. It is just too large. Therefore, let's throw an `IllegalArgumentExeception`.

```
if (amount > balance)
{
    throw new IllegalArgumentExeception("Amount exceeds balance");
}
```

### Syntax 11.1

#### Throwing an Exception

**Syntax**    `throw exceptionObject;`

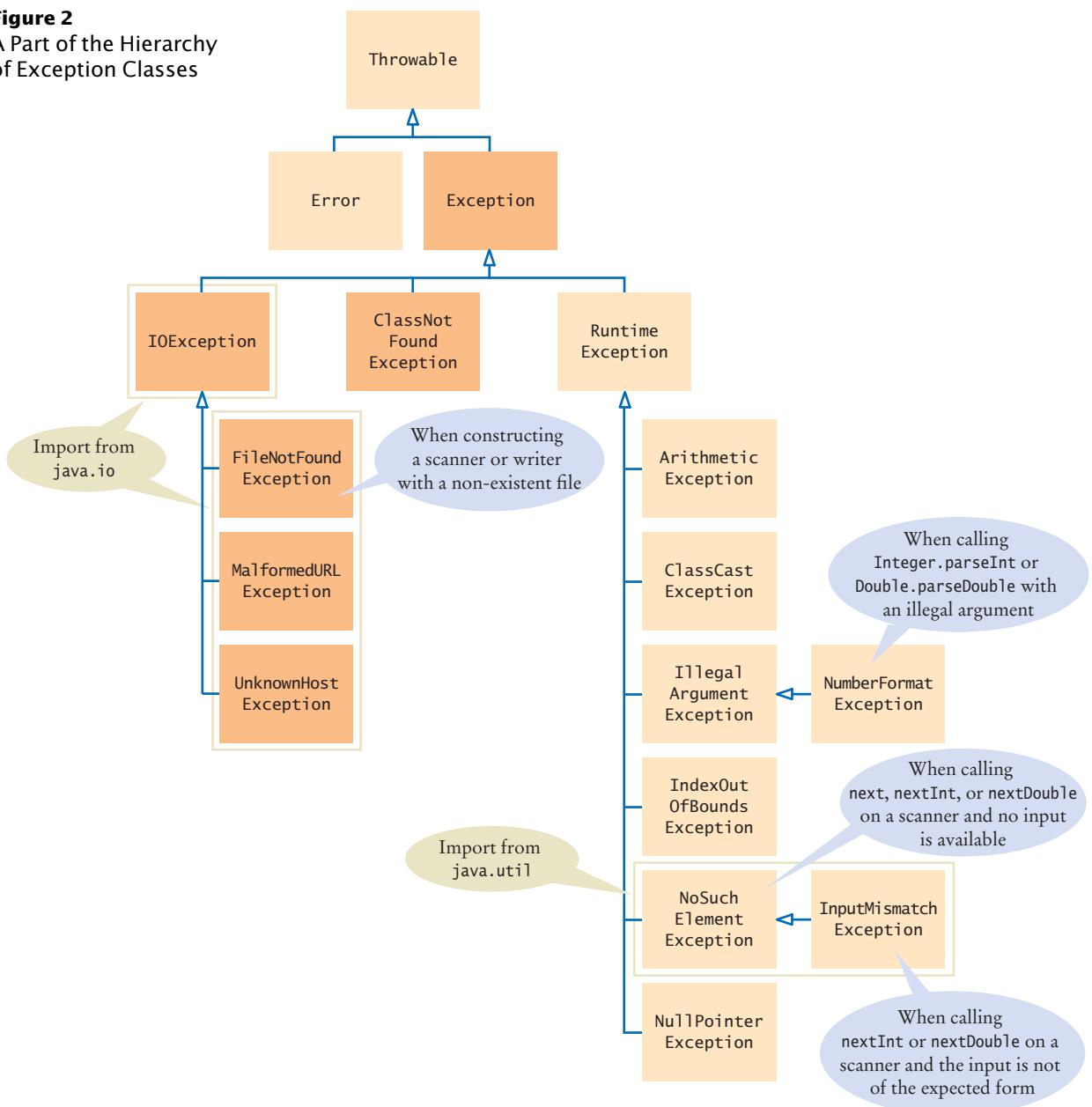
A new exception object is constructed, then thrown.

```
if (amount > balance)
{
    throw new IllegalArgumentExeception("Amount exceeds balance");
    balance = balance - amount;
}
```

Most exception objects can be constructed with an error message.

This line is not executed when the exception is thrown.

**Figure 2**  
A Part of the Hierarchy  
of Exception Classes



When you throw an exception, processing continues in an exception handler.

When you **throw an exception**, execution does not continue with the next statement but with an **exception handler**. That is the topic of the next section.

*When you throw an exception, the normal control flow is terminated. This is similar to a circuit breaker that cuts off the flow of electricity in a dangerous situation.*



© Lisa F. Young/Stockphoto.

## 11.4.2 Catching Exceptions

Place the statements that can cause an exception inside a try block, and the handler inside a catch clause.

Every exception should be handled somewhere in your program. If an exception has no handler, an error message is printed, and your program terminates. Of course, such an unhandled exception is confusing to program users.

You handle exceptions with the try/catch statement. Place the statement into a location of your program that knows how to handle a particular exception. The **try statement** contains one or more statements that may cause an exception of the kind that you are willing to handle. Each catch clause contains the handler for an exception type. Here is an example:

```
try
{
    String filename = . . .;
    Scanner in = new Scanner(new File(filename));
    String input = in.next();
    int value = Integer.parseInt(input);
    .
    .
}
catch (IOException exception)
{
    exception.printStackTrace();
}
catch (NumberFormatException exception)
{
    System.out.println(exception.getMessage());
}
```

## Syntax 11.2 Catching Exceptions

*Syntax*

```
try
{
    statement
    statement
    .
    .
}
catch (ExceptionClass exceptionObject)
{
    statement
    statement
    .
    .
}
```

When an IOException is thrown, execution resumes here.

Additional catch clauses can appear here. Place more specific exceptions before more general ones.

```
try
{
    Scanner in = new Scanner(new File("input.txt"));
    String input = in.next();
    process(input);
}
catch (IOException exception)
{
    System.out.println("Could not open input file");
}
catch (Exception except)
{
    System.out.println(except.getMessage());
```

This constructor can throw a FileNotFoundException.

This is the exception that was thrown.

A FileNotFoundException is a special case of an IOException.

Three exceptions may be thrown in this try block:

- The Scanner constructor can throw a `FileNotFoundException`.
- `Scanner.next` can throw a `NoSuchElementException`.
- `Integer.parseInt` can throw a `NumberFormatException`.

If any of these exceptions is actually thrown, then the rest of the instructions in the try block are skipped. Here is what happens for the various exception types:

- If a `FileNotFoundException` is thrown, then the catch clause for the `IOException` is executed. (If you look at Figure 2, you will note that `FileNotFoundException` is a descendant of `IOException`.) If you want to show the user a different message for a `FileNotFoundException`, you must place the catch clause *before* the clause for an `IOException`.
- If a `NumberFormatException` occurs, then the second catch clause is executed.
- A `NoSuchElementException` is *not caught* by any of the catch clauses. The exception remains thrown until it is caught by another try block.

Each catch clause contains a handler. When the `catch (IOException exception)` block is executed, then some method in the try block has failed with an `IOException` (or one of its descendants).

In this handler, we produce a printout of the chain of method calls that led to the exception, by calling

```
exception.printStackTrace()
```

In the second exception handler, we call `exception.getMessage()` to retrieve the message associated with the exception. When the `parseInt` method throws a `NumberFormatException`, the message contains the string that it was unable to format. When you throw an exception, you can provide your own message string. For example, when you call

```
throw new IllegalArgumentException("Amount exceeds balance");
```

the message of the exception is the string provided in the constructor.

In these sample catch clauses, we merely inform the user of the source of the problem. Often, it is better to give the user another chance to provide a correct input—see Section 11.5 for a solution.



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bje06code](http://wiley.com/go/bje06code) to download a program that demonstrates catching exceptions.



© Andraz Cerar/Stockphoto

*You should only catch those exceptions that you can handle.*

### 11.4.3 Checked Exceptions

In Java, the exceptions that you can throw and catch fall into three categories.

- Internal errors are reported by descendants of the type `Error`. One example is the `OutOfMemoryError`, which is thrown when all available computer memory has been used up. These are fatal errors that happen rarely, and we will not consider them in this book.
- Descendants of `RuntimeException`, such as `IndexOutOfBoundsException` or `IllegalArgumentException` indicate errors in your code. They are called **unchecked exceptions**.

Checked exceptions are due to external circumstances that the programmer cannot prevent. The compiler checks that your program handles these exceptions.

- All other exceptions are **checked exceptions**. These exceptions indicate that something has gone wrong for some external reason beyond your control. In Figure 2, the checked exceptions are shaded in a darker color.

Why have two kinds of exceptions? A checked exception describes a problem that can occur, no matter how careful you are. For example, an `IOException` can be caused by forces beyond your control, such as a disk error or a broken network connection. The compiler takes checked exceptions very seriously and ensures that they are handled. Your program will not compile if you don't indicate how to deal with a checked exception.

The unchecked exceptions, on the other hand, are your fault. The compiler does not check whether you handle an unchecked exception, such as an `IndexOutOfBoundsException`. After all, you should check your index values rather than install a handler for that exception.

If you have a handler for a checked exception in the same method that may throw it, then the compiler is satisfied. For example,

```
try
{
    File inFile = new File(filename);
    Scanner in = new Scanner(inFile); // Throws FileNotFoundException
    . .
}
catch (FileNotFoundException exception) // Exception caught here
{
    . .
}
```

#### FULL CODE EXAMPLE

Go to [wiley.com/go/bje06code](http://wiley.com/go/bje06code) to download a program that demonstrates throwing and catching checked exceptions.

Add a `throws` clause to a method that can throw a checked exception.

However, it commonly happens that the current method *cannot handle* the exception. In that case, you need to tell the compiler that you are aware of this exception and that you want your method to be terminated when it occurs. You supply the method with a `throws` clause:

```
public void readData(String filename) throws FileNotFoundException
{
    File inFile = new File(filename);
    Scanner in = new Scanner(inFile);
    . .
}
```

The **`throws` clause** signals to the caller of your method that it may encounter a `FileNotFoundException`. Then the caller needs to make the same decision—handle the exception, or declare that the exception may be thrown.

It sounds somehow irresponsible not to handle an exception when you know that it happened. Actually, the opposite is true. Java provides an exception handling facility so that an exception can be sent to the *appropriate* handler. Some methods detect errors, some methods handle them, and some methods just pass them along. The `throws` clause simply ensures that no exceptions get lost along the way.



*Just as trucks with large or hazardous loads carry warning signs, the `throws` clause warns the caller that an exception may occur.*

## Syntax 11.3 The throws Clause

**Syntax**    *modifiers returnType methodName(parameterType parameterName, . . .)*  
*throws ExceptionClass, ExceptionClass, . . .*

```
public void readData(String filename)
    throws FileNotFoundException, NumberFormatException
```

You **must** specify all checked exceptions  
that this method may throw.

You may also list unchecked exceptions.

### 11.4.4 Closing Resources

When you use a resource that must be closed, such as a `PrintWriter`, you need to be careful in the presence of exceptions. Consider this sequence of statements:

```
PrintWriter out = new PrintWriter(filename);
writeData(out);
out.close(); // May never get here
```

Now suppose that one of the methods before the last line throws an exception. Then the call to `close` is never executed! This is a problem—data that was written to the stream may never end up in the file.

The remedy is to use the **try-with-resources statement**. Declare the `PrintWriter` variable in a `try` statement, like this:

```
try (PrintWriter out = new PrintWriter(filename))
{
    writeData(out);
} // out.close() is always called
```

When the `try` block is completed, the `close` method is called on the variable. If no exception has occurred, this happens when the `writeData` method returns. However, if an exception occurs, the `close` method is invoked before the exception is passed to its handler.

The try-with-resources statement ensures that a resource is closed when the statement ends normally or due to an exception.

## Syntax 11.4

### The try-with-resources Statement

**Syntax**    `try (Type1 variable1 = expression1; Type2 variable2 = expression2; . . .)`  
`{`  
 `. . .`  
`}`

This code may  
throw exceptions.

```
try (PrintWriter out = new PrintWriter(filename))
{
    writeData(out);
}
```

Implements the  
AutoCloseable  
interface.

At this point, `out.close()` is called,  
even when an exception occurs.

You can declare multiple variables in a try-with-resources statement, like this:

```
try (Scanner in = new Scanner(inFile); PrintWriter out = new PrintWriter(outFile))
{
    while (in.hasNextLine())
    {
        String input = in.nextLine();
        String result = process(input);
        out.println(result);
    }
} // Both in.close() and out.close() are called here
```

Use the try-with-resources statement whenever you work with a Scanner or PrintWriter to make sure that these resources are closed properly.

More generally, you can declare variables of any class that implements the AutoCloseable interface in a try-with-resources statement. You will find other AutoCloseable classes in Chapters 21 and 24.



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bje06code](http://wiley.com/go/bje06code) to download a program that demonstrates closing resources.

*All visitors to a foreign country have to go through passport control, no matter what happened on their trip. Similarly, the try-with-resources statement ensures that a resource is closed, even when an exception has occurred.*



© archives/Stockphoto.

### 11.4.5 Designing Your Own Exception Types

Sometimes none of the standard exception types describe your particular error condition well enough. In that case, you can design your own exception class. Consider a bank account. Let's report an InsufficientFundsException when an attempt is made to withdraw an amount from a bank account that exceeds the current balance.

```
if (amount > balance)
{
    throw new InsufficientFundsException(
        "withdrawal of " + amount + " exceeds balance of " + balance);
}
```

To describe an error condition, provide a subclass of an existing exception class.

Now you need to provide the InsufficientFundsException class. Should it be a checked or an unchecked exception? Is it the fault of some external event, or is it the fault of the programmer? We take the position that the programmer could have prevented the exceptional condition—after all, it would have been an easy matter to check whether `amount <= account.getBalance()` before calling the `withdraw` method. Therefore, the exception should be an unchecked exception and extend the `RuntimeException` class or one of its subclasses.

It is a good idea to extend an appropriate class in the exception hierarchy. For example, we can consider an `InsufficientFundsException` a special case of an `IllegalArgumentException`. This enables other programmers to catch the exception as an `IllegalArgumentException` if they are not interested in the exact nature of the problem.

It is customary to provide two constructors for an exception class: a constructor with no arguments and a constructor that accepts a message string describing the reason for the exception.

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download code with custom exception types.

**SELF CHECK**

Here is the declaration of the exception class:

```
public class InsufficientFundsException extends IllegalArgumentException
{
    public InsufficientFundsException() {}

    public InsufficientFundsException(String message)
    {
        super(message);
    }
}
```

When the exception is caught, its message string can be retrieved using the `getMessage` method of the `Throwable` class.

- 16.** Suppose `balance` is 100 and `amount` is 200. What is the value of `balance` after these statements?

```
if (amount > balance)
{
    throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

- 17.** When depositing an amount into a bank account, we don't have to worry about overdrafts—except when the amount is negative. Write a statement that throws an appropriate exception in that case.

- 18.** Consider the method

```
public static void main(String[] args)
{
    try
    {
        Scanner in = new Scanner(new File("input.txt"));
        int value = in.nextInt();
        System.out.println(value);
    }
    catch (IOException exception)
    {
        System.out.println("Error opening file.");
    }
}
```

Suppose the file with the given file name exists and has no contents. Trace the flow of execution.

- 19.** Why is an `ArrayIndexOutOfBoundsException` not a checked exception?

- 20.** Is there a difference between catching checked and unchecked exceptions?

- 21.** What is wrong with the following code, and how can you fix it?

```
public static void writeAll(String[] lines, String filename)
{
    PrintWriter out = new PrintWriter(filename);
    for (String line : lines)
    {
        out.println(line.toUpperCase());
    }
    out.close();
}
```

- 22.** What is the purpose of the call `super(message)` in the second `InsufficientFundsException` constructor?

- 23.** Suppose you read bank account data from a file. Contrary to your expectation, the next input value is not of type `double`. You decide to implement a `BadDataException`. Which exception class should you extend?

**Practice It** Now you can try these exercises at the end of the chapter: R11.8, R11.9, R11.10.

#### Programming Tip 11.1



#### Throw Early, Catch Late

When a method detects a problem that it cannot solve, it is better to throw an exception rather than try to come up with an imperfect fix. For example, suppose a method expects to read a number from a file, and the file doesn't contain a number. Simply using a zero value would be a poor choice because it hides the actual problem and perhaps causes a different problem elsewhere.

Throw an exception as soon as a problem is detected. Catch it only when the problem can be handled.

Conversely, a method should only catch an exception if it can really remedy the situation. Otherwise, the best remedy is simply to have the exception propagate to its caller, allowing it to be caught by a competent handler.

These principles can be summarized with the slogan “throw early, catch late”.

#### Programming Tip 11.2



#### Do Not Squelch Exceptions

When you call a method that throws a checked exception and you haven't specified a handler, the compiler complains. In your eagerness to continue your work, it is an understandable impulse to shut the compiler up by squelching the exception:

```
try
{
    Scanner in = new Scanner(new File(filename));
    // Compiler complained about FileNotFoundException
    ...
}
catch (FileNotFoundException e) {} // So there!
```

The do-nothing exception handler fools the compiler into thinking that the exception has been handled. In the long run, this is clearly a bad idea. Exceptions were designed to transmit problem reports to a competent handler. Installing an incompetent handler simply hides an error condition that could be serious.

#### Programming Tip 11.3



#### Do Throw Specific Exceptions

When throwing an exception, you should choose an exception class that describes the situation as closely as possible. For example, it would be a bad idea to simply throw a `RuntimeException` object when a bank account has insufficient funds. This would make it far too difficult to catch the exception. After all, if you caught all exceptions of type `RuntimeException`, your catch clause would also be activated by exceptions of the type `NullPointerException`, `ArrayIndexOutOfBoundsException`, and so on. You would then need to carefully examine the exception object and attempt to deduce whether the exception was caused by insufficient funds.

If the standard library does not have an exception class that describes your particular error situation, simply provide a new exception class.

**Special Topic 11.6****Assertions**

An **assertion** is a condition that you believe to be true at all times in a particular program location. An assertion check tests whether an assertion is true. Here is a typical assertion check:

```
public double deposit (double amount)
{
    assert amount >= 0;
    balance = balance + amount;
}
```

In this method, the programmer expects that the quantity `amount` can never be negative. When the assertion is correct, no harm is done, and the program works in the normal way. If, for some reason, the assertion fails, and assertion checking is enabled, then the `assert` statement throws an exception of type `AssertionError`, causing the program to terminate.

However, if assertion checking is disabled, then the assertion is never checked, and the program runs at full speed. By default, assertion checking is disabled when you execute a program.

To execute a program with assertion checking turned on, use this command:

```
java -enableassertions MainClass
```

You can also use the shortcut `-ea` instead of `-enableassertions`. You should turn assertion checking on during program development and testing.

**Special Topic 11.7****The try/finally Statement**

You saw in Section 11.4 how to ensure that a resource is closed when an exception occurs. The `try-with-resources` statement calls the `close` methods of variables declared within the statement header. You should always use the `try-with-resources` statement when closing resources.

It can happen that you need to do some cleanup other than calling a `close` method. In that case, use the `try/finally` statement:

```
public double deposit (double amount)
try
{
    . .
}
finally
{
    Cleanup. // This code is executed whether or not an exception occurs
}
```

If the body of the `try` statement completes without an exception, the cleanup happens. If an exception is thrown, the cleanup happens and the exception is then propagated to its handler.

The `try/finally` statement is rarely required because most Java library classes that require cleanup implement the `AutoCloseable` interface. However, you will see a use of this statement in Chapter 22.

## 11.5 Application: Handling Input Errors

This section walks through a complete example of a program with exception handling. The program asks a user for the name of a file. The file is expected to contain data values. The first line of the file contains the total number of values, and the remaining lines contain the data.

When designing a program, ask yourself what kinds of exceptions can occur.

For each exception, you need to decide which part of your program can competently handle it.

A typical input file looks like this:

```
3
1.45
-2.1
0.05
```

What can go wrong? There are two principal risks.

- The file might not exist.
- The file might have data in the wrong format.

Who can detect these faults? The Scanner constructor will throw an exception when the file does not exist. The methods that process the input values need to throw an exception when they find an error in the data format.

What exceptions can be thrown? The Scanner constructor throws a FileNotFoundException when the file does not exist, which is appropriate in our situation. When the file data is in the wrong format, we will throw a BadDataException, a custom checked exception class. We use a checked exception because corruption of a data file is beyond the control of the programmer.

Who can remedy the faults that the exceptions report? Only the main method of the DataAnalyzer program interacts with the user. It catches the exceptions, prints appropriate error messages, and gives the user another chance to enter a correct file.

### section\_5/DataAnalyzer.java

```
1 import java.io.FileNotFoundException;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 /**
6     This program reads a file containing numbers and analyzes its contents.
7     If the file doesn't exist or contains strings that are not numbers, an
8     error message is displayed.
9 */
10 public class DataAnalyzer
11 {
12     public static void main(String[] args)
13     {
14         Scanner in = new Scanner(System.in);
15         DataSetReader reader = new DataSetReader();
16
17         boolean done = false;
18         while (!done)
19         {
20             try
21             {
22                 System.out.println("Please enter the file name: ");
23                 String filename = in.next();
24
25                 double[] data = reader.readFile(filename);
26                 double sum = 0;
27                 for (double d : data) { sum = sum + d; }
28                 System.out.println("The sum is " + sum);
29                 done = true;
30             }
31             catch (FileNotFoundException exception)
32             {
```

```

33         System.out.println("File not found.");
34     }
35     catch (BadDataException exception)
36     {
37         System.out.println("Bad data: " + exception.getMessage());
38     }
39     catch (IOException exception)
40     {
41         exception.printStackTrace();
42     }
43 }
44 }
45 }
```

The catch clauses in the `main` method give a human-readable error report if the file was not found or bad data was encountered.

The following `readFile` method of the `DataSetReader` class constructs the `Scanner` object and calls the `readData` method. It is unconcerned with any exceptions. If there is a problem with the input file, it simply passes the exception to its caller.

```

public double[] readFile(String filename) throws IOException
{
    File inFile = new File(filename);
    try (Scanner in = new Scanner(inFile))
    {
        readData(in);
        return data;
    }
}
```

The method throws an `IOException`, the common superclass of `FileNotFoundException` (thrown by the `Scanner` constructor) and `BadDataException` (thrown by the `readData` method).

Next, here is the `readData` method of the `DataSetReader` class. It reads the number of values, constructs an array, and calls `readValue` for each data value.

```

private void readData(Scanner in) throws BadDataException
{
    if (!in.hasNextInt())
    {
        throw new BadDataException("Length expected");
    }
    int numberOfValues = in.nextInt();
    data = new double[numberOfValues];

    for (int i = 0; i < numberOfValues; i++)
    {
        readValue(in, i);
    }

    if (in.hasNext())
    {
        throw new BadDataException("End of file expected");
    }
}
```

This method checks for two potential errors. The file might not start with an integer, or it might have additional data after reading all values.

However, this method makes no attempt to catch any exceptions. Plus, if the `readValue` method throws an exception—which it will if there aren't enough values in the file—the exception is simply passed on to the caller.

Here is the `readValue` method:

```
private void readValue(Scanner in, int i) throws BadDataException
{
    if (!in.hasNextDouble())
    {
        throw new BadDataException("Data value expected");
    }
    data[i] = in.nextDouble();
}
```

To see the exception handling at work, look at a specific error scenario:

1. `DataAnalyzer.main` calls `DataSetReader.readFile`.
2. `readFile` calls `readData`.
3. `readData` calls `readValue`.
4. `readValue` doesn't find the expected value and throws a `BadDataException`.
5. `readValue` has no handler for the exception and terminates immediately.
6. `readData` has no handler for the exception and terminates immediately.
7. `readFile` has no handler for the exception and terminates immediately after closing the `Scanner` object.
8. `DataAnalyzer.main` has a handler for a `BadDataException`. That handler prints a message to the user. Afterward, the user is given another chance to enter a file name. Note that the statements computing the sum of the values have been skipped.

This example shows the separation between error detection (in the `DataSetReader.readValue` method) and error handling (in the `DataAnalyzer.main` method). In between the two are the `readData` and `readFile` methods, which just pass exceptions along.

### **section\_5/DataSetReader.java**

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 /**
6     Reads a data set from a file. The file must have the format
7     numberOfValues
8     value1
9     value2
10    ...
11 */
12 public class DataSetReader
13 {
14     private double[] data;
15
16 /**
17     Reads a data set.
18     @param filename the name of the file holding the data
19     @return the data in the file
20 */
21     public double[] readFile(String filename) throws IOException
22 {
```

```

23     File inFile = new File(filename);
24     try (Scanner in = new Scanner(inFile))
25     {
26         readData(in);
27         return data;
28     }
29 }
30
31 /**
32  * Reads all data.
33  * @param in the scanner that scans the data
34 */
35 private void readData(Scanner in) throws BadDataException
36 {
37     if (!in.hasNextInt())
38     {
39         throw new BadDataException("Length expected");
40     }
41     int numberofValues = in.nextInt();
42     data = new double[numberofValues];
43
44     for (int i = 0; i < numberofValues; i++)
45     {
46         readValue(in, i);
47     }
48
49     if (in.hasNext())
50     {
51         throw new BadDataException("End of file expected");
52     }
53 }
54
55 /**
56  * Reads one data value.
57  * @param in the scanner that scans the data
58  * @param i the position of the value to read
59 */
60 private void readValue(Scanner in, int i) throws BadDataException
61 {
62     if (!in.hasNextDouble())
63     {
64         throw new BadDataException("Data value expected");
65     }
66     data[i] = in.nextDouble();
67 }
68 }
```

### section\_5/BadDataException.java

```

1 import java.io.IOException;
2
3 /**
4  * This class reports bad input data.
5 */
6 public class BadDataException extends IOException
7 {
8     public BadDataException() {}
9     public BadDataException(String message)
10    {
11        super(message);
12    }
13}
```

12      }  
13      }

**SELF CHECK**

24. Why doesn't the `DataSetReader.readFile` method catch any exceptions?
25. Suppose the user specifies a file that exists and is empty. Trace the flow of execution.
26. If the `readValue` method had to throw a `NoSuchElementException` instead of a `BadDataException` when the next input isn't a floating-point number, how would the implementation change?
27. What happens to the `Scanner` object if the `readData` method throws an exception?
28. What happens to the `Scanner` object if the `readData` method doesn't throw an exception?

**Practice It** Now you can try these exercises at the end of the chapter: R11.16, R11.17, E11.13.



## Computing & Society 11.2 The Ariane Rocket Incident

The European Space Agency (ESA), Europe's counterpart to NASA, had developed a rocket model named Ariane that it had successfully used several times to launch satellites and scientific experiments into space. However, when a new version, the Ariane 5, was launched on June 4, 1996, from ESA's launch site in Kourou, French Guiana, the rocket veered off course about 40 seconds after liftoff. Flying at an angle of more than 20 degrees, rather than straight up, exerted such an aerodynamic force that the boosters separated, which triggered the automatic self-destruction mechanism. The rocket blew itself up.

The ultimate cause of this accident was an unhandled exception! The rocket contained two identical devices (called inertial reference systems) that processed flight data from measuring devices and turned the data into information about the rocket position.

The onboard computer used the position information for controlling the boosters. The same inertial reference systems and computer software had worked fine on the Ariane 4.

However, due to design changes to the rocket, one of the sensors measured a larger acceleration force than had been encountered in the Ariane 4. That value, expressed as a floating-point value, was stored in a 16-bit integer (like a short variable in Java). Unlike Java, the Ada language, used for the device software, generates an exception if a floating-point number is too large to be converted to an integer. Unfortunately, the programmers of the device had decided that this situation would never happen and didn't provide an exception handler.

When the overflow did happen, the exception was triggered and, because there was no handler, the device shut itself off. The onboard computer sensed

the failure and switched over to the backup device. However, that device had shut itself off for exactly the same reason, something that the designers of the rocket had not expected. They figured that the devices might fail for mechanical reasons, but the chance of them having the same mechanical failure was remote. At that point, the rocket was without reliable position information and went off course.

Perhaps it would have been better if the software hadn't been so thorough? If it had ignored the overflow, the device wouldn't have been shut off. It would have computed bad data. But then the device would have reported wrong position data, which could have been just as fatal. Instead, a correct implementation should have caught overflow exceptions and come up with some strategy to recompute the flight data. Clearly, giving up was not a reasonable option in this context.

The advantage of the exception-handling mechanism is that it makes these issues explicit to programmers—something to think about when you curse the Java compiler for complaining about uncaught exceptions.



*The Explosion of the Ariane Rocket*

## CHAPTER SUMMARY

### Develop programs that read and write files.

- Use the Scanner class for reading text files.
- When writing text files, use the PrintWriter class and the print/println/printf methods.
- Close all files when you are done processing them.

### Be able to process text in files.

- The next method reads a string that is delimited by white space.
- The Character class has methods for classifying characters.
- The nextLine method reads an entire line.
- If a string contains the digits of a number, you use the Integer.parseInt or Double.parseDouble method to obtain the number value.

### Process the command line arguments of a program.



- Programs that start from the command line receive the command line arguments in the main method.

### Use exception handling to transfer control from an error location to an error handler.



- To signal an exceptional condition, use the throw statement to throw an exception object.
- When you throw an exception, processing continues in an exception handler.
- Place the statements that can cause an exception inside a try block, and the handler inside a catch clause.
- Checked exceptions are due to external circumstances that the programmer cannot prevent. The compiler checks that your program handles these exceptions.
- Add a throws clause to a method that can throw a checked exception.
- The try-with-resources statement ensures that a resource is closed when the statement ends normally or due to an exception.
- To describe an error condition, provide a subclass of an existing exception class.
- Throw an exception as soon as a problem is detected. Catch it only when the problem can be handled.



### Use exception handling in a program that processes input.

- When designing a program, ask yourself what kinds of exceptions can occur.
- For each exception, you need to decide which part of your program can competently handle it.

## STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

|                                    |                                  |
|------------------------------------|----------------------------------|
| java.io.File                       | java.lang.RuntimeException       |
| java.io.FileNotFoundException      | java.lang.String                 |
| java.io.IOException                | replaceAll                       |
| java.io.PrintWriter                | split                            |
| close                              | java.lang.Throwable              |
| java.lang.AutoCloseable            | getMessage                       |
| java.lang.Character                | printStackTrace                  |
| isDigit                            | java.net.URL                     |
| isLetter                           | openStream                       |
| isLowerCase                        | java.util.InputMismatchException |
| isUpperCase                        | java.util.NoSuchElementException |
| isWhiteSpace                       | java.util.Scanner                |
| java.lang.Double                   | close                            |
| parseDouble                        | hasNextLine                      |
| java.lang.Error                    | nextLine                         |
| java.lang.Integer                  | useDelimiter                     |
| parseInt                           | javax.swing.JFileChooser         |
| java.lang.IllegalArgumentException | getSelectedFile                  |
| java.lang.NullPointerException     | showOpenDialog                   |
| java.lang.NumberFormatException    | showSaveDialog                   |

## REVIEW EXERCISES

- ■ **R11.1** What happens if you try to open a file for reading that doesn't exist? What happens if you try to open a file for writing that doesn't exist?
- ■ **R11.2** What happens if you try to open a file for writing, but the file or device is write-protected (sometimes called read-only)? Try it out with a short test program.
- **R11.3** What happens when you write to a `PrintWriter` without closing it? Produce a test program that demonstrates how you can lose data.
- **R11.4** How do you open a file whose name contains a backslash, like `c:\temp\output.dat`?
- **R11.5** If a program `Woozle` is started with the command  

```
java Woozle -Dname=piglet -I\eeeyore -v heff.txt a.txt lump.txt
```

what are the values of `args[0]`, `args[1]`, and so on?
- **R11.6** What is the difference between throwing an exception and catching an exception?
- **R11.7** What is a checked exception? What is an unchecked exception? Give an example for each. Which exceptions do you need to declare with the `throws` reserved word?
- ■ **R11.8** Why don't you need to declare that your method might throw an `IndexOutOfBoundsException`?
- ■ **R11.9** When your program executes a `throw` statement, which statement is executed next?
- ■ **R11.10** What happens if an exception does not have a matching `catch` clause?
- ■ **R11.11** What can your program do with the exception object that a `catch` clause receives?
- ■ **R11.12** Is the type of the exception object always the same as the type declared in the `catch` clause that catches it? If not, why not?

- **R11.13** What is the purpose of the try-with-resources statement? Give an example of how it can be used.
- **R11.14** What happens when an exception is thrown, a try-with-resources statement calls close, and that call throws an exception of a different kind than the original one? Which one is caught by a surrounding catch clause? Write a sample program to try it out.
- **R11.15** Which exceptions can the next and nextInt methods of the Scanner class throw? Are they checked exceptions or unchecked exceptions?
- **R11.16** Suppose the program in Section 11.5 reads a file containing the following values:

```
1
2
3
4
```

What is the outcome? How could the program be improved to give a more accurate error report?

- **R11.17** Can the readFile method in Section 11.5 throw a NullPointerException? If so, how?
- **R11.18** The following code tries to close the writer without using a try-with-resources statement:

```
PrintWriter out = new PrintWriter(filename);
try
{
    Write output.
    out.close();
}
catch (IOException exception)
{
    out.close();
}
```

What is the disadvantage of this approach? (*Hint:* What happens when the PrintWriter constructor or the close method throws an exception?)

## PRACTICE EXERCISES

- **E11.1** Write a program that carries out the following tasks:
  - Open a file with the name hello.txt.**
  - Store the message "Hello, World!" in the file.**
  - Close the file.**
  - Open the same file again.**
  - Read the message into a string variable and print it.**
- **E11.2** Write a program that reads a file, removes any blank lines, and writes the non-blank lines back to the same file.
- **E11.3** Write a program that reads a file, removes any blank lines at the beginning or end of the file, and writes the remaining lines back to the same file.
- **E11.4** Write a program that reads a file containing text. Read each line and send it to the output file, preceded by *line numbers*. If the input file is

```

Mary had a little lamb
Whose fleece was white as snow.
And everywhere that Mary went,
The lamb was sure to go!

```

then the program produces the output file

```

/* 1 */ Mary had a little lamb
/* 2 */ Whose fleece was white as snow.
/* 3 */ And everywhere that Mary went,
/* 4 */ The lamb was sure to go!

```



© Chris Price/iStockphoto.

The line numbers are enclosed in /\* \*/ delimiters so that the program can be used for numbering Java source files.

Prompt the user for the input and output file names.

- **E11.5** Repeat Exercise E11.4, but allow the user to specify the file name on the command-line. If the user doesn't specify any file name, then prompt the user for the name.
- **E11.6** Write a program that reads a file containing two columns of floating-point numbers. Prompt the user for the file name. Print the average of each column.
- **E11.7** Write a program that asks the user for a file name and prints the number of characters, words, and lines in that file.
- **E11.8** Write a program `Find` that searches all files specified on the command line and prints out all lines containing a specified word. For example, if you call

```
java Find ring report.txt address.txt Homework.java
```

then the program might print

```

report.txt: has broken up an international ring of DVD bootleggers that
address.txt: Kris Kringle, North Pole
address.txt: Homer Simpson, Springfield
Homework.java: String filename;

```

The specified word is always the first command line argument.

- **E11.9** Write a program that checks the spelling of all words in a file. It should read each word of a file and check whether it is contained in a word list. A word list is available on Macintosh and Linux systems in the file /usr/share/dict/words. (If you don't have that file on your computer, use the file ch19/words.txt in the companion code for this book.) The program should print out all words that it cannot find in the word list.
- **E11.10** Write a program that replaces each line of a file with its reverse. For example, if you run

```
java Reverse HelloPrinter.java
```

then the contents of `HelloPrinter.java` are changed to

```

retnirPolleH ssalc cilbup
{
)sgra ][gnirtS(niam diov citats cilbup
{
wodniw elosnoc eht ni gniteerg a yalpsiD //

;)"!dlrow ,olleH"(nltinirp.tuo.metsyS
}
}

```

Of course, if you run `Reverse` twice on the same file, you get back the original file.

- E11.11** Write a program that reads each line in a file, reverses its lines, and writes them to another file. For example, if the file `input.txt` contains the lines

```
Mary had a little lamb
Its fleece was white as snow
And everywhere that Mary went
The Lamb was sure to go.
```

and you run

```
reverse input.txt output.txt
```

then `output.txt` contains

```
The Lamb was sure to go.
And everywhere that Mary went
Its fleece was white as snow
Mary had a little lamb
```

- E11.12** Get the data for names in prior decades from the Social Security Administration. Paste the table data in files named `babynames80s.txt`, etc. Modify the `worked_example_1/BabyNames.java` program so that it prompts the user for a file name. The numbers in the files have comma separators, so modify the program to handle them. Can you spot a trend in the frequencies?

- E11.13** Write a program that asks the user to input a set of floating-point values. When the user enters a value that is not a number, give the user a second chance to enter the value. After two chances, quit reading input. Add all correctly specified values and print the sum when the user is done entering data. Use exception handling to detect improper inputs.

- E11.14** Modify the `BankAccount` class to throw an `IllegalArgumentException` when the account is constructed with a negative balance, when a negative amount is deposited, or when an amount that is not between 0 and the current balance is withdrawn. Write a test program that causes all three exceptions to occur and that catches them all.

- E11.15** Repeat Exercise E11.14, but throw exceptions of three exception types that you provide.

- E11.16** Modify the `DataSetReader` class so that you do not call `hasNextInt` or `hasNextDouble`. Simply have `nextInt` and `nextDouble` throw a `NoSuchElementException` and catch it in the `main` method.

## PROGRAMMING PROJECTS

- P11.1** Write a program that reads in `worked_example_1/babynames.txt` and produces two files, `boynames.txt` and `girlnames.txt`, separating the data for the boys and girls.

- P11.2** Write a program that reads a file in the same format as `worked_example_1/babynames.txt` and prints all names that are both boy and girl names (such as Alexis or Morgan).

- P11.3** Using the mechanism described in Special Topic 11.1, write a program that reads all data from a web page and writes them to a file. Prompt the user for the web page URL and the file.

- P11.4** The CSV (or *comma-separated values*) format is commonly used for tabular data. Each table row is a line, with columns separated by commas. Items may be enclosed

in quotation marks, and they must be if they contain commas or quotation marks. Quotation marks inside quoted fields are doubled. Here is a line with four fields:

```
1729, San Francisco, "Hello, World", "He asked: ""Quo vadis?"""
```

Implement a class CSVReader that reads a CSV file, and provide methods

```
int numberOfRows()
int numberOfFields(int row)
String field(int row, int column)
```

- P11.5** Find an interesting data set in CSV format (or in spreadsheet format, save then use a spreadsheet to save the data as CSV). Using the CSVReader class from the Exercise P11.4, read the data and compute a summary, such as the maximum, minimum, or average of one of the columns.
- P11.6** Using the mechanism described in Special Topic 11.1, write a program that reads all data from a web page and prints all hyperlinks of the form

```
<a href="link">link text</a>
```

Extra credit if your program can follow the links that it finds and find links in those web pages as well. (This is the method that search engines such as Google use to find web sites.)

- P11.7** Write a program that reads in a set of coin descriptions from a file. The input file has the format

```
coinName1 coinValue1
coinName2 coinValue2
...
```

Add a method

```
void read(Scanner in) throws FileNotFoundException
```

to the Coin class of Section 8.2. Throw an exception if the current line is not properly formatted. Then implement a method

```
static ArrayList<Coin> readFile(String filename) throws FileNotFoundException
```

In the main method, call readFile. If an exception is thrown, give the user a chance to select another file. If you read all coins successfully, print the total value.

- P11.8** Design a class Bank that contains a number of bank accounts. Each account has an account number and a current balance. Add an accountNumber field to the BankAccount class. Store the bank accounts in an array list. Write a readfile method of the Bank class for reading a file with the format

```
accountNumber1 balance1
accountNumber2 balance2
...
```

Implement read methods for the Bank and BankAccount classes. Write a sample program to read in a file with bank accounts, then print the account with the highest balance. If the file is not properly formatted, give the user a chance to select another file.

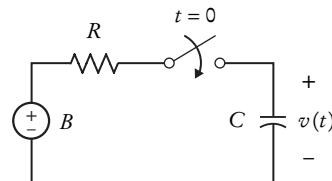
- Business P11.9** A hotel salesperson enters sales in a text file. Each line contains the following, separated by semicolons: The name of the client, the service sold (such as Dinner, Conference, Lodging, and so on), the amount of the sale, and the date of that event. Write a program that reads such a file and displays the total amount for each service category. Display an error if the file does not exist or the format is incorrect.

**Business P11.10** Write a program that reads a text file as described in Exercise P11.9, and that writes a separate file for each service category, containing the entries for that category. Name the output files Dinner.txt, Conference.txt, and so on.

**Business P11.11** A store owner keeps a record of daily cash transactions in a text file. Each line contains three items: The invoice number, the cash amount, and the letter P if the amount was paid or R if it was received. Items are separated by spaces. Write a program that prompts the store owner for the amount of cash at the beginning and end of the day, and the name of the file. Your program should check whether the actual amount of cash at the end of the day equals the expected value.

**Science P11.12** After the switch in the figure below closes, the voltage (in volts) across the capacitor is represented by the equation

$$v(t) = B \left(1 - e^{-t/(RC)}\right)$$



Suppose the parameters of the electric circuit are  $B = 12$  volts,  $R = 500 \Omega$ , and  $C = 0.25 \mu\text{F}$ . Consequently

$$v(t) = 12 \left(1 - e^{-0.008t}\right)$$

where  $t$  has units of  $\mu\text{s}$ . Read a file params.txt containing the values for  $B$ ,  $R$ ,  $C$ , and the starting and ending values for  $t$ . Write a file rc.txt of values for the time  $t$  and the corresponding capacitor voltage  $v(t)$ , where  $t$  goes from the given starting value to the given ending value in 100 steps. In our example, if  $t$  goes from 0 to 1,000  $\mu\text{s}$ , the twelfth entry in the output file would be:

110 7.02261

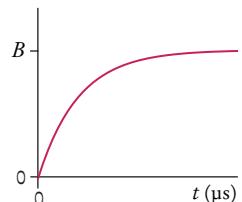
**Science P11.13** The figure at right shows a plot of the capacitor voltage from the circuit shown in Exercise P11.12. The capacitor voltage increases from 0 volts to  $B$  volts. The “rise time” is defined as the time required for the capacitor voltage to change from  $v_1 = 0.05 \times B$  to  $v_2 = 0.95 \times B$ .

The file rc.txt contains a list of values of time  $t$  and the corresponding capacitor voltage  $v(t)$ . A time in  $\mu\text{s}$  and the corresponding voltage in volts are printed on the same line. For example, the line

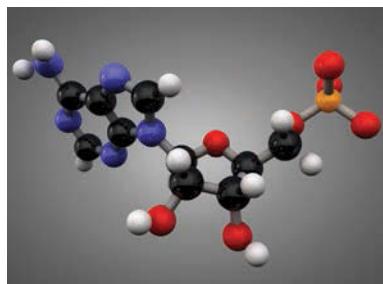
110 7.02261

indicates that the capacitor voltage is 7.02261 volts when the time is 110  $\mu\text{s}$ . The time is increasing in the data file.

Write a program that reads the file rc.txt and uses the data to calculate the rise time. Approximate  $B$  by the voltage in the last line of the file, and find the data points that are closest to  $0.05 \times B$  and  $0.95 \times B$ .



**Science P11.14** Suppose a file contains bond energies and bond lengths for covalent bonds in the following format:



© Chris Dascher/iStockphoto.

| Single, double, or triple bond | Bond energy (kJ/mol) | Bond length (nm) |
|--------------------------------|----------------------|------------------|
| C C                            | 370                  | 0.154            |
| C  C                           | 680                  | 0.13             |
| C   C                          | 890                  | 0.12             |
| C H                            | 435                  | 0.11             |
| C N                            | 305                  | 0.15             |
| C O                            | 360                  | 0.14             |
| C F                            | 450                  | 0.14             |
| C Cl                           | 340                  | 0.18             |
| O H                            | 500                  | 0.10             |
| O O                            | 220                  | 0.15             |
| O Si                           | 375                  | 0.16             |
| N H                            | 430                  | 0.10             |
| N O                            | 250                  | 0.12             |
| F F                            | 160                  | 0.14             |
| H H                            | 435                  | 0.074            |

Write a program that accepts data from one column and returns the corresponding data from the other columns in the stored file. If input data matches different rows, then return all matching row data. For example, a bond length input of 0.12 should return triple bond C|||C and bond energy 890 kJ/mol *and* single bond N|O and bond energy 250 kJ/mol.

### ANSWERS TO SELF-CHECK QUESTIONS

1. When the `PrintWriter` object is created, the output file is emptied. Sadly, that is the same file as the input file. The input file is now empty and the `while` loop exits immediately.

2. The program throws a `FileNotFoundException` and terminates.

3. **Open a scanner for the file.**  
**For each number in the scanner**  
**Add the number to an array.**  
**Close the scanner.**  
**Set total to 0.**

**Open a print writer for the file.**

**For each number in the array**

**Write the number to the print writer.**

**Add the number to total.**

**Write total to the print writer.**

**Close the print writer.**

4. Add a variable count that is incremented whenever a number is read. At the end, print the average, not the total, as

```
out.printf("Average: %.2f\n", total / count);
```

Because the string "Average" is three characters longer than "Total", change the other output to `out.printf("%18.2f\n", value)`.

5. Add a variable count that is incremented whenever a number is read. Only write a new line when it is even.

```
count++;
out.printf("%8.2f", value);
if (count % 2 == 0) { out.println(); }
```

At the end of the loop, write a new line if count is odd, then write the total:

```
if (count % 2 == 1) { out.println(); }
out.printf("Total: %10.2f\n", total);
```

6. word is "Hello," and input is "World!"  
 7. Because 995.0 is not an integer, the call `in.hasNextInt()` returns false, and the call `in.nextInt()` is skipped. The value of `number` stays 0, and `input` is set to the string "995.0".  
 8. `x1` is set to 6000000. Because a dollar sign is not considered a part of a floating-point number in Java, the second call to `nextDouble` causes an input mismatch exception and `x2` is not set.  
 9. Read them as strings, and convert those strings to numbers that are not equal to N/A:

```
String input = in.next();
if (!input.equals("N/A"))
{
    double value = Double.parseDouble(input);
    Process value.
}
```

10. Locate the last character of the country name:

```
int j = i - 1;
while (Character.isWhiteSpace(line.charAt(j)))
{
    j--;
}
```

Then extract the country name:

```
String countryName = line.substring(0, j + 1);
```

11. `args[0]` is "-d" and `args[1]` is "file1.txt"

- 12.

| key | inFile    | outFile | i | arg       |
|-----|-----------|---------|---|-----------|
| 3   | null      | null    | 0 | -d        |
| -3  | file1.txt |         | 1 | file1.txt |
|     |           |         | 2 |           |
|     |           |         |   |           |

Then the program prints a message

Usage: java CaesarCipher [-d] infile outfile

13. The program will run correctly. The loop that parses the options does not depend on the positions in which the options appear.

14. FDHVDU

15. Add the lines

```
else if (option == 'k')
{
    key = Integer.parseInt(
        args[i].substring(2));
}
```

after line 27 and update the usage information.

16. It is still 100. The last statement was not executed because the exception was thrown.

17. if (amount < 0)
{
 throw new IllegalArgumentException(
 "Negative amount");
}

18. The Scanner constructor succeeds because the file exists. The `nextInt` method throws a `NoSuchElementException`. This is *not* an `IOException`. Therefore, the error is not caught. Because there is no other handler, an error message is printed and the program terminates.

19. Because programmers should simply check that their array index values are valid instead of trying to handle an `ArrayIndexOutOfBoundsException`.

20. No. You can catch both exception types in the same way, as you can see in the code example on page 542.

21. There are two mistakes. The `PrintWriter` constructor can throw a `FileNotFoundException`. You should supply a `throws` clause. And if one of the array elements is `null`, a `NullPointerException` is thrown. In that case, the `out.close()` statement is never executed. You should use a `try-with-resources` statement.

22. To pass the exception message string to the `IllegalArgumentException` superclass.

23. Because file corruption is beyond the control of the programmer, this should be a checked exception, so it would be wrong to extend `RuntimeException` or `IllegalArgumentException`. Because the error is related to input, `IOException` would be a good choice.

24. It would not be able to do much with them. The `DataSetReader` class is a reusable class that

may be used for systems with different languages and different user interfaces. Thus, it cannot engage in a dialog with the program user.

25. `DataAnalyzer.main` calls `DataSetReader.readFile`, which calls `readData`. The call `in.hasNextInt()` returns `false`, and `readData` throws a `BadDataException`. The `readFile` method doesn't catch it, so it propagates back to `main`, where it is caught.
26. It could simply be

```
private void readValue(Scanner in, int i)
{
    data[i] = in.nextDouble();
}
```

The `nextDouble` method throws a `NoSuchElementException` or an `InputMismatchException` (which is a subclass of `NoSuchElementException`) when the next input isn't a floating-point number. That exception isn't a checked exception, so it need not be declared.

27. The `close` method is called on the `Scanner` object before the exception is propagated to its handler.
28. The `close` method is called on the `Scanner` object before the `readFile` method returns to its caller.



## WORKED EXAMPLE 11.1

## Analyzing Baby Names



**Problem Statement** The Social Security Administration publishes lists of the most popular baby names on their web site <http://www.ssa.gov/OACT/babynames/>. When querying the 1,000 most popular names for a given decade, the browser displays the result on the screen (see the *Querying Baby Names* figure below).

To save the data as text, one simply selects it and pastes the result into a file. The book's code contains a file `babynames.txt` with the data for the 1990s.

Each line in the file contains seven entries:

- The rank (from 1 to 1,000)
- The name, frequency, and percentage of the male name of that rank
- The name, frequency, and percentage of the female name of that rank

For example, the line

```
10 Joseph 260365 1.2681 Megan 160312 0.8168
```

shows that the 10th most common boy's name was Joseph, with 260,365 births, or 1.2681 percent of all births during that period. The 10th most common girl's name was Megan. Why are there many more Josephs than Megans? Parents seem to use a wider set of girl's names, making each one of them less frequent.

Your task is to test that conjecture, by determining the names given to the top 50 percent of boys and girls in the list. Simply print boy and girl names, together with their ranks, until you reach the 50 percent limit.



© Nancy Ross/Stockphoto.

### Step 1 Understand the processing task.

To process each line, we first read the rank. We then read three values (name, count, and percentage) for the boy's name. Then we repeat that step for girls. To stop processing after reaching 50 percent, we can add up the frequencies and stop when they reach 50 percent.

We need separate totals for boys and girls. When a total reaches 50 percent, we stop printing. When both totals reach 50 percent, we stop reading.

The following pseudocode describes our processing task.

```
boyTotal = 0
girlTotal = 0
While boyTotal < 50 or girlTotal < 50
    Read the rank and print it.

    Read the boy name, count, and percentage.
    If boyTotal < 50
        Print boy name.
        Add percentage to boyTotal.

    Repeat for girl part.
```

### Step 2 Determine which files you need to read and write.

We only need to read a single file, `babynames.txt`. We were not asked to save the output to a file, so we will just send it to `System.out`.

### Step 3 Choose a mechanism for obtaining the file names.

We do not need to prompt the user for the file name.

| Rank | Name        | Number  | Percent | Name      | Number  | Percent |
|------|-------------|---------|---------|-----------|---------|---------|
| 1    | Michael     | 462,085 | 2.2506  | Jessica   | 302,962 | 1.5436  |
| 2    | Christopher | 381,250 | 1.7595  | Ashley    | 301,702 | 1.5372  |
| 3    | Matthew     | 351,477 | 1.7119  | Emily     | 237,133 | 1.2082  |
| 4    | Joshua      | 328,955 | 1.6022  | Sarah     | 224,000 | 1.1413  |
| 5    | Jacob       | 298,016 | 1.4515  | Samantha  | 223,913 | 1.1408  |
| 6    | Nicholas    | 275,222 | 1.3405  | Amanda    | 190,901 | 0.9726  |
| 7    | Andrew      | 272,600 | 1.3277  | Brittany  | 190,779 | 0.9720  |
| 8    | Daniel      | 271,734 | 1.3235  | Elizabeth | 172,383 | 0.8783  |
| 9    | Tyler       | 262,218 | 1.2771  | Taylor    | 168,977 | 0.8609  |
| 10   | Joseph      | 260,365 | 1.2681  | Megan     | 160,312 | 0.8168  |
| 11   | Brandon     | 259,299 | 1.2629  | Hannah    | 158,647 | 0.8083  |
| 12   | David       | 253,193 | 1.2332  | Kayla     | 155,844 | 0.7940  |

Querying Baby Names

**Step 4** Choose between line, word, and character-based input.

The Social Security Administration data do not contain names with spaces, such as “Mary Jane”. Therefore, each data record contains exactly seven entries, as shown in the screen capture above. This input can be safely processed by reading words and numbers.

**Step 5** With line-oriented input, extract the required data.

We can skip this step because we don’t read a line at a time.

But suppose you decided in Step 4 to choose line-oriented input. Then you would need to break the input line into seven strings, converting five of them to numbers. This is quite tedious and it might well make you reconsider your choice.

**Step 6** Use classes and methods to factor out common tasks.

In the pseudocode, we wrote **Repeat for girl part**. Clearly, there is a common task that calls for a helper method. It involves three tasks:

- Read the name, count, and percentage.
- Print the name if the total is less than 50 percent.
- Add the percentage to the total.

We use a helper class RecordReader for this purpose and construct two objects, one each for processing the boy and girl names. Each RecordReader maintains a separate total, updates it by adding the current percentage, and prints names until the limit has been reached. Our main processing loop then becomes

```
RecordReader boys = new RecordReader(LIMIT);
RecordReader girls = new RecordReader(LIMIT);
```

```

        while (boys.hasMore() || girls.hasMore())
    {
        int rank = in.nextInt();
        System.out.print(rank + " ");
        boys.process(in);
        girls.process(in);
        System.out.println();
    }
}

```

Here is the code of the `process` method:

```

/*
 * Reads an input record and prints the name if the current total is less than the limit.
 * @param in the input stream
 */
public void process(Scanner in)
{
    String name = in.next();
    int count = in.nextInt();
    double percent = in.nextDouble();

    if (total < limit) { System.out.print(name + " "); }
    total = total + percent;
}


```

The complete program is shown below.

Have a look at the program output. Remarkably, only 69 boy names and 153 girl names account for half of all births. That's good news for those who are in the business of producing personalized doodads. Exercise E11.12 asks you to study how this distribution has changed over the years.

### [worked\\_example\\_1/BabyNames.java](#)

```

1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.Scanner;
4
5 public class BabyNames
6 {
7     public static final double LIMIT = 50;
8
9     public static void main(String[] args) throws FileNotFoundException
10    {
11        try (Scanner in = new Scanner(new File("babynames.txt")))
12        {
13            RecordReader boys = new RecordReader(LIMIT);
14            RecordReader girls = new RecordReader(LIMIT);
15
16            while (boys.hasMore() || girls.hasMore())
17            {
18                int rank = in.nextInt();
19                System.out.print(rank + " ");
20                boys.process(in);
21                girls.process(in);
22                System.out.println();
23            }
24        }
25    }
26}

```

**worked\_example\_1/RecordReader.java**

```
1 import java.util.Scanner;
2
3 /**
4  * This class processes baby name records.
5 */
6 public class RecordReader
7 {
8     private double total;
9     private double limit;
10
11    /**
12     * Constructs a RecordReader with a zero total.
13     */
14    public RecordReader(double aLimit)
15    {
16        total = 0;
17        limit = aLimit;
18    }
19
20    /**
21     * Reads an input record and prints the name if the current total is less
22     * than the limit.
23     * @param in the input stream
24     */
25    public void process(Scanner in)
26    {
27        String name = in.next();
28        int count = in.nextInt();
29        double percent = in.nextDouble();
30
31        if (total < limit) { System.out.print(name + " "); }
32        total = total + percent;
33    }
34
35    /**
36     * Checks whether there are more inputs to process.
37     * @return true if the limit has not yet been reached
38     */
39    public boolean hasMore()
40    {
41        return total < limit;
42    }
43 }
```

# OBJECT-ORIENTED DESIGN

## CHAPTER GOALS

- To learn how to discover new classes and methods
- To use CRC cards for class discovery
- To identify inheritance, aggregation, and dependency relationships between classes
- To describe class relationships using UML class diagrams
- To apply object-oriented design techniques to building complex programs



© Petrea Alexandru/iStockphoto.

## CHAPTER CONTENTS

### 12.1 CLASSES AND THEIR RESPONSIBILITIES 566

### 12.2 RELATIONSHIPS BETWEEN CLASSES 569

**HT1** Using CRC Cards and UML Diagrams in Program Design 572

**ST1** Attributes and Methods in UML Diagrams 573

**ST2** Multiplicities 574

**ST3** Aggregation, Association, and Composition 574

### 12.3 APPLICATION: PRINTING AN INVOICE 575

**C&S** Databases and Privacy 586

**WE1** Simulating an Automatic Teller Machine 



© Petrea Alexandru/iStockphoto.

Successfully implementing a software system—as simple as your next homework project or as complex as the next air traffic monitoring system—requires a great deal of planning and design. In fact, for larger projects, the amount of time spent on planning and design is much greater than the amount of time spent on programming and testing.

Do you find that most of your homework time is spent in front of the computer, keying in code and fixing bugs? If so, you can probably save time by focusing on a better design before you start coding. This chapter tells you how to approach the design of an object-oriented program in a systematic manner.

## 12.1 Classes and Their Responsibilities

When you design a program, you work from a *requirements specification*, a description of what your program should do. The designer's task is to discover structures that make it possible to implement the requirements in a computer program. In the following sections, we will examine the steps of the design process.

### 12.1.1 Discovering Classes

To discover classes, look for nouns in the problem description.

When you solve a problem using objects and classes, you need to determine the classes required for the implementation. You may be able to reuse existing classes, or you may need to implement new ones.

One simple approach for discovering classes and methods is to look for the nouns and verbs in the requirements specification. Often, *nouns* correspond to classes, and *verbs* correspond to methods.

For example, suppose your job is to print an invoice such as the one in Figure 1.

| INVOICE  |     |         |         |
|--|-----|---------|---------|
| Sam's Small Appliances<br>100 Main Street<br>Anytown, CA 98765 |     |         |         |
| Item   | Qty | Price   | Total   |
| Toaster  | 3   | \$29.95 | \$89.85 |
| Hair Dryer   | 1   | \$24.95 | \$24.95 |
| Car Vacuum   | 2   | \$19.99 | \$39.98 |

AMOUNT DUE: \$154.78

**Figure 1**  
An Invoice

Concepts from the problem domain are good candidates for classes.

Obvious classes that come to mind are `Invoice`, `LineItem`, and `Customer`. It is a good idea to keep a list of *candidate classes* on a whiteboard or a sheet of paper. As you brainstorm, simply put all ideas for classes onto the list. You can always cross out the ones that weren't useful after all.

In general, concepts from the problem domain, be it science, business, or a game, often make good classes. Examples are

- `Cannonball`
- `CashRegister`
- `Monster`

The name for such a class should be a noun that describes the concept.

Not all classes can be discovered from the program requirements. Most complex programs need classes for tactical purposes, such as file or database access, user interfaces, control mechanisms, and so on.

Some of the classes that you need may already exist, either in the standard library or in a program that you developed previously. You also may be able to use inheritance to extend existing classes into classes that match your needs.

A common error is to overdo the class discovery process. For example, should an address be an object of an `Address` class, or should it simply be a string? There is no perfect answer—it depends on the task that you want to solve. If your software needs to analyze addresses (for example, to determine shipping costs), then an `Address` class is an appropriate design. However, if your software will never need such a capability, you should not waste time on an overly complex design. It is your job to find a balanced design; one that is neither too limiting nor excessively general.

### 12.1.2 The CRC Card Method

Once you have identified a set of classes, you define the behavior for each class. Find out what methods you need to provide for each class in order to solve the programming problem. A simple rule for finding these methods is to look for *verbs* in the task description, then match the verbs to the appropriate objects. For example, in the invoice program, a class needs to compute the amount due. Now you need to figure out *which class* is responsible for this method. Do customers compute what they owe? Do invoices total up the amount due? Do the items total themselves up? The best choice is to make “compute amount due” the responsibility of the `Invoice` class.

*In a class scheduling system, potential classes from the problem domain include `Class`, `LectureHall`, `Instructor`, and `Student`.*



© Oleg Prikhodko/Stockphoto.

A CRC card describes a class, its responsibilities, and its collaborating classes.

An excellent way to carry out this task is the “**CRC card** method.” **CRC** stands for “classes”, “responsibilities”, “collaborators”, and in its simplest form, the method works as follows: Use an index card for each *class* (see Figure 2). As you think about verbs in the task description that indicate methods, you pick the card of the class that you think should be responsible, and write that *responsibility* on the card.

For each responsibility, you record which other classes are needed to fulfill it. Those classes are the **collaborators**.

For example, suppose you decide that an invoice should compute the amount due. Then you write “compute amount due” on the left-hand side of an index card with the title **Invoice**.

If a class can carry out that responsibility by itself, do nothing further. But if the class needs the help of other classes, write the names of these collaborators on the right-hand side of the card.

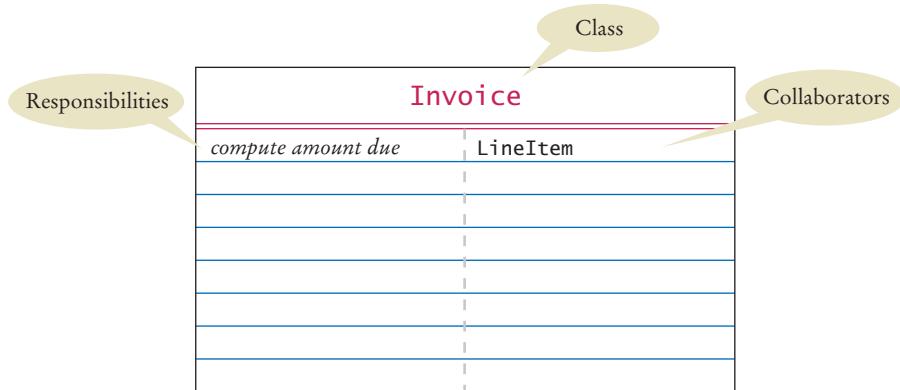
To compute the total, the invoice needs to ask each line item about its total price. Therefore, the `LineItem` class is a collaborator.

This is a good time to look up the index card for the `LineItem` class. Does it have a “get total price” method? If not, add one.

How do you know that you are on the right track? For each responsibility, ask yourself how it can actually be done, using the responsibilities written on the various cards. Many people find it helpful to group the cards on a table so that the collaborators are close to each other, and to simulate tasks by moving a token (such as a coin) from one card to the next to indicate which object is currently active.

Keep in mind that the responsibilities that you list on the CRC card are on a *high level*. Sometimes a single responsibility may need two or more Java methods for carrying it out. Some researchers say that a CRC card should have no more than three distinct responsibilities.

The CRC card method is informal on purpose, so that you can be creative and discover classes and their properties. Once you find that you have settled on a good set of classes, you will want to know how they are related to each other. Can you find classes with common properties, so that some responsibilities can be taken care of by a common superclass? Can you organize classes into clusters that are independent of each other? Finding class relationships and documenting them with diagrams is the topic of Section 12.2.



**Figure 2** A CRC Card

**SELF CHECK**

1. What is the 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`?
3. Suppose the invoice is to be saved to a file. Name a likely collaborator.
4. Looking at the invoice in Figure 1, what is a likely responsibility of the `Customer` class?
5. What do you do if a CRC card has ten responsibilities?

**Practice It** Now you can try these exercises at the end of the chapter: R12.4, R12.8.

## 12.2 Relationships Between Classes

When designing a program, it is useful to document the relationships between classes. This helps you in a number of ways. For example, if you find classes with common behavior, you can save effort by placing the common behavior into a superclass. If you know that some classes are *not* related to each other, you can assign different programmers to implement each of them, without worrying that one of them has to wait for the other.

In the following sections, we will describe the most common types of relationships.

### 12.2.1 Dependency

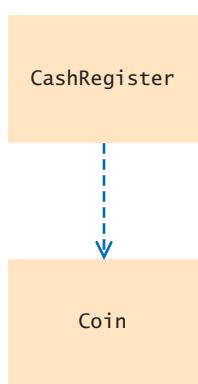
A class depends on another class if it uses objects of that class.

Many classes need other classes in order to do their jobs. For example, in Section 8.2.2, we described a design of a `CashRegister` class that depends on the `Coin` class to determine the value of the payment.

The dependency relationship is sometimes nicknamed the “knows about” relationship. The cash register in Section 8.2.2 knows that there are coin objects. In contrast, the `Coin` class does *not* depend on the `CashRegister` class. Coins have no idea that they are being collected in cash registers, and they can carry out their work without ever calling any method in the `CashRegister` class.

As you saw in Section 8.2, dependency is denoted by a dashed line with a ➤-shaped open arrow tip. The arrow tip points to the class on which the other class depends. Figure 3 shows a class diagram indicating that the `CashRegister` class depends on the `Coin` class.

If many classes of a program depend on each other, then we say that the **coupling** between classes is high. Conversely, if there are few dependencies between classes, then we say that the coupling is low (see Figure 4).



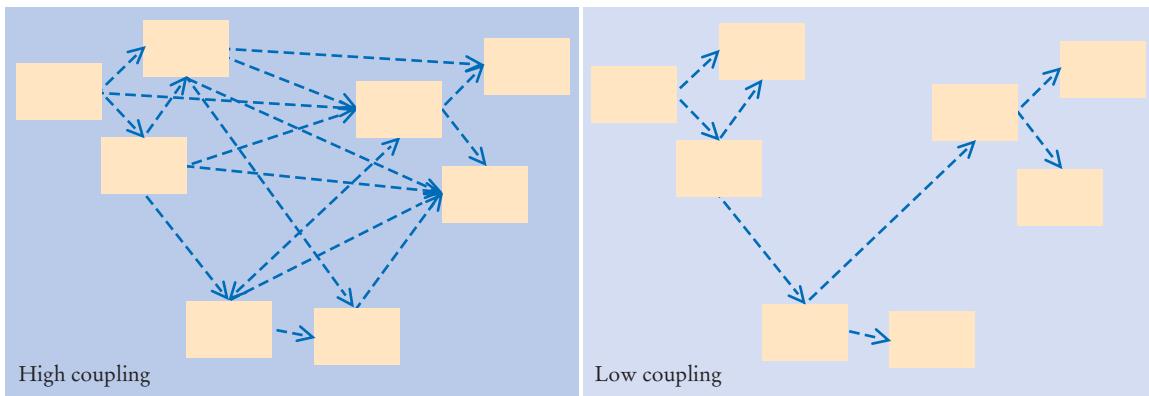
**Figure 3**

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



Too many dependencies make a system difficult to manage.

© visual7/iStockphoto

**Figure 4** High and Low Coupling Between Classes

It is a good practice to minimize the coupling (i.e., dependency) between classes.

A class aggregates another if its objects contain objects of the other class.

Why does coupling matter? If the `Coin` class changes in the next release of the program, all the classes that depend on it may be affected. If the change is drastic, the coupled classes must all be updated. Furthermore, if we would like to use a class in another program, we have to take with it all the classes on which it depends. Thus, we want to remove unnecessary coupling between classes.

## 12.2.2 Aggregation

Another fundamental relationship between classes is the “aggregation” relationship (which is informally known as the “has-a” relationship).

The **aggregation** relationship states that objects of one class contain objects of another class. Consider a quiz that is made up of questions. Because each quiz has one or more questions, we say that the class `Quiz` *aggregates* the class `Question`. In the UML notation, aggregation is denoted by a line with a diamond-shaped symbol attached to the aggregating class (see Figure 5).

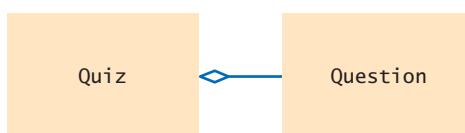
Finding out about aggregation is very helpful for deciding how to implement classes. For example, when you implement the `Quiz` class, you will want to store the questions of a quiz as an instance variable.

Because a quiz can have any number of questions, an array or array list is a good choice for collecting them:

```
public class Quiz
{
    private ArrayList<Question> questions;
    ...
}
```

Aggregation is a stronger form of dependency. If a class has objects of another class, it certainly knows about the other class. However, the converse is not true. For example, a class may use the `Scanner` class without ever declaring an instance variable of

**Figure 5**  
Class Diagram  
Showing Aggregation



A car has a motor and tires.  
In object-oriented design,  
this “has-a” relationship  
is called aggregation.



© bojan fatur/Stockphoto.

class Scanner. The class may simply construct a local variable of type Scanner, or its methods may receive Scanner objects as arguments. This use is not aggregation because the objects of the class don’t contain Scanner objects—they just create or receive them for the duration of a single method.

Generally, you need aggregation when an object needs to remember another object *between method calls*.

### 12.2.3 Inheritance

Inheritance is a relationship between a more general class (the superclass) and a more specialized class (the subclass). This relationship is often described as the “*is-a*” relationship. Every truck *is a* vehicle. Every savings account *is a* bank account.

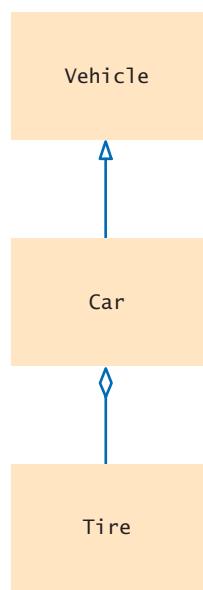
Inheritance is sometimes abused. For example, consider a Tire class that describes a car tire. Should the class Tire be a subclass of a class Circle? It sounds convenient. There are quite a few useful methods in the Circle class—for example, the Tire class may inherit methods that compute the radius, perimeter, and center point, which should come in handy when drawing tire shapes. Though it may be convenient for the programmer, this arrangement makes no sense conceptually. It isn’t true that every tire is a circle. Tires are car parts, whereas circles are geometric objects. There is a relationship between tires and circles, though. A tire *has a* circle as its boundary. Use aggregation:

```
public class Tire
{
    private String rating;
    private Circle boundary;
    ...
}
```

Aggregation (the *has-a* relationship) denotes that objects of one class contain references to objects of another class.

Here is another example: Every car *is a* vehicle. Every car *has a* tire (in fact, it typically has four or, if you count the spare, five). Thus, you would use inheritance from Vehicle and use aggregation of Tire objects (see Figure 6 for the UML diagram):

```
public class Car extends Vehicle
{
    private Tire[] tires;
    ...
}
```



**Figure 6**  
UML Notation for Inheritance and Aggregation

You need to be able to distinguish the UML notation for inheritance, interface implementation, aggregation, and dependency.

The arrows in the UML notation can get confusing. Table 1 shows a summary of the four UML relationship symbols that we use in this book.

**Table 1** UML Relationship Symbols

| Relationship             | Symbol | Line Style | Arrow Tip |
|--------------------------|--------|------------|-----------|
| Inheritance              | →      | Solid      | Triangle  |
| Interface Implementation | →      | Dotted     | Triangle  |
| Aggregation              | ○→     | Solid      | Diamond   |
| Dependency               | →      | Dotted     | Open      |

### SELF CHECK



6. Consider the `CashRegisterTester` class of Section 8.2. On which classes does it depend?
7. Consider the `Question` and `ChoiceQuestion` objects of Chapter 9. How are they related?
8. Consider the `Quiz` class described in Section 12.2.2. Suppose a quiz contains a mixture of `Question` and `ChoiceQuestion` objects. Which classes does the `Quiz` class depend on?
9. Why should coupling be minimized between classes?
10. In an e-mail system, messages are stored in a mailbox. Draw a UML diagram that shows the appropriate aggregation relationship.
11. You are implementing a system to manage a library, keeping track of which books are checked out by whom. Should the `Book` class aggregate `Patron` or the other way around?
12. In a library management system, what would be the relationship between classes `Patron` and `Author`?

**Practice It** Now you can try these exercises at the end of the chapter: R12.5, R12.6, R12.10.

### HOW TO 12.1



### Using CRC Cards and UML Diagrams in Program Design

Before writing code for a complex problem, you need to design a solution. The methodology introduced in this chapter suggests that you follow a design process that is composed of the following tasks:

- Discover classes.
- Determine the responsibilities of each class.
- Describe the relationships between the classes.

CRC cards and UML diagrams help you discover and record this information.

#### Step 1 Discover classes.

Highlight the nouns in the problem description. Make a list of the nouns. Cross out those that don't seem to be reasonable candidates for classes.

**Step 2** Discover responsibilities.

Make a list of the major tasks that your system needs to fulfill. From those tasks, pick one that is not trivial and that is intuitive to you. Find a class that is responsible for carrying out that task. Make an index card and write the name and the task on it. Now ask yourself how an object of the class can carry out the task. It probably needs help from other objects. Then make CRC cards for the classes to which those objects belong and write the responsibilities on them.

Don't be afraid to cross out, move, split, or merge responsibilities. Rip up cards if they become too messy. This is an informal process.

You are done when you have walked through all major tasks and are satisfied that they can all be solved with the classes and responsibilities that you discovered.

**Step 3** Describe relationships.

Make a class diagram that shows the relationships between all the classes that you discovered.

Start with inheritance—the *is-a* relationship between classes. Is any class a specialization of another? If so, draw inheritance arrows. Keep in mind that many designs, especially for simple programs, don't use inheritance extensively.

The “collaborators” column of the CRC card tells you which classes are used by that class. Draw dependency arrows for the collaborators on the CRC cards.

Some dependency relationships give rise to aggregations. For each of the dependency relationships, ask yourself: How does the object locate its collaborator? Does it navigate to it directly because it stores a reference? In that case, draw an aggregation arrow. Or is the collaborator a method parameter variable or return value? Then simply draw a dependency arrow.

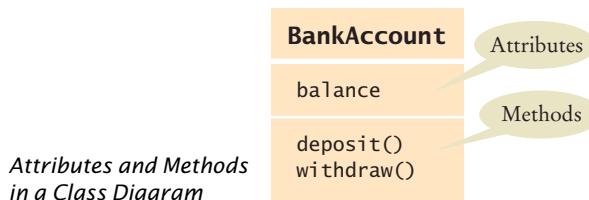
## Special Topic 12.1

**Attributes and Methods in UML Diagrams**

Sometimes it is useful to indicate class *attributes* and *methods* in a class diagram. An **attribute** is an externally observable property that objects of a class have. For example, *name* and *price* would be attributes of the *Product* class. Usually, attributes correspond to instance variables. But they don't have to—a class may have a different way of organizing its data. For example, a *GregorianCalendar* object from the Java library has attributes *day*, *month*, and *year*, and it would be appropriate to draw a UML diagram that shows these attributes. However, the class doesn't actually have instance variables that store these quantities. Instead, it internally represents all dates by counting the milliseconds from January 1, 1970—an implementation detail that a class user certainly doesn't need to know about.

You can indicate attributes and methods in a class diagram by dividing a class rectangle into three compartments, with the class name in the top, attributes in the middle, and methods in the bottom (see the figure below). You need not list *all* attributes and methods in a particular diagram. Just list the ones that are helpful for understanding whatever point you are making with a particular diagram.

Also, don't list as an attribute what you also draw as an aggregation. If you denote by aggregation the fact that a *Car* has *Tire* objects, don't add an attribute *tires*.



**Special Topic 12.2****Multiplicities**

Some designers like to write *multiplicities* at the end(s) of an aggregation relationship to denote how many objects are aggregated. The notations for the most common multiplicities are:

- any number (zero or more): \*
- one or more: 1..\*
- zero or one: 0..1
- exactly one: 1

The figure below shows that a customer has one or more bank accounts.



*An Aggregation Relationship with Multiplicities*

**Special Topic 12.3****Aggregation, Association, and Composition**

Some designers find the aggregation or *has-a* terminology unsatisfactory. For example, consider customers of a bank. Does the bank “have” customers? Do the customers “have” bank accounts, or does the bank “have” them? Which of these “has” relationships should be modeled by aggregation? This line of thinking can lead us to premature implementation decisions.

Early in the design phase, it makes sense to use a more general relationship between classes called **association**. A class is associated with another if you can *navigate* from objects of one class to objects of the other class. For example, given a *Bank* object, you can navigate to *Customer* objects, perhaps by accessing an instance variable, or by making a database lookup.

The UML notation for an association relationship is a solid line, with optional arrows that show in which directions you can navigate the relationship. You can also add words to the line ends to further explain the nature of the relationship. The figure below shows that you can navigate from *Bank* objects to *Customer* objects, but you cannot navigate the other way around. That is, in this particular design, the *Customer* class has no mechanism to determine in which banks it keeps its money.



*An Association Relationship*

The UML standard also recognizes a stronger form of the aggregation relationship called **composition**, where the aggregated objects do not have an existence independent of the containing object. For example, composition models the relationship between a bank and its accounts. If a bank closes, the account objects cease to exist as well. In the UML notation, composition looks like aggregation with a filled-in diamond.



*A Composition Relationship*

Frankly, the differences between aggregation, association, and composition can be confusing, even to experienced designers. If you find the distinction helpful, by all means use the relationship that you find most appropriate. But don't spend time pondering subtle differences between these concepts. From the practical point of view of a Java programmer, it is useful to know when objects of one class have references to objects of another class. The aggregation or *has-a* relationship accurately describes this phenomenon.

## 12.3 Application: Printing an Invoice

In this book, we discuss a five-part program development process that is particularly well suited for beginning programmers:

1. Gather requirements.
2. Use CRC cards to find classes, responsibilities, and collaborators.
3. Use UML diagrams to record class relationships.
4. Use javadoc to document method behavior.
5. Implement your program.

There isn't a lot of notation to learn. The class diagrams are simple to draw. The deliverables of the design phase are obviously useful for the implementation phase—you simply take the source files and start adding the method code. Of course, as your projects get more complex, you will want to learn more about formal design methods. There are many techniques to describe object scenarios, call sequencing, the large-scale structure of programs, and so on, that are very beneficial even for relatively simple projects. *The Unified Modeling Language User Guide* gives a good overview of these techniques.

In this section, we will walk through the object-oriented design technique with a very simple example. In this case, the methodology may feel overblown, but it is a good introduction to the mechanics of each step. You will then be better prepared for the more complex programs that you will encounter in the future.

### 12.3.1 Requirements

Start the development process by gathering and documenting program requirements.

Before you begin designing a solution, you should gather all requirements for your program in plain English. Write down what your program should do. It is helpful to include typical scenarios in addition to a general description.

The task of our sample program is to print out an invoice. An invoice describes the charges for a set of products in certain quantities. (We omit complexities such as dates, taxes, and invoice and customer numbers.) The program simply prints the billing address, all line items, and the amount due. Each line item contains the description and unit price of a product, the quantity ordered, and the total price.



© Scott Cramer/Stockphoto.

An invoice lists the charges for each item and the amount due.

## I N V O I C E

Sam's Small Appliances  
100 Main Street  
Anytown, CA 98765

| Description | Price | Qty | Total |
|-------------|-------|-----|-------|
| Toaster     | 29.95 | 3   | 89.85 |
| Hair dryer  | 24.95 | 1   | 24.95 |
| Car vacuum  | 19.99 | 2   | 39.98 |

AMOUNT DUE: \$154.78

Also, in the interest of simplicity, we do not provide a user interface. We just supply a test program that adds line items to the invoice and then prints it.

### 12.3.2 CRC Cards

Use CRC cards to find classes, responsibilities, and collaborators.

When designing an object-oriented program, you need to discover classes. Classes correspond to nouns in the requirements specification. In this problem, it is pretty obvious what the nouns are:

| Invoice     | Address | LineItem | Product |
|-------------|---------|----------|---------|
| Description | Price   | Quantity | Total   |

(Of course, Toaster doesn't count—it is the description of a LineItem object and therefore a data value, not the name of a class.)

Description and price are attributes of the Product class. What about the quantity? The quantity is not an attribute of a Product. Just as in the printed invoice, let's have a class LineItem that records the product and the quantity (such as "3 toasters").

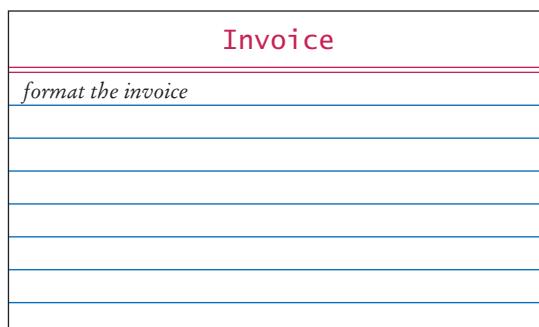
The total and amount due are computed—not stored anywhere. Thus, they don't lead to classes.

After this process of elimination, we are left with four candidates for classes:

|         |         |          |         |
|---------|---------|----------|---------|
| Invoice | Address | LineItem | Product |
|---------|---------|----------|---------|

Each of them represents a useful concept, so let's make them all into classes.

The purpose of the program is to print an invoice. However, the Invoice class won't necessarily know whether to display the output in `System.out`, in a text area, or in a file. Therefore, let's relax the task slightly and make the invoice responsible for *formatting* the invoice. The result is a string (containing multiple lines) that can be printed out or displayed. Record that responsibility on a CRC card:

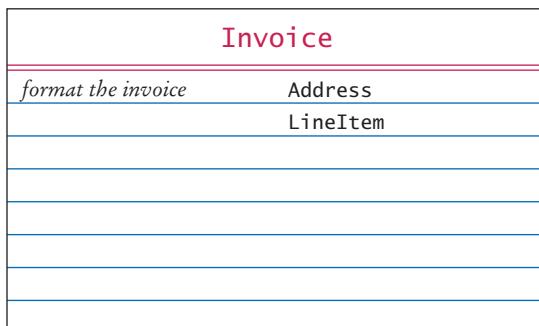


How does an invoice format itself? It must format the billing address, format all line items, and then add the amount due. How can the invoice format an address? It can't—that really is the responsibility of the `Address` class. This leads to a second CRC card:



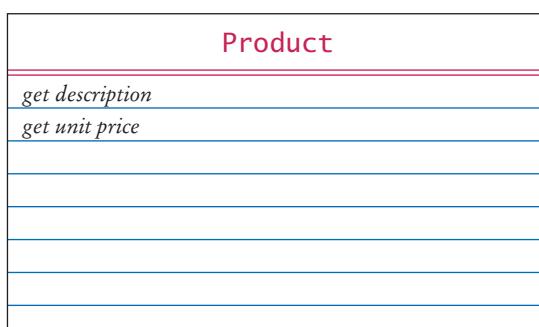
Similarly, formatting of a line item is the responsibility of the `LineItem` class.

The `format` method of the `Invoice` class calls the `format` methods of the `Address` and `LineItem` classes. Whenever a method uses another class, you list that other class as a collaborator. In other words, `Address` and `LineItem` are collaborators of `Invoice`:



When formatting the invoice, the invoice also needs to compute the total amount due. To obtain that amount, it must ask each line item about the total price of the item.

How does a line item obtain that total? It must ask the product for the unit price, and then multiply it by the quantity. That is, the `Product` class must reveal the unit price, and it is a collaborator of the `LineItem` class.



| LineItem        |         |
|-----------------|---------|
| format the item | Product |
| get total price |         |
|                 |         |
|                 |         |
|                 |         |
|                 |         |
|                 |         |

Finally, the invoice must be populated with products and quantities, so that it makes sense to format the result. That too is a responsibility of the `Invoice` class.

| Invoice                    |          |
|----------------------------|----------|
| format the invoice         | Address  |
| add a product and quantity | LineItem |
|                            | Product  |
|                            |          |
|                            |          |
|                            |          |
|                            |          |

We now have a set of CRC cards that completes the CRC card process.

### 12.3.3 UML Diagrams

Use UML diagrams to record class relationships.

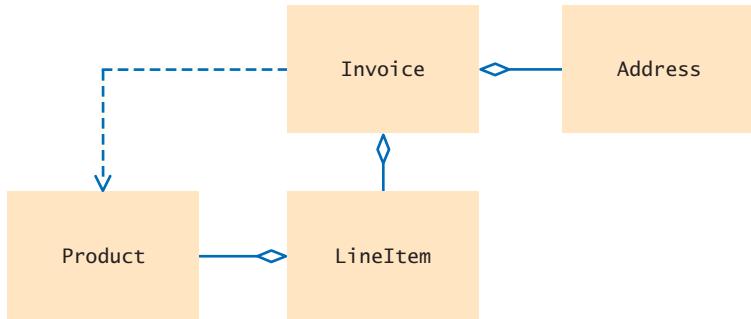
After you have discovered classes and their relationships with CRC cards, you should record your findings in a UML diagram. The dependency relationships come from the collaboration column on the CRC cards. Each class depends on the classes with which it collaborates. In our example, the `Invoice` class collaborates with the `Address`, `LineItem`, and `Product` classes. The `LineItem` class collaborates with the `Product` class.

Now ask yourself which of these dependencies are actually aggregations. How does an invoice know about the address, line item, and product objects with which it collaborates? An invoice object must hold references to the address and the line items when it formats the invoice. But an invoice object need not hold a reference to a product object when adding a product. The product is turned into a line item, and then it is the item's responsibility to hold a reference to it.

Therefore, the `Invoice` class aggregates the `Address` and `LineItem` classes. The `LineItem` class aggregates the `Product` class. However, there is no *has-a* relationship between an invoice and a product. An invoice doesn't store products directly—they are stored in the `LineItem` objects.

There is no inheritance in this example.

Figure 7 shows the class relationships that we discovered.



**Figure 7** The Relationships Between the Invoice Classes

### 12.3.4 Method Documentation

Use javadoc comments (with the method bodies left blank) to record the behavior of classes.

The final step of the design phase is to write the documentation of the discovered classes and methods. Simply write a Java source file for each class, write the method comments for those methods that you have discovered, and leave the bodies of the methods blank.

```


    /**
     * Describes an invoice for a set of purchased products.
     */
    public class Invoice
    {
        /**
         * Adds a charge for a product to this invoice.
         * @param aProduct the product that the customer ordered
         * @param quantity the quantity of the product
         */
        public void add(Product aProduct, int quantity)
        {

        }

        /**
         * Formats the invoice.
         * @return the formatted invoice
         */
        public String format()
        {
        }
    }

    /**
     * Describes a quantity of an article to purchase.
     */
    public class LineItem
    {
        /**
         * Computes the total cost of this line item.
         * @return the total price
         */
        public double getTotalPrice()
        {
        }
    }


```

```

    }

    /**
     * Formats this item.
     * @return a formatted string of this item
    */
    public String format()
    {
    }
}

/**
 * Describes a product with a description and a price.
 */
public class Product
{
    /**
     * Gets the product description.
     * @return the description
    */
    public String getDescription()
    {
    }

    /**
     * Gets the product price.
     * @return the unit price
    */
    public double getPrice()
    {
    }
}

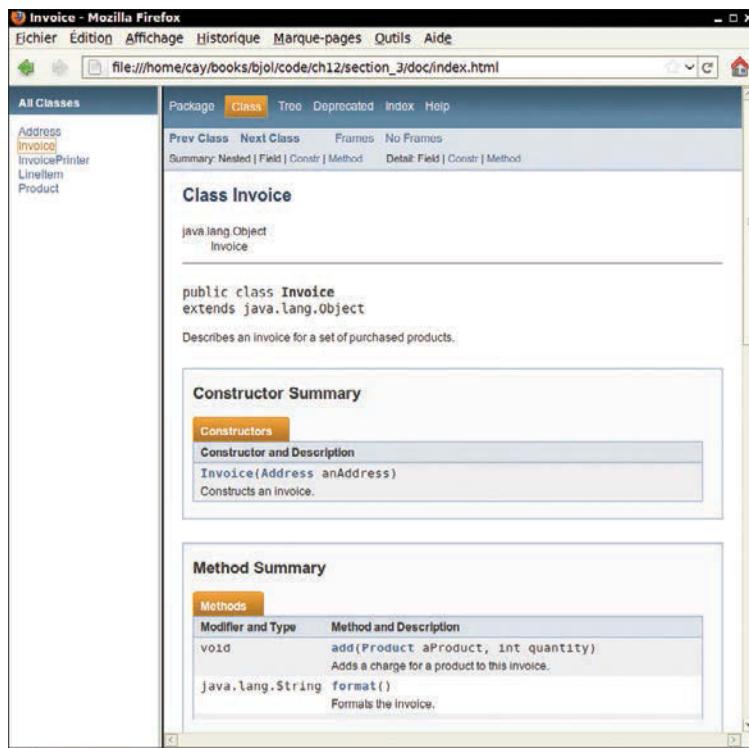
/**
 * Describes a mailing address.
 */
public class Address
{
    /**
     * Formats the address.
     * @return the address as a string with three lines
    */
    public String format()
    {
    }
}

```

Then run the javadoc program to obtain a neatly formatted version of your documentation in HTML format (see Figure 8).

This approach for documenting your classes has a number of advantages. You can share the HTML documentation with others if you work in a team. You use a format that is immediately useful—Java source files that you can carry into the implementation phase. And, most importantly, you supply the comments for the key methods—a task that less prepared programmers leave for later, and often neglect for lack of time.

**Figure 8**  
Class Documentation  
in HTML Format



### 12.3.5 Implementation

After completing the design, implement your classes.

After you have completed the object-oriented design, you are ready to implement the classes.

You already have the method parameter variables and comments from the previous step. Now look at the UML diagram to add instance variables. Aggregated classes yield instance variables. Start with the `Invoice` class. An invoice aggregates `Address` and `LineItem`. Every invoice has one billing address, but it can have many line items. To store multiple `LineItem` objects, you can use an array list. Now you have the instance variables of the `Invoice` class:

```
public class Invoice
{
    private Address billingAddress;
    private ArrayList<LineItem> items;
    ...
}
```

A line item needs to store a `Product` object and the product quantity. That leads to the following instance variables:

```
public class LineItem
{
    private int quantity;
    private Product theProduct;
    ...
}
```

The methods themselves are now easy to implement. Here is a typical example. You already know what the `getTotalPrice` method of the `LineItem` class needs to do—get the unit price of the product and multiply it with the quantity:

```
 /**
     * Computes the total cost of this line item.
     * @return the total price
 */
public double getTotalPrice()
{
    return theProduct.getPrice() * quantity;
}
```

We will not discuss the other methods in detail—they are equally straightforward.

Finally, you need to supply constructors, another routine task.

The entire program is shown below. It is a good practice to go through it in detail and match up the classes and methods to the CRC cards and UML diagram.

Worked Example 12.1 ([at wiley.com/go/bjeo6examples](http://wiley.com/go/bjeo6examples)) demonstrates the design process with a more challenging problem, a simulated automatic teller machine. You should download and study that example as well.

In this chapter, you learned a systematic approach for building a relatively complex program. However, object-oriented design is definitely not a spectator sport. To really learn how to design and implement programs, you have to gain experience by repeating this process with your own projects. It is quite possible that you don't immediately home in on a good solution and that you need to go back and reorganize your classes and responsibilities. That is normal and only to be expected. The purpose of the object-oriented design process is to spot these problems in the design phase, when they are still easy to rectify, instead of in the implementation phase, when massive reorganization is more difficult and time consuming.

### section\_3/InvoicePrinter.java

```
1 /**
2  * This program demonstrates the invoice classes by
3  * printing a sample invoice.
4 */
5 public class InvoicePrinter
6 {
7     public static void main(String[] args)
8     {
9         Address samsAddress
10        = new Address("Sam's Small Appliances",
11                      "100 Main Street", "Anytown", "CA", "98765");
12
13        Invoice samsInvoice = new Invoice(samsAddress);
14        samsInvoice.add(new Product("Toaster", 29.95), 3);
15        samsInvoice.add(new Product("Hair dryer", 24.95), 1);
16        samsInvoice.add(new Product("Car vacuum", 19.99), 2);
17
18        System.out.println(samsInvoice.format());
19    }
20}
```

### section\_3/Invoice.java

```
1 import java.util.ArrayList;
2
```

```

3  /**
4   * Describes an invoice for a set of purchased products.
5  */
6  public class Invoice
7  {
8      private Address billingAddress;
9      private ArrayList<LineItem> items;
10
11     /**
12      Constructs an invoice.
13      @param anAddress the billing address
14     */
15     public Invoice(Address anAddress)
16     {
17         items = new ArrayList<LineItem>();
18         billingAddress = anAddress;
19     }
20
21     /**
22      Adds a charge for a product to this invoice.
23      @param aProduct the product that the customer ordered
24      @param quantity the quantity of the product
25     */
26     public void add(Product aProduct, int quantity)
27     {
28         LineItem anItem = new LineItem(aProduct, quantity);
29         items.add(anItem);
30     }
31
32     /**
33      Formats the invoice.
34      @return the formatted invoice
35     */
36     public String format()
37     {
38         String r = "I N V O I C E\n"
39                     + billingAddress.format()
40                     + String.format("\n%-30s%8s%5s%8s\n",
41                         "Description", "Price", "Qty", "Total");
42
43         for (LineItem item : items)
44         {
45             r = r + item.format() + "\n";
46         }
47
48         r = r + String.format("\nAMOUNT DUE: $%.2f", getAmountDue());
49
50         return r;
51     }
52
53     /**
54      Computes the total amount due.
55      @return the amount due
56     */
57     private double getAmountDue()
58     {
59         double amountDue = 0;
60         for (LineItem item : items)
61         {
62             amountDue = amountDue + item.getTotalPrice();

```

```

63     }
64     return amountDue;
65   }
66 }
```

**section\_3/LineItem.java**

```

1  /**
2   * Describes a quantity of an article to purchase.
3  */
4  public class LineItem
5  {
6    private int quantity;
7    private Product theProduct;
8
9    /**
10     * Constructs an item from the product and quantity.
11     * @param aProduct the product
12     * @param aQuantity the item quantity
13    */
14    public LineItem(Product aProduct, int aQuantity)
15    {
16      theProduct = aProduct;
17      quantity = aQuantity;
18    }
19
20   /**
21    * Computes the total cost of this line item.
22    * @return the total price
23   */
24   public double getTotalPrice()
25   {
26     return theProduct.getPrice() * quantity;
27   }
28
29   /**
30    * Formats this item.
31    * @return a formatted string of this line item
32   */
33   public String format()
34   {
35     return String.format("%-30s%8.2f%5d%8.2f",
36       theProduct.getDescription(), theProduct.getPrice(),
37       quantity, getTotalPrice());
38   }
39 }
```

**section\_3/Product.java**

```

1  /**
2   * Describes a product with a description and a price.
3  */
4  public class Product
5  {
6    private String description;
7    private double price;
8
9    /**
10     * Constructs a product from a description and a price.
11     * @param aDescription the product description
12   }
```

```

12     @param aPrice the product price
13 */
14 public Product(String aDescription, double aPrice)
15 {
16     description = aDescription;
17     price = aPrice;
18 }
19
20 /**
21     Gets the product description.
22     @return the description
23 */
24 public String getDescription()
25 {
26     return description;
27 }
28
29 /**
30     Gets the product price.
31     @return the unit price
32 */
33 public double getPrice()
34 {
35     return price;
36 }
37 }
```

### section\_3/Address.java

```

1 /**
2     Describes a mailing address.
3 */
4 public class Address
5 {
6     private String name;
7     private String street;
8     private String city;
9     private String state;
10    private String zip;
11
12 /**
13     Constructs a mailing address.
14     @param aName the recipient name
15     @param aStreet the street
16     @param aCity the city
17     @param aState the two-letter state code
18     @param aZip the ZIP postal code
19 */
20 public Address(String aName, String aStreet,
21                 String aCity, String aState, String aZip)
22 {
23     name = aName;
24     street = aStreet;
25     city = aCity;
26     state = aState;
27     zip = aZip;
28 }
29
30 /**
31     Formats the address.
32 }
```

```

32     @return the address as a string with three lines
33     */
34     public String format()
35     {
36         return name + "\n" + street + "\n"
37         + city + ", " + state + " " + zip;
38     }
39 }
```

**SELF CHECK**

13. Which class is responsible for computing the amount due? What are its collaborators for this task?
14. Why do the format methods return `String` objects instead of directly printing to `System.out`?

**Practice It** Now you can try these exercises at the end of the chapter: R12.15, E12.4, E12.5.

**Computing & Society 12.1 Databases and Privacy**

Most companies use computers to keep huge databases of customer records and other business information. Databases not only lower the cost of doing business, they improve the quality of service that companies can offer. Nowadays it is almost unimaginable how time-consuming it used to be to withdraw money from a bank branch or to make travel reservations.

As these databases became ubiquitous, they started creating problems for citizens. Consider the “no fly list” maintained by the U.S. government, which lists names used by suspected terrorists. On March 1, 2007, Professor Walter Murphy, a constitutional scholar of Princeton University and a decorated former Marine, was denied a boarding pass. The airline employee asked him, “Have you been in any peace marches? We ban a lot of people from flying because of that.” As Murphy tells it, “I explained that I had not so marched but had, in September 2006, given a lecture at Princeton, televised and put on the Web, highly critical of George Bush for his many violations of the constitution. ‘That’ll do it,’ the man said.”

We do not actually know if Professor Murphy’s name was on the list because he was critical of the Bush administration or because some other potentially dangerous person had traveled under the same name. Travelers with similar misfortunes had serious

difficulties trying to get themselves off the list.

Problems such as these have become commonplace. Companies and the government routinely merge multiple databases, derive information about us that may be quite inaccurate, and then use that information to make decisions. An insurance company may deny coverage, or charge a higher premium, if it finds that you have too many relatives with a certain disease. You may be denied a job because of a credit or medical report. You do not usually know what information about you is stored or how it is used. In cases where the information can be checked—such as credit reports—it is often difficult to correct errors.

Another issue of concern is privacy. Most people do something, at one time or another in their lives, that they do not want everyone to know about. As judge Louis Brandeis wrote in 1928, “Privacy is the right to be alone—the most comprehensive of rights, and the right most valued by civilized man.” When employers can see your old Facebook posts, divorce lawyers have access to toll road records, and Google mines your e-mails and searches to present you “targeted” advertising, you have little privacy left.

The 1948 “universal declaration of human rights” by the United Nations states, “No one shall be subjected to arbitrary interference with his privacy,

family, home or correspondence, nor to attacks upon his honour and reputation. Everyone has the right to the protection of the law against such interference or attacks.” The United States has surprisingly few legal protections against privacy invasion, apart from federal laws protecting student records and video rentals (the latter was passed after a Supreme Court nominee’s video rental records were published). Other industrialized countries have gone much further and recognize every citizen’s right to control what information about them should be communicated to others and under what circumstances.



© Greg Nichols/iStockphoto.

*If you pay road or bridge tolls with an electronic pass, your records may not be private.*



## WORKED EXAMPLE 12.1



## Simulating an Automatic Teller Machine

Learn to apply the object-oriented design methodology to the simulation of an automatic teller machine that works with both a console-based and graphical user interface. Go to [wiley.com/go/bjebexamples](http://wiley.com/go/bjebexamples) and download Worked Example 12.1.



© Mark Evans/  
iStockphoto.

## CHAPTER SUMMARY

## Recognize how to discover classes and their responsibilities.



- To discover classes, look for nouns in the problem description.
- Concepts from the problem domain are good candidates for classes.
- A CRC card describes a class, its responsibilities, and its collaborating classes.

## Categorize class relationships and produce UML diagrams that describe them.



- A class depends on another class if it uses objects of that class.
- It is a good practice to minimize the coupling (i.e., dependency) between classes.
- A class aggregates another if its objects contain objects of the other class.
- Inheritance (the *is-a* relationship) is sometimes inappropriately used when the *has-a* relationship would be more appropriate.
- Aggregation (the *has-a* relationship) denotes that objects of one class contain references to objects of another class.
- You need to be able to distinguish the UML notation for inheritance, interface implementation, aggregation, and dependency.

## Apply an object-oriented development process to designing a program.

- Start the development process by gathering and documenting program requirements.
- Use CRC cards to find classes, responsibilities, and collaborators.
- Use UML diagrams to record class relationships.
- Use javadoc comments (with the method bodies left blank) to record the behavior of classes.
- After completing the design, implement your classes.

## REVIEW EXERCISES

- R12.1** List the steps in the process of object-oriented design that this chapter recommends for student use.
- R12.2** Give a rule of thumb for how to find classes when designing a program.
- R12.3** Give a rule of thumb for how to find methods when designing a program.
- R12.4** After discovering a method, why is it important to identify the object that is *responsible* for carrying out the action?

**■■ R12.5** What relationship is appropriate between the following classes: aggregation, inheritance, or neither?

- |  |   |
|--|---|
| <b>a.</b> University–Student<br><b>b.</b> Student–TeachingAssistant<br><b>c.</b> Student–Freshman<br><b>d.</b> Student–Professor | <b>e.</b> Car–Door<br><b>f.</b> Truck–Vehicle<br><b>g.</b> Traffic–TrafficSign<br><b>h.</b> TrafficSign–Color |
|--|---|

**■■ R12.6** Every BMW is a vehicle. Should a class `BMW` inherit from the class `Vehicle`? BMW is a vehicle manufacturer. Does that mean that the class `BMW` should inherit from the class `VehicleManufacturer`?

**■■ R12.7** Some books on object-oriented programming recommend using inheritance so that the class `Circle` extends the class `java.awt.Point`. Then the `Circle` class inherits the `setLocation` method from the `Point` superclass. Explain why the `setLocation` method need not be overridden in the subclass. Why is it nevertheless not a good idea to have `Circle` inherit from `Point`? Conversely, would inheriting `Point` from `Circle` fulfill the *is-a* rule? Would it be a good idea?

**■ R12.8** Write CRC cards for the `Coin` and `CashRegister` classes described in Section 8.2.

**■ R12.9** Write CRC cards for the `Quiz` and `Question` classes in Section 12.2.2.

**■■ R12.10** Draw a UML diagram for the `Quiz`, `Question`, and `ChoiceQuestion` classes. The `Quiz` class is described in Section 12.2.2.

**■■■ R12.11** A file contains a set of records describing countries. Each record consists of the name of the country, its population, and its area. Suppose your task is to write a program that reads in such a file and prints

- The country with the largest area.
- The country with the largest population.
- The country with the largest population density (people per square kilometer).

Think through the problems that you need to solve. What classes and methods will you need? Produce a set of CRC cards, a UML diagram, and a set of javadoc comments.

**■■■ R12.12** Discover classes and methods for generating a student report card that lists all classes, grades, and the grade point average for a semester. Produce a set of CRC cards, a UML diagram, and a set of javadoc comments.

**■■ R12.13** Consider the following problem description:

**Users place coins in a vending machine and select a product by pushing a button. If the inserted coins are sufficient to cover the purchase price of the product, the product is dispensed and change is given. Otherwise, the inserted coins are returned to the user.**

What classes should you use to implement a solution?

**■■ R12.14** Consider the following problem description:

**Employees receive their biweekly paychecks. They are paid their hourly rates for each hour worked; however, if they worked more than 40 hours per week, they are paid overtime at 150 percent of their regular wage.**

What classes should you use to implement a solution?

- R12.15** Consider the following problem description:

**Customers order products from a store. Invoices are generated to list the items and quantities ordered, payments received, and amounts still due. Products are shipped to the shipping address of the customer, and invoices are sent to the billing address.**

Draw a UML diagram showing the aggregation relationships between the classes `Invoice`, `Address`, `Customer`, and `Product`.

## PRACTICE EXERCISES

- E12.1** Enhance the invoice-printing program by providing for two kinds of line items: One kind describes products that are purchased in certain numerical quantities (such as “3 toasters”), another describes a fixed charge (such as “shipping: \$5.00”). *Hint:* Use inheritance. Produce a UML diagram of your modified implementation.
- E12.2** The invoice-printing program is somewhat unrealistic because the formatting of the `LineItem` objects won’t lead to good visual results when the prices and quantities have varying numbers of digits. Enhance the `format` method in two ways: Accept an `int[]` array of column widths as an argument. Use the `NumberFormat` class to format the currency values.
- E12.3** The invoice-printing program has an unfortunate flaw—it mixes “application logic” (the computation of total charges) and “presentation” (the visual appearance of the invoice). To appreciate this flaw, imagine the changes that would be necessary to draw the invoice in HTML for presentation on the Web. Reimplement the program, using a separate `InvoiceFormatter` class to format the invoice. That is, the `Invoice` and `LineItem` methods are no longer responsible for formatting. However, they will acquire other responsibilities, because the `InvoiceFormatter` class needs to query them for the values that it requires.
- E12.4** Write a program that teaches arithmetic to a young child. The program tests addition and subtraction. In level 1, it tests only addition of numbers less than ten whose sum is less than ten. In level 2, it tests addition of arbitrary one-digit numbers. In level 3, it tests subtraction of one-digit numbers with a nonnegative difference.  
Generate random problems and get the player’s input. The player gets up to two tries per problem. Advance from one level to the next when the player has achieved a score of five points.
- E12.5** Implement a simple e-mail messaging system. A message has a recipient, a sender, and a message text. A mailbox can store messages. Supply a number of mailboxes for different users and a user interface for users to log in, send messages to other users, read their own messages, and log out. Follow the design process that was described in this chapter.
- E12.6** Modify the implementation of the classes in the ATM simulation in Worked Example 12.1 so that the bank manages a collection of bank accounts and a separate collection of customers. Allow joint accounts in which some accounts can have more than one customer.

## PROGRAMMING PROJECTS

- P12.1** Write a program that simulates a vending machine. Products can be purchased by inserting coins with a value at least equal to the cost of the product. A user selects a product from a list of available products, adds coins, and either gets the product or gets the coins returned. The coins are returned if insufficient money was supplied or if the product is sold out. The machine does not give change if too much money was added. Products can be restocked and money removed by an operator. Follow the design process that was described in this chapter. Your solution should include a class `VendingMachine` that is not coupled with the `Scanner` or `PrintStream` classes.
- P12.2** Write a program to design an appointment calendar. An appointment includes the date, starting time, ending time, and a description; for example,

```
Dentist 2016/10/1 17:30 18:30
CS1 class 2016/10/2 08:30 10:00
```

Supply a user interface to add appointments, remove canceled appointments, and print out a list of appointments for a particular day. Follow the design process that was described in this chapter. Your solution should include a class `AppointmentCalendar` that is not coupled with the `Scanner` or `PrintStream` classes.

- P12.3** Write a program that administers and grades quizzes. A quiz consists of questions. There are four types of questions: text questions, number questions, choice questions with a single answer, and choice questions with multiple answers. When grading a text question, ignore leading or trailing spaces and letter case. When grading a numeric question, accept a response that is approximately the same as the answer.
- A quiz is specified in a text file. Each question starts with a letter indicating the question type (T, N, S, M), followed by a line containing the question text. The next line of a non-choice question contains the answer. Choice questions have a list of choices that is terminated by a blank line. Each choice starts with + (correct) or - (incorrect). Here is a sample file:

```
T
Which Java reserved word is used to declare a subclass?
extends
S
What is the original name of the Java language?
- *7
- C--
+ Oak
- Gosling

M
Which of the following types are supertypes of Rectangle?
- PrintStream
+ Shape
+ RectangularShape
+ Object
- String

N
What is the square root of 2?
1.41421356
```

Your program should read in a quiz file, prompt the user for responses to all questions, and grade the responses. Follow the design process described in this chapter.

**■■ P12.4** Produce a requirements document for a program that allows a company to send out personalized mailings, either by e-mail or through the postal service. Template files contain the message text, together with variable fields (such as Dear [Title] [Last Name] ...). A database (stored as a text file) contains the field values for each recipient. Use HTML as the output file format. Then design and implement the program.

**■■■ P12.5** Write a tic-tac-toe game that allows a human player to play against the computer. Your program will play many turns against a human opponent, and it will learn. When it is the computer's turn, the computer randomly selects an empty field, except that it won't ever choose a losing combination. For that purpose, your program must keep an array of losing combinations. Whenever the human wins, the immediately preceding combination is stored as losing. For example, suppose that X = computer and O = human.

Suppose the current combination is

|   |   |   |
|---|---|---|
| O | X | X |
|   | O |   |
|   |   |   |

Now it is the human's turn, who will of course choose

|   |   |   |
|---|---|---|
| O | X | X |
|   | O |   |
|   |   | O |

The computer should then remember the preceding combination

|   |   |   |
|---|---|---|
| O | X | X |
|   | O |   |
|   |   | O |

as a losing combination. As a result, the computer will never again choose that combination from

|   |   |  |    |   |   |
|---|---|--|----|---|---|
| O | X |  | or | O | X |
|   | O |  |    | O |   |
|   |   |  |    |   |   |

Discover classes and supply a UML diagram before you begin to program.

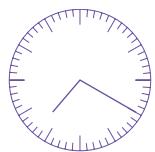
**■■■ Business P12.6** *Airline seating.* Write a program that assigns seats on an airplane. Assume the airplane has 20 seats in first class (5 rows of 4 seats each, separated by an aisle) and 90 seats in economy class (15 rows of 6 seats each, separated by an aisle). Your program should take three commands: add passengers, show seating, and quit. When passengers are added, ask for the class (first or economy), the number of passengers traveling together (1 or 2 in first class; 1 to 3 in economy), and the seating preference (aisle or window in first class; aisle, center, or window in economy). Then try to find a match and assign the seats. If no match exists, print a message. Your solution should include a class `Airplane` that is not coupled with the `Scanner` or `PrintStream` classes. Follow the design process that was described in this chapter.

**■■■ Business P12.7** In an airplane, each passenger has a touch screen for ordering a drink and a snack. Some items are free and some are not. The system prepares two reports for speeding up service:

1. A list of all seats, ordered by row, showing the charges that must be collected.
2. A list of how many drinks and snacks of each type must be prepared for the front and the rear of the plane.

Follow the design process that was described in this chapter to identify classes, and implement a program that simulates the system.

**■■■ Graphics P12.8**



An Analog Clock

Implement a program to teach a young child to read the clock. In the game, present an analog clock, such as the one shown at left. Generate random times and display the clock. Accept guesses from the player. Reward the player for correct guesses. After two incorrect guesses, display the correct answer and make a new random time. Implement several levels of play. In level 1, only show full hours. In level 2, show quarter hours. In level 3, show five-minute multiples, and in level 4, show any number of minutes. After a player has achieved five correct guesses at one level, advance to the next level.

**■■■ Graphics P12.9**

Write a program that can be used to design a suburban scene, with houses, streets, and cars. Users can add houses and cars of various colors to a street. Write more specific requirements that include a detailed description of the user interface. Then, discover classes and methods, provide UML diagrams, and implement your program.

**■■■ Graphics P12.10**

Write a simple graphics editor that allows users to add a mixture of shapes (ellipses, rectangles, and lines in different colors) to a panel. Supply commands to load and save the picture. Discover classes, supply a UML diagram, and implement your program.

### ANSWERS TO SELF-CHECK QUESTIONS

1. Look for nouns in the problem description.
2. Yes (`ChessBoard`) and no (`MovePiece`).
3. `PrintStream`.
4. To produce the shipping address of the customer.
5. Rework the responsibilities so that they are at a higher level, or come up with more classes to handle the responsibilities.
6. The `CashRegisterTester` class depends on the `CashRegister`, `Coin`, and `System` classes.
7. The `ChoiceQuestion` class inherits from the `Question` class.
8. The `Quiz` class depends on the `Question` class but probably not `ChoiceQuestion`, if we assume that the methods of the `Quiz` class manipulate generic `Question` objects, as they did in Chapter 9.
9. If a class doesn't depend on another, it is not affected by interface changes in the other class.
10. 

```

graph LR
    Mailbox[Mailbox] --> Message[Message]
  
```

11. Typically, a library system wants to track which books a patron has checked out, so it makes more sense to have `Patron` aggregate `Book`. However, there is not always one true answer in design. If you feel strongly that it is important to identify the patron who checked out a particular book (perhaps to notify the patron to return it because it was requested by someone else), then you can argue that the aggregation should go the other way around.

12. There would be no relationship.

13. The `Invoice` class is responsible for computing the amount due. It collaborates with the `LineItem` class.

14. This design decision reduces coupling. It enables us to reuse the classes when we want to show the invoice in a dialog box or on a web page.



## WORKED EXAMPLE 12.1

## Simulating an Automatic Teller Machine



In this Worked Example, we apply the object-oriented design methodology to the simulation of an automatic teller machine (ATM).

**Problem Statement** Simulate an ATM that handles checking and savings accounts. Provide both a console-based and graphical user interface.



© Mark Evans/iStockphoto.

### Step 1 Gather requirements.

The purpose of this project is to simulate an automatic teller machine. The ATM is used by the customers of a bank. Each customer has two accounts: a checking account and a savings account. Each customer also has a customer number and a personal identification number (PIN); both are required to gain access to the accounts. (In a real ATM, the customer number would be recorded on the magnetic strip of the ATM card. In this simulation, the customer will need to type it in.) With the ATM, customers can select an account (checking or savings). The balance of the selected account is displayed. Then the customer can deposit and withdraw money. This process is repeated until the customer chooses to exit.

The details of the user interaction depend on the user interface that we choose for the simulation. We will develop two separate interfaces: a graphical interface that closely mimics an actual ATM (see Figure 9), and a text-based interface that allows you to test the ATM and bank classes without being distracted by GUI programming.

In the GUI interface, the ATM has a keypad to enter numbers, a display to show messages, and a set of buttons, labeled A, B, and C, whose function depends on the state of the machine.

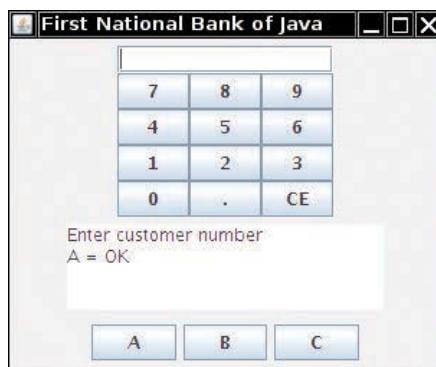
Specifically, the user interaction is as follows. When the ATM starts up, it expects a user to enter a customer number. The display shows the following message:

```
Enter customer number
A = OK
```

The user enters the customer number on the keypad and presses the A button. The display message changes to

```
Enter PIN
A = OK
```

Next, the user enters the PIN and presses the A button again. If the customer number and ID match those of one of the customers in the bank, then the customer can proceed. If not, the user is again prompted to enter the customer number.



**Figure 9**  
Graphical User Interface for  
the Automatic Teller Machine

## WE2 Chapter 12 Object-Oriented Design

If the customer has been authorized to use the system, then the display message changes to

```
Select Account  
A = Checking  
B = Savings  
C = Exit
```

If the user presses the C button, the ATM reverts to its original state and asks the next user to enter a customer number.

If the user presses the A or B buttons, the ATM remembers the selected account, and the display message changes to

```
Balance = balance of selected account  
Enter amount and select transaction  
A = Withdraw  
B = Deposit  
C = Cancel
```

If the user presses the A or B buttons, the value entered in the keypad is withdrawn from or deposited into the selected account. (This is just a simulation, so no money is dispensed and no deposit is accepted.) Afterward, the ATM reverts to the preceding state, allowing the user to select another account or to exit.

If the user presses the C button, the ATM reverts to the preceding state without executing any transaction.

In the text-based interaction, we read input from `System.in` instead of the buttons. Here is a typical dialog:

```
Enter customer number: 1  
Enter PIN: 1234  
A=Checking, B=Savings, C=Quit: A  
Balance=0.0  
A=Deposit, B=Withdrawal, C=Cancel: A  
Amount: 1000  
A=Checking, B=Savings, C=Quit: C
```

In our solution, only the user interface classes are affected by the choice of user interface. The remainder of the classes can be used for both solutions—they are decoupled from the user interface.

Because this is a simulation, the ATM does not actually communicate with a bank. It simply loads a set of customer numbers and PINs from a file. All accounts are initialized with a zero balance.

### Step 2 Use CRC cards to find classes, responsibilities, and collaborators.

We will again follow the recipe of Section 12.2 and show how to discover classes, responsibilities, and relationships and how to obtain a detailed design for the ATM program.

Recall that the first rule for finding classes is “Look for nouns in the problem description”. Here is a list of the nouns:

```
ATM  
User  
Keypad  
Display  
Display message  
Button  
State  
Bank account  
Checking account  
Savings account  
Customer  
Customer number  
PIN  
Bank
```

Of course, not all of these nouns will become names of classes, and we may yet discover the need for classes that aren't in this list, but it is a good start.

Users and customers represent the same concept in this program. Let's use a class `Customer`. A customer has two bank accounts, and we will require that a `Customer` object should be able to locate these accounts. (Another possible design would make the `Bank` class responsible for locating the accounts of a given customer—see Exercise E12.6.)

A customer also has a customer number and a PIN. We can, of course, require that a customer object give us the customer number and the PIN. But perhaps that isn't so secure. Instead, simply require that a customer object, when given a customer number and a PIN, will tell us whether it matches its own information or not.

| Customer                    |
|-----------------------------|
| <i>get accounts</i>         |
| <i>match number and PIN</i> |
|                             |
|                             |
|                             |
|                             |
|                             |

A bank contains a collection of customers. When a user walks up to the ATM and enters a customer number and PIN, it is the job of the bank to find the matching customer. How can the bank do this? It needs to check for each customer whether its customer number and PIN match. Thus, it needs to call the *match number and PIN* method of the `Customer` class that we just discovered. Because the *find customer* method calls a `Customer` method, it collaborates with the `Customer` class. We record that fact in the right-hand column of the CRC card.

When the simulation starts up, the bank must also be able to read customer information from a file.

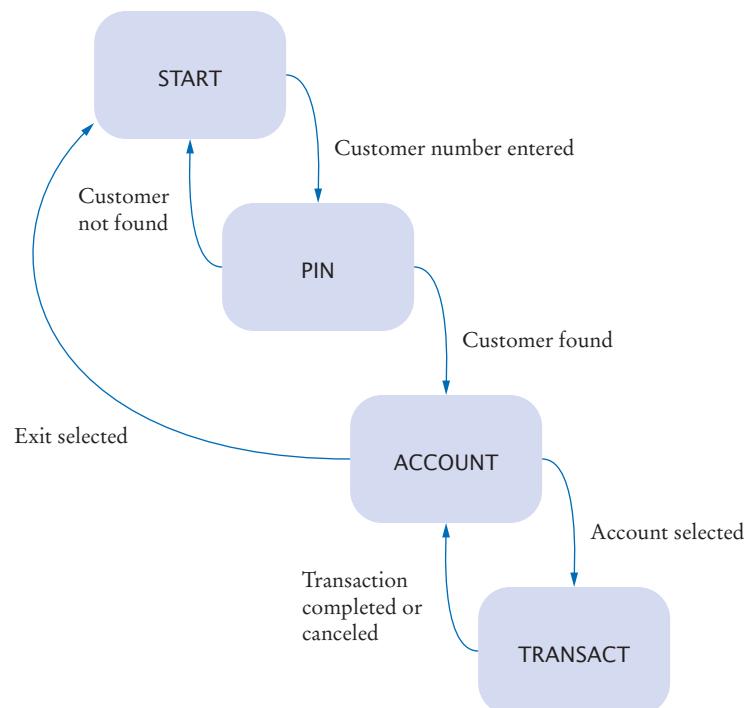
| Bank                  |
|-----------------------|
| <i>find customer</i>  |
| <i>Customer</i>       |
| <i>read customers</i> |
|                       |
|                       |
|                       |
|                       |
|                       |

The `BankAccount` class is our familiar class with methods to get the balance and to deposit and withdraw money.

In this program, there is nothing that distinguishes checking accounts from savings accounts. The ATM does not add interest or deduct fees. Therefore, we decide not to implement separate subclasses for checking and savings accounts.

Finally, we are left with the `ATM` class itself. An important notion of the ATM is the **state**. The current machine state determines the text of the prompts and the function of the buttons. For example, when you first log in, you use the A and B buttons to select an account. Next, you use the same buttons to choose between deposit and withdrawal. The ATM must remember the current state so that it can correctly interpret the buttons.

**Figure 10**  
State Diagram for  
the ATM Class



There are four states:

1. **START**: Enter customer ID
2. **PIN**: Enter PIN
3. **ACCOUNT**: Select account
4. **TRANSACT**: Select transaction

To understand how to move from one state to the next, it is useful to draw a **state diagram** (Figure 10). The UML notation has standardized shapes for state diagrams. Draw states as rectangles with rounded corners. Draw state changes as arrows, with labels that indicate the reason for the change.

The user must type a valid customer number and PIN. Then the ATM can ask the bank to find the customer. This calls for a *select customer* method. It collaborates with the bank, asking the bank for the customer that matches the customer number and PIN. Next, there must be a *select account* method that asks the current customer for the checking or savings account. Finally, the ATM must carry out the selected transaction on the current account.

| ATM                 |             |
|---------------------|-------------|
| manage state        | Customer    |
| select customer     | Bank        |
| select account      | BankAccount |
| execute transaction |             |
|                     |             |
|                     |             |
|                     |             |

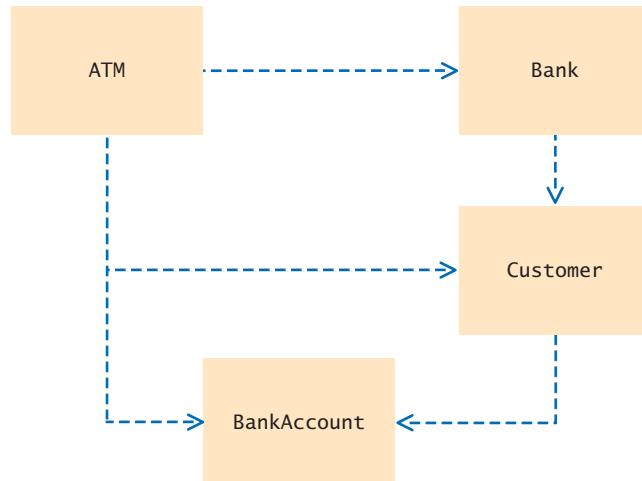
Of course, discovering these classes and methods was not as neat and orderly as it appears from this discussion. When I designed these classes for this book, it took me several trials and many torn cards to come up with a satisfactory design. It is also important to remember that there is seldom one best design.

This design has several advantages. The classes describe clear concepts. The methods are sufficient to implement all necessary tasks. (I mentally walked through every ATM usage scenario to verify that.) There are not too many collaboration dependencies between the classes. Thus, I was satisfied with this design and proceeded to the next step.

### Step 3 Use UML diagrams to record class relationships.

To draw the dependencies, use the “collaborator” columns from the CRC cards. Looking at those columns, you find that the dependencies are as follows:

- ATM knows about Bank, Customer, and BankAccount.
- Bank knows about Customer.
- Customer knows about BankAccount.



It is easy to see some of the aggregation relationships. A bank has customers, and each customer has two bank accounts.

Does the ATM class aggregate Bank? To answer this question, ask yourself whether an ATM object needs to store a reference to a bank object. Does it need to locate the same bank object across multiple method calls? Indeed it does. Therefore, aggregation is the appropriate relationship.

Does an ATM aggregate customers? Clearly, the ATM is not responsible for storing all of the bank's customers. That's the bank's job. But in our design, the ATM remembers the *current* customer. If a customer has logged in, subsequent commands refer to the same customer. The ATM needs to either store a reference to the customer, or ask the bank to look up the object whenever it needs the current customer. It is a design decision: either store the object, or look it up when needed. We will decide to store the current customer object. That is, we will use aggregation. Note that the choice of aggregation is not an automatic consequence of the problem description—it is a design decision.

Similarly, we will decide to store the current bank account (checking or savings) that the user selects. Therefore, we have an aggregation relationship between ATM and BankAccount.

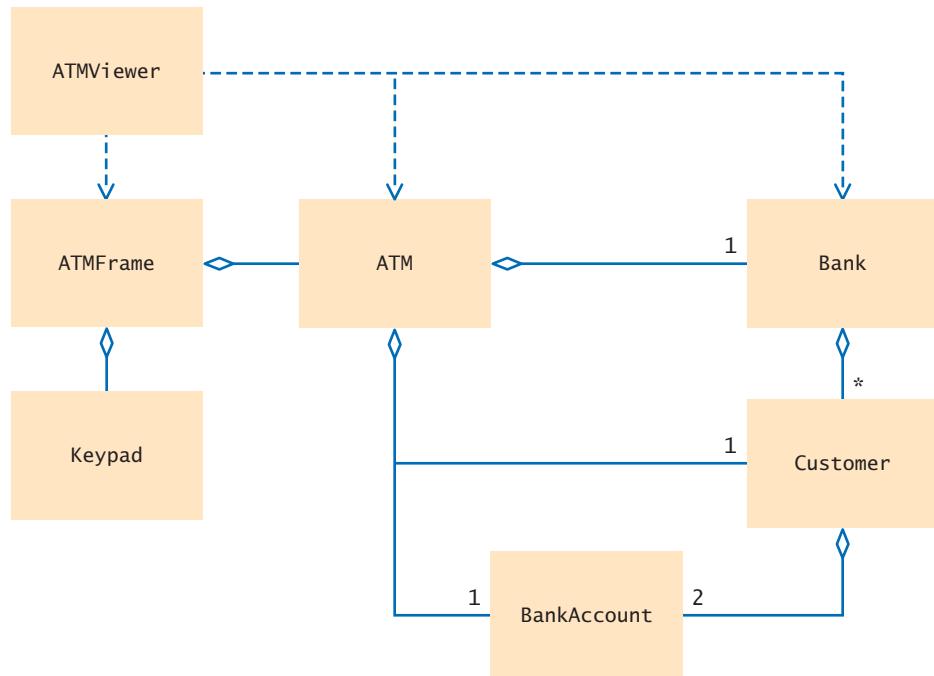
**Figure 11** Relationships Between the ATM Classes

Figure 11 shows the relationships between these classes, using the graphical user interface. (The console user interface uses a single class **ATMSimulator** instead of the **ATMFrame**, **ATMViewer**, and **Keypad** classes.)

The class diagram is a good tool to visualize dependencies. Look at the GUI classes. They are completely independent from the rest of the ATM system. You can replace the GUI with a console interface, and you can take out the **Keypad** class and use it in another application. Also, the **Bank**, **BankAccount**, and **Customer** classes, although dependent on each other, don't know anything about the **ATM** class. That makes sense—you can have banks without ATMs. As you can see, when you analyze relationships, you look for both the absence and presence of relationships.

#### Step 4 Use javadoc to document method behavior.

Now you are ready for the final step of the design phase: documenting the classes and methods that you discovered. Here is a part of the documentation for the **ATM** class:

```

/**
 * An ATM that accesses a bank.
 */
public class ATM
{
    . . .

    /**
     * Constructs an ATM for a given bank.
     * @param aBank the bank to which this ATM connects
     */
    public ATM(Bank aBank) { }

    /**
     * Sets the current customer number
     * and sets state to PIN.
     * (Precondition: state is START)
     */
  
```

```

    @param number the customer number
*/
public void setCustomerNumber(int number) { }

/**
   Finds customer in bank.
   If found sets state to ACCOUNT, else to START.
   (Precondition: state is PIN)
   @param pin the PIN of the current customer
*/
public void selectCustomer(int pin) { }

/**
   Sets current account to checking or savings. Sets
   state to TRANSACT.
   (Precondition: state is ACCOUNT or TRANSACT)
   @param account one of CHECKING or SAVINGS
*/
public void selectAccount(int account) { }

/**
   Withdraws amount from current account.
   (Precondition: state is TRANSACT)
   @param value the amount to withdraw
*/
public void withdraw(double value) { }
...
}

```

Then run the javadoc utility to turn this documentation into HTML format.

For conciseness, we omit the documentation of the other classes, but they are shown at the end of this example.

### Step 5 Implement your program.

Finally, the time has come to implement the ATM simulator. The implementation phase is very straightforward and should take *much less time than the design phase*.

A good strategy for implementing the classes is to go “bottom-up”. Start with the classes that don’t depend on others, such as Keypad and BankAccount. Then implement a class such as Customer that depends only on the BankAccount class. This “bottom-up” approach allows you to test your classes individually. You will find the implementations of these classes at the end of this section.

The most complex class is the ATM class. In order to implement the methods, you need to declare the necessary instance variables. From the class diagram, you can tell that the ATM has a bank object. It becomes an instance variable of the class:

```

public class ATM
{
    private Bank theBank;
}

```

From the description of the ATM states, it is clear that we require additional instance variables to store the current state, customer, and bank account:

```

public class ATM
{
    private int state;
    private Customer currentCustomer;
    private BankAccount currentAccount;
}

```

## WE8 Chapter 12 Object-Oriented Design

Most methods are very straightforward to implement. Consider the `selectCustomer` method. From the design documentation, we have the description

```
/**  
 * Finds customer in bank.  
 * If found sets state to ACCOUNT, else to START.  
 * (Precondition: state is PIN)  
 * @param pin the PIN of the current customer  
 */
```

This description can be almost literally translated to Java instructions:

```
public void selectCustomer(int pin)  
{  
    currentCustomer = theBank.findCustomer(customerNumber, pin);  
    if (currentCustomer == null)  
    {  
        state = START;  
    }  
    else  
    {  
        state = ACCOUNT;  
    }  
}
```

We won't go through a method-by-method description of the ATM program. You should take some time and compare the actual implementation to the CRC cards and the UML diagram.

### worked\_example\_1/BankAccount.java

```
1  /**  
2   * A bank account has a balance that can be changed by  
3   * deposits and withdrawals.  
4  */  
5  public class BankAccount  
6  {  
7      private double balance;  
8  
9      /**  
10       * Constructs a bank account with a zero balance.  
11      */  
12      public BankAccount()  
13      {  
14          balance = 0;  
15      }  
16  
17      /**  
18       * Constructs a bank account with a given balance.  
19       * @param initialBalance the initial balance  
20      */  
21      public BankAccount(double initialBalance)  
22      {  
23          balance = initialBalance;  
24      }  
25  
26      /**  
27       * Deposits money into the account.  
28       * @param amount the amount of money to withdraw  
29      */  
30      public void deposit(double amount)  
31      {
```

```

32     balance = balance + amount;
33 }
34
35 /**
36  * Withdraws money from the account.
37  * @param amount the amount of money to deposit
38 */
39 public void withdraw(double amount)
40 {
41     balance = balance - amount;
42 }
43
44 /**
45  * Gets the account balance.
46  * @return the account balance
47 */
48 public double getBalance()
49 {
50     return balance;
51 }
52 }
```

**worked\_example\_1/Customer.java**

```

1  /**
2   * A bank customer with a checking and a savings account.
3   */
4  public class Customer
5  {
6      private int customerNumber;
7      private int pin;
8      private BankAccount checkingAccount;
9      private BankAccount savingsAccount;
10
11 /**
12  * Constructs a customer with a given number and PIN.
13  * @param aNumber the customer number
14  * @param aPin the personal identification number
15  */
16 public Customer(int aNumber, int aPin)
17 {
18     customerNumber = aNumber;
19     pin = aPin;
20     checkingAccount = new BankAccount();
21     savingsAccount = new BankAccount();
22 }
23
24 /**
25  * Tests if this customer matches a customer number
26  * and PIN.
27  * @param aNumber a customer number
28  * @param aPin a personal identification number
29  * @return true if the customer number and PIN match
30  */
31 public boolean match(int aNumber, int aPin)
32 {
33     return customerNumber == aNumber && pin == aPin;
34 }
35
36 */
```

```

37     Gets the checking account of this customer.
38     @return the checking account
39 */
40     public BankAccount getCheckingAccount()
41     {
42         return checkingAccount;
43     }
44
45 /**
46     Gets the savings account of this customer.
47     @return the checking account
48 */
49     public BankAccount getSavingsAccount()
50     {
51         return savingsAccount;
52     }
53 }
```

**worked\_example\_1/Bank.java**

```

1 import java.io.File;
2 import java.io.IOException;
3 import java.util.ArrayList;
4 import java.util.Scanner;
5
6 /**
7     A bank contains customers.
8 */
9 public class Bank
10 {
11     private ArrayList<Customer> customers;
12
13 /**
14     Constructs a bank with no customers.
15 */
16     public Bank()
17     {
18         customers = new ArrayList<Customer>();
19     }
20
21 /**
22     Reads the customer numbers and pins.
23     @param filename the name of the customer file
24 */
25     public void readCustomers(String filename)
26             throws IOException
27     {
28         Scanner in = new Scanner(new File(filename));
29         while (in.hasNext())
30         {
31             int number = in.nextInt();
32             int pin = in.nextInt();
33             Customer c = new Customer(number, pin);
34             addCustomer(c);
35         }
36         in.close();
37     }
38
39 /**
```

```

40     Adds a customer to the bank.
41     @param c the customer to add
42 */
43 public void addCustomer(Customer c)
44 {
45     customers.add(c);
46 }
47
48 /**
49     Finds a customer in the bank.
50     @param aNumber a customer number
51     @param aPin a personal identification number
52     @return the matching customer, or null if no customer
53         matches
54 */
55 public Customer findCustomer(int aNumber, int aPin)
56 {
57     for (Customer c : customers)
58     {
59         if (c.match(aNumber, aPin))
60         {
61             return c;
62         }
63     }
64     return null;
65 }
66 }
```

**worked\_example\_1/ATM.java**

```

1 /**
2     An ATM that accesses a bank.
3 */
4 public class ATM
5 {
6     public static final int CHECKING = 1;
7     public static final int SAVINGS = 2;
8
9     private int state;
10    private int customerNumber;
11    private Customer currentCustomer;
12    private BankAccount currentAccount;
13    private Bank theBank;
14
15    public static final int START = 1;
16    public static final int PIN = 2;
17    public static final int ACCOUNT = 3;
18    public static final int TRANSACT = 4;
19
20 /**
21     Constructs an ATM for a given bank.
22     @param aBank the bank to which this ATM connects
23 */
24 public ATM(Bank aBank)
25 {
26     theBank = aBank;
27     reset();
28 }
```

```

30  /**
31   *      Resets the ATM to the initial state.
32  */
33  public void reset()
34  {
35     customerNumber = -1;
36     currentAccount = null;
37     state = START;
38 }
39
40 /**
41   *      Sets the current customer number
42   *      and sets state to PIN.
43   *      (Precondition: state is START)
44   *      @param number the customer number
45  */
46  public void setCustomerNumber(int number)
47  {
48     customerNumber = number;
49     state = PIN;
50 }
51
52 /**
53   *      Finds customer in bank.
54   *      If found sets state to ACCOUNT, else to START.
55   *      (Precondition: state is PIN)
56   *      @param pin the PIN of the current customer
57  */
58  public void selectCustomer(int pin)
59  {
60     currentCustomer
61     = theBank.findCustomer(customerNumber, pin);
62     if (currentCustomer == null)
63     {
64         state = START;
65     }
66     else
67     {
68         state = ACCOUNT;
69     }
70 }
71
72 /**
73   *      Sets current account to checking or savings. Sets
74   *      state to TRANSACT.
75   *      (Precondition: state is ACCOUNT or TRANSACT)
76   *      @param account one of CHECKING or SAVINGS
77  */
78  public void selectAccount(int account)
79  {
80     if (account == CHECKING)
81     {
82         currentAccount = currentCustomer.getCheckingAccount();
83     }
84     else
85     {
86         currentAccount = currentCustomer.getSavingsAccount();
87     }
88     state = TRANSACT;
89 }

```

```

90
91      /**
92       Withdraws amount from current account.
93       (Precondition: state is TRANSACT)
94       @param value the amount to withdraw
95     */
96     public void withdraw(double value)
97     {
98       currentAccount.withdraw(value);
99     }
100
101    /**
102     Deposits amount to current account.
103     (Precondition: state is TRANSACT)
104     @param value the amount to deposit
105   */
106   public void deposit(double value)
107   {
108     currentAccount.deposit(value);
109   }
110
111  /**
112   Gets the balance of the current account.
113   (Precondition: state is TRANSACT)
114   @return the balance
115 */
116 public double getBalance()
117 {
118   return currentAccount.getBalance();
119 }
120
121 /**
122  Moves back to the previous state.
123 */
124 public void back()
125 {
126   if (state == TRANSACT)
127   {
128     state = ACCOUNT;
129   }
130   else if (state == ACCOUNT)
131   {
132     state = PIN;
133   }
134   else if (state == PIN)
135   {
136     state = START;
137   }
138 }
139
140 /**
141  Gets the current state of this ATM.
142  @return the current state
143 */
144 public int getState()
145 {
146   return state;
147 }
148 }
```

The following class implements a console-based user interface for the ATM.

### **worked\_example\_1/ATMSimulator.java**

```

1 import java.io.IOException;
2 import java.util.Scanner;
3
4 /**
5   A text-based simulation of an automatic teller machine.
6 */
7 public class ATMSimulator
8 {
9     public static void main(String[] args)
10    {
11        ATM theATM;
12        try
13        {
14            Bank theBank = new Bank();
15            theBank.readCustomers("customers.txt");
16            theATM = new ATM(theBank);
17        }
18        catch (IOException e)
19        {
20            System.out.println("Error opening accounts file.");
21            return;
22        }
23
24        Scanner in = new Scanner(System.in);
25
26        while (true)
27        {
28            int state = theATM.getState();
29            if (state == ATM.START)
30            {
31                System.out.print("Enter customer number: ");
32                int number = in.nextInt();
33                theATM.setCustomerNumber(number);
34            }
35            else if (state == ATM.PIN)
36            {
37                System.out.print("Enter PIN: ");
38                int pin = in.nextInt();
39                theATM.selectCustomer(pin);
40            }
41            else if (state == ATM.ACCOUNT)
42            {
43                System.out.print("A=Checking, B=Savings, C=Quit: ");
44                String command = in.next();
45                if (command.equalsIgnoreCase("A"))
46                {
47                    theATM.selectAccount(ATM.CHECKING);
48                }
49                else if (command.equalsIgnoreCase("B"))
50                {
51                    theATM.selectAccount(ATM.SAVINGS);
52                }
53                else if (command.equalsIgnoreCase("C"))
54                {
55                    theATM.reset();
56                }
57            }
58        }
59    }
60}
```

```

57         else
58     {
59         System.out.println("Illegal input!");
60     }
61 }
62 else if (state == ATM.TRANSACT)
63 {
64     System.out.println("Balance=" + theATM.getBalance());
65     System.out.print("A=Deposit, B=Withdrawal, C=Cancel: ");
66     String command = in.next();
67     if (command.equalsIgnoreCase("A"))
68     {
69         System.out.print("Amount: ");
70         double amount = in.nextDouble();
71         theATM.deposit(amount);
72         theATM.back();
73     }
74     else if (command.equalsIgnoreCase("B"))
75     {
76         System.out.print("Amount: ");
77         double amount = in.nextDouble();
78         theATM.withdraw(amount);
79         theATM.back();
80     }
81     else if (command.equalsIgnoreCase("C"))
82     {
83         theATM.back();
84     }
85     else
86     {
87         System.out.println("Illegal input!");
88     }
89 }
90 }
91 }
92 }

```

### Program Run

```

Enter customer number: 1
Enter PIN: 1234
A=Checking, B=Savings, C=Quit: A
Balance=0.0
A=Deposit, B=Withdrawal, C=Cancel: A
Amount: 1000
A=Checking, B=Savings, C=Quit: C
...

```

Here are the user-interface classes for the GUI version of the user interface.

### [worked\\_example\\_1/KeyPad.java](#)

```

1 import java.awt.BorderLayout;
2 import java.awt.GridLayout;
3 import java.awt.event.ActionEvent;
4 import java.awt.event.ActionListener;
5 import javax.swing.JButton;
6 import javax.swing.JPanel;
7 import javax.swing.JTextField;

```

```
8  /**
9   * A component that lets the user enter a number, using
10  * a button pad labeled with digits.
11  */
12 public class KeyPad extends JPanel
13 {
14     private JPanel buttonPanel;
15     private JButton clearButton;
16     private JTextField display;
17
18     /**
19      * Constructs the keypad panel.
20     */
21     public KeyPad()
22     {
23         setLayout(new BorderLayout());
24
25         // Add display field
26
27         display = new JTextField();
28         add(display, "North");
29
30         // Make button panel
31
32         buttonPanel = new JPanel();
33         buttonPanel.setLayout(new GridLayout(4, 3));
34
35         // Add digit buttons
36
37         addButton("7");
38         addButton("8");
39         addButton("9");
40         addButton("4");
41         addButton("5");
42         addButton("6");
43         addButton("1");
44         addButton("2");
45         addButton("3");
46         addButton("0");
47         addButton(".");
48
49         // Add clear entry button
50
51         clearButton = new JButton("CE");
52         buttonPanel.add(clearButton);
53
54         class ClearButtonListener implements ActionListener
55         {
56             public void actionPerformed(ActionEvent event)
57             {
58                 display.setText("");
59             }
60         }
61         ActionListener listener = new ClearButtonListener();
62
63         clearButton.addActionListener(new
64             ClearButtonListener());
65
66         add(buttonPanel, "Center");
67 }
```

```

68 }
69
70 /**
71  * Adds a button to the button panel.
72  * @param label the button label
73 */
74 private void addButton(final String label)
75 {
76     class DigitButtonListener implements ActionListener
77     {
78         public void actionPerformed(ActionEvent event)
79         {
80             // Don't add two decimal points
81             if (label.equals("."))
82                 && display.getText().indexOf(".") != -1)
83             {
84                 return;
85             }
86
87             // Append label text to button
88             display.setText(display.getText() + label);
89         }
90     }
91
92     JButton button = new JButton(label);
93     buttonPanel.add(button);
94     ActionListener listener = new DigitButtonListener();
95     button.addActionListener(listener);
96 }
97
98 /**
99  * Gets the value that the user entered.
100 * @return the value in the text field of the keypad
101 */
102 public double getValue()
103 {
104     return Double.parseDouble(display.getText());
105 }
106
107 /**
108  * Clears the display.
109 */
110 public void clear()
111 {
112     display.setText("");
113 }
114 }

```

**worked\_example\_1/ATMFrame.java**

```

1 import java.awt.FlowLayout;
2 import java.awt.GridLayout;
3 import java.awt.event.ActionEvent;
4 import java.awt.event.ActionListener;
5 import javax.swing.JButton;
6 import javax.swing.JFrame;
7 import javax.swing.JPanel;
8 import javax.swing.JTextArea;
9
10 /**

```

```
11     A frame displaying the components of an ATM.
12  */
13 public class ATMFrame extends JFrame
14 {
15     private static final int FRAME_WIDTH = 300;
16     private static final int FRAME_HEIGHT = 300;
17
18     private JButton aButton;
19     private JButton bButton;
20     private JButton cButton;
21
22     private KeyPad pad;
23     private JTextArea display;
24
25     private ATM theATM;
26
27     /**
28      Constructs the user interface of the ATM frame.
29     */
30     public ATMFrame(ATM anATM)
31     {
32         theATM = anATM;
33
34         // Construct components
35         pad = new KeyPad();
36
37         display = new JTextArea(4, 20);
38
39         aButton = new JButton(" A ");
40         aButton.addActionListener(new AButtonListener());
41
42         bButton = new JButton(" B ");
43         bButton.addActionListener(new BButtonListener());
44
45         cButton = new JButton(" C ");
46         cButton.addActionListener(new CButtonListener());
47
48         // Add components
49
50         JPanel buttonPanel = new JPanel();
51         buttonPanel.add(aButton);
52         buttonPanel.add(bButton);
53         buttonPanel.add(cButton);
54
55         setLayout(new FlowLayout());
56         add(pad);
57         add(display);
58         add(buttonPanel);
59         showState();
60
61         setSize(FRAME_WIDTH, FRAME_HEIGHT);
62     }
63
64     /**
65      Updates display message.
66     */
67     public void showState()
68     {
69         int state = theATM.getState();
70         pad.clear();
```

```

71   if (state == ATM.START)
72   {
73     display.setText("Enter customer number\nA = OK");
74   }
75   else if (state == ATM.PIN)
76   {
77     display.setText("Enter PIN\nA = OK");
78   }
79   else if (state == ATM.ACCOUNT)
80   {
81     display.setText("Select Account\n"
82                   + "A = Checking\nB = Savings\nC = Exit");
83   }
84   else if (state == ATM.TRANSACT)
85   {
86     display.setText("Balance = "
87                   + theATM.getBalance()
88                   + "\nEnter amount and select transaction\n"
89                   + "A = Withdraw\nB = Deposit\nC = Cancel");
90   }
91 }
92
93 class AButtonListener implements ActionListener
94 {
95   public void actionPerformed(ActionEvent event)
96   {
97     int state = theATM.getState();
98     if (state == ATM.START)
99     {
100       theATM.setCustomerNumber((int) pad.getValue());
101     }
102     else if (state == ATM.PIN)
103     {
104       theATM.selectCustomer((int) pad.getValue());
105     }
106     else if (state == ATM.ACCOUNT)
107     {
108       theATM.selectAccount(ATM.CHECKING);
109     }
110     else if (state == ATM.TRANSACT)
111     {
112       theATM.withdraw(pad.getValue());
113       theATM.back();
114     }
115     showState();
116   }
117 }
118
119 class BButtonListener implements ActionListener
120 {
121   public void actionPerformed(ActionEvent event)
122   {
123     int state = theATM.getState();
124     if (state == ATM.ACCOUNT)
125     {
126       theATM.selectAccount(ATM.SAVINGS);
127     }
128     else if (state == ATM.TRANSACT)
129     {
130       theATM.deposit(pad.getValue());
131     }
132   }
133 }
```

```

131             theATM.back();
132         }
133     }
134 }
135 }
136
137 class CButtonListener implements ActionListener
138 {
139     public void actionPerformed(ActionEvent event)
140     {
141         int state = theATM.getState();
142         if (state == ATM.ACCOUNT)
143         {
144             theATM.reset();
145         }
146         else if (state == ATM.TRANSACT)
147         {
148             theATM.back();
149         }
150         showState();
151     }
152 }
153 }
```

**worked\_example\_1/ATMViewer.java**

```

1 import java.io.IOException;
2 import javax.swing.JFrame;
3 import javax.swing.JOptionPane;
4
5 /**
6  * A graphical simulation of an automatic teller machine.
7 */
8 public class ATMViewer
9 {
10    public static void main(String[] args)
11    {
12        ATM theATM;
13
14        try
15        {
16            Bank theBank = new Bank();
17            theBank.readCustomers("customers.txt");
18            theATM = new ATM(theBank);
19        }
20        catch (IOException e)
21        {
22            JOptionPane.showMessageDialog(null, "Error opening accounts file.");
23            return;
24        }
25
26        JFrame frame = new ATMFrame(theATM);
27        frame.setTitle("First National Bank of Java");
28        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29        frame.setVisible(true);
30    }
31 }
```

# RECURSION

## CHAPTER GOALS

- To learn to “think recursively”
- To be able to use recursive helper methods
- To understand the relationship between recursion and iteration
- To understand when the use of recursion affects the efficiency of an algorithm
- To analyze problems that are much easier to solve by recursion than by iteration
- To process data with recursive structures using mutual recursion



© Nicolae Popovici/iStockphoto.

## CHAPTER CONTENTS

### 13.1 TRIANGLE NUMBERS 594

**CE1** Infinite Recursion 598

**CE2** Tracing Through Recursive Methods 598

**HT1** Thinking Recursively 599

**WE1** Finding Files

### 13.2 RECURSIVE HELPER METHODS 602

### 13.3 THE EFFICIENCY OF RECURSION 604

### 13.4 PERMUTATIONS 609

**C&S** The Limits of Computation 612

### 13.5 MUTUAL RECURSION 614

### 13.6 BACKTRACKING 620

**WE2** Towers of Hanoi



© Nicolae Popovici/iStockphoto.

The method of recursion is a powerful technique for breaking up complex computational problems into simpler, often smaller, ones. The term “recursion” refers to the fact that the same computation recurs, or occurs repeatedly, as the problem is solved. Recursion is often the most natural way of thinking about a problem, and there are some computations that are very difficult to perform without recursion. This chapter shows you both simple and complex examples of recursion and teaches you how to “think recursively”.

## 13.1 Triangle Numbers

We begin this chapter with a very simple example that demonstrates the power of thinking recursively. In this example, we will look at triangle shapes such as this one:



We'd like to compute the area of a triangle of width  $n$ , assuming that each [] square has area 1. The area of the triangle is sometimes called the  $n^{\text{th}}$  triangle number. For example, as you can tell from looking at the triangle above, the third triangle number is 6.

You may know that there is a very simple formula to compute these numbers, but you should pretend for now that you don't know about it. The ultimate purpose of this section is not to compute triangle numbers, but to learn about the concept of **recursion** by working through a simple example.

Here is the outline of the class that we will develop:

```
public class Triangle
{
    private int width;

    public Triangle(int aWidth)
    {
        width = aWidth;
    }

    public int getArea()
    {
        ...
    }
}
```

If the width of the triangle is 1, then the triangle consists of a single square, and its area is 1. Let's take care of this case first:

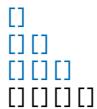
```
public int getArea()
{
    if (width == 1) { return 1; }
    ...
}
```



© Davis Mantei/iStockphoto.

*Using the same method as the one in this section, you can compute the volume of a Mayan pyramid.*

To deal with the general case, consider this picture:



Suppose we knew the area of the smaller, colored triangle. Then we could easily compute the area of the larger triangle as

$\text{smallerArea} + \text{width}$

How can we get the smaller area? Let's make a smaller triangle and ask it!

```
Triangle smallerTriangle = new Triangle(width - 1);
int smallerArea = smallerTriangle.getArea();
```

Now we can complete the `getArea` method:

```
public int getArea()
{
    if (width == 1) { return 1; }
    else
    {
        Triangle smallerTriangle = new Triangle(width - 1);
        int smallerArea = smallerTriangle.getArea();
        return smallerArea + width;
    }
}
```

A recursive computation solves a problem by using the solution to the same problem with simpler inputs.

Here is an illustration of what happens when we compute the area of a triangle of width 4:

- The `getArea` method makes a smaller triangle of width 3.
- It calls `getArea` on that triangle.
  - That method makes a smaller triangle of width 2.
  - It calls `getArea` on that triangle.
    - That method makes a smaller triangle of width 1.
    - It calls `getArea` on that triangle.
      - That method returns 1.
      - The method returns `smallerArea + width = 1 + 2 = 3`.
    - The method returns `smallerArea + width = 3 + 3 = 6`.
  - The method returns `smallerArea + width = 6 + 4 = 10`.

This solution has one remarkable aspect. To solve the area problem for a triangle of a given width, we use the fact that we can solve the same problem for a lesser width. This is called a *recursive* solution.

The call pattern of a **recursive method** looks complicated, and the key to the successful design of a recursive method is *not to think about it*. Instead, look at the `getArea` method one more time and notice how utterly reasonable it is. If the width is 1, then, of course, the area is 1. The next part is just as reasonable. Compute the area of the smaller triangle *and don't think about why that works*. Then the area of the larger triangle is clearly the sum of the smaller area and the width.

There are two key requirements to make sure that the recursion is successful:

- Every recursive call must simplify the computation in some way.
- There must be special cases to handle the simplest computations directly.

For a recursion to terminate, there must be special cases for the simplest values.

The `getArea` method calls itself again with smaller and smaller width values. Eventually the width must reach 1, and there is a special case for computing the area of a triangle with width 1. Thus, the `getArea` method always succeeds.

Actually, you have to be careful. What happens when you call the area of a triangle with width -1? It computes the area of a triangle with width -2, which computes the area of a triangle with width -3, and so on. To avoid this, the `getArea` method should return 0 if the width is  $\leq 0$ .

Recursion is not really necessary to compute the triangle numbers. The area of a triangle equals the sum

$$1 + 2 + 3 + \dots + \text{width}$$

Of course, we can program a simple loop:

```
double area = 0;
for (int i = 1; i <= width; i++)
{
    area = area + i;
}
```

Many simple recursions can be computed as loops. However, loop equivalents for more complex recursions—such as the one in our next example—can be complex.

Actually, in this case, you don't even need a loop to compute the answer. The sum of the first  $n$  integers can be computed as

$$1 + 2 + \dots + n = n \times (n + 1)/2$$

Thus, the area equals

$$\text{width} * (\text{width} + 1) / 2$$

Therefore, neither recursion nor a loop is required to solve this problem. The recursive solution is intended as a “warm-up” to introduce you to the concept of recursion.

### **section\_1/Triangle.java**

```
1  /**
2   * A triangular shape composed of stacked unit squares like this:
3   * []
4   * []
5   * []
6   * ...
7  */
8  public class Triangle
9  {
10     private int width;
11
12    /**
13     * Constructs a triangular shape.
14     * @param aWidth the width (and height) of the triangle
15     */
16    public Triangle(int aWidth)
17    {
18        width = aWidth;
19    }
20
21    /**
22     * Computes the area of the triangle.
23     * @return the area
24     */
25    public int getArea()
26    {
```

```

27     if (width <= 0) { return 0; }
28     else if (width == 1) { return 1; }
29     else
30     {
31         Triangle smallerTriangle = new Triangle(width - 1);
32         int smallerArea = smallerTriangle.getArea();
33         return smallerArea + width;
34     }
35 }
36 }
```

**section\_1/TriangleTester.java**

```

1  public class TriangleTester
2  {
3      public static void main(String[] args)
4      {
5          Triangle t = new Triangle(10);
6          int area = t.getArea();
7          System.out.println("Area: " + area);
8          System.out.println("Expected: 55");
9      }
10 }
```

**Program Run**

Area: 55  
Expected: 55

**SELF CHECK**

- Why is the statement `else if (width == 1) { return 1; }` in the final version of the `getArea` method unnecessary?
- How would you modify the program to recursively compute the area of a square?
- In some cultures, numbers containing the digit 8 are lucky numbers. What is wrong with the following method that tries to test whether a number is lucky?

```
public static boolean isLucky(int number)
{
    int lastDigit = number % 10;
    if (lastDigit == 8) { return true; }
    else
    {
        return isLucky(number / 10); // Test the number without the last digit
    }
}
```

- In order to compute a power of two, you can take the next-lower power and double it. For example, if you want to compute  $2^{11}$  and you know that  $2^{10} = 1024$ , then  $2^{11} = 2 \times 1024 = 2048$ . Write a recursive method `public static int pow2(int n)` that is based on this observation.
- Consider the following recursive method:

```
public static int mystery(int n)
{
    if (n <= 0) { return 0; }
    else
    {
```

```

        int smaller = n - 1;
        return mystery(smaller) + n * n;
    }
}

```

What is `mystery(4)`?

**Practice It** Now you can try these exercises at the end of the chapter: E13.1, E13.2, E13.12.

### Common Error 13.1



### Infinite Recursion

A common programming error is an *infinite recursion*: a method calling itself over and over with no end in sight. The computer needs some amount of memory for bookkeeping for each call. After some number of calls, all memory that is available for this purpose is exhausted. Your program shuts down and reports a “stack overflow”.

Infinite recursion happens either because the arguments don’t get simpler or because a special terminating case is missing. For example, suppose the `getArea` method was allowed to compute the area of a triangle with width 0. If it weren’t for the special test, the method would construct triangles with width -1, -2, -3, and so on.

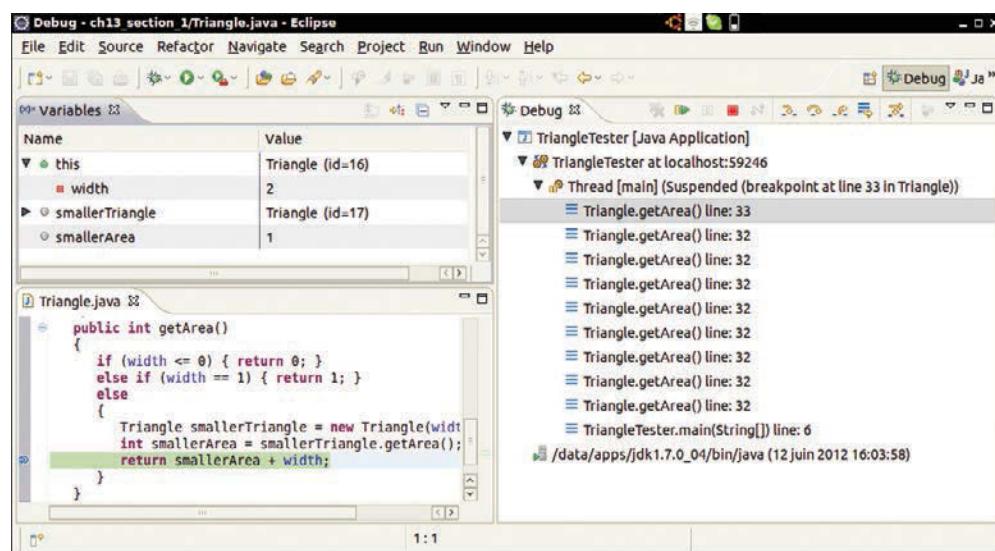
### Common Error 13.2



### Tracing Through Recursive Methods

Debugging a recursive method can be somewhat challenging. When you set a **breakpoint** in a recursive method, the program stops as soon as that program line is encountered in *any call to the recursive method*. Suppose you want to debug the recursive `getArea` method of the `Triangle` class. Debug the `TriangleTester` program and run until the beginning of the `getArea` method. Inspect the `width` instance variable. It is 10.

Remove the breakpoint and now run until the statement `return smallerArea + width;` (see Figure 1). When you inspect `width` again, its value is 2! That makes no sense. There was no instruction that changed the value of `width`. Is that a bug with the debugger?



**Figure 1** Debugging a Recursive Method

No. The program stopped in the first recursive call to `getArea` that reached the return statement. If you are confused, look at the **call stack** (top right in the figure). You will see that nine calls to `getArea` are pending.

You can debug recursive methods with the debugger. You just need to be particularly careful, and watch the call stack to understand which nested call you currently are in.

## HOW TO 13.1



## Thinking Recursively

Solving a problem recursively requires a different mindset than solving it by programming a loop. In fact, it helps if you pretend to be a bit lazy, asking others to do most of the work for you. If you need to solve a complex problem, pretend that “someone else” will do most of the heavy lifting and solve the problem for simpler inputs. Then you only need to figure out how you can turn the solutions with simpler inputs into a solution for the whole problem. To illustrate the technique of recursion, let us consider the following problem.

**Problem Statement** Test whether a sentence is a *palindrome*—a string that is equal to itself when you reverse all characters.



© Nikada/iStockphoto.

*Thinking recursively is easy if you can recognize a subtask that is similar to the original task.*

Typical examples of palindromes are

- A man, a plan, a canal—Panama!
  - Go hang a salami, I’m a lasagna hog
- and, of course, the oldest palindrome of all:
- Madam, I’m Adam

When testing for a palindrome, we match upper- and lowercase letters, and ignore all spaces and punctuation marks.

We want to implement the `isPalindrome` method in the following class:

```
public class Palindromes
{
    ...
    /**
     * Tests whether a text is a palindrome.
     * @param text a string that is being checked
     * @return true if text is a palindrome, false otherwise
     */
    public static boolean isPalindrome(String text)
    {
        ...
    }
}
```

**Step 1** Consider various ways to simplify inputs.

In your mind, focus on a particular input or set of inputs for the problem that you want to solve. Think how you can simplify the inputs in such a way that the same problem can be applied to the simpler input.

When you consider simpler inputs, you may want to remove just a little bit from the original input—maybe remove one or two characters from a string, or remove a small portion of a

geometric shape. But sometimes it is more useful to cut the input in half and then see what it means to solve the problem for both halves.

In the palindrome test problem, the input is the string that we need to test. How can you simplify the input? Here are several possibilities:

- Remove the first character.
- Remove the last character.
- Remove both the first and last characters.
- Remove a character from the middle.
- Cut the string into two halves.

These simpler inputs are all potential inputs for the palindrome test.

### **Step 2** Combine solutions with simpler inputs into a solution of the original problem.

In your mind, consider the solutions of your problem for the simpler inputs that you discovered in Step 1. Don't worry *how* those solutions are obtained. Simply have faith that the solutions are readily available. Just say to yourself: These are simpler inputs, so someone else will solve the problem for me.

Now think how you can turn the solution for the simpler inputs into a solution for the input that you are currently thinking about. Maybe you need to add a small quantity, related to the quantity that you lopped off to arrive at the simpler input. Maybe you cut the original input in half and have solutions for each half. Then you may need to add both solutions to arrive at a solution for the whole.

Consider the methods for simplifying the inputs for the palindrome test. Cutting the string in half doesn't seem a good idea. If you cut

"Madam, I'm Adam"

in half, you get two strings:

"Madam, I"

and

"'m Adam"

Neither of them is a palindrome. Cutting the input in half and testing whether the halves are palindromes seems a dead end.

The most promising simplification is to remove the first *and* last characters. Removing the M at the front and the m at the back yields

"adam, I'm Ada"

Suppose you can verify that the shorter string is a palindrome. Then *of course* the original string is a palindrome—we put the same letter in the front and the back. That's extremely promising. A word is a palindrome if

- The first and last letters match (ignoring letter case)
- and
- The word obtained by removing the first and last letters is a palindrome.

Again, don't worry how the test works for the shorter string. It just works.

There is one other case to consider. What if the first or last letter of the word is not a letter? For example, the string

"A man, a plan, a canal, Panama!"

ends in a ! character, which does not match the A in the front. But we should ignore non-letters when testing for palindromes. Thus, when the last character is not a letter but the first character is a letter, it doesn't make sense to remove both the first and the last characters. That's not a problem. Remove only the last character. If the shorter string is a palindrome, then it stays a palindrome when you attach a nonletter.

The same argument applies if the first character is not a letter. Now we have a complete set of cases.

- If the first and last characters are both letters, then check whether they match. If so, remove both and test the shorter string.
- Otherwise, if the last character isn't a letter, remove it and test the shorter string.
- Otherwise, the first character isn't a letter. Remove it and test the shorter string.

In all three cases, you can use the solution to the simpler problem to arrive at a solution to your problem.

### Step 3

Find solutions to the simplest inputs.

A recursive computation keeps simplifying its inputs. Eventually it arrives at very simple inputs. To make sure that the recursion comes to a stop, you must deal with the simplest inputs separately. Come up with special solutions for them, which is usually very easy.

However, sometimes you get into philosophical questions dealing with *degenerate* inputs: empty strings, shapes with no area, and so on. Then you may want to investigate a slightly larger input that gets reduced to such a trivial input and see what value you should attach to the degenerate inputs so that the simpler value, when used according to the rules you discovered in Step 2, yields the correct answer.

Let's look at the simplest strings for the palindrome test:

- Strings with two characters
- Strings with a single character
- The empty string

We don't need a special solution for strings with two characters. Step 2 still applies to those strings—either or both of the characters are removed. But we do need to worry about strings of length 0 and 1. In those cases, Step 2 can't apply. There aren't two characters to remove.

The empty string is a palindrome—it's the same string when you read it backwards. If you find that too artificial, consider a string "mm". According to the rule discovered in Step 2, this string is a palindrome if the first and last characters of that string match and the remainder—that is, the empty string—is also a palindrome. Therefore, it makes sense to consider the empty string a palindrome.

A string with a single letter, such as "I", is a palindrome. How about the case in which the character is not a letter, such as "?"? Removing the ! yields the empty string, which is a palindrome. Thus, we conclude that all strings of length 0 or 1 are palindromes.

### Step 4

Implement the solution by combining the simple cases and the reduction step.

Now you are ready to implement the solution. Make separate cases for the simple inputs that you considered in Step 3. If the input isn't one of the simplest cases, then implement the logic you discovered in Step 2.

Here is the `isPalindrome` method:

```
public static boolean isPalindrome(String text)
{
    int length = text.length();

    // Separate case for shortest strings.
    if (length <= 1) { return true; }
    else
    {
        // Get first and last characters, converted to lowercase.
        char first = Character.toLowerCase(text.charAt(0));
        char last = Character.toLowerCase(text.charAt(length - 1));

        if (Character.isLetter(first) && Character.isLetter(last))
        {
            // Both are letters.
            if (first == last)
            {

```

```

        // Remove both first and last character.
        String shorter = text.substring(1, length - 1);
        return isPalindrome(shorter);
    }
    else
    {
        return false;
    }
}
else if (!Character.isLetter(last))
{
    // Remove last character.
    String shorter = text.substring(0, length - 1);
    return isPalindrome(shorter);
}
else
{
    // Remove first character.
    String shorter = text.substring(1);
    return isPalindrome(shorter);
}
}
}

```

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download the complete Palindromes class.

**WORKED EXAMPLE 13.1****Finding Files**

Learn how to use recursion to find all files with a given extension in a directory tree. Go to [wiley.com/go/bjeo6examples](http://wiley.com/go/bjeo6examples) and download Worked Example 13.1.



## 13.2 Recursive Helper Methods

Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.

Sometimes it is easier to find a recursive solution if you change the original problem slightly. Then the original problem can be solved by calling a recursive helper method.

Here is a typical example: In the palindrome test of How To 13.1, it is a bit inefficient to construct new string objects in every step. Consider the following change in the problem: Instead of testing whether the entire sentence is a palindrome, let's check whether a substring is a palindrome:

```

/**
 * Tests whether a substring is a palindrome.
 * @param text a string that is being checked
 * @param start the index of the first character of the substring
 * @param end the index of the last character of the substring
 * @return true if the substring is a palindrome
 */
public static boolean isPalindrome(String text, int start, int end)

```



© geremne/iStockphoto.

*Sometimes, a task can be solved by handing it off to a recursive helper method.*

This method turns out to be even easier to implement than the original test. In the recursive calls, simply adjust the start and end parameter variables to skip over matching letter pairs and characters that are not letters. There is no need to construct new String objects to represent the shorter strings.

```
public static boolean isPalindrome(String text, int start, int end)
{
    // Separate case for substrings of length 0 and 1.
    if (start >= end) { return true; }
    else
    {
        // Get first and last characters, converted to lowercase.
        char first = Character.toLowerCase(text.charAt(start));
        char last = Character.toLowerCase(text.charAt(end));

        if (Character.isLetter(first) && Character.isLetter(last))
        {
            if (first == last)
            {
                // Test substring that doesn't contain the matching letters.
                return isPalindrome(text, start + 1, end - 1);
            }
            else
            {
                return false;
            }
        }
        else if (!Character.isLetter(last))
        {
            // Test substring that doesn't contain the last character.
            return isPalindrome(text, start, end - 1);
        }
        else
        {
            // Test substring that doesn't contain the first character.
            return isPalindrome(text, start + 1, end);
        }
    }
}
```

You should still supply a method to solve the whole problem—the user of your method shouldn't have to know about the trick with the substring positions. Simply call the helper method with positions that test the entire string:

```
public static boolean isPalindrome(String text)
{
    return isPalindrome(text, 0, text.length() - 1);
}
```

#### FULL CODE EXAMPLE

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download the `Palindromes` class with a helper method.

Note that this call is *not* a recursive method call. The `isPalindrome(String)` method calls the helper method `isPalindrome(String, int, int)`. In this example, we use **overloading** to declare two methods with the same name. The `isPalindrome` method with just a `String` parameter variable is the method that we expect the public to use. The second method, with one `String` and two `int` parameter variables, is the recursive helper method. If you prefer, you can avoid overloaded methods by choosing a different name for the helper method, such as `substringIsPalindrome`.

Use the technique of recursive helper methods whenever it is easier to solve a recursive problem that is equivalent to the original problem—but more amenable to a recursive solution.



6. Do we have to give the same name to both `isPalindrome` methods?
7. When does the recursive `isPalindrome` method stop calling itself?
8. To compute the sum of the values in an array, add the first value to the sum of the remaining values, computing recursively. Design a recursive helper method to solve this problem.
9. How can you write a recursive method `public static void sum(int[] a)` without needing a helper function? Why is this less efficient?

**Practice It** Now you can try these exercises at the end of the chapter: E13.6, E13.9, E13.13.

## 13.3 The Efficiency of Recursion

As you have seen in this chapter, recursion can be a powerful tool for implementing complex algorithms. On the other hand, recursion can lead to algorithms that perform poorly. In this section, we will analyze the question of when recursion is beneficial and when it is inefficient.

Consider the Fibonacci sequence: a sequence of numbers defined by the equation

$$\begin{aligned}f_1 &= 1 \\f_2 &= 1 \\f_n &= f_{n-1} + f_{n-2}\end{aligned}$$



*In most cases, iterative and recursive approaches have comparable efficiency.*

© pagadesign/iStockphoto.

That is, each value of the sequence is the sum of the two preceding values. The first ten terms of the sequence are

1, 1, 2, 3, 5, 8, 13, 21, 34, 55

It is easy to extend this sequence indefinitely. Just keep appending the sum of the last two values of the sequence. For example, the next entry is  $34 + 55 = 89$ .

We would like to write a method that computes  $f_n$  for any value of  $n$ . Here we translate the definition directly into a recursive method:

### section\_3/RecursiveFib.java

```

1 import java.util.Scanner;
2
3 /**
4  * This program computes Fibonacci numbers using a recursive method.
5 */
6 public class RecursiveFib
7 {
8     public static void main(String[] args)
9     {
10        Scanner in = new Scanner(System.in);
11        System.out.print("Enter n: ");
12        int n = in.nextInt();

```

```

13     for (int i = 1; i <= n; i++)
14     {
15         long f = fib(i);
16         System.out.println("fib(" + i + ") = " + f);
17     }
18 }
19
20 /**
21  * Computes a Fibonacci number.
22  * @param n an integer
23  * @return the nth Fibonacci number
24 */
25 public static long fib(int n)
26 {
27     if (n <= 2) { return 1; }
28     else { return fib(n - 1) + fib(n - 2); }
29 }
30 }
31 }
```

### Program Run

```

Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
...
fib(50) = 12586269025
```

That is certainly simple, and the method will work correctly. But watch the output closely as you run the test program. The first few calls to the `fib` method are fast. For larger values, though, the program pauses an amazingly long time between outputs.

That makes no sense. Armed with pencil, paper, and a pocket calculator you could calculate these numbers pretty quickly, so it shouldn't take the computer anywhere near that long.

To find out the problem, let us insert **trace messages** into the method:

### section\_3/RecursiveFibTracer.java

```

1 import java.util.Scanner;
2
3 /**
4  * This program prints trace messages that show how often the
5  * recursive method for computing Fibonacci numbers calls itself.
6 */
7 public class RecursiveFibTracer
8 {
9     public static void main(String[] args)
10    {
11        Scanner in = new Scanner(System.in);
12        System.out.print("Enter n: ");
13        int n = in.nextInt();
14 }
```

```
15     long f = fib(n);
16
17     System.out.println("fib(" + n + ") = " + f);
18 }
19
20 /**
21  * Computes a Fibonacci number.
22  * @param n an integer
23  * @return the nth Fibonacci number
24 */
25 public static long fib(int n)
26 {
27     System.out.println("Entering fib: n = " + n);
28     long f;
29     if (n <= 2) { f = 1; }
30     else { f = fib(n - 1) + fib(n - 2); }
31     System.out.println("Exiting fib: n = " + n
32                         + " return value = " + f);
33     return f;
34 }
35 }
```

### Program Run

```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Exiting fib: n = 6 return value = 8
fib(6) = 8
```

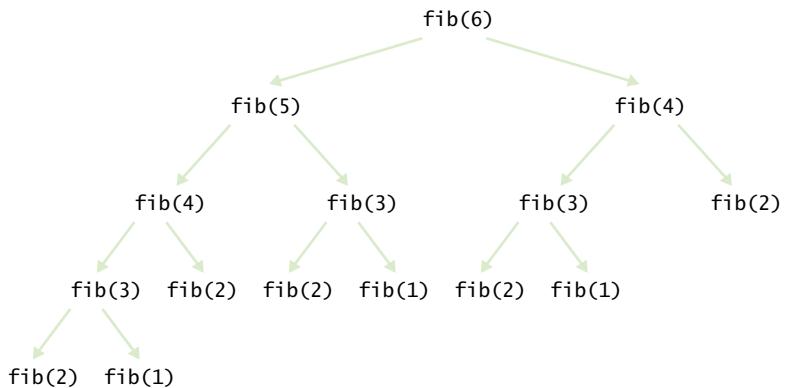
**Figure 2** Call Pattern of the Recursive fib Method

Figure 2 shows the pattern of recursive calls for computing  $\text{fib}(6)$ . Now it is becoming apparent why the method takes so long. It is computing the same values over and over. For example, the computation of  $\text{fib}(6)$  calls  $\text{fib}(4)$  twice and  $\text{fib}(3)$  three times. That is very different from the computation we would do with pencil and paper. There we would just write down the values as they were computed and add up the last two to get the next one until we reached the desired entry; no sequence value would ever be computed twice.

If we imitate the pencil-and-paper process, then we get the following program:

### section\_3/LoopFib.java

```

1  import java.util.Scanner;
2
3  /**
4   * This program computes Fibonacci numbers using an iterative method.
5  */
6  public class LoopFib
7  {
8      public static void main(String[] args)
9      {
10         Scanner in = new Scanner(System.in);
11         System.out.print("Enter n: ");
12         int n = in.nextInt();
13
14         for (int i = 1; i <= n; i++)
15         {
16             long f = fib(i);
17             System.out.println("fib(" + i + ") = " + f);
18         }
19     }
20
21     /**
22      Computes a Fibonacci number.
23      @param n an integer
24      @return the nth Fibonacci number
25     */
26     public static long fib(int n)
27     {
28         if (n <= 2) { return 1; }
29         else
30         {
  
```

```

31     long olderValue = 1;
32     long oldValue = 1;
33     long newValue = 1;
34     for (int i = 3; i <= n; i++)
35     {
36         newValue = oldValue + olderValue;
37         oldValue = oldValue;
38         oldValue = newValue;
39     }
40     return newValue;
41 }
42 }
43 }
```

### Program Run

```

Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
...
fib(50) = 12586269025
```

This method runs *much* faster than the recursive version.

In this example of the `fib` method, the recursive solution was easy to program because it followed the mathematical definition, but it ran far more slowly than the iterative solution, because it computed many intermediate results multiple times.

Can you always speed up a recursive solution by changing it into a loop? Frequently, the iterative and recursive solution have essentially the same performance. For example, here is an iterative solution for the palindrome test:

```

public static boolean isPalindrome(String text)
{
    int start = 0;
    int end = text.length() - 1;
    while (start < end)
    {
        char first = Character.toLowerCase(text.charAt(start));
        char last = Character.toLowerCase(text.charAt(end));

        if (Character.isLetter(first) && Character.isLetter(last))
        {
            // Both are letters.
            if (first == last)
            {
                start++;
                end--;
            }
            else { return false; }
        }
        if (!Character.isLetter(last)) { end--; }
        if (!Character.isLetter(first)) { start++; }
    }
    return true;
}
```

Occasionally, a recursive solution is much slower than its iterative counterpart. However, in most cases, the recursive solution is only slightly slower.

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bjelo6code](http://wiley.com/go/bjelo6code) to download the LoopPalindromes class.

In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.

**SELF CHECK**

10. Is it faster to compute the triangle numbers recursively, as shown in Section 13.1, or is it faster to use a loop that computes  $1 + 2 + 3 + \dots + \text{width}$ ?
11. You can compute the factorial function either with a loop, using the definition that  $n! = 1 \times 2 \times \dots \times n$ , or recursively, using the definition that  $0! = 1$  and  $n! = (n - 1)! \times n$ . Is the recursive approach inefficient in this case?
12. To compute the sum of the values in an array, you can split the array in the middle, recursively compute the sums of the halves, and add the results. Compare the performance of this algorithm with that of a loop that adds the values.

**Practice It**

Now you can try these exercises at the end of the chapter: R13.7, R13.9, E13.7, E13.27.

## 13.4 Permutations

The permutations of a string can be obtained more naturally through recursion than with a loop.

In this section, we will study a more complex example of recursion that would be difficult to program with a simple loop. (As Exercise P13.3 shows, it is possible to avoid the recursion, but the resulting solution is quite complex, and no faster).

We will design a method that lists all permutations of a string. A permutation is simply a rearrangement of the letters in the string. For example, the string "eat" has six permutations (including the original string itself):

|       |       |
|-------|-------|
| "eat" | "ate" |
| "eta" | "tea" |
| "aet" | "tae" |



© Jeanine Groenwald/  
iStockphoto.

*Using recursion, you can find all arrangements of a set of objects.*

Now we need a way to generate the permutations recursively. Consider the string "eat". Let's simplify the problem. First, we'll generate all permutations that start with the letter 'e', then those that start with 'a', and finally those that start with 't'. How do we generate the permutations that start with 'e'? We need to know the permutations of the substring "at". But that's the same problem—to generate all permutations—with a simpler input, namely the shorter string "at". Thus, we can use recursion. Generate the permutations of the substring "at". They are

```
"at"
"ta"
```

For each permutation of that substring, prepend the letter 'e' to get the permutations of "eat" that start with 'e', namely

```
"eat"
"eta"
```

Now let's turn our attention to the permutations of "eat" that start with 'a'. We need to produce the permutations of the remaining letters, "et". They are:

```
"et"
"te"
```

We add the letter 'a' to the front of the strings and obtain

```
"aet"
"ate"
```

We generate the permutations that start with 't' in the same way.

That's the idea. The implementation is fairly straightforward. In the `permutations` method, we loop through all positions in the word to be permuted. For each of them, we compute the shorter word that is obtained by removing the *i*th letter:

```
String shorter = word.substring(0, i) + word.substring(i + 1);
```

We compute the permutations of the shorter word:

```
ArrayList<String> shorterPermutations = permutations(shorter);
```

Finally, we add the removed letter to the front of all permutations of the shorter word.

```
for (String s : shorterPermutations)
{
    result.add(word.charAt(i) + s);
}
```

As always, we have to provide a special case for the simplest strings. The simplest possible string is the empty string, which has a single permutation—itself.

Here is the complete `Permutations` class:

#### [section\\_4/Permutations.java](#)

```
1 import java.util.ArrayList;
2
3 /**
4     This class computes permutations of a string.
5 */
6 public class Permutations
7 {
8     public static void main(String[] args)
9     {
10         for (String s : permutations("eat"))
11         {
```

```

12         System.out.println(s);
13     }
14 }
15 /**
16 * Gets all permutations of a given word.
17 * @param word the string to permute
18 * @return a list of all permutations
19 */
20 public static ArrayList<String> permutations(String word)
21 {
22     ArrayList<String> result = new ArrayList<String>();
23
24     // The empty string has a single permutation: itself
25     if (word.length() == 0)
26     {
27         result.add(word);
28         return result;
29     }
30     else
31     {
32         // Loop through all character positions
33         for (int i = 0; i < word.length(); i++)
34         {
35             // Form a shorter word by removing the ith character
36             String shorter = word.substring(0, i) + word.substring(i + 1);
37
38             // Generate all permutations of the simpler word
39             ArrayList<String> shorterPermutations = permutations(shorter)
40
41             // Add the removed character to the front of
42             // each permutation of the simpler word
43             for (String s : shorterPermutations)
44             {
45                 result.add(word.charAt(i) + s);
46             }
47         }
48         // Return all permutations
49         return result;
50     }
51 }
52 }
53 }
```

### Program Run

```

eat
eta
aet
ate
tea
tae
```

Compare the `Permutations` and `Triangle` classes. Both of them work on the same principle. When they work on a more complex input, they first solve the problem for a simpler input. Then they combine the result for the simpler input with additional work to deliver the results for the more complex input. There really is no particular complexity behind that process as long as you think about the solution on that level only. However, behind the scenes, the simpler input creates even simpler input,

which creates yet another simplification, and so on, until one input is so simple that the result can be obtained without further help. It is interesting to think about this process, but it can also be confusing. What's important is that you can focus on the one level that matters—putting a solution together from the slightly simpler problem, ignoring the fact that the simpler problem also uses recursion to get its results.



## Computing & Society 13.1 The Limits of Computation

Have you ever wondered how your instructor or grader makes sure your programming homework is correct? In all likelihood, they look at your solution and perhaps run it with some test inputs. But usually they have a correct solution available. That suggests that there might be an easier way. Perhaps they could feed your program and their correct program into a “program comparator”, a computer program that analyzes both programs and determines whether they both compute the same results. Of course, your solution and the program that is known to be correct need not be identical—what matters is that they produce the same output when given the same input.

How could such a program comparator work? Well, the Java compiler knows how to read a program and make sense of the classes, methods, and statements. So it seems plausible that someone could, with some effort, write a program that reads two Java programs, analyzes what they do, and determines whether they solve the same task. Of course, such a program would be very attractive to instructors, because it could automate the grading process. Thus, even though no such program exists today, it might be tempting to try to develop one and sell it to universities around the world.

However, before you start raising venture capital for such an effort, you should know that theoretical computer scientists have proven that it is impossible to develop such a program, *no matter how hard you try*.

There are quite a few of these unsolvable problems. The first one,

called the *halting problem*, was discovered by the British researcher Alan Turing in 1936. Because his research occurred before the first actual computer was constructed, Turing had to devise a theoretical device, the **Turing machine**, to explain how computers could work. The Turing machine consists of a long magnetic tape, a read/write head, and a program that has numbered instructions of the form: “If the current symbol under the head is *x*, then replace it with *y*, move the head one unit left or right, and continue with instruction *n*” (see the figure on the next page). Interestingly enough, with only these instructions, you can program just as much as with Java, even though it is incredibly tedious to do so. Theoretical computer scientists like Turing machines because they can be described using nothing more than the laws of mathematics.

Expressed in terms of Java, the halting problem states: “It is impossible to write a program with two inputs, namely the source code of an arbitrary Java program *P* and a string *I*, that decides whether the program *P*, when executed with the input *I*, will halt—that is, the program will not get into an infinite loop with the given input”. Of course, for some kinds of programs and inputs, it is possible to decide whether the program halts with the given input. The halting problem asserts that it is impossible to come up with a single decision-making algorithm that works with all programs and inputs. Note that you can't simply run the program *P* on the input *I* to settle this question. If the program runs for 1,000 days, you don't know that the

program is in an infinite loop. Maybe you just have to wait another day for it to stop.

Such a “halt checker”, if it could be written, might also be useful for grading homework. An instructor could use it to screen student submissions to see if they get into an infinite loop with a particular input, and then stop checking them. However, as Turing demonstrated, such a program cannot be written. His argument is ingenious and quite simple.

Suppose a “halt checker” program existed. Let's call it *H*. From *H*, we will develop another program, the “killer” program *K*. *K* does the following computation. Its input is a string containing the source code for a program *R*. It then applies the halt checker on the input program *R* and the input string *R*. That is, it checks whether the program *R* halts if its input is its own source code. It sounds bizarre to feed a program to itself, but it isn't impossible.



Science Photo Library/Photo Researchers.

Alan Turing

**SELF CHECK**

13. What are all permutations of the four-letter word beat?
14. Our recursion for the permutation generator stops at the empty string. What simple modification would make the recursion stop at strings of length 0 or 1?
15. Why isn't it easy to develop an iterative solution for the permutation generator?

**Practice It** Now you can try these exercises at the end of the chapter: E13.14, E13.15.

For example, the Java compiler is written in Java, and you can use it to compile itself. Or, as a simpler example, a word counting program can count the words in its own source code.

When *K* gets the answer from *H* that *R* halts when applied to itself, it is programmed to enter an infinite loop. Otherwise *K* exits. In Java, the program might look like this:

```
public class Killer
{
    public static void main(
        String[] args)
    {
        String r = read program input;
        HaltChecker checker =
            new HaltChecker();
        if (checker.check(r, r))
        {
            while (true)
                { // Infinite loop
                }
        }
        else
        {
            return;
        }
    }
}
```

Now ask yourself: What does the halt checker answer when asked whether *K* halts when given *K* as the input? Maybe it finds out that *K* gets into an infinite loop with such an input. But wait, that can't be right. That would mean that *checker.check(r, r)* returns false when *r* is the program code of *K*. As you can plainly see, in that case, the killer method returns, so *K* didn't get into an infinite loop. That shows that *K* must halt when analyzing itself, so

*checker.check(r, r)* should return true. But then the killer method doesn't terminate—it goes into an infinite loop. That shows that it is logically impossible to implement a program that can check whether *every* program halts on a particular input.

It is sobering to know that there are *limits to computing*. There are problems that no computer program, no matter how ingenious, can answer.

Theoretical computer scientists are working on other research involving the nature of computation. One important question that remains unsettled

to this day deals with problems that in practice are very time-consuming to solve. It may be that these problems are intrinsically hard, in which case it would be pointless to try to look for better algorithms. Such theoretical research can have important practical applications. For example, right now, nobody knows whether the most common encryption schemes used today could be broken by discovering a new algorithm. Knowing that no fast algorithms exist for breaking a particular code could make us feel more comfortable about the security of encryption.

Program

| Instruction number | If tape symbol is | Replace with | Then move head | Then go to instruction |
|--------------------|-------------------|--------------|----------------|------------------------|
| 1                  | 0                 | 2            | right          | 2                      |
|                    | 1                 | 1            | left           | 4                      |
| 2                  | 0                 | 0            | right          | 2                      |
|                    | 1                 | 1            | right          | 2                      |
| 3                  | 2                 | 0            | left           | 3                      |
|                    | 0                 | 0            | left           | 3                      |
| 4                  | 1                 | 1            | left           | 3                      |
|                    | 2                 | 2            | right          | 1                      |
| 4                  | 1                 | 1            | right          | 5                      |
|                    | 2                 | 0            | left           | 4                      |

Control unit

Read/write head



The Turing Machine

## 13.5 Mutual Recursion

In a mutual recursion, a set of cooperating methods calls each other repeatedly.

In the preceding examples, a method called itself to solve a simpler problem. Sometimes, a set of cooperating methods calls each other in a recursive fashion. In this section, we will explore such a **mutual recursion**. This technique is significantly more advanced than the simple recursion that we discussed in the preceding sections.

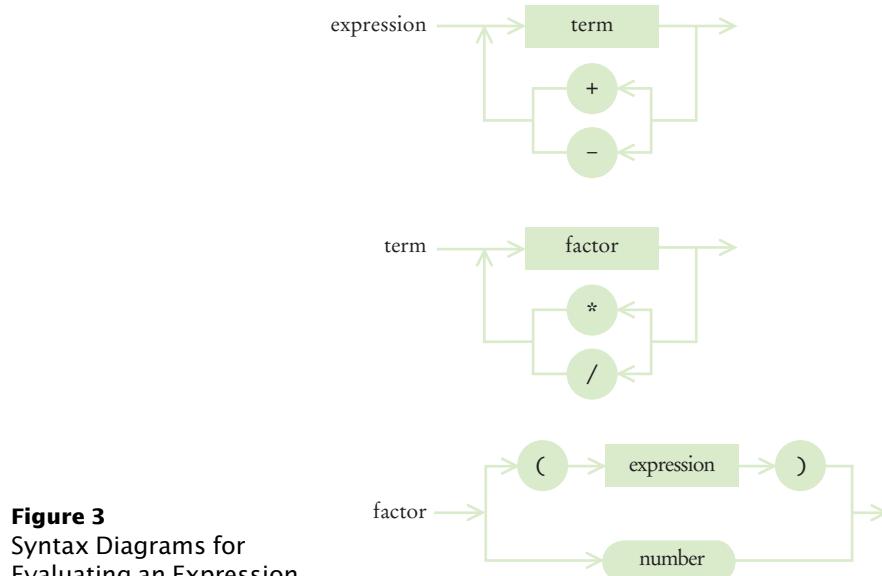
We will develop a program that can compute the values of arithmetic expressions such as

```
3+4*5
(3+4)*5
1-(2-(3-(4-5)))
```

Computing such an expression is complicated by the fact that \* and / bind more strongly than + and -, and that parentheses can be used to group subexpressions.

Figure 3 shows a set of **syntax diagrams** that describes the syntax of these expressions. To see how the syntax diagrams work, consider the expression  $3+4*5$ :

- Enter the *expression* syntax diagram. The arrow points directly to *term*, giving you no alternative.
- Enter the *term* syntax diagram. The arrow points to *factor*, again giving you no choice.
- Enter the *factor* diagram. You have two choices: to follow the top branch or the bottom branch. Because the first input token is the number 3 and not a (, follow the bottom branch.
- Accept the input token because it matches the number. The unprocessed input is now  $+4*5$ .
- Follow the arrow out of *number* to the end of *factor*. As in a method call, you now back up, returning to the end of the *factor* element of the *term* diagram.
- Now you have another choice—to loop back in the *term* diagram, or to exit. The next input token is a +, and it matches neither the \* nor the / that would be required to loop back. So you exit, returning to *expression*.



- Again, you have a choice, to loop back or to exit. Now the + matches one of the choices in the loop. Accept the + in the input and move back to the *term* element. The remaining input is  $4*5$ .

In this fashion, an expression is broken down into a sequence of terms, separated by + or -, each term is broken down into a sequence of factors, each separated by \* or /, and each factor is either a parenthesized expression or a number. You can draw this breakdown as a tree. Figure 4 shows how the expressions  $3+4*5$  and  $(3+4)*5$  are derived from the syntax diagram.

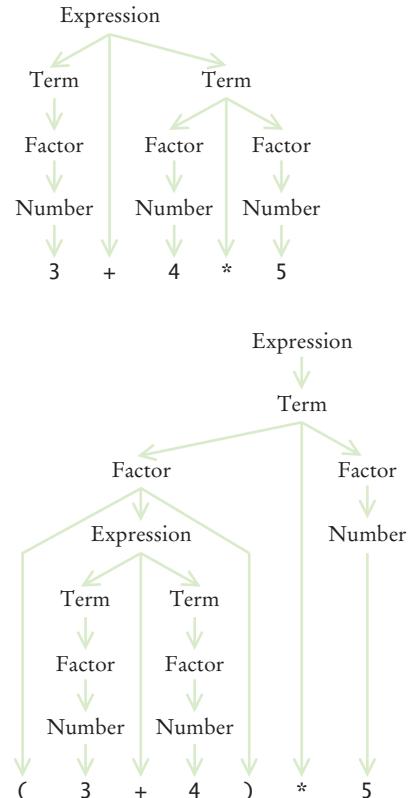
Why do the syntax diagrams help us compute the value of the tree? If you look at the syntax trees, you will see that they accurately represent which operations should be carried out first. In the first tree, 4 and 5 should be multiplied, and then the result should be added to 3. In the second tree, 3 and 4 should be added, and the result should be multiplied by 5.

At the end of this section, you will find the implementation of the `Evaluator` class, which evaluates these expressions. The `Evaluator` makes use of an `ExpressionTokenizer` class, which breaks up an input string into **tokens**—numbers, operators, and parentheses. (For simplicity, we only accept positive integers as numbers, and we don't allow spaces in the input.)

When you call `nextToken`, the next input token is returned as a string. We also supply another method, `peekToken`, which allows you to see the next token without consuming it. To see why the `peekToken` method is necessary, consider the syntax diagram of the *term* type. If the next token is a "\*" or "/", you want to continue adding and subtracting terms. But if the next token is another character, such as a "+" or "-", you want to stop without actually consuming it, so that the token can be considered later.

To compute the value of an expression, we implement three methods: `getExpressionValue`, `getTermValue`, and `getFactorValue`. The `getExpressionValue` method first calls `getTermValue` to get the value of the first term of the expression. Then it checks whether the next input token is one of + or -. If so, it calls `getTermValue` again and adds or subtracts it.

```
public int getExpressionValue()
{
    int value = getTermValue();
    boolean done = false;
    while (!done)
    {
        String next = tokenizer.peekToken();
        if ("+".equals(next) || "-".equals(next))
        {
```



**Figure 4**  
Syntax Trees for Two Expressions

```

        tokenizer.nextToken(); // Discard "+" or "-"
        int value2 = getTermValue();
        if ("+".equals(next)) { value = value + value2; }
        else { value = value - value2; }
    }
    else
    {
        done = true;
    }
}
return value;
}

```

The `getTermValue` method calls `getFactorValue` in the same way, multiplying or dividing the factor values.

Finally, the `getFactorValue` method checks whether the next input is a number, or whether it begins with a `(` token. In the first case, the value is simply the value of the number. However, in the second case, the `getFactorValue` method makes a recursive call to `getExpressionValue`. Thus, the three methods are mutually recursive.

```

public int getFactorValue()
{
    int value;
    String next = tokenizer.peekToken();
    if ("(".equals(next))
    {
        tokenizer.nextToken(); // Discard "("
        value = getExpressionValue();
        tokenizer.nextToken(); // Discard ")"
    }
    else
    {
        value = Integer.parseInt(tokenizer.nextToken());
    }
    return value;
}

```

To see the mutual recursion clearly, trace through the expression  $(3+4)*5$ :

- `getExpressionValue` calls `getTermValue`
  - `getTermValue` calls `getFactorValue`
    - `getFactorValue` consumes the `(` input
    - `getFactorValue` calls `getExpressionValue`
      - `getExpressionValue` returns eventually with the value of 7, having consumed  $3 + 4$ . This is the recursive call.
      - `getFactorValue` consumes the `)` input
      - `getFactorValue` returns 7
    - `getTermValue` consumes the inputs `*` and `5` and returns 35
    - `getExpressionValue` returns 35

As always with a recursive solution, you need to ensure that the recursion terminates. In this situation, that is easy to see when you consider the situation in which `getExpressionValue` calls itself. The second call works on a shorter subexpression than the original expression. At each recursive call, at least some of the tokens of the input string are consumed, so eventually the recursion must come to an end.

**section\_5/Evaluator.java**

```

1  /**
2   * A class that can compute the value of an arithmetic expression.
3  */
4  public class Evaluator
5  {
6      private ExpressionTokenizer tokenizer;
7
8      /**
9       * Constructs an evaluator.
10      * @param anExpression a string containing the expression
11      * to be evaluated
12     */
13    public Evaluator(String anExpression)
14    {
15        tokenizer = new ExpressionTokenizer(anExpression);
16    }
17
18    /**
19     * Evaluates the expression.
20     * @return the value of the expression
21    */
22    public int getExpressionValue()
23    {
24        int value = getTermValue();
25        boolean done = false;
26        while (!done)
27        {
28            String next = tokenizer.peekToken();
29            if ("+".equals(next) || "-".equals(next))
30            {
31                tokenizer.nextToken(); // Discard "+" or "-"
32                int value2 = getTermValue();
33                if ("+".equals(next)) { value = value + value2; }
34                else { value = value - value2; }
35            }
36            else
37            {
38                done = true;
39            }
40        }
41        return value;
42    }
43
44    /**
45     * Evaluates the next term found in the expression.
46     * @return the value of the term
47    */
48    public int getTermValue()
49    {
50        int value = getFactorValue();
51        boolean done = false;
52        while (!done)
53        {
54            String next = tokenizer.peekToken();
55            if ("*".equals(next) || "/".equals(next))
56            {
57                tokenizer.nextToken();
58                int value2 = getFactorValue();

```

```

59         if ("*".equals(next)) { value = value * value2; }
60     } else { value = value / value2; }
61   }
62   else
63   {
64     done = true;
65   }
66 }
67 return value;
68 }
69 /**
70 Evaluates the next factor found in the expression.
71 @return the value of the factor
72 */
73 public int getFactorValue()
74 {
75   int value;
76   String next = tokenizer.peekToken();
77   if ("(".equals(next))
78   {
79     tokenizer.nextToken(); // Discard "("
80     value = getExpressionValue();
81     tokenizer.nextToken(); // Discard ")"
82   }
83   else
84   {
85     value = Integer.parseInt(tokenizer.nextToken());
86   }
87   return value;
88 }
89 }
90 }
```

**section\_5/ExpressionTokenizer.java**

```

1 /**
2  This class breaks up a string describing an expression
3  into tokens: numbers, parentheses, and operators.
4 */
5 public class ExpressionTokenizer
6 {
7   private String input;
8   private int start; // The start of the current token
9   private int end; // The position after the end of the current token
10
11 /**
12  Constructs a tokenizer.
13  @param anInput the string to tokenize
14 */
15 public ExpressionTokenizer(String anInput)
16 {
17   input = anInput;
18   start = 0;
19   end = 0;
20   nextToken(); // Find the first token
21 }
22
23 /**
24  Peeks at the next token without consuming it.
```

```

25     @return the next token or null if there are no more tokens
26 */
27 public String peekToken()
28 {
29     if (start >= input.length()) { return null; }
30     else { return input.substring(start, end); }
31 }
32
33 /**
34  * Gets the next token and moves the tokenizer to the following token.
35  * @return the next token or null if there are no more tokens
36 */
37 public String nextToken()
38 {
39     String r = peekToken();
40     start = end;
41     if (start >= input.length()) { return r; }
42     if (Character.isDigit(input.charAt(start)))
43     {
44         end = start + 1;
45         while (end < input.length()
46                 && Character.isDigit(input.charAt(end)))
47         {
48             end++;
49         }
50     }
51     else
52     {
53         end = start + 1;
54     }
55     return r;
56 }
57 }
```

### section\_5/ExpressionCalculator.java

```

1 import java.util.Scanner;
2
3 /**
4  * This program calculates the value of an expression
5  * consisting of numbers, arithmetic operators, and parentheses.
6 */
7 public class ExpressionCalculator
8 {
9     public static void main(String[] args)
10    {
11        Scanner in = new Scanner(System.in);
12        System.out.print("Enter an expression: ");
13        String input = in.nextLine();
14        Evaluator e = new Evaluator(input);
15        int value = e.getExpressionValue();
16        System.out.println(input + " = " + value);
17    }
18 }
```

### Program Run

Enter an expression: 3+4\*5  
3+4\*5=23



- 16.** What is the difference between a term and a factor? Why do we need both concepts?
- 17.** Why does the expression evaluator use mutual recursion?
- 18.** What happens if you try to evaluate the illegal expression  $3+4^*)5$ ? Specifically, which method throws an exception?

**Practice It** Now you can try these exercises at the end of the chapter: R13.13, E13.21.

## 13.6 Backtracking

Backtracking examines partial solutions, abandoning unsuitable ones and returning to consider other candidates.

Backtracking is a problem solving technique that builds up partial solutions that get increasingly closer to the goal. If a partial solution cannot be completed, one abandons it and returns to examining the other candidates.

Backtracking can be used to solve crossword puzzles, escape from mazes, or find solutions to systems that are constrained by rules. In order to employ backtracking for a particular problem, we need two characteristic properties:

- 1.** A procedure to examine a partial solution and determine whether to
  - Accept it as an actual solution.
  - Abandon it (either because it violates some rules or because it is clear that it can never lead to a valid solution).
  - Continue extending it.
- 2.** A procedure to extend a partial solution, generating one or more solutions that come closer to the goal.

Backtracking can then be expressed with the following recursive algorithm:

```

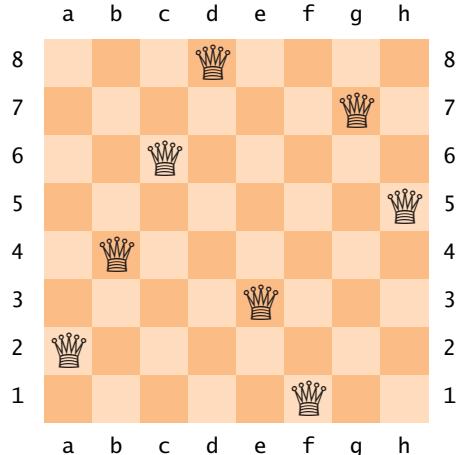
Solve(partialSolution)
  Examine(partialSolution).
  If accepted
    Add partialSolution to the list of solutions.
  Else if continuing
    For each p in extend(partialSolution)
      Solve(p).
  
```

Of course, the processes of examining and extending a partial solution depend on the nature of the problem.



© Lanica Klein/iStockphoto.

*In a backtracking algorithm, one explores all paths toward a solution. When one path is a dead end, one needs to backtrack and try another choice.*



**Figure 5** A Solution to the Eight Queens Problem

As an example, we will develop a program that finds all solutions to the eight queens problem: the task of positioning eight queens on a chess board so that none of them attacks another according to the rules of chess. In other words, there are no two queens on the same row, column, or diagonal. Figure 5 shows a solution.

In this problem, it is easy to examine a partial solution. If two queens attack another, reject it. Otherwise, if it has eight queens, accept it. Otherwise, continue.

It is also easy to extend a partial solution. Simply add another queen on an empty square.

However, in the interest of efficiency, we will be a bit more systematic about the extension process. We will place the first queen in row 1, the next queen in row 2, and so on.

We provide a class `PartialSolution` that collects the queens in a partial solution, and that has methods to examine and extend the solution:

```
public class PartialSolution
{
    private Queen[] queens;

    public int examine() { . . . }
    public PartialSolution[] extend() { . . . }
}
```

The `examine` method simply checks whether two queens attack each other:

```
public int examine()
{
    for (int i = 0; i < queens.length; i++)
    {
        for (int j = i + 1; j < queens.length; j++)
        {
            if (queens[i].attacks(queens[j])) { return ABANDON; }
        }
    }
    if (queens.length == NQUEENS) { return ACCEPT; }
    else { return CONTINUE; }
}
```

The extend method takes a given solution and makes eight copies of it. Each copy gets a new queen in a different column.

```
public PartialSolution[] extend()
{
    // Generate a new solution for each column
    PartialSolution[] result = new PartialSolution[NQUEENS];
    for (int i = 0; i < result.length; i++)
    {
        int size = queens.length;

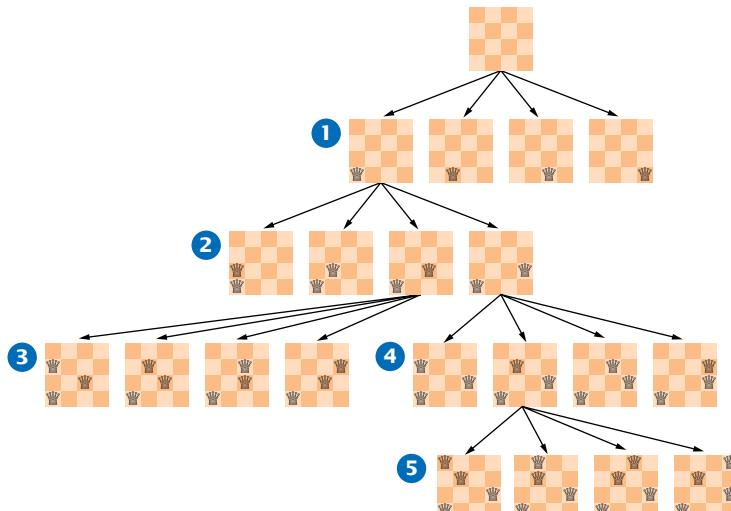
        // The new solution has one more row than this one
        result[i] = new PartialSolution(size + 1);

        // Copy this solution into the new one
        for (int j = 0; j < size; j++)
        {
            result[i].queens[j] = queens[j];
        }

        // Append the new queen into the ith column
        result[i].queens[size] = new Queen(size, i);
    }
    return result;
}
```

You will find the Queen class at the end of the section. The only challenge is to determine when two queens attack each other diagonally. Here is an easy way of checking that. Compute the slope and check whether it is  $\pm 1$ . This condition can be simplified as follows:

$$\begin{aligned} (\text{row}_2 - \text{row}_1)/(\text{column}_2 - \text{column}_1) &= \pm 1 \\ \text{row}_2 - \text{row}_1 &= \pm(\text{column}_2 - \text{column}_1) \\ |\text{row}_2 - \text{row}_1| &= |\text{column}_2 - \text{column}_1| \end{aligned}$$



**Figure 6** Backtracking in the Four Queens Problem

Have a close look at the `solve` method in the `EightQueens` class on page 625. The method is a straightforward translation of the pseudocode for backtracking. Note how there is nothing specific about the eight queens problem in this method—it works for any partial solution with an `examine` and `extend` method (see Exercise E13.22).

Figure 6 shows the `solve` method in action for a four queens problem. Starting from a blank board, there are four partial solutions with a queen in row 1 ①. When the queen is in column 1, there are four partial solutions with a queen in row 2 ②. Two of them are abandoned immediately. The other two lead to partial solutions with three queens ③ and ④, all but one of which are abandoned. One partial solution is extended to four queens, but all of those are abandoned as well ⑤. Then the algorithm backtracks, giving up on a queen in position a1, instead extending the solution with the queen in position b1 (not shown).

When you run the program, it lists 92 solutions, including the one in Figure 5. Exercise E13.23 asks you to remove those that are rotations or reflections of another.

### section\_6/PartialSolution.java

```

1 import java.util.Arrays;
2
3 /**
4  * A partial solution to the eight queens puzzle.
5 */
6 public class PartialSolution
7 {
8     private Queen[] queens;
9     private static final int NQUEENS = 8;
10
11    public static final int ACCEPT = 1;
12    public static final int ABANDON = 2;
13    public static final int CONTINUE = 3;
14
15    /**
16     * Constructs a partial solution of a given size.
17     * @param size the size
18     */
19    public PartialSolution(int size)
20    {
21        queens = new Queen[size];
22    }
23
24    /**
25     * Examines a partial solution.
26     * @return one of ACCEPT, ABANDON, CONTINUE
27     */
28    public int examine()
29    {
30        for (int i = 0; i < queens.length; i++)
31        {
32            for (int j = i + 1; j < queens.length; j++)
33            {
34                if (queens[i].attacks(queens[j])) { return ABANDON; }
35            }
36        }
37        if (queens.length == NQUEENS) { return ACCEPT; }
38        else { return CONTINUE; }
39    }
40

```

```

41  /**
42   * Yields all extensions of this partial solution.
43   * @return an array of partial solutions that extend this solution.
44   */
45  public PartialSolution[] extend()
46  {
47      // Generate a new solution for each column
48      PartialSolution[] result = new PartialSolution[NQUEENS];
49      for (int i = 0; i < result.length; i++)
50      {
51          int size = queens.length;
52
53          // The new solution has one more row than this one
54          result[i] = new PartialSolution(size + 1);
55
56          // Copy this solution into the new one
57          for (int j = 0; j < size; j++)
58          {
59              result[i].queens[j] = queens[j];
60          }
61
62          // Append the new queen into the ith column
63          result[i].queens[size] = new Queen(size, i);
64      }
65      return result;
66  }
67
68  public String toString() { return Arrays.toString(queens); }
69 }
```

### section\_6/Queen.java

```

1  /**
2   * A queen in the eight queens problem.
3   */
4  public class Queen
5  {
6      private int row;
7      private int column;
8
9      /**
10       * Constructs a queen at a given position.
11       * @param r the row
12       * @param c the column
13       */
14      public Queen(int r, int c)
15      {
16          row = r;
17          column = c;
18      }
19
20      /**
21       * Checks whether this queen attacks another.
22       * @param other the other queen
23       * @return true if this and the other queen are in the same
24       *         row, column, or diagonal
25       */
26      public boolean attacks(Queen other)
```

```

27     {
28         return row == other.row
29         || column == other.column
30         || Math.abs(row - other.row) == Math.abs(column - other.column);
31     }
32
33     public String toString()
34     {
35         return "" + "abcdefgh".charAt(column) + (row + 1) ;
36     }
37 }
```

### section\_6/EightQueens.java

```

1  /**
2   * This class solves the eight queens problem using backtracking.
3  */
4  public class EightQueens
5  {
6      public static void main(String[] args)
7      {
8          solve(new PartialSolution(0));
9      }
10
11     /**
12      Prints all solutions to the problem that can be extended from
13      a given partial solution.
14      @param sol the partial solution
15     */
16     public static void solve(PartialSolution sol)
17     {
18         int exam = sol.examine();
19         if (exam == PartialSolution.ACCEPT)
20         {
21             System.out.println(sol);
22         }
23         else if (exam == PartialSolution.CONTINUE)
24         {
25             for (PartialSolution p : sol.extend())
26             {
27                 solve(p);
28             }
29         }
30     }
31 }
```

### Program Run

```

[a1, e2, h3, f4, c5, g6, b7, d8]
[a1, f2, h3, c4, g5, d6, b7, e8]
[a1, g2, d3, f4, h5, b6, e7, c8]
. .
[f1, a2, e3, b4, h5, c6, g7, d8]
. .
[h1, c2, a3, f4, b5, e6, g7, d8]
[h1, d2, a3, c4, f5, b6, g7, e8]

(92 solutions)
```



19. Why does  $j$  begin at  $i + 1$  in the `examine` method?
20. Continue tracing the four queens problem as shown in Figure 6. How many solutions are there with the first queen in position  $a_2$ ?
21. How many solutions are there altogether for the four queens problem?

**Practice It** Now you can try these exercises at the end of the chapter: E13.22, E13.25, E13.26.

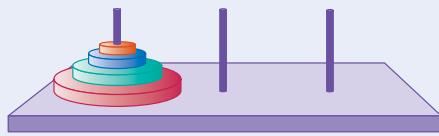


### WORKED EXAMPLE 13.2

#### Towers of Hanoi



No discussion of recursion would be complete without the “Towers of Hanoi”. Learn how to solve this classic puzzle with an elegant recursive solution. Go to [wiley.com/go/bjeo6examples](http://wiley.com/go/bjeo6examples) and download Worked Example 13.2.



### CHAPTER SUMMARY

#### Understand the control flow in a recursive computation.



- A recursive computation solves a problem by using the solution to the same problem with simpler inputs.
- For a recursion to terminate, there must be special cases for the simplest values.



#### Identify recursive helper methods for solving a problem.



- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.

#### Contrast the efficiency of recursive and non-recursive algorithms.



- Occasionally, a recursive solution is much slower than its iterative counterpart. However, in most cases, the recursive solution is only slightly slower.
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.

#### Review a complex recursion example that cannot be solved with a simple loop.

- The permutations of a string can be obtained more naturally through recursion than with a loop.



#### Recognize the phenomenon of mutual recursion in an expression evaluator.

- In a mutual recursion, a set of cooperating methods calls each other repeatedly.

**Use backtracking to solve problems that require trying out multiple paths.**

- Backtracking examines partial solutions, abandoning unsuitable ones and returning to consider other candidates.

**REVIEW EXERCISES**

- **R13.1** Define the terms
  - a. Recursion
  - b. Iteration
  - c. Infinite recursion
  - d. Recursive helper method
  
- **R13.2** Outline, but do not implement, a recursive solution for finding the smallest value in an array.
  
- **R13.3** Outline, but do not implement, a recursive solution for finding the  $k$ th smallest element in an array. *Hint:* Look at the elements that are less than the initial element. Suppose there are  $m$  of them. How should you proceed if  $k \leq m$ ? If  $k > m$ ?
  
- **R13.4** Outline, but do not implement, a recursive solution for sorting an array of numbers. *Hint:* First find the smallest value in the array.
  
- **R13.5** Outline, but do not implement, a recursive solution for sorting an array of numbers. *Hint:* First sort the subarray without the initial element.
  
- **R13.6** Write a recursive definition of  $x^n$ , where  $n \geq 0$ , similar to the recursive definition of the Fibonacci numbers. *Hint:* How do you compute  $x^n$  from  $x^{n-1}$ ? How does the recursion terminate?
  
- **R13.7** Improve upon Exercise R13.6 by computing  $x^n$  as  $(x^{n/2})^2$  if  $n$  is even. Why is this approach significantly faster? *Hint:* Compute  $x^{1023}$  and  $x^{1024}$  both ways.
  
- **R13.8** Write a recursive definition of  $n! = 1 \times 2 \times \dots \times n$ , similar to the recursive definition of the Fibonacci numbers.
  
- **R13.9** Find out how often the recursive version of `fib` calls itself. Keep a static variable `fibCount` and increment it once in every call to `fib`. What is the relationship between `fib(n)` and `fibCount`?
  
- **R13.10** Let  $\text{moves}(n)$  be the number of moves required to solve the Towers of Hanoi problem (see Worked Example 13.2). Find a formula that expresses  $\text{moves}(n)$  in terms of  $\text{moves}(n-1)$ . Then show that  $\text{moves}(n) = 2^n - 1$ .
  
- **R13.11** Outline, but do not implement, a recursive solution for generating all subsets of the set  $\{1, 2, \dots, n\}$ .
  
- **R13.12** Exercise P13.5 shows an iterative way of generating all permutations of the sequence  $(0, 1, \dots, n-1)$ . Explain why the algorithm produces the correct result.
  
- **R13.13** Trace the expression evaluator program from Section 13.5 with inputs  $3 - 4 + 5$ ,  $3 - (4 + 5)$ ,  $(3 - 4) * 5$ , and  $3 * 4 + 5 * 6$ .

## PRACTICE EXERCISES

- **E13.1** Given a class `Rectangle` with instance variables `width` and `height`, provide a recursive `getArea` method. Construct a rectangle whose width is one less than the original and call its `getArea` method.
- ■ **E13.2** Given a class `Square` with an instance variable `width`, provide a recursive `getArea` method. Construct a square whose width is one less than the original and call its `getArea` method.
- **E13.3** Write a recursive method for factoring an integer  $n$ . First, find a factor  $f$ , then recursively factor  $n/f$ .
- **E13.4** Write a recursive method for computing a string with the binary digits of a number. If  $n$  is even, then the last digit is 0. If  $n$  is odd, then the last digit is 1. Recursively obtain the remaining digits.
- **E13.5** Write a recursive method `String reverse(String text)` that reverses a string. For example, `reverse("Hello!")` returns the string "`!olleH`". Implement a recursive solution by removing the first character, reversing the remaining text, and combining the two.
- ■ **E13.6** Redo Exercise E13.5 with a recursive helper method that reverses a substring of the message text.
- **E13.7** Implement the `reverse` method of Exercise E13.5 as an iteration.
- ■ **E13.8** Use recursion to implement a method

```
public static boolean find(String text, String str)
```

that tests whether a given text contains a string. For example, `find("Mississippi", "sip")` returns true.

*Hint:* If the text starts with the string you want to match, then you are done. If not, consider the text that you obtain by removing the first character.

- ■ **E13.9** Use recursion to implement a method

```
public static int indexOf(String text, String str)
```

that returns the starting position of the first substring of the text that matches `str`. Return `-1` if `str` is not a substring of the text.

For example, `s.indexOf("Mississippi", "sip")` returns 6.

*Hint:* This is a bit trickier than Exercise E13.8, because you must keep track of how far the match is from the beginning of the text. Make that value a parameter variable of a helper method.

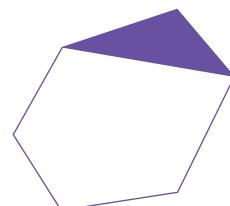
- **E13.10** Using recursion, find the largest element in an array.

*Hint:* Find the largest element in the subset containing all but the last element. Then compare that maximum to the value of the last element.

- **E13.11** Using recursion, compute the sum of all values in an array.

- ■ **E13.12** Using recursion, compute the area of a polygon. Cut off a triangle and use the fact that a triangle with corners  $(x_1, y_1)$ ,  $(x_2, y_2)$ ,  $(x_3, y_3)$  has area

$$\frac{|x_1y_2 + x_2y_3 + x_3y_1 - y_1x_2 - y_2x_3 - y_3x_1|}{2}$$



- ... E13.13** The following method was known to the ancient Greeks for computing square roots. Given a value  $x > 0$  and a guess  $g$  for the square root, a better guess is  $(g + x/g) / 2$ . Write a recursive helper method `public static squareRootGuess(double x, double g)`. If  $g^2$  is approximately equal to  $x$ , return  $g$ , otherwise, return `squareRootGuess` with the better guess. Then write a method `public static squareRoot(double x)` that uses the helper method.
- ... E13.14** Implement a `SubstringGenerator` that generates all substrings of a string. For example, the substrings of the string "rum" are the seven strings
- ```
"r", "ru", "rum", "u", "um", "m", ""
```
- Hint:* First enumerate all substrings that start with the first character. There are  $n$  of them if the string has length  $n$ . Then enumerate the substrings of the string that you obtain by removing the first character.
- ... E13.15** Implement a `SubsetGenerator` that generates all subsets of the characters of a string. For example, the subsets of the characters of the string "rum" are the eight strings
- ```
"rum", "ru", "rm", "r", "um", "u", "m", ""
```
- Note that the subsets don't have to be substrings—for example, "rm" isn't a substring of "rum".
- ... E13.16** Recursively generate all ways in which an array list can be split up into a sequence of nonempty sublists. For example, if you are given the array list [1, 7, 2, 9], return the following lists of lists:
- ```
[[[1], [7], [2], [9]], [[1, 7], [2], [9]], [[1], [7, 2], [9]], [[1, 7, 2], [9]],  
[[1], [7], [2, 9]], [[1, 7], [2, 9]], [[1], [7, 2, 9]], [[1, 7, 2, 9]]]
```
- Hint:* First generate all sublists of the list with the last element removed. The last element can either be a subsequence of length 1, or it can be added to the last subsequence.
- ... E13.17** Given an array list `a` of integers, recursively find all lists of elements of `a` whose sum is a given integer `n`.
- ... E13.18** Suppose you want to climb a staircase with  $n$  steps and you can take either one or two steps at a time. Recursively enumerate all paths. For example, if  $n$  is 5, the possible paths are:
- ```
[1, 2, 3, 4, 5], [1, 3, 4, 5], [1, 2, 4, 5], [1, 2, 3, 5], [1, 4, 5]
```
- ... E13.19** Repeat Exercise E13.18, where the climber can take up to  $k$  steps at a time.
- ... E13.20** Given an integer `price`, list all possible ways of paying for it with \$100, \$20, \$5, and \$1 bills, using recursion. Don't list duplicates.
- ... E13.21** Extend the expression evaluator in Section 13.5 so that it can handle the `%` operator as well as a “raise to a power” operator `^`. For example,  $2^3$  should evaluate to 8. As in mathematics, raising to a power should bind more strongly than multiplication:  $5 * 2^3$  is 40.
- ... E13.22** The backtracking algorithm will work for any problem whose partial solutions can be examined and extended. Provide a `PartialSolution` interface type with methods `examine` and `extend`, a `solve` method that works with this interface type, and a class `EightQueensPartialSolution` that implements the interface.

- **E13.23** Refine the program for solving the eight queens problem so that rotations and reflections of previously displayed solutions are not shown. Your program should display twelve unique solutions.
- **E13.24** Refine the program for solving the eight queens problem so that the solutions are written to an HTML file, using tables with black and white background for the board and the Unicode character ☜ '\u2655' for the white queen.
- **E13.25** Generalize the program for solving the eight queens problem to the  $n$  queens problem. Your program should prompt for the value of  $n$  and display the solutions.
- **E13.26** Using backtracking, write a program that solves summation puzzles in which each letter should be replaced by a digit, such as

send + more = money

Other examples are base + ball = games and kyoto + osaka = tokyo.

- **E13.27** The recursive computation of Fibonacci numbers can be speeded up significantly by keeping track of the values that have already been computed. Provide an implementation of the `fib` method that uses this strategy. Whenever you return a new value, also store it in an auxiliary array. However, before embarking on a computation, consult the array to find whether the result has already been computed. Compare the running time of your improved implementation with that of the original recursive implementation and the loop implementation.

## PROGRAMMING PROJECTS

- **P13.1** Phone numbers and PIN codes can be easier to remember when you find words that spell out the number on a standard phone pad. For example, instead of remembering the combination 5282, you can just think of JAVA.  
Write a recursive method that, given a number, yields all possible spellings (which may or may not be real words).
- **P13.2** Continue Exercise P13.1, checking the words against the `/usr/share/dict/words` file on your computer, or the `words.txt` file in the companion code for this book. For a given number, return only actual words.
- **P13.3** With a longer number, you may need more than one word to remember it on a phone pad. For example, 263-346-5282 is CODE IN JAVA. Using your work from Exercise P13.2, write a program that, given any number, lists all word sequences that spell the number on a phone pad.
- **P13.4** Change the `permutations` method of Section 13.4 (which computed all permutations at once) to a `PermutationIterator` (which computes them one at a time).

```
public class PermutationIterator
{
    public PermutationIterator(String s) { . . . }
    public String nextPermutation() { . . . }
    public boolean hasMorePermutations() { . . . }
}
```

Here is how you would print out all permutations of the string "eat":

```
PermutationIterator iter = new PermutationIterator("eat");
while (iter.hasMorePermutations())
{
```

```

        System.out.println(iter.nextPermutation());
    }
}

```

Now we need a way to iterate through the permutations recursively. Consider the string "eat". As before, we'll generate all permutations that start with the letter 'e', then those that start with 'a', and finally those that start with 't'. How do we generate the permutations that start with 'e'? Make another `PermutationIterator` object (called `tailIterator`) that iterates through the permutations of the substring "at". In the `nextPermutation` method, simply ask `tailIterator` what its next permutation is, and then add the 'e' at the front. However, there is one special case. When the tail generator runs out of permutations, all permutations that start with the current letter have been enumerated. Then

- Increment the current position.
- Compute the tail string that contains all letters except for the current one.
- Make a new permutation iterator for the tail string.

You are done when the current position has reached the end of the string.

**••• P13.5** The following class generates all permutations of the numbers  $0, 1, 2, \dots, n - 1$ , without using recursion.

```

public class NumberPermutationIterator
{
    private int[] a;

    public NumberPermutationIterator(int n)
    {
        a = new int[n];
        done = false;
        for (int i = 0; i < n; i++) { a[i] = i; }
    }

    public int[] nextPermutation()
    {
        if (a.length <= 1) { return a; }

        for (int i = a.length - 1; i > 0; i--)
        {
            if (a[i - 1] < a[i])
            {
                int j = a.length - 1;
                while (a[i - 1] > a[j]) { j--; }
                swap(i - 1, j);
                reverse(i, a.length - 1);
                return a;
            }
        }
        return a;
    }

    public boolean hasMorePermutations()
    {
        if (a.length <= 1) { return false; }
        for (int i = a.length - 1; i > 0; i--)
        {
            if (a[i - 1] < a[i]) { return true; }
        }
        return false;
    }
}

```

```

public void swap(int i, int j)
{
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

public void reverse(int i, int j)
{
    while (i < j) { swap(i, j); i++; j--; }
}
}

```

The algorithm uses the fact that the set to be permuted consists of distinct numbers. Thus, you cannot use the same algorithm to compute the permutations of the characters in a string. You can, however, use this class to get all permutations of the character positions and then compute a string whose  $i$ th character is `word.charAt(a[i])`. Use this approach to reimplement the `PermutationIterator` of Exercise P13.4 without recursion.

- P13.6** Implement an iterator that produces the moves for the Towers of Hanoi puzzle described in Worked Example 13.2. Provide methods `hasMoreMoves` and `nextMove`. The `nextMove` method should yield a string describing the next move. For example, the following code prints all moves needed to move five disks from peg 1 to peg 3:

```

DiskMover mover = new DiskMover(5, 1, 3);
while (mover.hasMoreMoves())
{
    System.out.println(mover.nextMove());
}

```

*Hint:* A disk mover that moves a single disk from one peg to another simply has a `nextMove` method that returns a string

Move disk from peg *source* to *target*

A disk mover with more than one disk to move must work harder. It needs another `DiskMover` to help it move the first  $d - 1$  disks. Then `nextMove` asks that disk mover for its next move until it is done. Then the `nextMove` method issues a command to move the  $d$ th disk. Finally, it constructs another disk mover that generates the remaining moves.

It helps to keep track of the state of the disk mover:

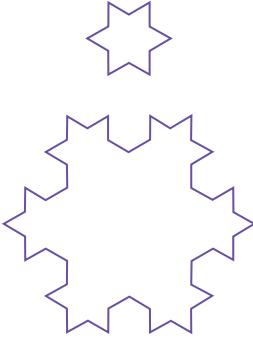
- BEFORE\_LARGEST: A helper mover moves the smaller pile to the other peg.
- LARGEST: Move the largest disk from the source to the destination.
- AFTER\_LARGEST: The helper mover moves the smaller pile from the other peg to the target.
- DONE: All moves are done.

- P13.7** *Escaping a Maze.* You are currently located inside a maze. The walls of the maze are indicated by asterisks (\*).

Use the following recursive approach to check whether you can escape from the maze: If you are at an exit, return true. Recursively check whether you can escape from one of the empty neighboring locations without visiting the current location. This method merely tests whether there is a path out of the maze. Extra credit if you can print out a path that leads to an exit.



- P13.8** Using the `PartialSolution` interface and `solve` method from Exercise E13.22, provide a class `MazePartialSolution` for solving the maze escape problem of Exercise P13.7.

- ... P13.9** The expression evaluator in Section 13.5 returns the value of an expression. Modify the evaluator so that it returns an instance of the Expression interface with five implementing classes, Constant, Sum, Difference, Product, and Quotient. The Expression interface has a method `int value()`. The Constant class stores a number, which is returned by the `value` method. The other four classes store two arguments of type Expression, and their `value` method returns the sum, difference, product, and quotient of the values of the arguments. Write a test program that reads an expression string, translates it into an Expression object, and prints the result of calling `value`.
- ... P13.10** Refine the expression evaluator of Exercise P13.9 so that expressions can contain the variable `x`. For example, `3*x*x+4*x+5` is a valid expression. Change the Expression interface so that its `value` method has as parameter the value that `x` should take. Add a class Variable that denotes an `x`. Write a program that reads an expression string and a value for `x`, translates the expression string into an Expression object, and prints the result of calling `value(x)`.
- ... P13.11** Add a `toString` method to the Expression class (as described in Exercises P13.9 and P13.10) that returns a string representation of the expression. It is ok to use more parentheses than required in mathematical notation. For example, for the expression `3*x*x+5`, you can print `((3*x)*x)+5`.
- ... P13.12** Write a program that reads an expression involving integers and the variable `x` into an Expression object, and then computes the derivative. Add a method `Expression derivative()` to the Expression interface. Use the rules from calculus for computing the derivative of a sum, difference, product, quotient, constant, or variable. Don't simplify the result. Print the resulting expression. For example, when reading `x * x`, you should print `((1*x)+(x*1))`.
- ... Graphics P13.13** *The Koch Snowflake.* A snowflake-like shape is recursively defined as follows. Start with an equilateral triangle:
- 
- Next, increase the size by a factor of three and replace each straight line with four line segments:
- 
- Repeat the process:
- Write a program that draws the iterations of the snowflake shape. Supply a button that, when clicked, produces the next iteration.

## ANSWERS TO SELF-CHECK QUESTIONS

- Suppose we omit the statement. When computing the area of a triangle with width 1, we compute the area of the triangle with width 0 as 0, and then add 1 to arrive at the correct area.

2. You would compute the smaller area recursively, then return

```
smallerArea + width + width - 1.
```



Of course, it would be simpler to compute the area simply as `width * width`. The results are identical because

$$1 + 0 + 2 + 1 + 3 + 2 + \dots + n + n - 1 = \\ \frac{n(n+1)}{2} + \frac{(n-1)n}{2} = n^2$$

3. There is no provision for stopping the recursion. When a number  $< 10$  isn't 8, then the method should return `false` and stop.

```
4. public static int pow2(int n)
{
    if (n <= 0) { return 1; } // 20 is 1
    else { return 2 * pow2(n - 1); }
}
```

```
5. mystery(4) calls mystery(3)
mystery(3) calls mystery(2)
mystery(2) calls mystery(1)
mystery(1) calls mystery(0)
mystery(0) returns 0.
mystery(1) returns 0 + 1 * 1 = 1
mystery(2) returns 1 + 2 * 2 = 5
mystery(3) returns 5 + 3 * 3 = 14
mystery(4) returns 14 + 4 * 4 = 30
```

6. No—the second one could be given a different name such as `substringIsPalindrome`.

7. When `start >= end`, that is, when the investigated string is either empty or has length 1.

8. The method `sumHelper(int[] a, int start)` adds `a[start]` and `sumHelper(a, start + 1)`.

9. `sum(a)` can make a new array `smaller` containing `a[1] ... a[a.length - 1]` and compute `a[0] + sum(smaller)`. But it is inefficient to make a copy of the array in each step.

10. The loop is slightly faster. It is even faster to simply compute `width * (width + 1) / 2`.

11. No, the recursive solution is about as efficient as the iterative approach. Both require  $n - 1$  multiplications to compute  $n!$ .

12. The recursive algorithm performs about as well as the loop. Unlike the recursive Fibonacci algorithm, this algorithm doesn't call itself again on the same input. For example, the sum of the array 1 4 9 16 25 36 49 64 is computed as the sum of 1 4 9 16 and 25 36 49 64, then as the sums of 1 4, 9 16, 25 36, and 49 64, which can be computed directly.

13. They are b followed by the six permutations of eat, e followed by the six permutations of bat, a followed by the six permutations of bet, and t followed by the six permutations of bea.

14. Simply change `if (word.length() == 0)` to `if (word.length() <= 1)`, because a word with a single letter is also its sole permutation.

15. An iterative solution would have a loop whose body computes the next permutation from the previous ones. But there is no obvious mechanism for getting the next permutation. For example, if you already found permutations eat, eta, and aet, it is not clear how you use that information to get the next permutation. Actually, there is an ingenious mechanism for doing just that, but it is far from obvious—see Exercise P13.5.

16. Factors are combined by multiplicative operators (\* and /); terms are combined by additive operators (+, -). We need both so that multiplication can bind more strongly than addition.

17. To handle parenthesized expressions, such as `2+3*(4+5)`. The subexpression `4+5` is handled by a recursive call to `getExpressionValue`.

18. The `Integer.parseInt` call in `getFactorValue` throws an exception when it is given the string ")".

19. We want to check whether any `queen[i]` attacks any `queen[j]`, but attacking is symmetric. That is, we can choose to compare only those for which  $i < j$  (or, alternatively, those for which  $i > j$ ). We don't want to call the `attacks` method when  $i$  equals  $j$ ; it would return true.

20. One solution:



21. Two solutions: The one from Self Check 20, and its mirror image.



## WORKED EXAMPLE 13.1



## Finding Files

**Problem Statement** Your task is to print the names of all files in a directory tree that end in a given extension.

To solve this task, you need to know two methods of the `File` class. A `File` object can represent either a directory or a regular file, and the method

```
boolean isDirectory()
```

tells you which it is. The method

```
File[] listFiles()
```

yields an array of all `File` objects describing the files and directories in a given directory. These can be directories or files.

For example, consider the directory tree at right, and suppose the `File` object `f` represents the `code` directory. Then `f.isDirectory()` returns true, and `f.listFiles()` returns an array containing `File` objects describing `code/ch01`, `code/ch02`, and `code/ch03`.



### Step 1

Consider various ways to simplify inputs.

Our problem has two inputs: A `File` object representing a directory tree, and an extension. Clearly, nothing is gained from manipulating the extension. However, there is an obvious way of chopping up the directory tree:

- Consider all files in the root level of the directory tree.
- Then consider each tree formed by a subdirectory.

This leads to a valid strategy. Find matching files in the root directory, and then recursively find them in each child subdirectory.

```

For each File object in the root
  If the File object is a directory
    Recursively find files in that directory.
  Else if the name ends in the extension
    Print the name.

```

### Step 2

Combine solutions with simpler inputs into a solution of the original problem.

We are asked to simply print the files that we find, so there aren't any results to combine.

Had we been asked to produce an array list of the found files, we would place all matches of the root directory into an array list and add all results from the subdirectories into the same list.

**Step 3** Find solutions to the simplest inputs.

The simplest input is a file that isn't a directory. In that case, we simply check whether it ends in the given extension, and if so, print it.

**Step 4** Implement the solution by combining the simple cases and the reduction step.

We design a class `FileFinder` with a method for finding the matching files:

```
public class FileFinder
{
    private File[] children;

    /**
     * Constructs a file finder for a given directory tree.
     * @param startingDirectory the starting directory of the tree
     */
    public FileFinder(File startingDirectory)
    {
        children = startingDirectory.listFiles();
    }

    /**
     * Prints all files whose names end in a given extension.
     * @param extension a file extension (such as ".java")
     */
    public void find(String extension)
    {
        . . .
    }
}
```

In our case, the reduction step is simply to look at the files and subdirectories:

```
For each child in children
  If the child is a directory
    Recursively find files in the child.
  Else
    If the name of child ends in extension
      Print the name.
```

Here is the complete `FileFinder.find` method:

```
/**
 * Prints all files whose names end in a given extension.
 * @param extension a file extension (such as ".java")
 */
public void find(String extension)
{
    for (File child : children)
    {
        String fileName = child.toString();
        if (child.isDirectory())
        {
            FileFinder finder = new FileFinder(child);
            finder.find(extension);
        }
        else if (fileName.endsWith(extension))
        {
            System.out.println(fileName);
        }
    }
}
```

`FileFinderDemo.java` in the `worked_example_1` directory completes the solution.

In this solution, we used a class for each directory. Alternatively, we can use a recursive static method:

```
/*
 * Prints all files whose names end in a given extension.
 * @param aFile a file or directory
 * @param extension a file extension (such as ".java")
 */
public static void find(File aFile, String extension)
{
    if (aFile.isDirectory())
    {
        for (File child : aFile.listFiles())
        {
            find(child, extension);
        }
    }
    else
    {
        String fileName = aFile.toString();
        if (fileName.endsWith(extension))
        {
            System.out.println(fileName);
        }
    }
}
```

The basic approach is the same. For a file, we check whether it ends in the given extension. If so, we print it. For a directory, we look at all the files and directories inside.

Here, we chose to accept either a file or directory. For that reason, the calling pattern is subtly different. In our first solution, recursive calls are only made on directories. In the second solution, the method is called recursively on all elements of the array returned by `listFiles()`. The recursion ends right away for files.

A complete solution is in the `ch13/worked_example_1` folder of your companion code.

### worked\_example\_1/FileFinder2.java

```
1 import java.io.File;
2
3 public class FileFinder2
4 {
5     public static void main(String[] args)
6     {
7         File startingDirectory = new File("/home/myname");
8         find(startingDirectory, ".java");
9     }
10
11 /**
12  * Prints all files whose names end in a given extension.
13  * @param aFile a file or directory
14  * @param extension a file extension (such as ".java")
15  */
16 public static void find(File aFile, String extension)
17 {
18     if (aFile.isDirectory())
19     {
20         for (File child : aFile.listFiles())
21         {
22             find(child, extension);
23         }
24     }
25 }
```

## WE4 Chapter 13 Recursion

```
23     }
24 }
25 else
26 {
27     String fileName = aFile.toString();
28     if (fileName.endsWith(extension))
29     {
30         System.out.println(fileName);
31     }
32 }
33 }
34 }
```

---



## WORKED EXAMPLE 13.2



## Towers of Hanoi

**Problem Statement** The “Towers of Hanoi” puzzle has a board with three pegs and a stack of disks of decreasing size, initially on the first peg (see Figure 7).

The goal is to move all disks to the third peg. One disk can be moved at one time, from any peg to any other peg. You can place smaller disks only on top of larger ones, not the other way around.

Legend has it that a temple (presumably in Hanoi) contains such an assembly, with sixty-four golden disks, which the priests move in the prescribed fashion. When they have arranged all disks on the third peg, the world will come to an end.

Let us help out by writing a program that prints instructions for moving the disks.

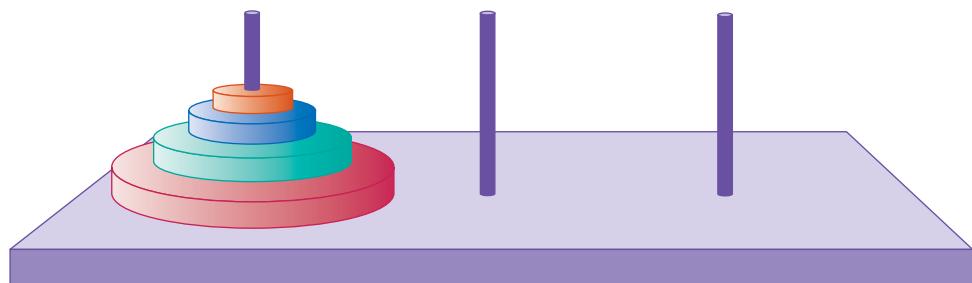


Figure 7 Towers of Hanoi

Consider the problem of moving  $d$  disks from peg  $p_1$  to peg  $p_2$ , where  $p_1$  and  $p_2$  are 1, 2, or 3, and  $p_1 \neq p_2$ . Since  $1 + 2 + 3 = 6$ , we can get the index of the remaining peg as  $p_3 = 6 - p_1 - p_2$ .

Now we can move the disks as follows:

- Move the top  $d - 1$  disks from  $p_1$  to  $p_3$
- Move one disk (the one on the bottom of the pile of  $d$  disks) from  $p_1$  to  $p_2$
- Move the  $d - 1$  disks that were parked on  $p_3$  to  $p_2$

The first and third step need to be handled recursively, but because we move one fewer disk, the recursion will eventually terminate.

It is very straightforward to translate the algorithm into Java. For the second step, we simply print out the instruction for the priest, something like

Move disk from peg 1 to 3

### worked\_example\_2/TowersOfHanoiInstructions.java

```

1  /**
2   * This program prints instructions for solving a Towers of Hanoi puzzle.
3  */
4  public class TowersOfHanoiInstructions
5  {
6      public static void main(String[] args)
7      {
8          move(5, 1, 3);
9      }
10
11 /**
12  * Print instructions for moving a pile of disks from one peg to another.
13  * @param disks the number of disks to move
14  * @param from the peg from which to move the disks
15  * @param to the peg to which to move the disks

```

```

16  */
17  public static void move(int disks, int from, int to)
18  {
19      if (disks > 0)
20      {
21          int other = 6 - from - to;
22          move(disks - 1, from, other);
23          System.out.println("Move disk from peg " + from + " to " + to);
24          move(disks - 1, other, to);
25      }
26  }
27 }
```

**Program Run**

```

Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 3 to 1
Move disk from peg 2 to 1
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 3 to 2
Move disk from peg 3 to 1
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
```

These instructions may suffice for the priests, but unfortunately it is not easy for us to see what is going on. Let's improve the program so that it actually carries out the instructions and shows the contents of the towers after each move.

We use a class `Tower` that manages the disks in one tower. Each disk is represented as an integer indicating its size from 1 to  $n$ , the number of disks in the puzzle.

We provide methods to remove the top disk, to add a disk to the top, and to show the contents of the tower as a list of disk sizes, for example, [5, 4, 1].

**worked\_example\_2/Tower.java**

```

1 import java.util.ArrayList;
2
3 /**
4  * A tower containing disks in the Towers of Hanoi puzzle.
5 */
6 public class Tower
7 {
8     private ArrayList<Integer> disks;
9
10    /**
11     * Constructs a tower holding a given number of disks of decreasing size.
12     * @param ndisks the number of disks
13    */
14    public Tower(int ndisks)
15    {
16        disks = new ArrayList<Integer>();
17        for (int d = ndisks; d >= 1; d--) { disks.add(d); }
18    }
19
20    /**
21     * Removes the top disk from this tower.
22     * @return the size of the removed disk
23    */
24    public int remove()
25    {
26        return disks.remove(disks.size() - 1);
27    }
28
29    /**
30     * Adds a disk to this tower.
31     * @param size the size of the disk to add
32    */
33    public void add(int size)
34    {
35        if (disks.size() > 0 && disks.get(disks.size() - 1) < size)
36        {
37            throw new IllegalStateException("Disk is too large");
38        }
39        disks.add(size);
40    }
41
42    public String toString() { return disks.toString(); }
43}

```

A TowersOfHanoi puzzle has three towers:

```

public class TowersOfHanoi
{
    private Tower[] towers;

    public TowersOfHanoi(int ndisks)
    {
        towers = new Tower[3];
        towers[0] = new Tower(ndisks);
        towers[1] = new Tower(0);
        towers[2] = new Tower(0);
    }
    . .

```

## WE8 Chapter 13 Recursion

```
}
```

Its move method first carries out the move, then prints the contents of the towers:

```
public void move(int disks, int from, int to)
{
    if (disks > 0)
    {
        int other = 3 - from - to;
        move(disks - 1, from, other);
        towers[to].add(towers[from].remove());
        System.out.println(Arrays.toString(towers));
        move(disks - 1, other, to);
    }
}
```

Here, we changed the index values to 0, 1, 2. Therefore, the index of the other peg is  $3 - \text{from} - \text{to}$ .

Here is the main method:

```
public static void main(String[] args)
{
    final int NDISKS = 5;
    TowersOfHanoi towers = new TowersOfHanoi(NDISKS);
    towers.move(NDISKS, 0, 2);
}
```

The program output is

```
[[5, 4, 3, 2], [], [1]]
[[5, 4, 3], [2], [1]]
[[5, 4, 3], [2, 1], []]
[[5, 4], [2, 1], [3]]
[[5, 4, 1], [2], [3]]
[[5, 4, 1], [], [3, 2]]
[[5, 4], [], [3, 2, 1]]
[[5], [4], [3, 2, 1]]
[[5], [4, 1], [3, 2]]
[[5, 2], [4, 1], [3]]
[[5, 2, 1], [4], [3]]
[[5, 2, 1], [4, 3], []]
[[5, 2], [4, 3], [1]]
[[5], [4, 3, 2], [1]]
[[5], [4, 3, 2, 1], []]
[[], [4, 3, 2, 1], [5]]
[[1], [4, 3, 2], [5]]
[[1], [4, 3], [5, 2]]
[[], [4, 3], [5, 2, 1]]
[[3], [4], [5, 2, 1]]
[[3], [4, 1], [5, 2]]
[[3, 2], [4, 1], [5]]
[[3, 2, 1], [4], [5]]
[[3, 2, 1], [], [5, 4]]
[[3, 2], [], [5, 4, 1]]
[[3], [2], [5, 4, 1]]
[[3], [2, 1], [5, 4]]
[[], [2, 1], [5, 4, 3]]
[[1], [2], [5, 4, 3]]
[[1], [], [5, 4, 3, 2]]
[[], [], [5, 4, 3, 2, 1]]
```

That's better. Now you can see how the disks move. You can check that all moves are legal—the disk size always decreases.

You can see that it takes  $31 = 2^5 - 1$  moves to solve the puzzle for 5 disks. With 64 disks, it takes  $2^{64} - 1 = 18446744073709551615$  moves. If the priests can move one disk per second, it takes about 585 billion years to finish the job. Because the earth is about 4.5 billion years old at the time this book is written, we don't have to worry too much whether the world will really come to an end when they are done.

### worked\_example\_2/TowersOfHanoiDemo.java

```

1 import java.util.ArrayList;
2 import java.util.Arrays;
3
4 /**
5     This program shows a solution for a Towers of Hanoi puzzle.
6 */
7 public class TowersOfHanoiDemo
8 {
9     public static void main(String[] args)
10    {
11        final int NDISKS = 5;
12        TowersOfHanoi towers = new TowersOfHanoi(NDISKS);
13        towers.move(NDISKS, 0, 2);
14    }
15 }
```

### worked\_example\_2/TowersOfHanoi.java

```

1 import java.util.Arrays;
2
3 /**
4     A Towers of Hanoi puzzle with three towers.
5 */
6 public class TowersOfHanoi
7 {
8     private Tower[] towers;
9
10    /**
11     Constructs a puzzle in which the first tower has a given number of disks.
12     @param ndisks the number of disks
13    */
14    public TowersOfHanoi(int ndisks)
15    {
16        towers = new Tower[3];
17        towers[0] = new Tower(ndisks);
18        towers[1] = new Tower(0);
19        towers[2] = new Tower(0);
20    }
21
22    /**
23     Moves a pile of disks from one peg to another and displays the movement.
24     @param disks the number of disks to move
25     @param from the peg from which to move the disks
26     @param to the peg to which to move the disks
27    */
28    public void move(int disks, int from, int to)
29    {
30        if (disks > 0)
31        {
32            int other = 3 - from - to;
33            move(disks - 1, from, other);
34        }
35    }
36 }
```

```
34     towers[to].add(towers[from].remove());
35     System.out.println(Arrays.toString(towers));
36     move(disks - 1, other, to);
37   }
38 }
39 }
```

---

# SORTING AND SEARCHING

## CHAPTER GOALS

- To study several sorting and searching algorithms
- To appreciate that algorithms for the same task can differ widely in performance
- To understand the big-Oh notation
- To estimate and compare the performance of algorithms
- To write code to measure the running time of a program



© Volkan Ersoy/iStockphoto.

## CHAPTER CONTENTS

|   |     |
|---|-----|
| <b>14.1 SELECTION SORT</b>  | 636 |
| <b>14.2 PROFILING THE SELECTION SORT ALGORITHM</b>                    | 639 |
| <b>14.3 ANALYZING THE PERFORMANCE OF THE SELECTION SORT ALGORITHM</b> | 642 |
| <b>ST1</b> Oh, Omega, and Theta                                       | 644 |
| <b>ST2</b> Insertion Sort   | 645 |
| <b>14.4 MERGE SORT</b>  | 647 |
| <b>14.5 ANALYZING THE MERGE SORT ALGORITHM</b>                        | 650 |
| <b>ST3</b> The Quicksort Algorithm                                    | 652 |
| <b>14.6 SEARCHING</b>   | 654 |
| <b>C&amp;S</b> The First Programmer                                   | 658 |

|  |     |
|--|-----|
| <b>14.7 PROBLEM SOLVING: ESTIMATING THE RUNNING TIME OF AN ALGORITHM</b> | 659 |
|--|-----|

|   |     |
|---|-----|
| <b>14.8 SORTING AND SEARCHING IN THE JAVA LIBRARY</b> | 664 |
|---|-----|

|   |     |
|---|-----|
| <b>CE1</b> The compareTo Method Can Return Any Integer, Not Just -1, 0, and 1 | 666 |
| <b>ST4</b> The Comparator Interface   | 666 |
| <b>J81</b> Comparators with Lambda Expressions                                | 667 |
| <b>WE1</b> Enhancing the Insertion Sort Algorithm                             | 668 |





© Volkan Ersoy/iStockphoto.

One of the most common tasks in data processing is sorting. For example, an array of employees often needs to be displayed in alphabetical order or sorted by salary. In this chapter, you will learn several sorting methods as well as techniques for comparing their performance. These techniques are useful not just for sorting algorithms, but also for analyzing other algorithms.

Once an array of elements is sorted, one can rapidly locate individual elements. You will study the *binary search* algorithm that carries out this fast lookup.

## 14.1 Selection Sort

In this section, we show you the first of several sorting algorithms. A *sorting algorithm* rearranges the elements of a collection so that they are stored in sorted order. To keep the examples simple, we will discuss how to sort an array of integers before going on to sorting strings or more complex data. Consider the following array a:

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] |
| 11  | 9   | 17  | 5   | 12  |

The selection sort algorithm sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.

An obvious first step is to find the smallest element. In this case the smallest element is 5, stored in  $a[3]$ . We should move the 5 to the beginning of the array. Of course, there is already an element stored in  $a[0]$ , namely 11. Therefore we cannot simply move  $a[3]$  into  $a[0]$  without moving the 11 somewhere else. We don't yet know where the 11 should end up, but we know for certain that it should not be in  $a[0]$ . We simply get it out of the way by *swapping* it with  $a[3]$ :

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] |
| 5   | 9   | 17  | 11  | 12  |
|     |     |     | ↑   | ↑   |

Now the first element is in the correct place. The darker color in the figure indicates the portion of the array that is already sorted.

*In selection sort, pick the smallest element and swap it with the first one. Pick the smallest element of the remaining ones and swap it with the next one, and so on.*



© Zone Creative/iStockphoto.

Next we take the minimum of the remaining entries  $a[1] \dots a[4]$ . That minimum value, 9, is already in the correct place. We don't need to do anything in this case and can simply extend the sorted area by one to the right:

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] |
| 5   | 9   | 11  | 11  | 12  |

Repeat the process. The minimum value of the unsorted region is 11, which needs to be swapped with the first value of the unsorted region, 17:

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] |
| 5   | 9   | 11  | 17  | 12  |

Now the unsorted region is only two elements long, but we keep to the same successful strategy. The minimum value is 12, and we swap it with the first value, 17:

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] |
| 5   | 9   | 11  | 12  | 17  |

That leaves us with an unprocessed region of length 1, but of course a region of length 1 is always sorted. We are done.

Let's program this algorithm, called **selection sort**. For this program, as well as the other programs in this chapter, we will use a utility method to generate an array with random entries. We place it into a class `ArrayUtil` so that we don't have to repeat the code in every example. To show the array, we call the static `toString` method of the `Arrays` class in the Java library and print the resulting string (see Section 7.3.4). We also add a method for swapping elements to the `ArrayUtil` class. (See Section 7.3.8 for details about swapping array elements.)

This algorithm will sort any array of integers. If speed were not an issue, or if there were no better sorting method available, we could stop the discussion of sorting right here. As the next section shows, however, this algorithm, while entirely correct, shows disappointing performance when run on a large data set.

Special Topic 14.2 discusses insertion sort, another simple sorting algorithm.

### section\_1/SelectionSorter.java

```

1  /**
2   * The sort method of this class sorts an array, using the selection
3   * sort algorithm.
4  */
5  public class SelectionSorter
6  {
7      /**
8       * Sorts an array, using selection sort.
9       * @param a the array to sort
10    */
11   public static void sort(int[] a)
12   {
13       for (int i = 0; i < a.length - 1; i++)
14       {
15           int minPos = minimumPosition(a, i);
16           ArrayUtil.swap(a, minPos, i);
17       }
18   }
}

```

```

19
20  /**
21   * Finds the smallest element in a tail range of the array.
22   * @param a the array to sort
23   * @param from the first position in a to compare
24   * @return the position of the smallest element in the
25   * range a[from] . . . a[a.length - 1]
26  */
27  private static int minimumPosition(int[] a, int from)
28  {
29      int minPos = from;
30      for (int i = from + 1; i < a.length; i++)
31      {
32          if (a[i] < a[minPos]) { minPos = i; }
33      }
34      return minPos;
35  }
36 }

```

**section\_1/SelectionSortDemo.java**

```

1 import java.util.Arrays;
2
3 /**
4  * This program demonstrates the selection sort algorithm by
5  * sorting an array that is filled with random numbers.
6  */
7 public class SelectionSortDemo
8 {
9     public static void main(String[] args)
10    {
11        int[] a = ArrayUtil.randomIntArray(20, 100);
12        System.out.println(Arrays.toString(a));
13
14        SelectionSorter.sort(a);
15
16        System.out.println(Arrays.toString(a));
17    }
18 }

```

**section\_1/ArrayUtil.java**

```

1 import java.util.Random;
2
3 /**
4  * This class contains utility methods for array manipulation.
5  */
6 public class ArrayUtil
7 {
8     private static Random generator = new Random();
9
10    /**
11     * Creates an array filled with random values.
12     * @param length the length of the array
13     * @param n the number of possible random values
14     * @return an array filled with length numbers between
15     * 0 and n - 1
16    */
17    public static int[] randomIntArray(int length, int n)
18    {

```

```

19     int[] a = new int[length];
20     for (int i = 0; i < a.length; i++)
21     {
22         a[i] = generator.nextInt(n);
23     }
24
25     return a;
26 }
27
28 /**
29  * Swaps two entries of an array.
30  * @param a the array
31  * @param i the first position to swap
32  * @param j the second position to swap
33 */
34 public static void swap(int[] a, int i, int j)
35 {
36     int temp = a[i];
37     a[i] = a[j];
38     a[j] = temp;
39 }
40

```

**Program Run**

```
[65, 46, 14, 52, 38, 2, 96, 39, 14, 33, 13, 4, 24, 99, 89, 77, 73, 87, 36, 81]
[2, 4, 13, 14, 14, 24, 33, 36, 38, 39, 46, 52, 65, 73, 77, 81, 87, 89, 96, 99]
```

**SELF CHECK**

1. Why do we need the `temp` variable in the `swap` method? What would happen if you simply assigned `a[i]` to `a[j]` and `a[j]` to `a[i]`?
2. What steps does the selection sort algorithm go through to sort the sequence `6 5 4 3 2 1`?
3. How can you change the selection sort algorithm so that it sorts the elements in descending order (that is, with the largest element at the beginning of the array)?
4. Suppose we modified the selection sort algorithm to start at the end of the array, working toward the beginning. In each step, the current position is swapped with the minimum. What is the result of this modification?

**Practice It**

Now you can try these exercises at the end of the chapter: R14.2, R14.12, E14.1, E14.2.

## 14.2 Profiling the Selection Sort Algorithm

To measure the performance of a program, you could simply run it and use a stopwatch to measure how long it takes. However, most of our programs run very quickly, and it is not easy to time them accurately in this way. Furthermore, when a program takes a noticeable time to run, a certain amount of that time may simply be used for loading the program from disk into memory and displaying the result (for which we should not penalize it).

In order to measure the running time of an algorithm more accurately, we will create a `Stopwatch` class. This class works like a real stopwatch. You can start it, stop

it, and read out the elapsed time. The class uses the `System.currentTimeMillis` method, which returns the milliseconds that have elapsed since midnight at the start of January 1, 1970. Of course, you don't care about the absolute number of seconds since this historical moment, but the *difference* of two such counts gives us the number of milliseconds in a given time interval.

Here is the code for the `StopWatch` class:

### section\_2/StopWatch.java

```

1  /**
2   * A stopwatch accumulates time when it is running. You can
3   * repeatedly start and stop the stopwatch. You can use a
4   * stopwatch to measure the running time of a program.
5  */
6  public class StopWatch
7  {
8      private long elapsedTime;
9      private long startTime;
10     private boolean isRunning;
11
12     /**
13      Constructs a stopwatch that is in the stopped state
14      and has no time accumulated.
15     */
16     public StopWatch()
17     {
18         reset();
19     }
20
21     /**
22      Starts the stopwatch. Time starts accumulating now.
23     */
24     public void start()
25     {
26         if (isRunning) { return; }
27         isRunning = true;
28         startTime = System.currentTimeMillis();
29     }
30
31     /**
32      Stops the stopwatch. Time stops accumulating and is
33      is added to the elapsed time.
34     */
35     public void stop()
36     {
37         if (!isRunning) { return; }
38         isRunning = false;
39         long endTime = System.currentTimeMillis();
40         elapsedTime = elapsedTime + endTime - startTime;
41     }
42
43     /**
44      Returns the total elapsed time.
45      @return the total elapsed time
46     */
47     public long getElapsedTime()
48     {
49         if (isRunning)
50         {

```

```

51     long endTime = System.currentTimeMillis();
52     return elapsedTime + endTime - startTime;
53   }
54   else
55   {
56     return elapsedTime;
57   }
58 }
59
60 /**
61  * Stops the watch and resets the elapsed time to 0.
62 */
63 public void reset()
64 {
65   elapsedTime = 0;
66   isRunning = false;
67 }
68 }
```

Here is how to use the stopwatch to measure the sorting algorithm's performance:

### **section\_2/SelectionSortTimer.java**

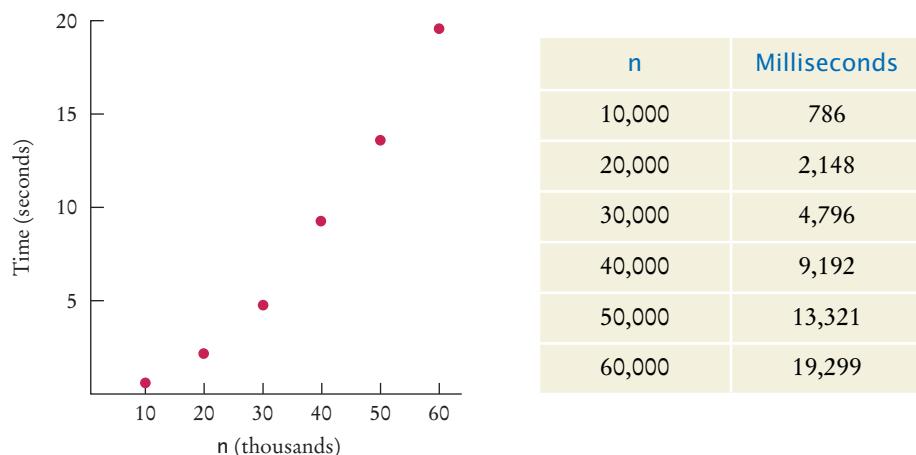
```

1 import java.util.Scanner;
2
3 /**
4  * This program measures how long it takes to sort an
5  * array of a user-specified size with the selection
6  * sort algorithm.
7 */
8 public class SelectionSortTimer
9 {
10   public static void main(String[] args)
11   {
12     Scanner in = new Scanner(System.in);
13     System.out.print("Enter array size: ");
14     int n = in.nextInt();
15
16     // Construct random array
17
18     int[] a = ArrayUtil.randomIntArray(n, 100);
19
20     // Use stopwatch to time selection sort
21
22     StopWatch timer = new StopWatch();
23
24     timer.start();
25     SelectionSorter.sort(a);
26     timer.stop();
27
28     System.out.println("Elapsed time: "
29                     + timer.getElapsedTime() + " milliseconds");
30   }
31 }
```

### **Program Run**

```

Enter array size: 50000
Elapsed time: 13321 milliseconds
```

**Figure 1** Time Taken by Selection Sort

To measure the running time of a method, get the current time immediately before and after the method call.

By starting to measure the time just before sorting, and stopping the stopwatch just after, you get the time required for the sorting process, without counting the time for input and output.

The table in Figure 1 shows the results of some sample runs. These measurements were obtained with an Intel processor with a clock speed of 2 GHz, running Java 6 on the Linux operating system. On another computer the actual numbers will look different, but the relationship between the numbers will be the same.

The graph in Figure 1 shows a plot of the measurements. As you can see, when you double the size of the data set, it takes about four times as long to sort it.



5. Approximately how many seconds would it take to sort a data set of 80,000 values?
6. Look at the graph in Figure 1. What mathematical shape does it resemble?

**Practice It** Now you can try these exercises at the end of the chapter: E14.3, E14.9.

## 14.3 Analyzing the Performance of the Selection Sort Algorithm

Let us count the number of operations that the program must carry out to sort an array with the selection sort algorithm. We don't actually know how many machine operations are generated for each Java instruction, or which of those instructions are more time-consuming than others, but we can make a simplification. We will simply count how often an array element is *visited*. Each visit requires about the same amount of work by other operations, such as incrementing subscripts and comparing values.

Let  $n$  be the size of the array. First, we must find the smallest of  $n$  numbers. To achieve that, we must visit  $n$  array elements. Then we swap the elements, which takes

two visits. (You may argue that there is a certain probability that we don't need to swap the values. That is true, and one can refine the computation to reflect that observation. As we will soon see, doing so would not affect the overall conclusion.) In the next step, we need to visit only  $n - 1$  elements to find the minimum. In the following step,  $n - 2$  elements are visited to find the minimum. The last step visits two elements to find the minimum. Each step requires two visits to swap the elements. Therefore, the total number of visits is

$$\begin{aligned} n + 2 + (n - 1) + 2 + \cdots + 2 + 2 &= (n + (n - 1) + \cdots + 2) + (n - 1) \cdot 2 \\ &= (2 + \cdots + (n - 1) + n) + (n - 1) \cdot 2 \\ &= \frac{n(n + 1)}{2} - 1 + (n - 1) \cdot 2 \end{aligned}$$

because

$$1 + 2 + \cdots + (n - 1) + n = \frac{n(n + 1)}{2}$$

After multiplying out and collecting terms of  $n$ , we find that the number of visits is

$$\frac{1}{2}n^2 + \frac{5}{2}n - 3$$

We obtain a quadratic equation in  $n$ . That explains why the graph of Figure 1 looks approximately like a parabola.

Now simplify the analysis further. When you plug in a large value for  $n$  (for example, 1,000 or 2,000), then  $\frac{1}{2}n^2$  is 500,000 or 2,000,000. The lower term,  $\frac{5}{2}n - 3$ , doesn't contribute much at all; it is only 2,497 or 4,997, a drop in the bucket compared to the hundreds of thousands or even millions of comparisons specified by the  $\frac{1}{2}n^2$  term. We will just ignore these lower-level terms. Next, we will ignore the constant factor  $\frac{1}{2}$ . We are not interested in the actual count of visits for a single  $n$ . We want to compare the ratios of counts for different values of  $n$ . For example, we can say that sorting an array of 2,000 numbers requires four times as many visits as sorting an array of 1,000 numbers:

$$\frac{\left(\frac{1}{2} \cdot 2000^2\right)}{\left(\frac{1}{2} \cdot 1000^2\right)} = 4$$

The factor  $\frac{1}{2}$  cancels out in comparisons of this kind. We will simply say, "The number of visits is of order  $n^2$ ." That way, we can easily see that the number of comparisons increases fourfold when the size of the array doubles:  $(2n)^2 = 4n^2$ .

To indicate that the number of visits is of order  $n^2$ , computer scientists often use **big-Oh notation**: The number of visits is  $O(n^2)$ . This is a convenient shorthand. (See Special Topic 14.1 for a formal definition.)

To turn a polynomial expression such as

$$\frac{1}{2}n^2 + \frac{5}{2}n - 3$$

into big-Oh notation, simply locate the fastest-growing term,  $n^2$ , and ignore its constant coefficient, no matter how large or small it may be.

We observed before that the actual number of machine operations, and the actual amount of time that the computer spends on them, is approximately proportional to the number of element visits. Maybe there are about 10 machine operations

Computer scientists use the big-Oh notation to describe the growth rate of a function.

Selection sort is an  $O(n^2)$  algorithm. Doubling the data set means a fourfold increase in processing time.

(increments, comparisons, memory loads, and stores) for every element visit. The number of machine operations is then approximately  $10 \times \frac{1}{2}n^2$ . As before, we aren't interested in the coefficient, so we can say that the number of machine operations, and hence the time spent on the sorting, is of the order  $n^2$  or  $O(n^2)$ .

The sad fact remains that doubling the size of the array causes a fourfold increase in the time required for sorting it with selection sort. When the size of the array increases by a factor of 100, the sorting time increases by a factor of 10,000. To sort an array of a million entries (for example, to create a telephone directory), takes 10,000 times as long as sorting 10,000 entries. If 10,000 entries can be sorted in about 3/4 of a second (as in our example), then sorting one million entries requires well over two hours. We will see in the next section how one can dramatically improve the performance of the sorting process by choosing a more sophisticated algorithm.

### SELF CHECK



7. If you increase the size of a data set tenfold, how much longer does it take to sort it with the selection sort algorithm?
8. How large does  $n$  need to be so that  $\frac{1}{2}n^2$  is bigger than  $\frac{5}{2}n - 3$ ?
9. Section 7.3.6 has two algorithms for removing an element from an array of length  $n$ . How many array visits does each algorithm require on average?
10. Describe the number of array visits in Self Check 9 using the big-Oh notation.
11. What is the big-Oh running time of checking whether an array is already sorted?
12. Consider this algorithm for sorting an array. Set  $k$  to the length of the array. Find the maximum of the first  $k$  elements. Remove it, using the second algorithm of Section 7.3.6. Decrement  $k$  and place the removed element into the  $k$ th position. Stop if  $k$  is 1. What is the algorithm's running time in big-Oh notation?

**Practice It** Now you can try these exercises at the end of the chapter: R14.4, R14.6, R14.8.

### Special Topic 14.1



### Oh, Omega, and Theta

We have used the big-Oh notation somewhat casually in this chapter to describe the growth behavior of a function. Here is the formal definition of the big-Oh notation: Suppose we have a function  $T(n)$ . Usually, it represents the processing time of an algorithm for a given input of size  $n$ . But it could be any function. Also, suppose that we have another function  $f(n)$ . It is usually chosen to be a simple function, such as  $f(n) = n^k$  or  $f(n) = \log(n)$ , but it too can be any function. We write

$$T(n) = O(f(n))$$

if  $T(n)$  grows at a rate that is bounded by  $f(n)$ . More formally, we require that for all  $n$  larger than some threshold, the ratio  $T(n)/f(n) \leq C$  for some constant value  $C$ .

If  $T(n)$  is a polynomial of degree  $k$  in  $n$ , then one can show that  $T(n) = O(n^k)$ . Later in this chapter, we will encounter functions that are  $O(\log(n))$  or  $O(n \log(n))$ . Some algorithms take much more time. For example, one way of sorting a sequence is to compute all of its permutations, until you find one that is in increasing order. Such an algorithm takes  $O(n!)$  time, which is very bad indeed.

Table 1 shows common big-Oh expressions, sorted by increasing growth.

Strictly speaking,  $T(n) = O(f(n))$  means that  $T$  grows no faster than  $f$ . But it is permissible for  $T$  to grow much more slowly. Thus, it is technically correct to state that  $T(n) = n^2 + 5n - 3$  is  $O(n^3)$  or even  $O(n^{10})$ .

**Table 1** Common Big-Oh Growth Rates

| Big-Oh Expression | Name        |
|-------------------|-------------|
| $O(1)$            | Constant    |
| $O(\log(n))$      | Logarithmic |
| $O(n)$            | Linear      |
| $O(n \log(n))$    | Log-linear  |
| $O(n^2)$          | Quadratic   |
| $O(n^3)$          | Cubic       |
| $O(2^n)$          | Exponential |
| $O(n!)$           | Factorial   |

Computer scientists have invented additional notation to describe the growth behavior of functions more accurately. The expression

$$T(n) = \Omega(f(n))$$

means that  $T$  grows at least as fast as  $f$ , or, formally, that for all  $n$  larger than some threshold, the ratio  $T(n)/f(n) \geq C$  for some constant value  $C$ . (The  $\Omega$  symbol is the capital Greek letter omega.) For example,  $T(n) = n^2 + 5n - 3$  is  $\Omega(n^2)$  or even  $\Omega(n)$ .

The expression

$$T(n) = \Theta(f(n))$$

means that  $T$  and  $f$  grow at the same rate—that is, both  $T(n) = O(f(n))$  and  $T(n) = \Omega(f(n))$  hold. (The  $\Theta$  symbol is the capital Greek letter theta.)

The  $\Theta$  notation gives the most precise description of growth behavior. For example,  $T(n) = n^2 + 5n - 3$  is  $\Theta(n^2)$  but not  $\Theta(n)$  or  $\Theta(n^3)$ .

The notations are very important for the precise analysis of algorithms. However, in casual conversation it is common to stick with big-Oh, while still giving an estimate as good as one can make.

### Special Topic 14.2



### Insertion Sort

Insertion sort is another simple sorting algorithm. In this algorithm, we assume that the initial sequence

$a[0] \ a[1] \ \dots \ a[k]$

of an array is already sorted. (When the algorithm starts, we set  $k$  to 0.) We enlarge the initial sequence by inserting the next array element,  $a[k + 1]$ , at the proper location. When we reach the end of the array, the sorting process is complete.

For example, suppose we start with the array

11 9 16 5 7

Of course, the initial sequence of length 1 is already sorted. We now add  $a[1]$ , which has the value 9. The element needs to be inserted before the element 11. The result is

9 11 16 5 7

Next, we add  $a[2]$ , which has the value 16. This element does not have to be moved.

|   |    |    |   |   |
|---|----|----|---|---|
| 9 | 11 | 16 | 5 | 7 |
|---|----|----|---|---|

We repeat the process, inserting  $a[3]$  or 5 at the very beginning of the initial sequence.

|   |   |    |    |   |
|---|---|----|----|---|
| 5 | 9 | 11 | 16 | 7 |
|---|---|----|----|---|

Finally,  $a[4]$  or 7 is inserted in its correct position, and the sorting is completed.

The following class implements the insertion sort algorithm:

```
public class InsertionSorter
{
    /**
     * Sorts an array, using insertion sort.
     * @param a the array to sort
     */
    public static void sort(int[] a)
    {
        for (int i = 1; i < a.length; i++)
        {
            int next = a[i];
            // Move all larger elements up
            int j = i;
            while (j > 0 && a[j - 1] > next)
            {
                a[j] = a[j - 1];
                j--;
            }
            // Insert the element
            a[j] = next;
        }
    }
}
```

How efficient is this algorithm? Let  $n$  denote the size of the array. We carry out  $n - 1$  iterations. In the  $k$ th iteration, we have a sequence of  $k$  elements that is already sorted, and we need to insert a new element into the sequence. For each insertion, we need to visit the elements of the initial sequence until we have found the location in which the new element can be inserted. Then we need to move up the remaining elements of the sequence. Thus,  $k + 1$  array elements are visited. Therefore, the total number of visits is

$$2 + 3 + \dots + n = \frac{n(n+1)}{2} - 1$$

We conclude that insertion sort is an  $O(n^2)$  algorithm, on the same order of efficiency as selection sort.

Insertion sort has a desirable property: Its performance is  $O(n)$  if the array is already sorted—see Exercise R14.19. This is a useful property in practical applications, in which data sets are often partially sorted.

Insertion sort is an  $O(n^2)$  algorithm.

*Insertion sort is the method that many people use to sort playing cards. Pick up one card at a time and insert it so that the cards stay sorted.*



© Kirby Hamilton/iStockphoto.

**FULL CODE EXAMPLE**  
Go to [wiley.com/go/bje06code](http://wiley.com/go/bje06code) to download a program that illustrates sorting with insertion sort.

## 14.4 Merge Sort

In this section, you will learn about the **merge sort** algorithm, a much more efficient algorithm than selection sort. The basic idea behind merge sort is very simple.

Suppose we have an array of 10 integers. Let us engage in a bit of wishful thinking and hope that the first half of the array is already perfectly sorted, and the second half is too, like this:

|   |   |    |    |    |   |   |    |    |    |
|---|---|----|----|----|---|---|----|----|----|
| 5 | 9 | 10 | 12 | 17 | 1 | 8 | 11 | 20 | 32 |
|---|---|----|----|----|---|---|----|----|----|

Now it is simple to *merge* the two sorted arrays into one sorted array, by taking a new element from either the first or the second subarray, and choosing the smaller of the elements each time:

|   |   |    |    |    |   |   |    |    |    |
|---|---|----|----|----|---|---|----|----|----|
| 5 | 9 | 10 | 12 | 17 | 1 | 8 | 11 | 20 | 32 |
| 5 | 9 | 10 | 12 | 17 | 1 | 8 | 11 | 20 | 32 |
| 5 | 9 | 10 | 12 | 17 | 1 | 8 | 11 | 20 | 32 |
| 5 | 9 | 10 | 12 | 17 | 1 | 8 | 11 | 20 | 32 |
| 5 | 9 | 10 | 12 | 17 | 1 | 8 | 11 | 20 | 32 |
| 5 | 9 | 10 | 12 | 17 | 1 | 8 | 11 | 20 | 32 |
| 5 | 9 | 10 | 12 | 17 | 1 | 8 | 11 | 20 | 32 |
| 5 | 9 | 10 | 12 | 17 | 1 | 8 | 11 | 20 | 32 |
| 5 | 9 | 10 | 12 | 17 | 1 | 8 | 11 | 20 | 32 |
| 5 | 9 | 10 | 12 | 17 | 1 | 8 | 11 | 20 | 32 |

|   |   |   |   |    |    |    |    |    |    |
|---|---|---|---|----|----|----|----|----|----|
| 1 |   |   |   |    |    |    |    |    |    |
| 1 | 5 |   |   |    |    |    |    |    |    |
| 1 | 5 | 8 |   |    |    |    |    |    |    |
| 1 | 5 | 8 | 9 |    |    |    |    |    |    |
| 1 | 5 | 8 | 9 | 10 |    |    |    |    |    |
| 1 | 5 | 8 | 9 | 10 | 11 |    |    |    |    |
| 1 | 5 | 8 | 9 | 10 | 11 | 12 |    |    |    |
| 1 | 5 | 8 | 9 | 10 | 11 | 12 | 17 |    |    |
| 1 | 5 | 8 | 9 | 10 | 11 | 12 | 17 | 20 |    |
| 1 | 5 | 8 | 9 | 10 | 11 | 12 | 17 | 20 | 32 |

The merge sort algorithm sorts an array by cutting the array in half, recursively sorting each half, and then merging the sorted halves.

In fact, you may have performed this merging before if you and a friend had to sort a pile of papers. You and the friend split the pile in half, each of you sorted your half, and then you merged the results together.

That is all well and good, but it doesn't seem to solve the problem for the computer. It still must sort the first and second halves of the array, because it can't very well ask a few buddies to pitch in. As it turns out, though, if the computer keeps dividing the array into smaller and smaller subarrays, sorting each half and merging them back together, it carries out dramatically fewer steps than the selection sort requires.

Let's write a `MergeSorter` class that implements this idea. When the `MergeSorter` sorts an array, it makes two arrays, each half the size of the original, and sorts them recursively. Then it merges the two sorted arrays together:

```
public static void sort(int[] a)
{
    if (a.length <= 1) { return; }
    int[] first = new int[a.length / 2];
    int[] second = new int[a.length - first.length];
    // Copy the first half of a into first, the second half into second
    .
    .
    sort(first);
    sort(second);
    merge(first, second, a);
}
```



© Rich Legg/Stockphoto.

*In merge sort, one sorts each half, then merges the sorted halves.*

The `merge` method is tedious but quite straightforward. You will find it in the code that follows.

### section\_4/MergeSorter.java

```

1  /**
2   * The sort method of this class sorts an array, using the merge
3   * sort algorithm.
4  */
5  public class MergeSorter
6  {
7      /**
8       * Sorts an array, using merge sort.
9       * @param a the array to sort
10    */
11   public static void sort(int[] a)
12   {
13       if (a.length <= 1) { return; }
14       int[] first = new int[a.length / 2];
15       int[] second = new int[a.length - first.length];
16       // Copy the first half of a into first, the second half into second
17       for (int i = 0; i < first.length; i++)
18       {
19           first[i] = a[i];
20       }
21       for (int i = 0; i < second.length; i++)
22       {
23           second[i] = a[first.length + i];
24       }
25       sort(first);
26       sort(second);
27       merge(first, second, a);
28   }
29
30 /**
31  * Merges two sorted arrays into an array.
32  * @param first the first sorted array
33  * @param second the second sorted array
34  * @param a the array into which to merge first and second
35  */
36 private static void merge(int[] first, int[] second, int[] a)
37 {
38     int iFirst = 0; // Next element to consider in the first array
39     int iSecond = 0; // Next element to consider in the second array
40     int j = 0; // Next open position in a
41
42     // As long as neither iFirst nor iSecond past the end, move
43     // the smaller element into a
44     while (iFirst < first.length && iSecond < second.length)
45     {
46         if (first[iFirst] < second[iSecond])
47         {
48             a[j] = first[iFirst];
49             iFirst++;
50         }
51         else
52         {
53             a[j] = second[iSecond];
54             iSecond++;
55         }
56     }
57 }
```

```

55     }
56     j++;
57 }
58
59 // Note that only one of the two loops below copies entries
60 // Copy any remaining entries of the first array
61 while (iFirst < first.length)
62 {
63     a[j] = first[iFirst];
64     iFirst++; j++;
65 }
66 // Copy any remaining entries of the second half
67 while (iSecond < second.length)
68 {
69     a[j] = second[iSecond];
70     iSecond++; j++;
71 }
72 }
73 }

```

### section\_4/MergeSortDemo.java

```

1 import java.util.Arrays;
2
3 /**
4  * This program demonstrates the merge sort algorithm by
5  * sorting an array that is filled with random numbers.
6  */
7 public class MergeSortDemo
8 {
9     public static void main(String[] args)
10    {
11        int[] a = ArrayUtil.randomIntArray(20, 100);
12        System.out.println(Arrays.toString(a));
13
14        MergeSorter.sort(a);
15
16        System.out.println(Arrays.toString(a));
17    }
18 }

```

### Program Run

```
[8, 81, 48, 53, 46, 70, 98, 42, 27, 76, 33, 24, 2, 76, 62, 89, 90, 5, 13, 21]
[2, 5, 8, 13, 21, 24, 27, 33, 42, 46, 48, 53, 62, 70, 76, 81, 89, 90, 98]
```



13. Why does only one of the two `while` loops at the end of the `merge` method do any work?
14. Manually run the merge sort algorithm on the array 8 7 6 5 4 3 2 1.
15. The merge sort algorithm processes an array by recursively processing two halves. Describe a similar recursive algorithm for computing the sum of all elements in an array.

**Practice It** Now you can try these exercises at the end of the chapter: R14.13, E14.4, E14.14.

## 14.5 Analyzing the Merge Sort Algorithm

The merge sort algorithm looks a lot more complicated than the selection sort algorithm, and it appears that it may well take much longer to carry out these repeated subdivisions. However, the timing results for merge sort look much better than those for selection sort.

Figure 2 shows a table and a graph comparing both sets of performance data. As you can see, merge sort is a tremendous improvement. To understand why, let us estimate the number of array element visits that are required to sort an array with the merge sort algorithm. First, let us tackle the merge process that happens after the first and second halves have been sorted.

Each step in the merge process adds one more element to  $a$ . That element may come from first or second, and in most cases the elements from the two halves must be compared to see which one to take. We'll count that as 3 visits (one for  $a$  and one each for first and second) per element, or  $3n$  visits total, where  $n$  denotes the length of  $a$ . Moreover, at the beginning, we had to copy from  $a$  to first and second, yielding another  $2n$  visits, for a total of  $5n$ .

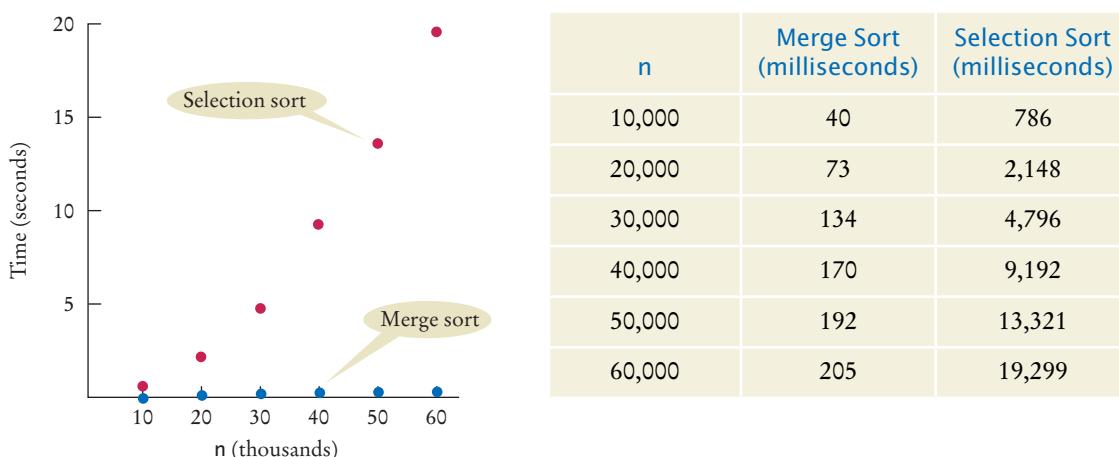
If we let  $T(n)$  denote the number of visits required to sort a range of  $n$  elements through the merge sort process, then we obtain

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 5n$$

because sorting each half takes  $T(n/2)$  visits. Actually, if  $n$  is not even, then we have one subarray of size  $(n - 1)/2$  and one of size  $(n + 1)/2$ . Although it turns out that this detail does not affect the outcome of the computation, we will nevertheless assume for now that  $n$  is a power of 2, say  $n = 2^m$ . That way, all subarrays can be evenly divided into two parts.

Unfortunately, the formula

$$T(n) = 2T\left(\frac{n}{2}\right) + 5n$$



**Figure 2** Time Taken by Selection Sort

does not clearly tell us the relationship between  $n$  and  $T(n)$ . To understand the relationship, let us evaluate  $T(n/2)$ , using the same formula:

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + 5\frac{n}{2}$$

Therefore

$$T(n) = 2 \times 2T\left(\frac{n}{4}\right) + 5n + 5n$$

Let us do that again:

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + 5\frac{n}{4}$$

hence

$$T(n) = 2 \times 2 \times 2T\left(\frac{n}{8}\right) + 5n + 5n + 5n$$

This generalizes from 2, 4, 8, to arbitrary powers of 2:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + 5nk$$

Recall that we assume that  $n = 2^m$ ; hence, for  $k = m$ ,

$$\begin{aligned} T(n) &= 2^m T\left(\frac{n}{2^m}\right) + 5nm \\ &= nT(1) + 5nm \\ &= n + 5n \log_2(n) \end{aligned}$$

Because  $n = 2^m$ , we have  $m = \log_2(n)$ .

To establish the growth order, we drop the lower-order term  $n$  and are left with  $5n \log_2(n)$ . We drop the constant factor 5. It is also customary to drop the base of the logarithm, because all logarithms are related by a constant factor. For example,

$$\log_2(x) = \log_{10}(x)/\log_{10}(2) \approx \log_{10}(x) \times 3.32193$$

Hence we say that merge sort is an  $O(n \log(n))$  algorithm.

Is the  $O(n \log(n))$  merge sort algorithm better than the  $O(n^2)$  selection sort algorithm? You bet it is. Recall that it took  $100^2 = 10,000$  times as long to sort a million records as it took to sort 10,000 records with the  $O(n^2)$  algorithm. With the  $O(n \log(n))$  algorithm, the ratio is

$$\frac{1,000,000 \log(1,000,000)}{10,000 \log(10,000)} = 100 \left( \frac{6}{4} \right) = 150$$

Suppose for the moment that merge sort takes the same time as selection sort to sort an array of 10,000 integers, that is,  $3/4$  of a second on the test machine. (Actually, it is much faster than that.) Then it would take about  $0.75 \times 150$  seconds, or under two minutes, to sort a million integers. Contrast that with selection sort, which would take over two hours for the same task. As you can see, even if it takes you several hours to learn about a better algorithm, that can be time well spent.

Merge sort is an  $O(n \log(n))$  algorithm. The  $n \log(n)$  function grows much more slowly than  $n^2$ .

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download a program for timing the merge sort algorithm.

**SELF CHECK**

16. Given the timing data for the merge sort algorithm in the table at the beginning of this section, how long would it take to sort an array of 100,000 values?
17. If you double the size of an array, how much longer will the merge sort algorithm take to sort the new array?

**Practice It** Now you can try these exercises at the end of the chapter: R14.7, R14.16, R14.18.

**Special Topic 14.3****The Quicksort Algorithm**

**Quicksort** is a commonly used algorithm that has the advantage over merge sort that no temporary arrays are required to sort and merge the partial results.

The quicksort algorithm, like merge sort, is based on the strategy of divide and conquer. To sort a range  $a[from] \dots a[to]$  of the array  $a$ , first rearrange the elements in the range so that no element in the range  $a[from] \dots a[p]$  is larger than any element in the range  $a[p + 1] \dots a[to]$ . This step is called *partitioning* the range.

For example, suppose we start with a range

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 5 | 3 | 2 | 6 | 4 | 1 | 3 | 7 |
|---|---|---|---|---|---|---|---|

Here is a partitioning of the range. Note that the partitions aren't yet sorted.

|   |   |   |   |   |  |  |   |   |   |
|---|---|---|---|---|--|--|---|---|---|
| 3 | 3 | 2 | 1 | 4 |  |  | 6 | 5 | 7 |
|---|---|---|---|---|--|--|---|---|---|

You'll see later how to obtain such a partition. In the next step, sort each partition, by recursively applying the same algorithm to the two partitions. That sorts the entire range, because the largest element in the first partition is at most as large as the smallest element in the second partition.

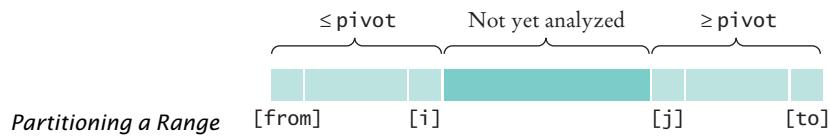
|   |   |   |   |   |  |  |   |   |   |
|---|---|---|---|---|--|--|---|---|---|
| 1 | 2 | 3 | 3 | 4 |  |  | 5 | 6 | 7 |
|---|---|---|---|---|--|--|---|---|---|

Quicksort is implemented recursively as follows:

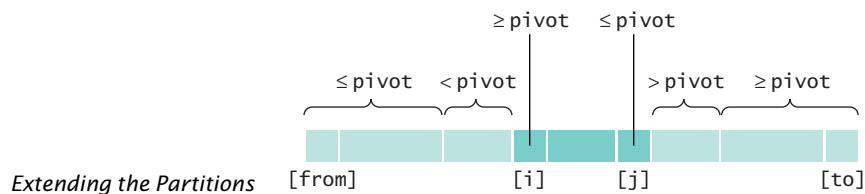
```
public static void sort(int[] a, int from, int to)
{
    if (from >= to) { return; }
    int p = partition(a, from, to);
    sort(a, from, p);
    sort(a, p + 1, to);
}
```

Let us return to the problem of partitioning a range. Pick an element from the range and call it the *pivot*. There are several variations of the quicksort algorithm. In the simplest one, we'll pick the first element of the range,  $a[from]$ , as the pivot.

Now form two regions  $a[from] \dots a[i]$ , consisting of values at most as large as the pivot and  $a[j] \dots a[to]$ , consisting of values at least as large as the pivot. The region  $a[i + 1] \dots a[j - 1]$  consists of values that haven't been analyzed yet. (See the figure below.) At the beginning, both the left and right areas are empty; that is,  $i = from - 1$  and  $j = to + 1$ .



Then keep incrementing  $i$  while  $a[i] < \text{pivot}$  and keep decrementing  $j$  while  $a[j] > \text{pivot}$ . The figure below shows  $i$  and  $j$  when that process stops.



Now swap the values in positions  $i$  and  $j$ , increasing both areas once more. Keep going while  $i < j$ . Here is the code for the partition method:

```
private static int partition(int[] a, int from, int to)
{
    int pivot = a[from];
    int i = from - 1;
    int j = to + 1;
    while (i < j)
    {
        i++; while (a[i] < pivot) { i++; }
        j--; while (a[j] > pivot) { j--; }
        if (i < j) { ArrayUtil.swap(a, i, j); }
    }
    return j;
}
```

On average, the quicksort algorithm is an  $O(n \log(n))$  algorithm. There is just one unfortunate aspect to the quicksort algorithm. Its *worst-case* run-time behavior is  $O(n^2)$ . Moreover, if the pivot element is chosen as the first element of the region, that worst-case behavior occurs when the input set is already sorted—a common situation in practice. By selecting the pivot element more cleverly, we can make it extremely unlikely for the worst-case behavior to occur. Such “tuned” quicksort algorithms are commonly used because their performance is generally excellent. For example, the sort method in the Arrays class uses a quicksort algorithm.

Another improvement that is commonly made in practice is to switch to insertion sort when the array is short, because the total number of operations using insertion sort is lower for short arrays. The Java library makes that switch if the array length is less than seven.



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download a program that demonstrates the quicksort algorithm.

*In quicksort, one partitions the elements into two groups, holding the smaller and larger elements. Then one sorts each group.*



© Christopher Furler/iStockphoto

# 14.6 Searching

Searching for an element in an array is an extremely common task. As with sorting, the right choice of algorithms can make a big difference.

## 14.6.1 Linear Search

Suppose you need to find your friend's telephone number. You look up the friend's name in the telephone book, and naturally you can find it quickly, because the telephone book is sorted alphabetically. Now suppose you have a telephone number and you must know to what party it belongs. You could of course call that number, but suppose nobody picks up on the other end. You could look through the telephone book, a number at a time, until you find the number. That would obviously be a tremendous amount of work, and you would have to be desperate to attempt it.

This thought experiment shows the difference between a search through an unsorted data set and a search through a sorted data set. The following two sections will analyze the difference formally.

If you want to find a number in a sequence of values that occur in arbitrary order, there is nothing you can do to speed up the search. You must simply look through all elements until you have found a match or until you reach the end. This is called a **linear** or **sequential** search.

How long does a linear search take? If we assume that the element  $v$  is present in the array  $a$ , then the average search visits  $n/2$  elements, where  $n$  is the length of the array. If it is not present, then all  $n$  elements must be inspected to verify the absence. Either way, a linear search is an  $O(n)$  algorithm.

Here is a class that performs linear searches through an array  $a$  of integers. When searching for a value, the search method returns the first index of the match, or  $-1$  if the value does not occur in  $a$ .

A linear search examines all values in an array until it finds a match or reaches the end.

A linear search locates a value in an array in  $O(n)$  steps.

### [section\\_6\\_1/LinearSearcher.java](#)

```

1  /**
2   * A class for executing linear searches in an array.
3  */
4  public class LinearSearcher
5  {
6      /**
7       * Finds a value in an array, using the linear search
8       * algorithm.
9       * @param a the array to search
10      * @param value the value to find
11      * @return the index at which the value occurs, or -1
12      * if it does not occur in the array
13     */
14    public static int search(int[] a, int value)
15    {
16        for (int i = 0; i < a.length; i++)
17        {
18            if (a[i] == value) { return i; }
19        }
20        return -1;
}

```

```
21     }
22 }
```

### section\_6\_1/LinearSearchDemo.java

```
1 import java.util.Arrays;
2 import java.util.Scanner;
3
4 /**
5  * This program demonstrates the linear search algorithm.
6 */
7 public class LinearSearchDemo
8 {
9     public static void main(String[] args)
10    {
11        int[] a = ArrayUtil.randomIntArray(20, 100);
12        System.out.println(Arrays.toString(a));
13        Scanner in = new Scanner(System.in);
14
15        boolean done = false;
16        while (!done)
17        {
18            System.out.print("Enter number to search for, -1 to quit: ");
19            int n = in.nextInt();
20            if (n == -1)
21            {
22                done = true;
23            }
24            else
25            {
26                int pos = LinearSearcher.search(a, n);
27                System.out.println("Found in position " + pos);
28            }
29        }
30    }
31 }
```

#### Program Run

```
[46, 99, 45, 57, 64, 95, 81, 69, 11, 97, 6, 85, 61, 88, 29, 65, 83, 88, 45, 88]
Enter number to search for, -1 to quit: 12
Found in position -1
Enter number to search for, -1 to quit: -1
```

## 14.6.2 Binary Search

Now let us search for an item in a data sequence that has been previously sorted. Of course, we could still do a linear search, but it turns out we can do much better than that.

Consider the following sorted array a. The data set is:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| 1   | 4   | 5   | 8   | 9   | 12  | 17  | 20  | 24  | 32  |

We would like to see whether the value 15 is in the data set. Let's narrow our search by finding whether the value is in the first or second half of the array. The last value

in the first half of the data set,  $a[4]$ , is 9, which is smaller than the value we are looking for. Hence, we should look in the second half of the array for a match, that is, in the sequence:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| 1   | 4   | 5   | 8   | 9   | 12  | 17  | 20  | 24  | 32  |

The middle element of this sequence is 20; hence, the value must be located in the sequence:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| 1   | 4   | 5   | 8   | 9   | 12  | 17  | 20  | 24  | 32  |

The last value of the first half of this very short sequence is 12, which is smaller than the value that we are searching, so we must look in the second half:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| 1   | 4   | 5   | 8   | 9   | 12  | 17  | 20  | 24  | 32  |

A binary search locates a value in a sorted array by determining whether the value occurs in the first or second half, then repeating the search in one of the halves.

It is trivial to see that we don't have a match, because  $15 \neq 17$ . If we wanted to insert 15 into the sequence, we would need to insert it just before  $a[6]$ .

This search process is called a **binary search**, because we cut the size of the search in half in each step. That cutting in half works only because we know that the sequence of values is sorted.

The following class implements binary searches in a sorted array of integers. The search method returns the position of the match if the search succeeds, or -1 if the value is not found in  $a$ . Here, we show a recursive version of the binary search algorithm.

### section\_6\_2/BinarySearcher.java

```

1  /**
2   * A class for executing binary searches in an array.
3   */
4  public class BinarySearcher
5  {
6      /**
7       * Finds a value in a range of a sorted array, using the binary
8       * search algorithm.
9       * @param a the array in which to search
10      * @param low the low index of the range
11      * @param high the high index of the range
12      * @param value the value to find
13      * @return the index at which the value occurs, or -1
14      * if it does not occur in the array
15     */
16    public int search(int[] a, int low, int high, int value)
17    {
18        if (low <= high)
19        {
20            int mid = (low + high) / 2;
21
22            if (a[mid] == value)
23            {
24                return mid;
25            }
26            else if (a[mid] < value )
27            {

```

```

28         return search(a, mid + 1, high, value);
29     }
30     else
31     {
32         return search(a, low, mid - 1, value);
33     }
34 }
35 else
36 {
37     return -1;
38 }
39 }
40 }
```

Now let's determine the number of visits to array elements required to carry out a binary search. We can use the same technique as in the analysis of merge sort. Because we look at the middle element, which counts as one visit, and then search either the left or the right subarray, we have

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Using the same equation,

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + 1$$

By plugging this result into the original equation, we get

$$T(n) = T\left(\frac{n}{4}\right) + 2$$

That generalizes to

$$T(n) = T\left(\frac{n}{2^k}\right) + k$$

As in the analysis of merge sort, we make the simplifying assumption that  $n$  is a power of 2,  $n = 2^m$ , where  $m = \log_2(n)$ . Then we obtain

$$T(n) = 1 + \log_2(n)$$

Therefore, binary search is an  $O(\log(n))$  algorithm.

That result makes intuitive sense. Suppose that  $n$  is 100. Then after each search, the size of the search range is cut in half, to 50, 25, 12, 6, 3, and 1. After seven comparisons we are done. This agrees with our formula, because  $\log_2(100) \approx 6.64386$ , and indeed the next larger power of 2 is  $2^7 = 128$ .

Because a binary search is so much faster than a linear search, is it worthwhile to sort an array first and then use a binary search? It depends. If you search the array only once, then it is more efficient to pay for an  $O(n)$  linear search than for an  $O(n \log(n))$  sort and an  $O(\log(n))$  binary search. But if you will be making many searches in the same array, then sorting it is definitely worthwhile.

A binary search locates a value in a sorted array in  $O(\log(n))$  steps.



18. Suppose you need to look through 1,000,000 records to find a telephone number. How many records do you expect to search before finding the number?
19. Why can't you use a "for each" loop for (int element : a) in the search method?
20. Suppose you need to look through a sorted array with 1,000,000 elements to find a value. Using the binary search algorithm, how many records do you expect to search before finding the value?

**Practice It** Now you can try these exercises at the end of the chapter: R14.14, E14.13, E14.15.



Before pocket calculators and personal computers existed, navigators and engineers used mechanical adding machines, slide rules, and tables of logarithms and trigonometric functions to speed up computations. Unfortunately, the tables—for which values had to be computed by hand—were notoriously inaccurate. The mathematician Charles Babbage (1791–1871) had the insight that if a machine could be constructed that produced printed tables automatically, both calculation and typesetting errors could be avoided. Babbage set out to develop a machine for this purpose, which he called a *Difference Engine*.



Topham/The Image Works

Replica of Babbage's Difference Engine

## Computing & Society 14.1 The First Programmer

because it used successive differences to compute polynomials. For example, consider the function  $f(x) = x^3$ . Write down the values for  $f(1), f(2), f(3)$ , and so on. Then take the *differences* between successive values:

|     |
|-----|
| 1   |
| 7   |
| 8   |
| 19  |
| 27  |
| 37  |
| 64  |
| 61  |
| 125 |
| 91  |
| 216 |

Repeat the process, taking the difference of successive values in the second column, and then repeat once again:

|     |
|-----|
| 1   |
| 7   |
| 8   |
| 12  |
| 19  |
| 6   |
| 27  |
| 18  |
| 6   |
| 37  |
| 6   |
| 64  |
| 24  |
| 6   |
| 61  |
| 6   |
| 125 |
| 30  |
| 91  |
| 216 |

Now the differences are all the same. You can retrieve the function values by a pattern of additions—you need to know the values at the fringe of the pattern and the constant difference. You can try it out yourself: Write the highlighted numbers on a sheet of paper and fill in the others by adding the numbers that are in the north and northwest positions.

This method was very attractive, because mechanical addition machines had been known for some time. They consisted of cog wheels, with 10 cogs per wheel, to represent digits, and mechanisms to handle the carry from one digit to the next. Mechanical multiplication machines, on the other hand, were fragile and unreliable. Babbage built a successful prototype of the Difference Engine and, with his own money and government grants, proceeded to build the table-printing machine. However, because of funding problems and the difficulty of building the machine to the required precision, it was never completed.

While working on the Difference Engine, Babbage conceived of a much grander vision that he called the *Analytical Engine*. The Difference Engine was designed to carry out a limited set of computations—it was no smarter than a pocket calculator is today. But Babbage realized that such a machine could be made *programmable* by storing programs as well as data. The internal storage of the Analytical Engine was to consist of 1,000 registers of 50 decimal digits each. Programs and constants were to be stored on punched cards—a technique that was, at that time, commonly used on looms for weaving patterned fabrics.

Ada Augusta, Countess of Lovelace (1815–1852), the only child of Lord Byron, was a friend and sponsor of Charles Babbage. Ada Lovelace was one of the first people to realize the potential of such a machine, not just for computing mathematical tables but for processing data that were not numbers. She is considered by many to be the world's first programmer.

# 14.7 Problem Solving: Estimating the Running Time of an Algorithm

In this chapter, you have learned how to estimate the running time of sorting algorithms. As you have seen, being able to differentiate between  $O(n \log(n))$  and  $O(n^2)$  running times has great practical implications. Being able to estimate the running times of other algorithms is an important skill. In this section, we will practice estimating the running time of array algorithms.

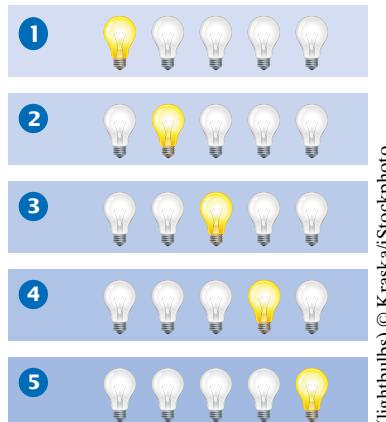
## 14.7.1 Linear Time

Let us start with a simple example, an algorithm that counts how many elements have a particular value:

```
int count = 0;
for (int i = 0; i < a.length; i++)
{
    if (a[i] == value) { count++; }
```

What is the running time in terms of  $n$ , the length of the array?

Start with looking at the pattern of array element visits. Here, we visit each element once. It helps to visualize this pattern. Imagine the array as a sequence of light bulbs. As the  $i$ th element gets visited, imagine the  $i$ th bulb lighting up.



(lightbulbs) © KraskaiStockphoto.

Now look at the work per visit. Does each visit involve a fixed number of actions, independent of  $n$ ? In this case, it does. There are just a few actions—read the element, compare it, maybe increment a counter.

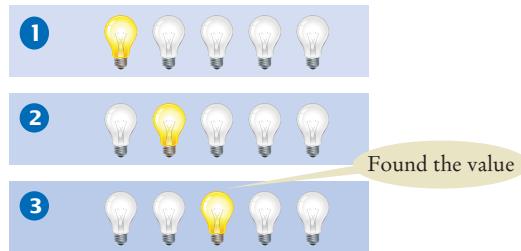
Therefore, the running time is  $n$  times a constant, or  $O(n)$ .

What if we don't always run to the end of the array? For example, suppose we want to check whether the value occurs in the array, without counting it:

```
boolean found = false;
for (int i = 0; !found && i < a.length; i++)
{
    if (a[i] == value) { found = true; }
```

A loop with  $n$  iterations has  $O(n)$  running time if each step consists of a fixed number of actions.

Then the loop can stop in the middle:



Is this still  $O(n)$ ? It is, because in some cases the match may be at the very end of the array. Also, if there is no match, one must traverse the entire array.

### 14.7.2 Quadratic Time

Now let's turn to a more interesting case. What if we do a lot of work with each visit? Here is an example: We want to find the most frequent element in an array.

Suppose the array is

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 8 | 7 | 5 | 7 | 7 | 5 | 4 |
|---|---|---|---|---|---|---|

It's obvious by looking at the values that 7 is the most frequent one. But now imagine an array with a few thousand values.

|   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |   |   |     |    |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|----|---|---|---|-----|----|---|---|---|---|---|
| 8 | 7 | 5 | 7 | 7 | 5 | 4 | 1 | 2 | 3 | 3 | 4 | 9 | 12 | 3 | 2 | 5 | ... | 11 | 9 | 2 | 3 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|----|---|---|---|-----|----|---|---|---|---|---|

We can count how often the value 8 occurs, then move on to count how often 7 occurs, and so on. For example, in the first array, 8 occurs once, and 7 occurs three times. Where do we put the counts? Let's put them into a second array of the same length.

|         |   |   |   |   |   |   |   |
|---------|---|---|---|---|---|---|---|
| a:      | 8 | 7 | 5 | 7 | 7 | 5 | 4 |
| counts: | 1 | 3 | 2 | 3 | 3 | 2 | 1 |

Then we take the maximum of the counts. It is 3. We look up where the 3 occurs in the counts, and find the corresponding value. Thus, the most common value is 7.

Let us first estimate how long it takes to compute the counts.

```
for (int i = 0; i < a.length; i++)
{
    counts[i] = Count how often a[i] occurs in a
}
```

A loop with  $n$  iterations has  $O(n^2)$  running time if each step takes  $O(n)$  time.

We still visit each array element once, but now the work per visit is much larger. As you have seen in the previous section, each counting action is  $O(n)$ . When we do  $O(n)$  work in each step, the total running time is  $O(n^2)$ .

This algorithm has three phases:

1. Compute all counts.
2. Compute the maximum.
3. Find the maximum in the counts.

We have just seen that the first phase is  $O(n^2)$ . Computing the maximum is  $O(n)$ —look at the algorithm in Section 7.3.3 and note that each step involves a fixed amount of work. Finally, we just saw that finding a value is  $O(n)$ .

The big-Oh running time for doing several steps in a row is the largest of the big-Oh times for each step.

How can we estimate the total running time from the estimates of each phase? Of course, the total time is the sum of the individual times, but for big-Oh estimates, we take the *maximum* of the estimates. To see why, imagine that we had actual equations for each of the times:

$$T_1(n) = an^2 + bn + c$$

$$T_2(n) = dn + e$$

$$T_3(n) = fn + g$$

Then the sum is

$$T(n) = T_1(n) + T_2(n) + T_3(n) = an^2 + (b + d + f)n + c + e + g$$

But only the largest term matters, so  $T(n)$  is  $O(n^2)$ .

Thus, we have found that our algorithm for finding the most frequent element is  $O(n^2)$ .

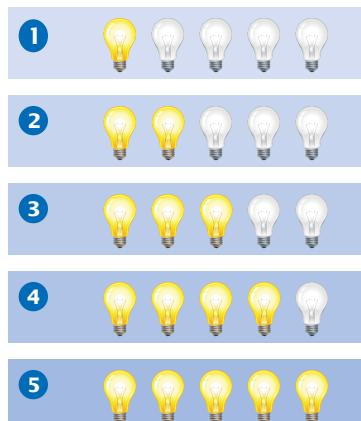
### 14.7.3 The Triangle Pattern

Let us see if we can speed up the algorithm from the preceding section. It seems wasteful to count elements again if we have already counted them.

Can we save time by eliminating repeated counting of the same element? That is, before counting  $a[i]$ , should we first check that it didn't occur in  $a[0] \dots a[i - 1]$ ?

Let us estimate the cost of these additional checks. In the  $i$ th step, the amount of work is proportional to  $i$ . That's not quite the same as in the preceding section, where you saw that a loop with  $n$  iterations, each of which takes  $O(n)$  time, is  $O(n^2)$ . Now each step just takes  $O(i)$  time.

To get an intuitive feel for this situation, look at the light bulbs again. In the second iteration, we visit  $a[0]$  again. In the third iteration, we visit  $a[0]$  and  $a[1]$  again, and so on. The light bulb pattern is



A loop with  $n$  iterations has  $O(n^2)$  running time if the  $i$ th step takes  $O(i)$  time.

If there are  $n$  light bulbs, about half of the square above, or  $n^2/2$  of them, light up. That's unfortunately still  $O(n^2)$ .

Here is another idea for time saving. When we count  $a[i]$ , there is no need to do the counting in  $a[0] \dots a[i - 1]$ . If  $a[i]$  never occurred before, we get an accurate

count by just looking at  $a[i] \dots a[n - 1]$ . And if it did, we already have an accurate count. Does that help us? Not really—it's the triangle pattern again, but this time in the other direction.



That doesn't mean that these improvements aren't worthwhile. If an  $O(n^2)$  algorithm is the best one can do for a particular problem, you still want to make it as fast as possible. However, we will not pursue this plan further because it turns out that we can do much better.

#### 14.7.4 Logarithmic Time

Logarithmic time estimates arise from algorithms that cut work in half in each step. You have seen this in the algorithms for binary search and merge sort, and you will see it again in Chapter 17.

In particular, when you use sorting or binary search in a phase of an algorithm, you will encounter logarithmic time in the big-Oh estimates.

Consider this idea for improving our algorithm for finding the most frequent element. Suppose we first *sort* the array:

8 | 7 | 5 | 7 | 7 | 5 | 4 → 4 | 5 | 5 | 7 | 7 | 7 | 8

That cost us  $O(n \log(n))$  time. If we can complete the algorithm in  $O(n)$  time, we will have found a better algorithm than the  $O(n^2)$  algorithm of the preceding sections.

To see why this is possible, imagine traversing the sorted array. As long as you find a value that was equal to its predecessor, you increment a counter. When you find a different value, save the counter and start counting anew:

|         |   |   |   |   |   |   |   |
|---------|---|---|---|---|---|---|---|
| a:      | 4 | 5 | 5 | 7 | 7 | 7 | 8 |
| counts: | 1 | 1 | 2 | 1 | 2 | 3 | 1 |

Or in code,

```
int count = 0;
for (int i = 0; i < a.length; i++)
{
    count++;
    if (i == a.length - 1 || a[i] != a[i + 1])
    {

```

An algorithm that cuts the size of work in half in each step runs in  $O(\log(n))$  time.

```

        counts[i] = count;
        count = 0;
    }
}

```

That's a constant amount of work per iteration, even though it visits two elements:



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bje06code](http://wiley.com/go/bje06code) to download a program for comparing the speed of algorithms that find the most frequent element.



#### SELF CHECK

$2n$  is still  $O(n)$ . Thus, we can compute the counts in  $O(n)$  time from a sorted array. The entire algorithm is now  $O(n \log(n))$ .

Note that we don't actually need to keep all counts, only the highest one that we encountered so far (see Exercise E14.11). That is a worthwhile improvement, but it does not change the big-Oh estimate of the running time.

21. What is the “light bulb pattern” of visits in the following algorithm to check whether an array is a palindrome?
 

```

for (int i = 0; i < a.length / 2; i++)
{
    if (a[i] != a[a.length - 1 - i]) { return false; }
}
return true;
      
```
22. What is the big-Oh running time of the following algorithm to check whether the first element is duplicated in an array?
 

```

for (int i = 1; i < a.length; i++)
{
    if (a[0] == a[i]) { return true; }
}
return false;
      
```
23. What is the big-Oh running time of the following algorithm to check whether an array has a duplicate value?
 

```

for (int i = 0; i < a.length; i++)
{
    for (j = i + 1; j < a.length; j++)
    {
        if (a[i] == a[j]) { return true; }
    }
}
return false;
      
```
24. Describe an  $O(n \log(n))$  algorithm for checking whether an array has duplicates.

25. What is the big-Oh running time of the following algorithm to find an element in an  $n \times n$  array?

```
for (int i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        if (a[i][j] == value) { return true; }
    }
}
return false;
```

26. If you apply the algorithm of Section 14.7.4 to an  $n \times n$  array, what is the big-Oh efficiency of finding the most frequent element in terms of  $n$ ?

**Practice It** Now you can try these exercises at the end of the chapter: R14.9, R14.15, R14.21, E14.11.

## 14.8 Sorting and Searching in the Java Library

When you write Java programs, you don't have to implement your own sorting algorithms. The `Arrays` and `Collections` classes provide sorting and searching methods that we will introduce in the following sections.

### 14.8.1 Sorting

The `Arrays` class implements a sorting method that you should use for your Java programs.

The `Arrays` class contains static sort methods to sort arrays of integers and floating-point numbers. For example, you can sort an array of integers simply as

```
int[] a = . . .;
Arrays.sort(a);
```

That `sort` method uses the quicksort algorithm—see Special Topic 14.3 for more information about that algorithm.

If your data are contained in an `ArrayList`, use the `Collections.sort` method instead; it uses the merge sort algorithm:

```
ArrayList<String> names = . . .;
Collections.sort(names);
```

The `Collections` class contains a sort method that can sort array lists.

### 14.8.2 Binary Search

The `Arrays` and `Collections` classes contain static `binarySearch` methods that implement the binary search algorithm, but with a useful enhancement. If a value is not found in the array, then the returned value is not  $-1$ , but  $-k - 1$ , where  $k$  is the position before which the element should be inserted. For example,

```
int[] a = { 1, 4, 9 };
int v = 7;
int pos = Arrays.binarySearch(a, v);
// Returns -3; v should be inserted before position 2
```

### 14.8.3 Comparing Objects

The sort method of the Arrays class sorts objects of classes that implement the Comparable interface.

In application programs, you often need to sort or search through collections of objects. Therefore, the `Arrays` and `Collections` classes also supply `sort` and `binarySearch` methods for objects. However, these methods cannot know how to compare arbitrary objects. Suppose, for example, that you have an array of `Country` objects. It is not obvious how the countries should be sorted. Should they be sorted by their names or by their areas? The `sort` and `binarySearch` methods cannot make that decision for you. Instead, they require that the objects belong to classes that implement the `Comparable` interface type that was introduced in Section 10.3. That interface has a single method:

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

The call

```
a.compareTo(b)
```

must return a negative number if `a` should come before `b`, 0 if `a` and `b` are the same, and a positive number otherwise.

Note that `Comparable` is a generic type, similar to the `ArrayList` type. With an `ArrayList`, the type parameter denotes the type of the elements. With `Comparable`, the type parameter is the type of the parameter of the `compareTo` method. Therefore, a class that implements `Comparable` will want to be “comparable to itself”. For example, the `Country` class implements `Comparable<Country>`.

Many classes in the standard Java library, including the `String` class, number wrappers, dates, and file paths, implement the `Comparable` interface.

You can implement the `Comparable` interface for your own classes as well. For example, to sort a collection of countries by area, the `Country` class would implement the `Comparable<Country>` interface and provide a `compareTo` method like this:

```
public class Country implements Comparable<Country>
{
    public int compareTo(Country other)
    {
        return Double.compare(area, other.area);
    }
}
```

The `compareTo` method compares countries by their area. Note the use of the helper method `Double.compare` (see Programming Tip 10.1) that returns a negative integer, 0, or a positive integer. This is easier than programming a three-way branch.

Now you can pass an array of countries to the `Arrays.sort` method:

```
Country[] countries = new Country[n];
// Add countries
Arrays.sort(countries); // Sorts by increasing area
```

Whenever you need to carry out sorting or searching, use the methods in the `Arrays` and `Collections` classes and not those that you write yourself. The library algorithms have been fully debugged and optimized. Thus, the primary purpose of this chapter was not to teach you how to implement practical sorting and searching algorithms. Instead, you have learned something more important, namely that different algorithms can vary widely in performance, and that it is worthwhile to learn more about the design and analysis of algorithms.



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download a program that demonstrates the Java library methods for sorting and searching.



27. Why can't the `Arrays.sort` method sort an array of `Rectangle` objects?
28. What steps would you need to take to sort an array of `BankAccount` objects by increasing balance?
29. Why is it useful that the `Arrays.binarySearch` method indicates the position where a missing element should be inserted?
30. Why does `Arrays.binarySearch` return  $-k - 1$  and not  $-k$  to indicate that a value is not present and should be inserted before position  $k$ ?

**Practice It** Now you can try these exercises at the end of the chapter: E14.12, E14.16, E14.17.

### Common Error 14.1



#### The `compareTo` Method Can Return Any Integer, Not Just `-1`, `0`, and `1`

The call `a.compareTo(b)` is allowed to return *any* negative integer to denote that `a` should come before `b`, not necessarily the value `-1`. That is, the test

```
if (a.compareTo(b) == -1) // Error!
```

is generally wrong. Instead, you should test

```
if (a.compareTo(b) < 0) // OK
```

Why would a `compareTo` method ever want to return a number other than `-1`, `0`, or `1`? Sometimes, it is convenient to just return the difference of two integers. For example, the `compareTo` method of the `String` class compares characters in matching positions:

```
char c1 = charAt(i);
char c2 = other.charAt(i);
```

If the characters are different, then the method simply returns their difference:

```
if (c1 != c2) { return c1 - c2; }
```

This difference is a negative number if `c1` is less than `c2`, but it is not necessarily the number `-1`.

Note that returning a difference only works if it doesn't overflow (see Programming Tip 10.1).

### Special Topic 14.4



#### The Comparator Interface

Sometimes you want to sort an array or array list of objects, but the objects don't belong to a class that implements the `Comparable` interface. Or, perhaps, you want to sort the array in a different order. For example, you may want to sort countries by name rather than by value.

You wouldn't want to change the implementation of a class simply to call `Arrays.sort`. Fortunately, there is an alternative. One version of the `Arrays.sort` method does not require that the objects belong to classes that implement the `Comparable` interface. Instead, you can supply arbitrary objects. However, you must also provide a *comparator* object whose job is to compare objects. The comparator object must belong to a class that implements the `Comparator` interface. That interface has a single method, `compare`, which compares two objects.

Just like `Comparable`, the `Comparator` interface is a parameterized type. The type parameter specifies the type of the `compare` parameter variables. For example, `Comparator<Country>` looks like this:

```
public interface Comparator<Country>
{
    int compare(Country a, Country b);
}
```

The call

```
comp.compare(a, b)
```

must return a negative number if a should come before b, 0 if a and b are the same, and a positive number otherwise. (Here, comp is an object of a class that implements Comparator<Country>.)

For example, here is a Comparator class for country:

```
public class CountryComparator implements Comparator<Country>
{
    public int compare(Country a, Country b)
    {
        return Double.compare(a.getArea(), b.getArea());
    }
}
```

To sort an array of countries by area, call

```
Arrays.sort(countries, new CountryComparator());
```

### Java 8 Note 14.1



#### Comparators with Lambda Expressions

Before Java 8, it was cumbersome to specify a comparator. You had to come up with a class that implements the Comparator interface, implement the compare method, and construct an object of that class. That was unfortunate because comparators are very useful for several algorithms, such as searching, sorting, and finding the maximum or minimum.

With lambda expressions, it is easier to specify a comparator. For example, to sort an array of words by increasing lengths, call

```
Arrays.sort(words, (v, w) -> v.length() - w.length());
```

There is a convenient shortcut for this case. Note that the comparison depends on a function that maps each string to a numeric value, namely its length. The static method Comparator.comparing constructs a comparator from a lambda expression. For example, you can call

```
Arrays.sort(words, Comparator.comparing(w -> w.length()));
```

A comparator is constructed that calls the supplied function on both objects that are to be compared, and then compares the function results.

The Comparator.comparing method takes care of many common cases. For example, to sort countries by area, call

```
Arrays.sort(countries, Comparator.comparing(c -> c.getArea()));
```



### WORKED EXAMPLE 14.1



#### Enhancing the Insertion Sort Algorithm

Learn how to implement an improvement of the insertion sort algorithm shown in Special Topic 14.2. The enhanced algorithm is called *Shell sort* after its inventor, Donald Shell. Go to [wiley.com/go/bjeo6examples](http://wiley.com/go/bjeo6examples) and download Worked Example 14.1.

## CHAPTER SUMMARY

### Describe the selection sort algorithm.



- The selection sort algorithm sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.

### Measure the running time of a method.

- To measure the running time of a method, get the current time immediately before and after the method call.

### Use the big-Oh notation to describe the running time of an algorithm.

- Computer scientists use the big-Oh notation to describe the growth rate of a function.
- Selection sort is an  $O(n^2)$  algorithm. Doubling the data set means a fourfold increase in processing time.
- Insertion sort is an  $O(n^2)$  algorithm.



### Describe the merge sort algorithm.



- The merge sort algorithm sorts an array by cutting the array in half, recursively sorting each half, and then merging the sorted halves.

### Contrast the running times of the merge sort and selection sort algorithms.

- Merge sort is an  $O(n \log(n))$  algorithm. The  $n \log(n)$  function grows much more slowly than  $n^2$ .

### Describe the running times of the linear search algorithm and the binary search algorithm.

- A linear search examines all values in an array until it finds a match or reaches the end.
- A linear search locates a value in an array in  $O(n)$  steps.
- A binary search locates a value in a sorted array by determining whether the value occurs in the first or second half, then repeating the search in one of the halves.
- A binary search locates a value in a sorted array in  $O(\log(n))$  steps.

### Practice developing big-Oh estimates of algorithms.

- A loop with  $n$  iterations has  $O(n)$  running time if each step consists of a fixed number of actions.
- A loop with  $n$  iterations has  $O(n^2)$  running time if each step takes  $O(n)$  time.
- The big-Oh running time for doing several steps in a row is the largest of the big-Oh times for each step.



- A loop with  $n$  iterations has  $O(n^2)$  running time if the  $i$ th step takes  $O(i)$  time.
- An algorithm that cuts the size of work in half in each step runs in  $O(\log(n))$  time.

### Use the Java library methods for sorting and searching data.

- The `Arrays` class implements a sorting method that you should use for your Java programs.
- The `Collections` class contains a `sort` method that can sort array lists.
- The `sort` method of the `Arrays` class sorts objects of classes that implement the `Comparable` interface.

### STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

`java.lang.System`  
`currentTimeMillis`  
`java.util.Arrays`  
`binarySearch`  
`sort`

`java.util.Collections`  
`binarySearch`  
`sort`  
`java.util.Comparator<T>`  
`compare`  
`comparing`

### REVIEW EXERCISES

- **R14.1** What is the difference between searching and sorting?
- **R14.2** *Checking against off-by-one errors.* When writing the selection sort algorithm of Section 14.1, a programmer must make the usual choices of `<` versus `<=`, `a.length` versus `a.length - 1`, and `from` versus `from + 1`. This is fertile ground for off-by-one errors. Conduct code walkthroughs of the algorithm with arrays of length 0, 1, 2, and 3 and check carefully that all index values are correct.
- **R14.3** For the following expressions, what is the order of the growth of each?
 

|  |   |
|--|---|
| <b>a.</b> $n^2 + 2n + 1$                   | <b>g.</b> $n + \log(n)$                 |
| <b>b.</b> $n^{10} + 9n^9 + 20n^8 + 145n^7$ | <b>h.</b> $n^2 + n \log(n)$             |
| <b>c.</b> $(n + 1)^4$                      | <b>i.</b> $2^n + n^2$                   |
| <b>d.</b> $(n^2 + n)^2$                    | <b>j.</b> $\frac{n^3 + 2n}{n^2 + 0.75}$ |
| <b>e.</b> $n + 0.001n^3$                   |   |
| <b>f.</b> $n^3 - 1000n^2 + 10^9$           |   |
- **R14.4** We determined that the actual number of visits in the selection sort algorithm is

$$T(n) = \frac{1}{2}n^2 + \frac{5}{2}n - 3$$

We characterized this method as having  $O(n^2)$  growth. Compute the actual ratios

$$\begin{aligned} & T(2,000)/T(1,000) \\ & T(4,000)/T(1,000) \\ & T(10,000)/T(1,000) \end{aligned}$$

and compare them with

$$f(2,000)/f(1,000)$$

$$f(4,000)/f(1,000)$$

$$f(10,000)/f(1,000)$$

where  $f(n) = n^2$ .

- **R14.5** Suppose algorithm  $A$  takes five seconds to handle a data set of 1,000 records. If the algorithm  $A$  is an  $O(n)$  algorithm, approximately how long will it take to handle a data set of 2,000 records? Of 10,000 records?
- ■ **R14.6** Suppose an algorithm takes five seconds to handle a data set of 1,000 records. Fill in the following table, which shows the approximate growth of the execution times depending on the complexity of the algorithm.

|        | $O(n)$ | $O(n^2)$ | $O(n^3)$ | $O(n \log(n))$ | $O(2^n)$ |
|--------|--------|----------|----------|----------------|----------|
| 1,000  | 5      | 5        | 5        | 5              | 5        |
| 2,000  |        |          |          |                |          |
| 3,000  |        | 45       |          |                |          |
| 10,000 |        |          |          |                |          |

For example, because  $3,000^2/1,000^2 = 9$ , the algorithm would take nine times as long, or 45 seconds, to handle a data set of 3,000 records.

- ■ **R14.7** Sort the following growth rates from slowest to fastest growth.

|          |                  |               |                  |
|----------|------------------|---------------|------------------|
| $O(n)$   | $O(\log(n))$     | $O(2^n)$      | $O(n\sqrt{n})$   |
| $O(n^3)$ | $O(n^2 \log(n))$ | $O(\sqrt{n})$ | $O(n^{\log(n)})$ |
| $O(n^n)$ | $O(n \log(n))$   |               |                  |

- **R14.8** What is the growth rate of the standard algorithm to find the minimum value of an array? Of finding both the minimum and the maximum?
- **R14.9** What is the big-Oh time estimate of the following method in terms of  $n$ , the length of  $a$ ? Use the “light bulb pattern” method of Section 14.7 to visualize your result.

```
public static void swap(int[] a)
{
    int i = 0;
    int j = a.length - 1;
    while (i < j)
    {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
        i++;
        j--;
    }
}
```

- **R14.10** A *run* is a sequence of adjacent repeated values (see Exercise R7.23). Describe an  $O(n)$  algorithm to find the length of the longest run in an array.
- **R14.11** Consider the task of finding the most frequent element in an array of length  $n$ . Here are three approaches:
- Sort the array, then find the longest run.
  - Allocate an array of counters of the same size as the original array. For each element, traverse the array and count how many other elements are equal to it, updating its counter. Then find the maximum count.
  - Keep variables for the most frequent element that you have seen so far and its frequency. For each index  $i$ , check whether  $a[i]$  occurs in  $a[0] \dots a[i - 1]$ . If not, count how often it occurs in  $a[i + 1] \dots a[n - 1]$ . If  $a[i]$  is more frequent than the most frequent element so far, update the variables.
- Describe the big-Oh efficiency of each approach.
- **R14.12** Trace a walkthrough of selection sort with these sets:
- 4 7 11 4 9 5 11 7 3 5
  - 7 6 8 7 5 9 0 11 10 5 8
- **R14.13** Trace a walkthrough of merge sort with these sets:
- 5 11 7 3 5 4 7 11 4 9
  - 9 0 11 10 5 8 -7 6 8 7 5
- **R14.14** Trace a walkthrough of:
- Linear search for 7 in -7 1 3 3 4 7 11 13
  - Binary search for 8 in -7 2 2 3 4 7 8 11 13
  - Binary search for 8 in -7 1 2 3 5 7 10 13
- **R14.15** Your task is to remove all duplicates from an array. For example, if the array has the values
- $$4 \ 7 \ 11 \ 4 \ 9 \ 5 \ 11 \ 7 \ 3 \ 5$$
- then the array should be changed to
- $$4 \ 7 \ 11 \ 9 \ 5 \ 3$$
- Here is a simple algorithm: Look at  $a[i]$ . Count how many times it occurs in  $a$ . If the count is larger than 1, remove it. What is the growth rate of the time required for this algorithm?
- **R14.16** Modify the merge sort algorithm to remove duplicates in the merging step to obtain an algorithm that removes duplicates from an array. Note that the resulting array does not have the same ordering as the original one. What is the efficiency of this algorithm?
- **R14.17** Consider the following algorithm to remove all duplicates from an array: Sort the array. For each element in the array, look at its next neighbor to decide whether it is present more than once. If so, remove it. Is this a faster algorithm than the one in Exercise R14.15?
- **R14.18** Develop an  $O(n \log(n))$  algorithm for removing duplicates from an array if the resulting array must have the same ordering as the original array. When a value occurs multiple times, all but its first occurrence should be removed.

- R14.19** Why does insertion sort perform significantly better than selection sort if an array is already sorted?
- R14.20** Consider the following speedup of the insertion sort algorithm of Special Topic 14.2. For each element, use the enhanced binary search algorithm that yields the insertion position for missing elements. Does this speedup have a significant impact on the efficiency of the algorithm?
- R14.21** Consider the following algorithm known as *bubble sort*:
- ```

While the array is not sorted
  For each adjacent pair of elements
    If the pair is not sorted
      Swap its elements.

```
- What is the big-Oh efficiency of this algorithm?
- R14.22** The *radix sort* algorithm sorts an array of  $n$  integers with  $d$  digits, using ten auxiliary arrays. First place each value  $v$  into the auxiliary array whose index corresponds to the last digit of  $v$ . Then move all values back into the original array, preserving their order. Repeat the process, now using the next-to-last (tens) digit, then the hundreds digit, and so on. What is the big-Oh time of this algorithm in terms of  $n$  and  $d$ ? When is this algorithm preferable to merge sort?
- R14.23** A *stable sort* does not change the order of elements with the same value. This is a desirable feature in many applications. Consider a sequence of e-mail messages. If you sort by date and then by sender, you'd like the second sort to preserve the relative order of the first, so that you can see all messages from the same sender in date order. Is selection sort stable? Insertion sort? Why or why not?
- R14.24** Give an  $O(n)$  algorithm to sort an array of  $n$  bytes (numbers between  $-128$  and  $127$ ). *Hint:* Use an array of counters.
- R14.25** You are given a sequence of arrays of words, representing the pages of a book. Your task is to build an index (a sorted array of words), each element of which has an array of sorted numbers representing the pages on which the word appears. Describe an algorithm for building the index and give its big-Oh running time in terms of the total number of words.
- R14.26** Given two arrays of  $n$  integers each, describe an  $O(n \log(n))$  algorithm for determining whether they have an element in common.
- R14.27** Given an array of  $n$  integers and a value  $v$ , describe an  $O(n \log(n))$  algorithm to find whether there are two values  $x$  and  $y$  in the array with sum  $v$ .
- R14.28** Given two arrays of  $n$  integers each, describe an  $O(n \log(n))$  algorithm for finding all elements that they have in common.
- R14.29** Suppose we modify the quicksort algorithm from Special Topic 14.3, selecting the middle element instead of the first one as pivot. What is the running time on an array that is already sorted?
- R14.30** Suppose we modify the quicksort algorithm from Special Topic 14.3, selecting the middle element instead of the first one as pivot. Find a sequence of values for which this algorithm has an  $O(n^2)$  running time.

## PRACTICE EXERCISES

- **E14.1** Modify the selection sort algorithm to sort an array of integers in descending order.
- **E14.2** Modify the selection sort algorithm to sort an array of coins by their value.
- **E14.3** Write a program that automatically generates the table of sample run times for the selection sort algorithm. The program should ask for the smallest and largest value of  $n$  and the number of measurements and then make all sample runs.
- **E14.4** Modify the merge sort algorithm to sort an array of strings in lexicographic order.
- **E14.5** Modify the selection sort algorithm to sort an array of objects that implement the `Measurable` interface from Chapter 10.
- **E14.6** Modify the selection sort algorithm to sort an array of objects that implement the `Comparable` interface (without a type parameter).
- **E14.7** Modify the selection sort algorithm to sort an array of objects, given a parameter of type `Comparator` (without a type parameter).
- **E14.8** Write a telephone lookup program. Read a data set of 1,000 names and telephone numbers from a file that contains the numbers in random order. Handle lookups by name and also reverse lookups by phone number. Use a binary search for both lookups.
- **E14.9** Implement a program that measures the performance of the insertion sort algorithm described in Special Topic 14.2.
- **E14.10** Implement the bubble sort algorithm described in Exercise R14.21.
- **E14.11** Implement the algorithm described in Section 14.7.4, but only remember the value with the highest frequency so far:
 

```
int mostFrequent = 0;
int highestFrequency = -1;
for (int i = 0; i < a.length; i++)
    Count how often a[i] occurs in a[i + 1] ... a[a.length - 1]
    If it occurs more often than highestFrequency
        highestFrequency = that count
        mostFrequent = a[i]
```
- **E14.12** Write a program that sorts an `ArrayList<Country>` in decreasing order so that the largest country is at the beginning of the array. Use a `Comparator`.
- **E14.13** Consider the binary search algorithm in Section 14.6. If no match is found, the `search` method returns  $-1$ . Modify the method so that if  $a$  is not found, the method returns  $-k - 1$ , where  $k$  is the position before which the element should be inserted. (This is the same behavior as `Arrays.binarySearch()`.)
- **E14.14** Implement the `sort` method of the merge sort algorithm without recursion, where the length of the array is a power of 2. First merge adjacent regions of size 1, then adjacent regions of size 2, then adjacent regions of size 4, and so on.
- **E14.15** Use insertion sort and the binary search from Exercise E14.13 to sort an array as described in Exercise R14.20. Implement this algorithm and measure its performance.

- **E14.16** Supply a class `Person` that implements the `Comparable` interface. Compare persons by their names. Ask the user to input ten names and generate ten `Person` objects. Using the `compareTo` method, determine the first and last person among them and print them.
- **E14.17** Sort an array list of strings by increasing *length*. Hint: Supply a `Comparator`.
- **E14.18** Sort an array list of strings by increasing length, and so that strings of the same length are sorted lexicographically. Hint: Supply a `Comparator`.

## PROGRAMMING PROJECTS

- **P14.1** It is common for people to name directories as `dir1`, `dir2`, and so on. When there are ten or more directories, the operating system displays them in dictionary order, as `dir1`, `dir10`, `dir11`, `dir12`, `dir2`, `dir3`, and so on. That is irritating, and it is easy to fix. Provide a comparator that compares strings that end in digit sequences in a way that makes sense to a human. First compare the part before the digits as strings, and then compare the numeric values of the digits.
- **P14.2** Sometimes, directory or file names have numbers in the middle, and there may be more than one number, for example, `sec3_14.txt` or `sec10_1.txt`. Provide a comparator that can compare such strings in a way that makes sense to humans. Break each string into strings not containing digits and digit groups. Then compare two strings by comparing the first non-digit groups as strings, the first digit groups as integers, and so on.
- **P14.3** The median  $m$  of a sequence of  $n$  elements is the element that would fall in the middle if the sequence was sorted. That is,  $e \leq m$  for half the elements, and  $m \leq e$  for the others. Clearly, one can obtain the median by sorting the sequence, but one can do quite a bit better with the following algorithm that finds the  $k$ th element of a sequence between  $a$  (inclusive) and  $b$  (exclusive). (For the median, use  $k = n / 2$ ,  $a = 0$ , and  $b = n$ .)

`select(k, a, b):`

Pick a pivot  $p$  in the subsequence between  $a$  and  $b$ .

Partition the subsequence elements into three subsequences: the elements  $< p, = p, > p$

Let  $n_1, n_2, n_3$  be the sizes of each of these subsequences.

if  $k < n_1$

    return `select(k, 0, n1)`.

else if ( $k > n_1 + n_2$ )

    return `select(k, n1 + n2, n)`.

else

    return  $p$ .

Implement this algorithm and measure how much faster it is for computing the median of a random large sequence, when compared to sorting the sequence and taking the middle element.

- **P14.4** Implement the following modification of the quicksort algorithm, due to Bentley and McIlroy. Instead of using the first element as the pivot, use an approximation of the median.

If  $n \leq 7$ , use the middle element. If  $n \leq 40$ , use the median of the first, middle, and last element. Otherwise compute the “pseudomedian” of the nine elements  $a[i * (n - 1) / 8]$ , where  $i$  ranges from 0 to 8. The pseudomedian of nine values is  $\text{med}(\text{med}(v_0, v_1, v_2), \text{med}(v_3, v_4, v_5), \text{med}(v_6, v_7, v_8))$ .

Compare the running time of this modification with that of the original algorithm on sequences that are nearly sorted or reverse sorted, and on sequences with many identical elements. What do you observe?

- **P14.5** Bentley and McIlroy suggest the following modification to the quicksort algorithm when dealing with data sets that contain many repeated elements.

Instead of partitioning as



(where  $\leq$  denotes the elements that are  $\leq$  the pivot), it is better to partition as



However, that is tedious to achieve directly. They recommend to partition as



and then swap the two  $=$  regions into the middle. Implement this modification and check whether it improves performance on data sets with many repeated elements.

- **P14.6** Implement the radix sort algorithm described in Exercise R14.22 to sort arrays of numbers between 0 and 999.
- **P14.7** Implement the radix sort algorithm described in Exercise R14.22 to sort arrays of numbers between 0 and 999. However, use a single auxiliary array, not ten.
- **P14.8** Implement the radix sort algorithm described in Exercise R14.22 to sort arbitrary `int` values (positive or negative).
- **P14.9** Implement the `sort` method of the merge sort algorithm without recursion, where the length of the array is an arbitrary number. Keep merging adjacent regions whose size is a power of 2, and pay special attention to the last area whose size is less.

## ANSWERS TO SELF-CHECK QUESTIONS

1. Dropping the `temp` variable would not work. Then  $a[i]$  and  $a[j]$  would end up being the same value.
2.  $1 \mid 5 \ 4 \ 3 \ 2 \ 6$   
 $1 \ 2 \mid 4 \ 3 \ 5 \ 6$   
 $1 \ 2 \ 3 \ 4 \ 5 \ 6$
3. In each step, find the *maximum* of the remaining elements and swap it with the current element (or see Self Check 4).
4. The modified algorithm sorts the array in descending order.
5. Four times as long as 40,000 values, or about 37 seconds.
6. A parabola.
7. It takes about 100 times longer.
8. If  $n$  is 4, then  $\frac{1}{2}n^2$  is 8 and  $\frac{5}{2}n - 3$  is 7.
9. The first algorithm requires one visit, to store the new element. The second algorithm requires  $T(p) = 2 \times (n - p - 1)$  visits, where  $p$  is the location at which the element is removed. We don't know where that element is, but if elements are removed at random locations, on average, half of the removals will be above the middle and half below, so we can assume an average  $p$  of  $n / 2$  and  $T(n) = 2 \times (n - n / 2 - 1) = n - 2$ .

10. The first algorithm is  $O(1)$ , the second  $O(n)$ .
11. We need to check that  $a[0] \leq a[1]$ ,  $a[1] \leq a[2]$ , and so on, visiting  $2n - 2$  elements. Therefore, the running time is  $O(n)$ .
12. Let  $n$  be the length of the array. In the  $k$ th step, we need  $k$  visits to find the minimum. To remove it, we need an average of  $k - 2$  visits (see Self Check 9). One additional visit is required to add it to the end. Thus, the  $k$ th step requires  $2k - 1$  visits. Because  $k$  goes from  $n$  to 2, the total number of visits is

$$\begin{aligned}2n - 1 + 2(n - 1) - 1 + \dots + 2 \cdot 3 - 1 + 2 \cdot 2 - 1 = \\2(n + (n - 1) + \dots + 3 + 2 + 1 - 1) - (n - 1) = \\n(n + 1) - 2 - n + 1 = n^2 - 3\end{aligned}$$

(because  $1 + 2 + 3 + \dots + (n - 1) + n = n(n + 1)/2$ ). Therefore, the total number of visits is  $O(n^2)$ .

13. When the preceding `while` loop ends, the loop condition must be false, that is, `iFirst >= first.length` or `iSecond >= second.length` (De Morgan's Law).
14. First sort 8 7 6 5. Recursively, first sort 8 7. Recursively, first sort 8. It's sorted. Sort 7. It's sorted. Merge them: 7 8. Do the same with 6 5 to get 5 6. Merge them to 5 6 7 8. Do the same with 4 3 2 1: Sort 4 3 by sorting 4 and 3 and merging them to 3 4. Sort 2 1 by sorting 2 and 1 and merging them to 1 2. Merge 3 4 and 1 2 to 1 2 3 4. Finally, merge 5 6 7 8 and 1 2 3 4 to 1 2 3 4 5 6 7 8.
15. If the array size is 1, return its only element as the sum. Otherwise, recursively compute the sum of the first and second subarray and return the sum of these two values.
16. Approximately  $(100,000 \cdot \log(100,000)) / (50,000 \cdot \log(50,000)) = 2 \cdot 5 / 4.7 = 2.13$  times the time required for 50,000 values. That's  $2.13 \cdot 192$  milliseconds or approximately 409 milliseconds.
17.  $\frac{2n \log(2n)}{n \log(n)} = 2 \frac{(1 + \log(2))}{\log(n)}$

For  $n > 2$ , that is a value  $< 3$ .

18. On average, you'd make 500,000 comparisons.
19. The search method returns the index at which the match occurs, not the data stored at that location.
20. You would search about 20. (The binary log of 1,024 is 10.)
21. 
22. It is an  $O(n)$  algorithm.
23. It is an  $O(n^2)$  algorithm—the number of visits follows a triangle pattern.
24. Sort the array, then make a linear scan to check for adjacent duplicates.
25. It is an  $O(n^2)$  algorithm—the outer and inner loops each have  $n$  iterations.
26. Because an  $n \times n$  array has  $m = n^2$  elements, and the algorithm in Section 14.7.4, when applied to an array with  $m$  elements, is  $O(m \log(m))$ , we have an  $O(n^2 \log(n))$  algorithm. Recall that  $\log(n^2) = 2 \log(n)$ , and the factor of 2 is irrelevant in the big-Oh notation.
27. The `Rectangle` class does not implement the `Comparable` interface.
28. The `BankAccount` class would need to implement the `Comparable` interface. Its `compareTo` method must compare the bank balances.
29. Then you know where to insert it so that the array stays sorted, and you can keep using binary search.
30. Otherwise, you would not know whether a value is present when the method returns 0.



## WORKED EXAMPLE 14.1

## Enhancing the Insertion Sort Algorithm



**Problem Statement** Implement an improvement of the insertion sort algorithm (in Special Topic 14.2) called *Shell sort* after its inventor, Donald Shell.

Shell sort is an enhancement of insertion sort that takes advantage of the fact that insertion sort is an  $O(n)$  algorithm if the array is already sorted. Shell sort brings parts of the array into sorted order, then runs an insertion sort over the entire array, so that the final sort doesn't do much work.

A key step in Shell sort is to arrange the sequence into rows and columns, and then to sort each column separately. For example, if the array is

65	46	14	52	38	2	96	39	14	33	13	4	24	99	89	77	73	87	36	81
----	----	----	----	----	---	----	----	----	----	----	---	----	----	----	----	----	----	----	----

and we arrange it into four columns, we get

65	46	14	52
38	2	96	39
14	33	13	4
24	99	89	77
73	87	36	81

Now we sort each column:

14	2	13	5
24	33	14	39
38	46	36	52
65	87	89	77
73	99	96	81

Put together as a single array, we get

14	2	13	5	24	33	14	39	38	46	36	52	65	87	89	77	73	99	96	81
----	---	----	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Note that the array isn't completely sorted, but many of the small numbers are now in front, and many of the large numbers are in the back.

We will repeat the process until the array is sorted. Each time, we use a different number of columns. Shell had originally used powers of two for the column counts. For example, on an array with 20 elements, he proposed using 16, 8, 4, 2, and finally one column. With one column, we have a plain insertion sort, so we know the array will be sorted. What is surprising is that the preceding sorts greatly speed up the process.

However, better sequences have been discovered. We will use the sequence of column counts

$$\begin{aligned} c_1 &= 1 \\ c_2 &= 4 \\ c_3 &= 13 \\ c_4 &= 40 \\ &\dots \\ c_{i+1} &= 3c_i + 1 \end{aligned}$$

That is, for an array with 20 elements, we first do a 13-sort, then a 4-sort, and then a 1-sort. This sequence is almost as good as the best known ones, and it is easy to compute.

We will not actually rearrange the array, but compute the locations of the elements of each column.

## WE2 Chapter 14 Sorting and Searching

For example, if the number of columns  $c$  is 4, the four columns are located in the array as follows:

65			38			14			24		73	
	46			2			33			99		87
		14			96			13			89	
			52			39			4		77	

Note that successive column elements have distance  $c$  from another. The  $k$ th column is made up of the elements  $a[k], a[k + c], a[k + 2 * c]$ , and so on.

Now let's adapt the insertion sort algorithm to sort such a column. The original algorithm was

```
for (int i = 1; i < a.length; i++)
{
    int next = a[i];
    // Move all larger elements up
    int j = i;
    while (j > 0 && a[j - 1] > next)
    {
        a[j] = a[j - 1];
        j--;
    }
    // Insert the element
    a[j] = next;
}
```

The outer loop visits the elements  $a[1], a[2]$ , and so on. In the  $k$ th column, the corresponding sequence is  $a[k + c], a[k + 2 * c]$ , and so on. That is, the outer loop becomes

```
for (int i = k + c; i < a.length; i = i + c)
```

In the inner loop, we originally visited  $a[j], a[j - 1]$ , and so on. We need to change that to  $a[j], a[j - c]$ , and so on. The inner loop becomes

```
while (j >= c && a[j - c] > next)
{
    a[j] = a[j - c];
    j = j - c;
}
```

Putting everything together, we get the following method:

```
/**
 * Sorts a column, using insertion sort.
 * @param a the array to sort
 * @param k the index of the first element in the column
 * @param c the gap between elements in the column
 */
public static void insertionSort(int[] a, int k, int c)
{
    for (int i = k + c; i < a.length; i = i + c)
    {
        int next = a[i];
        // Move all larger elements up
        int j = i;
        while (j >= c && a[j - c] > next)
        {
            a[j] = a[j - c];
            j = j - c;
        }
    }
}
```

```

        // Insert the element
        a[j] = next;
    }
}

```

Now we are ready to implement the Shell sort algorithm. First, we need to find out how many elements we need from the sequence of column counts. We generate the sequence values until they exceed the size of the array to be sorted.

```

ArrayList<Integer> columns = new ArrayList<Integer>();
int c = 1;
while (c < a.length)
{
    columns.add(c);
    c = 3 * c + 1;
}

```

For each column count, we sort all columns:

```

for (int s = columns.size() - 1; s >= 0; s--)
{
    c = columns.get(s);
    for (int k = 0; k < c; k++)
    {
        insertionSort(a, k, c);
    }
}

```

How good is the performance? Let's compare with the `Arrays.sort` method in the Java library.

```

int[] a = ArrayUtil.randomIntArray(n, 100);
int[] a2 = Arrays.copyOf(a, a.length);

StopWatch timer = new StopWatch();

timer.start();
ShellSorter.sort(a);
timer.stop();

System.out.println("Elapsed time with Shell sort: "
    + timer.getElapsedTime() + " milliseconds");

timer.reset();
timer.start();
Arrays.sort(a2);
timer.stop();

System.out.println("Elapsed time with Arrays.sort: "
    + timer.getElapsedTime() + " milliseconds");

if (!Arrays.equals(a, a2))
{
    throw new IllegalStateException("Incorrect sort result");
}

```

We make sure to sort the same array with both algorithms. Also, we check that the result of the Shell sort is correct by comparing it against the result of `Arrays.sort`.

Finally, we compare with the insertion sort algorithm.

The results show that Shell sort is a dramatic improvement over insertion sort:

```

Enter array size: 1000000
Elapsed time with Shell sort: 205 milliseconds

```

## WE4 Chapter 14 Sorting and Searching

```
Elapsed time with Arrays.sort: 101 milliseconds
Elapsed time with insertion sort: 148196 milliseconds
```

However, quicksort (which is used in `Arrays.sort`) outperforms Shell sort. For this reason, Shell sort is not used in practice, but it is still an interesting algorithm that is surprisingly effective.

You may also find it interesting to experiment with Shell's original column sizes. In the `sort` method, simply replace

```
c = 3 * c + 1;
```

with

```
c = 2 * c;
```

You will find that the algorithm is about three times slower than the improved sequence. That is still much faster than plain insertion sort.

You will find a program to demonstrate Shell sort and compare it to insertion sort in the `ch14/worked_example_1` folder of the book's companion code.

---

# THE JAVA COLLECTIONS FRAMEWORK

## CHAPTER GOALS

To learn how to use the collection classes supplied in the Java library

To use iterators to traverse collections

To choose appropriate collections for solving programming problems

To study applications of stacks and queues

## CHAPTER CONTENTS

### 15.1 AN OVERVIEW OF THE COLLECTIONS FRAMEWORK 678

### 15.2 LINKED LISTS 681

**C&S** Standardization 686

### 15.3 SETS 687

**PT1** Use Interface References to Manipulate Data Structures 691

### 15.4 MAPS 692

**J81** Updating Map Entries 694

**HT1** Choosing a Collection 694

**WE1** Word Frequency 

**ST1** Hash Functions 696



© nicholas belton/iStockphoto.

### 15.5 STACKS, QUEUES, AND PRIORITY QUEUES 698

### 15.6 STACK AND QUEUE APPLICATIONS 701

**WE2** Simulating a Queue of Waiting Customers 

**ST2** Reverse Polish Notation 709



© nicholas belton/iStockphoto.

If you want to write a program that collects objects (such as the stamps to the left), you have a number of choices. Of course, you can use an array list, but computer scientists have invented other mechanisms that may be better suited for the task. In this chapter, we introduce the collection classes and interfaces that the Java library offers. You will learn how to use the Java collection classes, and how to choose the most appropriate collection type for a problem.

## 15.1 An Overview of the Collections Framework

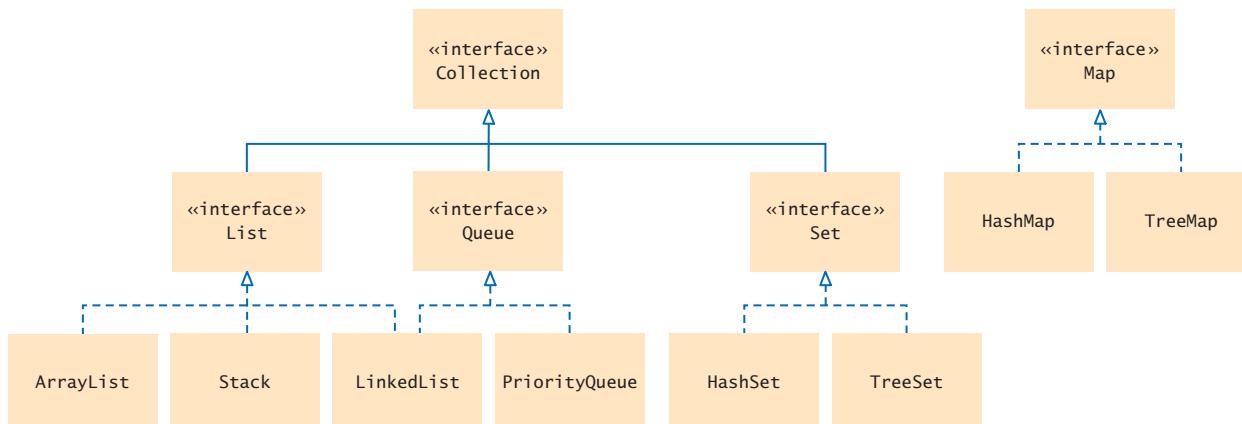
A collection groups together elements and allows them to be retrieved later.

When you need to organize multiple objects in your program, you can place them into a **collection**. The `ArrayList` class that was introduced in Chapter 7 is one of many collection classes that the standard Java library supplies. In this chapter, you will learn about the Java *collections framework*, a hierarchy of interface types and classes for collecting objects. Each interface type is implemented by one or more classes (see Figure 1).

At the root of the hierarchy is the `Collection` interface. That interface has methods for adding and removing elements, and so on. Table 1 on page 680 shows all the methods. Because all collections implement this interface, its methods are available for all collection classes. For example, the `size` method reports the number of elements in *any* collection.

The `List` interface describes an important category of collections. In Java, a *list* is a collection that remembers the order of its elements (see Figure 2). The `ArrayList` class implements the `List` interface. An `ArrayList` is simply a class containing an array that is expanded as needed. If you are not concerned about efficiency, you can use the `ArrayList` class whenever you need to collect objects. However, several common operations are inefficient with array lists. In particular, if an element is added or removed, the elements at larger positions must be moved.

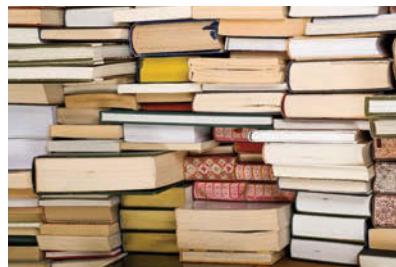
The Java library supplies another class, `LinkedList`, that also implements the `List` interface. Unlike an array list, a linked list allows efficient insertion and removal of elements in the middle of the list. We will discuss that class in the next section.



**Figure 1** Interfaces and Classes in the Java Collections Framework

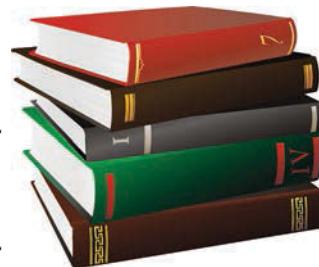


© Filip Fuxa/iStockphoto.

**Figure 2** A List of Books

© parema/iStockphoto.

A list is a collection that remembers the order of its elements.

**Figure 3** A Set of Books

© Vladimir Trenin/iStockphoto.

**Figure 4** A Stack of Books

You use a list whenever you want to retain the order that you established. For example, on your bookshelf, you may order books by topic. A list is an appropriate data structure for such a collection because the ordering matters to you.

However, in many applications, you don't really care about the order of the elements in a collection. Consider a mail-order dealer of books. Without customers browsing the shelves, there is no need to order books by topic. Such a collection without an intrinsic order is called a **set**—see Figure 3.

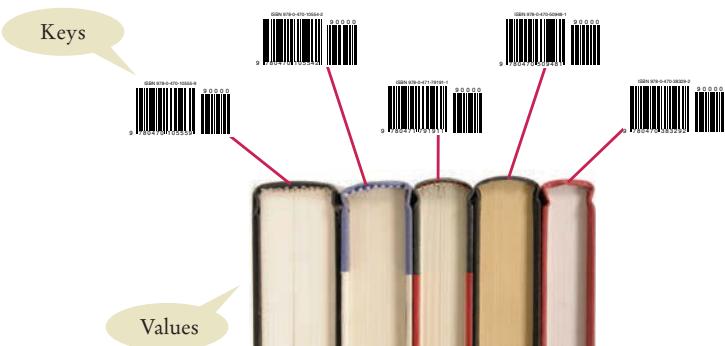
Because a set does not track the order of the elements, it can arrange the elements so that the operations of finding, adding, and removing elements become more efficient. Computer scientists have invented mechanisms for this purpose. The Java library provides classes that are based on two such mechanisms (called *hash tables* and *binary search trees*). You will learn in this chapter how to choose between them.

Another way of gaining efficiency in a collection is to reduce the number of operations. A **stack** remembers the order of its elements, but it does not allow you to insert elements in every position. You can add and remove elements only at the top—see Figure 4.

In a **queue**, you add items to one end (the tail) and remove them from the other end (the head). For example, you could keep a queue of books, adding required reading at the tail and taking a book from the head whenever you have time to read another one. A **priority queue** is an unordered collection that has an efficient operation for removing the element with the highest priority. You might use a priority queue for organizing your reading assignments. Whenever you have some time, remove the book with the highest priority and read it. We will discuss stacks, queues, and priority queues in Section 15.5.

Finally, a **map** manages associations between *keys* and *values*. Every key in the map has an associated value (see Figure 5). The map stores the keys, values, and the associations between them.

A map keeps associations between key and value objects.



**Figure 5**  
A Map from Bar Codes to Books

(books) © david franklin/iStockphoto.

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bjeo6code](http://www.wiley.com/go/bjeo6code) to download a sample program that demonstrates several collection classes.

For an example, consider a library that puts a bar code on each book. The program used to check books in and out needs to look up the book associated with each bar code. A map associating bar codes with books can solve this problem. We will discuss maps in Section 15.4.

Starting with this chapter, we will use the “diamond syntax” for constructing instances of generic classes (see Special Topic 7.5). For example, when constructing an array list of strings, we will use

```
ArrayList<String> coll = new ArrayList<>();
```

Note that there is an empty pair of brackets `<>` after `new ArrayList` on the right-hand side. The compiler infers from the left-hand side that an array list of strings is constructed.

**Table 1** The Methods of the Collection Interface

<code>Collection&lt;String&gt; coll = new ArrayList&lt;&gt;();</code>	The <code>ArrayList</code> class implements the <code>Collection</code> interface.
<code>coll = new TreeSet&lt;&gt;();</code>	The <code>TreeSet</code> class (Section 15.3) also implements the <code>Collection</code> interface.
<code>int n = coll.size();</code>	Gets the size of the collection. <code>n</code> is now 0.
<code>coll.add("Harry"); coll.add("Sally");</code>	Adds elements to the collection.
<code>String s = coll.toString();</code>	Returns a string with all elements in the collection. <code>s</code> is now [Harry, Sally].
<code>System.out.println(coll);</code>	Invokes the <code>toString</code> method and prints [Harry, Sally].
<code>coll.remove("Harry"); boolean b = coll.remove("Tom");</code>	Removes an element from the collection, returning <code>false</code> if the element is not present. <code>b</code> is false.
<code>b = coll.contains("Sally");</code>	Checks whether this collection contains a given element. <code>b</code> is now true.
<code>for (String s : coll) {     System.out.println(s); }</code>	You can use the “for each” loop with any collection. This loop prints the elements on separate lines.
<code>Iterator&lt;String&gt; iter = coll.iterator();</code>	You use an iterator for visiting the elements in the collection (see Section 15.2.3).

**SELF CHECK**

1. A grade book application stores a collection of quizzes. Should it use a list or a set?
2. A student information system stores a collection of student records for a university. Should it use a list or a set?
3. Why is a queue of books a better choice than a stack for organizing your required reading?
4. As you can see from Figure 1, the Java collections framework does not consider a map a collection. Give a reason for this decision.

**Practice It** Now you can try these exercises at the end of the chapter: R15.1, R15.2, R15.3.

## 15.2 Linked Lists

A **linked list** is a data structure used for collecting a sequence of objects that allows efficient addition and removal of elements in the middle of the sequence. In the following sections, you will learn how a linked list manages its elements and how you can use linked lists in your programs.

### 15.2.1 The Structure of Linked Lists

To understand the inefficiency of arrays and the need for a more efficient data structure, imagine a program that maintains a sequence of employee names. If an employee leaves the company, the name must be removed. In an array, the hole in the sequence needs to be closed up by moving all objects that come after it. Conversely, suppose an employee is added in the middle of the sequence. Then all names following the new hire must be moved toward the end. Moving a large number of elements can involve a substantial amount of processing time. A linked list structure avoids this movement.

A linked list consists of a number of nodes, each of which has a reference to the next node.

A linked list uses a sequence of *nodes*. A node is an object that stores an element and references to the neighboring nodes (see Figure 6).



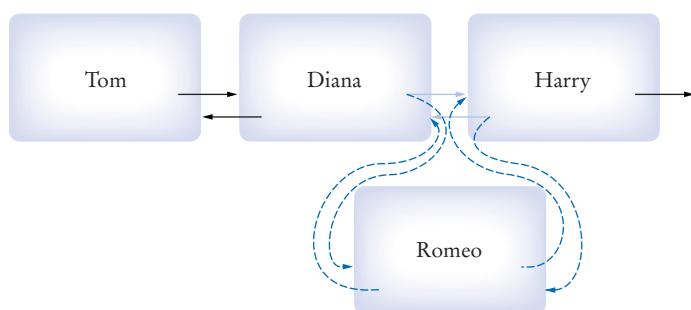
© andrea laurita/Stockphoto.

*Each node in a linked list is connected to the neighboring nodes.*



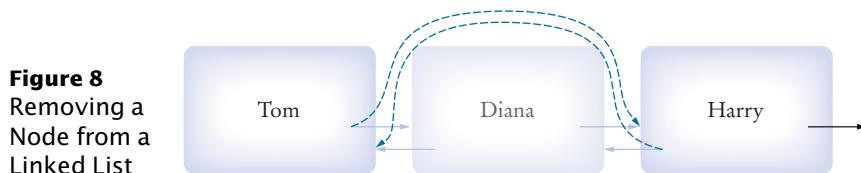
**Figure 6**  
A Linked List

When you insert a new node into a linked list, only the neighboring node references need to be updated (see Figure 7).



**Figure 7** Inserting a Node into a Linked List

The same is true when you remove a node (see Figure 8). What's the catch? Linked lists allow efficient insertion and removal, but element access can be inefficient.



Adding and removing elements at a given location in a linked list is efficient.

Visiting the elements of a linked list in sequential order is efficient, but random access is not.

For example, suppose you want to locate the fifth element. You must first traverse the first four. This is a problem if you need to access the elements in arbitrary order. The term “random access” is used in computer science to describe an access pattern in which elements are accessed in arbitrary (not necessarily random) order. In contrast, sequential access visits the elements in sequence.

Of course, if you mostly visit all elements in sequence (for example, to display or print the elements), the inefficiency of random access is not a problem. You use linked lists when you are concerned about the efficiency of inserting or removing elements and you rarely need element access in random order.

### 15.2.2 The LinkedList Class of the Java Collections Framework

The Java library provides a `LinkedList` class in the `java.util` package. It is a **generic class**, just like the `ArrayList` class. That is, you specify the type of the list elements in angle brackets, such as `LinkedList<String>` or `LinkedList<Employee>`.

Table 2 shows important methods of the `LinkedList` class. (Remember that the `LinkedList` class also inherits the methods of the `Collection` interface shown in Table 1.)

As you can see from Table 2, there are methods for accessing the beginning and the end of the list directly. However, to visit the other elements, you need a list **iterator**. We discuss iterators next.

**Table 2** Working with Linked Lists

<code>LinkedList&lt;String&gt; list = new LinkedList&lt;&gt;();</code>	An empty list.
<code>list.addLast("Harry");</code>	Adds an element to the end of the list. Same as <code>add</code> .
<code>list.addFirst("Sally");</code>	Adds an element to the beginning of the list. <code>list</code> is now [Sally, Harry].
<code>list.getFirst();</code>	Gets the element stored at the beginning of the list; here "Sally".
<code>list.getLast();</code>	Gets the element stored at the end of the list; here "Harry".
<code>String removed = list.removeFirst();</code>	Removes the first element of the list and returns it. <code>removed</code> is "Sally" and <code>list</code> is [Harry]. Use <code>removeLast</code> to remove the last element.
<code>ListIterator&lt;String&gt; iter = list.listIterator()</code>	Provides an iterator for visiting all list elements (see Table 3 on page 684).

### 15.2.3 List Iterators

You use a list iterator to access elements inside a linked list.

An iterator encapsulates a position anywhere inside the linked list. Conceptually, you should think of the iterator as pointing between two elements, just as the cursor in a word processor points between two characters (see Figure 9). In the conceptual view, think of each element as being like a letter in a word processor, and think of the iterator as being like the blinking cursor between letters.

You obtain a list iterator with the `listIterator` method of the `LinkedList` class:

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iterator = employeeNames.listIterator();
```

Note that the iterator class is also a generic type. A `ListIterator<String>` iterates through a list of strings; a `ListIterator<Book>` visits the elements in a `LinkedList<Book>`.

Initially, the iterator points before the first element. You can move the iterator position with the `next` method:

```
iterator.next();
```

The `next` method throws a `NoSuchElementException` if you are already past the end of the list. You should always call the iterator's `hasNext` method before calling `next`—it returns `true` if there is a next element.

```
if (iterator.hasNext())
{
    iterator.next();
}
```

The `next` method returns the element that the iterator is passing. When you use a `ListIterator<String>`, the return type of the `next` method is `String`. In general, the return type of the `next` method matches the list iterator's type parameter (which reflects the type of the elements in the list).

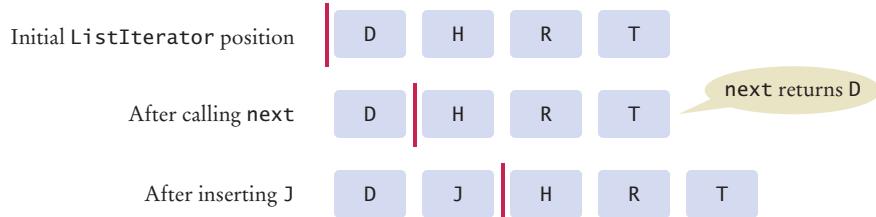
You traverse all elements in a linked list of strings with the following loop:

```
while (iterator.hasNext())
{
    String name = iterator.next();
    Do something with name.
}
```

As a shorthand, if your loop simply visits all elements of the linked list, you can use the “for each” loop:

```
for (String name : employeeNames)
{
    Do something with name.
}
```

Then you don't have to worry about iterators at all. Behind the scenes, the `for` loop uses an iterator to visit all list elements.



**Figure 9**  
A Conceptual View  
of the List Iterator

**Table 3** Methods of the Iterator and ListIterator Interfaces

<code>String s = iter.next();</code>	Assume that <code>iter</code> points to the beginning of the list [Sally] before calling <code>next</code> . After the call, <code>s</code> is "Sally" and the iterator points to the end.
<code>iter.previous();</code> <code>iter.set("Juliet");</code>	The <code>set</code> method updates the last element returned by <code>next</code> or <code>previous</code> . The list is now [Juliet].
<code>iter.hasNext()</code>	Returns <code>false</code> because the iterator is at the end of the collection.
<code>if (iter.hasPrevious())</code> { <code>s = iter.previous();</code> }	<code>hasPrevious</code> returns <code>true</code> because the iterator is not at the beginning of the list. <code>previous</code> and <code>hasPrevious</code> are <code>ListIterator</code> methods.
<code>iter.add("Diana");</code>	Adds an element before the iterator position ( <code>ListIterator</code> only). The list is now [Diana, Juliet].
<code>iter.next();</code> <code>iter.remove();</code>	<code>remove</code> removes the last element returned by <code>next</code> or <code>previous</code> . The list is now [Diana].

The nodes of the `LinkedList` class store two links: one to the next element and one to the previous one. Such a list is called a **doubly-linked list**. You can use the `previous` and `hasPrevious` methods of the `ListIterator` interface to move the iterator position backward.

The `add` method adds an object after the iterator, then moves the iterator position past the new element.

```
iterator.add("Juliet");
```

You can visualize insertion to be like typing text in a word processor. Each character is inserted after the cursor, then the cursor moves past the inserted character (see Figure 9). Most people never pay much attention to this—you may want to try it out and watch carefully how your word processor inserts characters.

The `remove` method removes the object that was returned by the last call to `next` or `previous`. For example, this loop removes all names that fulfill a certain condition:

```
while (iterator.hasNext())
{
    String name = iterator.next();
    if (condition is fulfilled for name)
    {
        iterator.remove();
    }
}
```

You have to be careful when calling `remove`. It can be called only *once* after calling `next` or `previous`, and you cannot call it immediately after a call to `add`. If you call the method improperly, it throws an `IllegalStateException`.

Table 3 summarizes the methods of the `ListIterator` interface. The `ListIterator` interface extends a more general `Iterator` interface that is suitable for arbitrary collections, not just lists. The table indicates which methods are specific to list iterators.

Following is a sample program that inserts strings into a list and then iterates through the list, adding and removing elements. Finally, the entire list is printed. The comments indicate the iterator position.

**section\_2/ListDemo.java**

```

1 import java.util.LinkedList;
2 import java.util.ListIterator;
3
4 /**
5  * This program demonstrates the LinkedList class.
6 */
7 public class ListDemo
8 {
9     public static void main(String[] args)
10    {
11        LinkedList<String> staff = new LinkedList<>();
12        staff.addLast("Diana");
13        staff.addLast("Harry");
14        staff.addLast("Romeo");
15        staff.addLast("Tom");
16
17        // | in the comments indicates the iterator position
18
19        ListIterator<String> iterator = staff.listIterator(); // |DHRT
20        iterator.next(); // D|HRT
21        iterator.next(); // DH|RT
22
23        // Add more elements after second element
24
25        iterator.add("Juliet"); // DHJ|RT
26        iterator.add("Nina"); // DHJN|RT
27
28        iterator.next(); // DHJNR|T
29
30        // Remove last traversed element
31
32        iterator.remove(); // DHJN|T
33
34        // Print all elements
35
36        System.out.println(staff);
37        System.out.println("Expected: [Diana, Harry, Juliet, Nina, Tom]");
38    }
39 }
```

**Program Run**

```
[Diana, Harry, Juliet, Nina, Tom]
Expected: [Diana, Harry, Juliet, Nina, Tom]
```



5. Do linked lists take more storage space than arrays of the same size?
6. Why don't we need iterators with arrays?
7. Suppose the list `letters` contains elements "A", "B", "C", and "D". Draw the contents of the list and the iterator position for the following operations:

```
ListIterator<String> iter = letters.iterator();
iter.next();
iter.next();
iter.remove();
iter.next();
iter.add("E");
```

```

iter.next();
iter.add("F");

```

8. Write a loop that removes all strings with length less than four from a linked list of strings called words.
9. Write a loop that prints every second element of a linked list of strings called words.

**Practice It** Now you can try these exercises at the end of the chapter: R15.5, R15.8, E15.1.



## Computing & Society 15.1 Standardization

You encounter the benefits of standardization every day. When you buy a light bulb, you can be assured that it fits the socket without having to measure the socket at home and the light bulb in the store. In fact, you may have experienced how painful the lack of standards can be if you have ever purchased a flashlight with nonstandard bulbs. Replacement bulbs for such a flashlight can be difficult and expensive to obtain.

Programmers have a similar desire for standardization. Consider the important goal of platform independence for Java programs. After you compile a Java program into class files, you can execute the class files on any computer that has a Java virtual machine. For this to work, the behavior of the virtual machine has to be strictly defined. If all virtual machines don't behave exactly the same way, then the slogan of "write once, run anywhere" turns into "write once, debug everywhere". In order for multiple implementors to create compatible virtual machines, the virtual machine needed to be *standardized*. That is, someone needed to create a definition of the virtual machine and its expected behavior.

Who creates standards? Some of the most successful standards have been created by volunteer groups such as the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C). The IETF standardizes protocols used in the Internet, such

as the protocol for exchanging e-mail messages. The W3C standardizes the Hypertext Markup Language (HTML), the format for web pages. These standards have been instrumental in the creation of the World Wide Web as an open platform that is not controlled by any one company.

Many programming languages, such as C++ and Scheme, have been standardized by independent standards organizations, such as the American National Standards Institute (ANSI) and the International Organization for Standardization—called ISO for short (not an acronym; see [http://www.iso.org/iso/about/discover-iso\\_isos-name.htm](http://www.iso.org/iso/about/discover-iso_isos-name.htm)). ANSI and ISO are associations of industry professionals who develop standards for everything from car tires to credit card shapes to programming languages.

Many standards are developed by dedicated experts from a multitude of vendors and users, with the objective of creating a set of rules that codifies best practices. But sometimes, standards are very contentious. By 2005, Microsoft started losing government contracts when its customers became concerned that many of their documents were stored in proprietary, undocumented formats. Instead of supporting existing standard formats, or working with an industry group to improve those standards, Microsoft wrote its own standard that simply codified what its product was currently doing, even though that format is widely regarded as being inconsistent and very complex. (The description of the format spans over 6,000 pages.) The company first proposed its standard to the European Computer

Manufacturers Association (ECMA), which approved it with minimal discussion. Then ISO "fast-tracked" it as an existing standard, bypassing the normal technical review mechanism.

For similar reasons, Sun Microsystems, the inventor of Java, never agreed to have a third-party organization standardize the Java language. Instead, they put in place their own standardization process, involving other companies but refusing to relinquish control.

Of course, many important pieces of technology aren't standardized at all. Consider the Windows operating system. Although Windows is often called a de-facto standard, it really is no standard at all. Nobody has ever attempted to define formally what the Windows operating system should do. The behavior changes at the whim of its vendor. That suits Microsoft just fine, because it makes it impossible for a third party to create its own version of Windows.

As a computer professional, there will be many times in your career when you need to make a decision whether to support a particular standard. Consider a simple example. In this chapter, you learn about the collection classes from the standard Java library. However, many computer scientists dislike these classes because of their numerous design issues. Should you use the Java collections in your own code, or should you implement a better set of collections? If you do the former, you have to deal with a design that is less than optimal. If you do the latter, other programmers may have a hard time understanding your code because they aren't familiar with your classes.



© Denis Vorob'yev/iStockphoto.

## 15.3 Sets

As you learned in Section 15.1, a **set** organizes its values in an order that is optimized for efficiency, which may not be the order in which you add elements. Inserting and removing elements is more efficient with a set than with a list.

In the following sections, you will learn how to choose a set implementation and how to work with sets.

### 15.3.1 Choosing a Set Implementation

The HashSet and TreeSet classes both implement the Set interface.

Set implementations arrange the elements so that they can locate them quickly.

You can form hash sets holding objects of type String, Integer, Double, Point, Rectangle, or Color.

The Set interface in the standard Java library has the same methods as the Collection interface, shown in Table 1. However, there is an essential difference between arbitrary collections and sets. A set does not admit duplicates. If you add an element to a set that is already present, the insertion is ignored.

The HashSet and TreeSet classes implement the Set interface. These two classes provide set implementations based on two different mechanisms, called **hash tables** and **binary search trees**. Both implementations arrange the set elements so that finding, adding, and removing elements is efficient, but they use different strategies.

The basic idea of a hash table is simple. Set elements are grouped into smaller collections of elements that share the same characteristic. You can imagine a hash set of books as having a group for each color, so that books of the same color are in the same group. To find whether a book is already present, you just need to check it against the books in the same color group. Actually, hash tables don't use colors, but integer values (called hash codes) that can be computed from the elements.

In order to use a hash table, the elements must have a method to compute those integer values. This method is called hashCode. The elements must also belong to a class with a properly defined equals method (see Section 9.5.2).

Many classes in the standard library implement these methods, for example String, Integer, Double, Point, Rectangle, Color, and all the collection classes. Therefore, you can form a HashSet<String>, HashSet<Rectangle>, or even a HashSet<HashSet<Integer>>.

Suppose you want to form a set of elements belonging to a class that you declared, such as a HashSet<Book>. Then you need to provide hashCode and equals methods for the class Book. There is one exception to this rule. If all elements are distinct (for example, if your program never has two Book objects with the same author and title), then you can simply inherit the hashCode and equals methods of the Object class.

*On this shelf, books of the same color are grouped together. Similarly, in a hash table, objects with the same hash code are placed in the same group.*



*A tree set keeps its elements in sorted order.*



© Volkan Ersoy/iStockphoto.

The `TreeSet` class uses a different strategy for arranging its elements. Elements are kept in sorted order. For example, a set of books might be arranged by height, or alphabetically by author and title. The elements are not stored in an array—that would make adding and removing elements too inefficient. Instead, they are stored in nodes, as in a linked list. However, the nodes are not arranged in a linear sequence but in a tree shape.

In order to use a `TreeSet`, it must be possible to compare the elements and determine which one is “larger”. You can use a `TreeSet` for classes such as `String` and `Integer` that implement the `Comparable` interface, which we discussed in Section 10.3. (That section also shows you how you can implement comparison methods for your own classes.)

As a rule of thumb, you should choose a `TreeSet` if you want to visit the set’s elements in sorted order. Otherwise choose a `HashSet`—as long as the hash function is well chosen, it is a bit more efficient.

When you construct a `HashSet` or `TreeSet`, store the reference in a `Set` variable. For example,

```
Set<String> names = new HashSet<>();
or
Set<String> names = new TreeSet<>();
```

After you construct the collection object, the implementation no longer matters; only the interface is important.

### 15.3.2 Working with Sets

You add and remove set elements with the `add` and `remove` methods:

```
names.add("Romeo");
names.remove("Juliet");
```

As in mathematics, a set collection in Java rejects duplicates. Adding an element has no effect if the element is already in the set. Similarly, attempting to remove an element that isn’t in the set is ignored.

The `contains` method tests whether an element is contained in the set:

```
if (names.contains("Juliet")) . . .
```

The `contains` method uses the `equals` method of the element type. If your set collects `String` or `Integer` objects, you don’t have to worry. Those classes provide an `equals` method. However, if you implemented the element type yourself, then you need to define the `equals` method—see Section 9.5.2.

Finally, to list all elements in the set, get an iterator. As with list iterators, you use the `next` and `hasNext` methods to step through the set.

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
```

You can form tree sets for any class that implements the `Comparable` interface, such as `String` or `Integer`.

Sets don’t have duplicates. Adding a duplicate of an element that is already present is ignored.

```

String name = iter.next();
Do something with name.
}

```

You can also use the “for each” loop instead of explicitly using an iterator:

```

for (String name : names)
{
    Do something with name.
}

```

A set iterator visits the elements in the order in which the set implementation keeps them.

You cannot add an element to a set at an iterator position.

A set iterator visits the elements in the order in which the set implementation keeps them. This is not necessarily the order in which you inserted them. The order of elements in a hash set seems quite random because the hash code spreads the elements into different groups. When you visit elements of a tree set, they always appear in sorted order, even if you inserted them in a different order.

There is an important difference between the `Iterator` that you obtain from a set and the `ListIterator` that a list yields. The `ListIterator` has an `add` method to add an element at the list iterator position. The `Iterator` interface has no such method. It makes no sense to add an element at a particular position in a set, because the set can order the elements any way it likes. Thus, you always add elements directly to a set, never to an iterator of the set.

However, you can remove a set element at an iterator position, just as you do with list iterators.

Also, the `Iterator` interface has no previous method to go backward through the elements. Because the elements are not ordered, it is not meaningful to distinguish between “going forward” and “going backward”.

**Table 4 Working with Sets**

<code>Set&lt;String&gt; names;</code>	Use the interface type for variable declarations.
<code>names = new HashSet&lt;&gt;();</code>	Use a <code>TreeSet</code> if you need to visit the elements in sorted order.
<code>names.add("Romeo");</code>	Now <code>names.size()</code> is 1.
<code>names.add("Fred");</code>	Now <code>names.size()</code> is 2.
<code>names.add("Romeo");</code>	<code>names.size()</code> is still 2. You can't add duplicates.
<code>if (names.contains("Fred"))</code>	The <code>contains</code> method checks whether a value is contained in the set. In this case, the method returns <code>true</code> .
<code>System.out.println(names);</code>	Prints the set in the format <code>[Fred, Romeo]</code> . The elements need not be shown in the order in which they were inserted.
<code>for (String name : names)</code> { . . . }	Use this loop to visit all elements of a set.
<code>names.remove("Romeo");</code>	Now <code>names.size()</code> is 1.
<code>names.remove("Juliet");</code>	It is not an error to remove an element that is not present. The method call has no effect.

The following program shows a practical application of sets. It reads in all words from a dictionary file that contains correctly spelled words and places them in a set. It then reads all words from a document—here, the book *Alice in Wonderland*—into a second set. Finally, it prints all words from that set that are not in the dictionary set. These are the potential misspellings. (As you can see from the output, we used an American dictionary, and words with British spelling, such as *clamour*, are flagged as potential errors.)

### section\_3/SpellCheck.java

```

1 import java.util.HashSet;
2 import java.util.Scanner;
3 import java.util.Set;
4 import java.io.File;
5 import java.io.FileNotFoundException;
6
7 /**
8 * This program checks which words in a file are not present in a dictionary.
9 */
10 public class SpellCheck
11 {
12     public static void main(String[] args)
13         throws FileNotFoundException
14     {
15         // Read the dictionary and the document
16
17         Set<String> dictionaryWords = readWords("words");
18         Set<String> documentWords = readWords("alice30.txt");
19
20         // Print all words that are in the document but not the dictionary
21
22         for (String word : documentWords)
23         {
24             if (!dictionaryWords.contains(word))
25             {
26                 System.out.println(word);
27             }
28         }
29     }
30
31 /**
32 * Reads all words from a file.
33 * @param filename the name of the file
34 * @return a set with all lowercased words in the file. Here, a
35 * word is a sequence of upper- and lowercase letters.
36 */
37 public static Set<String> readWords(String filename)
38     throws FileNotFoundException
39 {
40     Set<String> words = new HashSet<>();
41     Scanner in = new Scanner(new File(filename));
42     // Use any characters other than a-z or A-Z as delimiters
43     in.useDelimiter("[^a-zA-Z]+");
44     while (in.hasNext())
45     {
46         words.add(in.next().toLowerCase());
47     }
48     return words;

```

```
49     }
50 }
```

### Program Run

```
neighbouring
croqueted
pennyworth
dutchess
comfits
xii
dinn
clamour
...
```

### SELF CHECK



10. Arrays and lists remember the order in which you added elements; sets do not. Why would you want to use a set instead of an array or list?
11. Why are set iterators different from list iterators?
12. What is wrong with the following test to check whether the `Set<String>` s contains the elements "Tom", "Diana", and "Harry"?
 

```
if (s.toString().equals("[Tom, Diana, Harry]")) . . .
```
13. How can you correctly implement the test of Self Check 12?
14. Write a loop that prints all elements that are in both `Set<String>` s and `Set<String>` t.
15. Suppose you changed line 40 of the SpellCheck program to use a `TreeSet` instead of a `HashSet`. How would the output change?

### Practice It

Now you can try these exercises at the end of the chapter: E15.3, E15.12, E15.13.

### Programming Tip 15.1



### Use Interface References to Manipulate Data Structures

It is considered good style to store a reference to a `HashSet` or `TreeSet` in a variable of type `Set`:

```
Set<String> words = new HashSet<>();
```

This way, you have to change only one line if you decide to use a `TreeSet` instead.

If a method can operate on arbitrary collections, use the `Collection` interface type for the parameter variable:

```
public static void removeLongWords(Collection<String> words)
```

In theory, we should make the same recommendation for the `List` interface, namely to save `ArrayList` and `LinkedList` references in variables of type `List`. However, the `List` interface has get and set methods for random access, even though these methods are very inefficient for linked lists. You can't write efficient code if you don't know whether the methods that you are calling are efficient or not. This is plainly a serious design error in the standard library, and it makes the `List` interface somewhat unattractive.

## 15.4 Maps

The `HashMap` and `TreeMap` classes both implement the `Map` interface.

A **map** allows you to associate elements from a *key set* with elements from a *value collection*. You use a map when you want to look up objects by using a key. For example, Figure 10 shows a map from the names of people to their favorite colors.

Just as there are two kinds of set implementations, the Java library has two implementations for the `Map` interface: `HashMap` and `TreeMap`.

After constructing a `HashMap` or `TreeMap`, you can store the reference to the map object in a `Map` reference:

```
Map<String, Color> favoriteColors = new HashMap<>();
```

Use the `put` method to add an association:

```
favoriteColors.put("Juliet", Color.RED);
```

You can change the value of an existing association, simply by calling `put` again:

```
favoriteColors.put("Juliet", Color.BLUE);
```

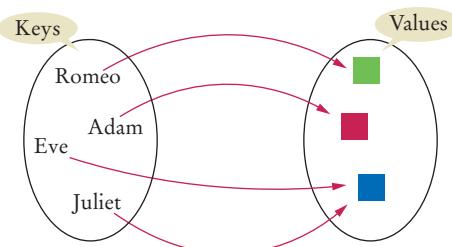
The `get` method returns the value associated with a key.

```
Color juliet'sFavoriteColor = favoriteColors.get("Juliet");
```

If you ask for a key that isn't associated with any values, the `get` method returns `null`.

To remove an association, call the `remove` method with the key:

```
favoriteColors.remove("Juliet");
```



**Figure 10** A Map

**Table 5 Working with Maps**

<code>Map&lt;String, Integer&gt; scores;</code>	Keys are strings, values are <code>Integer</code> wrappers. Use the interface type for variable declarations.
<code>scores = new TreeMap&lt;&gt;();</code>	Use a <code>HashMap</code> if you don't need to visit the keys in sorted order.
<code>scores.put("Harry", 90); scores.put("Sally", 95);</code>	Adds keys and values to the map.
<code>scores.put("Sally", 100);</code>	Modifies the value of an existing key.
<code>int n = scores.get("Sally"); Integer n2 = scores.get("Diana");</code>	Gets the value associated with a key, or <code>null</code> if the key is not present. <code>n</code> is 100, <code>n2</code> is <code>null</code> .
<code>System.out.println(scores);</code>	Prints <code>scores.toString()</code> , a string of the form {Harry=90, Sally=100}
<code>for (String key : scores.keySet()) {     Integer value = scores.get(key);     . . . }</code>	Iterates through all map keys and values.
<code>scores.remove("Sally");</code>	Removes the key and value.

To find all keys and values in a map, iterate through the key set and find the values that correspond to the keys.

Sometimes you want to enumerate all keys in a map. The `keySet` method yields the set of keys. You can then ask the key set for an iterator and get all keys. From each key, you can find the associated value with the `get` method. Thus, the following instructions print all key/value pairs in a map `m`:

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
    Color value = m.get(key);
    System.out.println(key + " -> " + value);
}
```

This sample program shows a map in action:

### section\_4/MapDemo.java

```
1 import java.awt.Color;
2 import java.util.HashMap;
3 import java.util.Map;
4 import java.util.Set;
5
6 /**
7     This program demonstrates a map that maps names to colors.
8 */
9 public class MapDemo
10 {
11     public static void main(String[] args)
12     {
13         Map<String, Color> favoriteColors = new HashMap<>();
14         favoriteColors.put("Juliet", Color.BLUE);
15         favoriteColors.put("Romeo", Color.GREEN);
16         favoriteColors.put("Adam", Color.RED);
17         favoriteColors.put("Eve", Color.BLUE);
18
19         // Print all keys and values in the map
20
21         Set<String> keySet = favoriteColors.keySet();
22         for (String key : keySet)
23         {
24             Color value = favoriteColors.get(key);
25             System.out.println(key + " : " + value);
26         }
27     }
28 }
```

### Program Run

```
Juliet : java.awt.Color[r=0,g=0,b=255]
Adam : java.awt.Color[r=255,g=0,b=0]
Eve : java.awt.Color[r=0,g=0,b=255]
Romeo : java.awt.Color[r=0,g=255,b=0]
```

### SELF CHECK



16. What is the difference between a set and a map?
17. Why is the collection of the keys of a map a set and not a list?
18. Why is the collection of the values of a map not a set?
19. Suppose you want to track how many times each word occurs in a document. Declare a suitable map variable.

**20.** What is a `Map<String, HashSet<String>>`? Give a possible use for such a structure.

**Practice It** Now you can try these exercises at the end of the chapter: R15.20, E15.4, E15.5.

### Java 8 Note 15.1

8

#### Updating Map Entries

Maps are commonly used for counting how often an item occurs. For example, Worked Example 15.1 uses a `Map<String, Integer>` to track how many times a word occurs in a file.

It is a bit tedious to deal with the special case of inserting the first value. Consider the following code from Worked Example 15.1:

```
Integer count = frequencies.get(word); // Get the old frequency count
// If there was none, put 1; otherwise, increment the count
if (count == null) { count = 1; }
else { count = count + 1; }
frequencies.put(word, count);
```

Java 8 adds a useful `merge` method to the `Map` interface. You specify

- A key.
- A value to be used if the key is not yet present.
- A function to compute the updated value if the key is present.

The function is specified as a lambda expression (see Java 8 Note 10.4). For example,

```
frequencies.merge(word, 1, (oldValue, value) -> oldValue + value);
```

does the same as the four lines of code above. If `word` is not present, the value is set to 1. Otherwise, the old value is incremented.

The `merge` method is also useful if the map values are sets or comma-separated strings—see Exercises E15.6 and E15.7.

### HOW TO 15.1



#### Choosing a Collection

Suppose you need to store objects in a collection. You have now seen a number of different data structures. This How To reviews how to pick an appropriate collection for your application.



© Tom Hahn/  
iStockphoto.

##### Step 1 Determine how you access the values.

You store values in a collection so that you can later retrieve them. How do you want to access individual values? You have several choices:

- Values are accessed by an integer position. Use an `ArrayList`.
- Values are accessed by a key that is not a part of the object. Use a map.
- Values are accessed only at one of the ends. Use a queue (for first-in, first-out access) or a stack (for last-in, first-out access).
- You don't need to access individual values by position. Refine your choice in Steps 3 and 4.

##### Step 2 Determine the element types or key/value types.

For a list or set, determine the type of the elements that you want to store. For example, if you collect a set of books, then the element type is `Book`.

Similarly, for a map, determine the types of the keys and the associated values. If you want to look up books by ID, you can use a `Map<Integer, Book>` or `Map<String, Book>`, depending on your ID type.

### Step 3

Determine whether element or key order matters.

When you visit elements from a collection or keys from a map, do you care about the order in which they are visited? You have several choices:

- Elements or keys must be sorted. Use a `TreeSet` or `TreeMap`. Go to Step 6.
- Elements must be in the same order in which they were inserted. Your choice is now narrowed down to a `LinkedList` or an `ArrayList`.
- It doesn't matter. As long as you get to visit all elements, you don't care in which order. If you chose a map in Step 1, use a `HashMap` and go to Step 5.

### Step 4

For a collection, determine which operations must be efficient.

You have several choices:

- Finding elements must be efficient. Use a `HashSet`.
- It must be efficient to add or remove elements at the beginning, or, provided that you are already inspecting an element there, another position. Use a `LinkedList`.
- You only insert or remove at the end, or you collect so few elements that you aren't concerned about speed. Use an `ArrayList`.

### Step 5

For hash sets and maps, decide whether you need to implement the `hashCode` and `equals` methods.

- If your elements or keys belong to a class that someone else implemented, check whether the class has its own `hashCode` and `equals` methods. If so, you are all set. This is the case for most classes in the standard Java library, such as `String`, `Integer`, `Rectangle`, and so on.
- If not, decide whether you can compare the elements by identity. This is the case if you never construct two distinct elements with the same contents. In that case, you need not do anything—the `hashCode` and `equals` methods of the `Object` class are appropriate.
- Otherwise, you need to implement your own `equals` and `hashCode` methods—see Section 9.5.2 and Special Topic 15.1.

### Step 6

If you use a tree, decide whether to supply a comparator.

Look at the class of the set elements or map keys. Does that class implement the `Comparable` interface? If so, is the sort order given by the `compareTo` method the one you want? If yes, then you don't need to do anything further. This is the case for many classes in the standard library, in particular for `String` and `Integer`.

If not, then your element class must implement the `Comparable` interface (Section 10.3), or you must declare a class that implements the `Comparator` interface (see Special Topic 14.4).



## WORKED EXAMPLE 15.1

### Word Frequency



Learn how to create a program that reads a text file and prints a list of all words in the file in alphabetical order, together with a count that indicates how often each word occurred in the file. Go to [wiley.com/go/bje06examples](http://wiley.com/go/bje06examples) and download Worked Example 15.1.



© Ermin Guttenberger/  
iStockphoto

## Special Topic 15.1

**Hash Functions**

If you use a hash set or hash map with your own classes, you may need to implement a hash function. A **hash function** is a function that computes an integer value, the **hash code**, from an object in such a way that different objects are likely to yield different hash codes. Because hashing is so important, the `Object` class has a `hashCode` method. The call

```
int h = x.hashCode();
```

computes the hash code of any object `x`. If you want to put objects of a given class into a `HashSet` or use the objects as keys in a `HashMap`, the class should override this method. The method should be implemented so that different objects are likely to have different hash codes.

For example, the `String` class declares a hash function for strings that does a good job of producing different integer values for different strings. Table 6 shows some examples of strings and their hash codes.

It is possible for two or more distinct objects to have the same hash code; this is called a *collision*. For example, the strings "Ugh" and "VII" happen to have the same hash code, but these collisions are very rare for strings (see Exercise P15.5).

The `hashCode` method of the `String` class combines the characters of a string into a numerical code. The code isn't simply the sum of the character values—that would not scramble the character values enough. Strings that are permutations of another (such as "eat" and "tea") would all have the same hash code.

Here is the method the standard library uses to compute the hash code for a string:

```
final int HASH_MULTIPLIER = 31;
int h = 0;
for (int i = 0; i < s.length(); i++)
{
    h = HASH_MULTIPLIER * h + s.charAt(i);
}
```

For example, the hash code of "eat" is

$$31 * (31 * 'e' + 'a') + 't' = 100184$$



© one clear vision/iStockphoto.

*A good hash function produces different hash values for each object so that they are scattered about in a hash table.*

A hash function computes an integer value from an object.

A good hash function minimizes *collisions*—identical hash codes for different objects.

**Table 6** Sample Strings and Their Hash Codes

String	Hash Code
"eat"	100184
"tea"	114704
"Juliet"	-2065036585
"Ugh"	84982
"VII"	84982

The hash code of "tea" is quite different, namely

$$31 * (31 * 't' + 'e') + 'a' = 114704$$

(Use the Unicode table from Appendix A to look up the character values: 'a' is 97, 'e' is 101, and 't' is 116.)

For your own classes, you should make up a hash code that combines the hash codes of the instance variables in a similar way. For example, let us declare a hashCode method for the Country class from Section 10.1.

There are two instance variables: the country name and the area. First, compute their hash codes. You know how to compute the hash code of a string. To compute the hash code of a floating-point number, first wrap the floating-point number into a Double object, and then compute its hash code.

Override hashCode methods in your own classes by combining the hash codes for the instance variables.

```
public class Country
{
    ...
    public int hashCode()
    {
        int h1 = name.hashCode();
        int h2 = new Double(area).hashCode();
        ...
    }
}
```

Then combine the two hash codes:

```
final int HASH_MULTIPLIER = 31;
int h = HASH_MULTIPLIER * h1 + h2;
return h;
```

However, it is easier to use the Objects.hash method which takes the hash codes of all arguments and combines them with a multiplier.

```
public int hashCode()
{
    return Objects.hash(name, area);
}
```

When you supply your own hashCode method for a class, you must also provide a compatible equals method. The equals method is used to differentiate between two objects that happen to have the same hash code.

The equals and hashCode methods must be *compatible* with each other. Two objects that are equal must yield the same hash code.

A class's hashCode method must be compatible with its equals method.

You get into trouble if your class declares an equals method but not a hashCode method. Suppose the Country class declares an equals method (checking that the name and area are the same), but no hashCode method. Then the hashCode method is inherited from the Object superclass. That method computes a hash code from the *memory location* of the object. Then it is very likely that two objects with the same contents will have different hash codes, in which case a hash set will store them as two distinct objects.

However, if you declare *neither* equals *nor* hashCode, then there is no problem. The equals method of the Object class considers two objects equal only if their memory location is the same. That is, the Object class has compatible equals and hashCode methods. Of course, then the notion of equality is very restricted: Only identical objects are considered equal. That can be a perfectly valid notion of equality, depending on your application.



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bjoe6code](http://wiley.com/go/bjoe6code) to download a program that demonstrates a hash set with objects of the Country class.

## 15.5 Stacks, Queues, and Priority Queues

In the following sections, we cover stacks, queues, and priority queues. These data structures each have a different policy for data removal. Removing an element yields the most recently added element, the least recently added, or the element with the highest priority.

### 15.5.1 Stacks

A stack is a collection of elements with “last-in, first-out” retrieval.

© budgetstockphoto/iStockphoto.



The Undo key pops commands off a stack so that the last command is the first to be undone.

A **stack** lets you insert and remove elements only at one end, traditionally called the *top* of the stack. New items can be added to the top of the stack. Items are removed from the top of the stack as well. Therefore, they are removed in the order that is opposite from the order in which they have been added, called *last-in, first-out* or *LIFO* order. For example, if you add items A, B, and C and then remove them, you obtain C, B, and A. With stacks, the addition and removal operations are called *push* and *pop*.

```
Stack<String> s = new Stack<>();
s.push("A"); s.push("B"); s.push("C");
while (s.size() > 0)
{
    System.out.print(s.pop() + " ");
} // Prints C B A
```

There are many applications for stacks in computer science. Consider the undo feature of a word processor. It keeps the issued commands in a stack. When you select “Undo”, the *last* command is undone, then the next-to-last, and so on.

Another important example is the **run-time stack** that a processor or virtual machine keeps to store the values of variables in nested methods. Whenever a new method is called, its parameter variables and local variables are pushed onto a stack. When the method exits, they are popped off again.

You will see other applications in Section 15.6.

The Java library provides a simple `Stack` class with methods `push`, `pop`, and `peek`—the latter gets the top element of the stack but does not remove it (see Table 7).



The last pancake that has been added to this stack will be the first one that is consumed.

© John Madden/iStockphoto.

**Table 7** Working with Stacks

<code>Stack&lt;Integer&gt; s = new Stack&lt;&gt;();</code>	Constructs an empty stack.
<code>s.push(1);</code> <code>s.push(2);</code> <code>s.push(3);</code>	Adds to the top of the stack; <code>s</code> is now [1, 2, 3]. (Following the <code>toString</code> method of the <code>Stack</code> class, we show the top of the stack at the end.)
<code>int top = s.pop();</code>	Removes the top of the stack; <code>top</code> is set to 3 and <code>s</code> is now [1, 2].
<code>head = s.peek();</code>	Gets the top of the stack without removing it; <code>head</code> is set to 2.

## 15.5.2 Queues

A queue is a collection of elements with “first-in, first-out” retrieval.

A **queue** lets you add items to one end of the queue (the *tail*) and remove them from the other end of the queue (the *head*). Queues yield items in a *first-in, first-out* or *FIFO* fashion. Items are removed in the same order in which they were added.

A typical application is a print queue. A printer may be accessed by several applications, perhaps running on different computers. If each of the applications tried to access the printer at the same time, the printout would be garbled. Instead, each application places its print data into a file and adds that file to the print queue. When the printer is done printing one file, it retrieves the next one from the queue. Therefore, print jobs are printed using the “first-in, first-out” rule, which is a fair arrangement for users of the shared printer.

The `Queue` interface in the standard Java library has methods `add` to add an element to the tail of the queue, `remove` to remove the head of the queue, and `peek` to get the head element of the queue without removing it (see Table 8).

The `LinkedList` class implements the `Queue` interface. Whenever you need a queue, simply initialize a `Queue` variable with a `LinkedList` object:

```
Queue<String> q = new LinkedList<>();
q.add("A"); q.add("B"); q.add("C");
while (q.size() > 0) { System.out.print(q.remove() + " ");}
// Prints A B C
```

The standard library provides several queue classes that we do not discuss in this book. Those classes are intended for work sharing when multiple activities (called threads) run in parallel.



To visualize a queue, think of people lining up.

**Table 8** Working with Queues

<code>Queue&lt;Integer&gt; q = new LinkedList&lt;&gt;();</code>	The <code>LinkedList</code> class implements the <code>Queue</code> interface.
<code>q.add(1);</code> <code>q.add(2);</code> <code>q.add(3);</code>	Adds to the tail of the queue; <code>q</code> is now [1, 2, 3].
<code>int head = q.remove();</code>	Removes the head of the queue; <code>head</code> is set to 1 and <code>q</code> is [2, 3].
<code>head = q.peek();</code>	Gets the head of the queue without removing it; <code>head</code> is set to 2.

## 15.5.3 Priority Queues

When removing an element from a priority queue, the element with the most urgent priority is retrieved.

A **priority queue** collects elements, each of which has a *priority*. A typical example of a priority queue is a collection of work requests, some of which may be more urgent than others. Unlike a regular queue, the priority queue does not maintain a first-in, first-out discipline. Instead, elements are retrieved according to their priority. In other words, new items can be inserted in any order. But whenever an item is removed, it is the item with the most urgent priority.



When you retrieve an item from a priority queue, you always get the most urgent one.



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bje06code](http://wiley.com/go/bje06code) to download programs that demonstrate stacks, queues, and priority queues.

It is customary to give low values to urgent priorities, with priority 1 denoting the most urgent priority. Thus, each removal operation extracts the *minimum* element from the queue.

For example, consider this code in which we add objects of a class `WorkOrder` into a priority queue. Each work order has a priority and a description.

```
PriorityQueue<WorkOrder> q = new PriorityQueue<>();
q.add(new WorkOrder(3, "Shampoo carpets"));
q.add(new WorkOrder(1, "Fix broken sink"));
q.add(new WorkOrder(2, "Order cleaning supplies"));
```

When calling `q.remove()` for the first time, the work order with priority 1 is removed. The next call to `q.remove()` removes the work order whose priority is highest among those remaining in the queue—in our example, the work order with priority 2. If there happen to be two elements with the same priority, the priority queue will break ties arbitrarily.

Because the priority queue needs to be able to tell which element is the smallest, the added elements should belong to a class that implements the `Comparable` interface. (See Section 10.3 for a description of that interface type.)

Table 9 shows the methods of the `PriorityQueue` class in the standard Java library.

**Table 9** Working with Priority Queues

<code>PriorityQueue&lt;Integer&gt; q = new PriorityQueue&lt;&gt;();</code>	This priority queue holds <code>Integer</code> objects. In practice, you would use objects that describe tasks.
<code>q.add(3); q.add(1); q.add(2);</code>	Adds values to the priority queue.
<code>int first = q.remove();</code> <code>int second = q.remove();</code>	Each call to <code>remove</code> removes the most urgent item: <code>first</code> is set to 1, <code>second</code> to 2.
<code>int next = q.peek();</code>	Gets the smallest value in the priority queue without removing it.



#### SELF CHECK

21. Why would you want to declare a variable as  
`Queue<String> q = new LinkedList<>();`  
 instead of simply declaring it as a linked list?
22. Why wouldn't you want to use an array list for implementing a queue?
23. What does this code print?  

```
Queue<String> q = new LinkedList<>();
q.add("A");
q.add("B");
q.add("C");
while (q.size() > 0) { System.out.print(q.remove() + " "); }
```
24. Why wouldn't you want to use a stack to manage print jobs?
25. In the sample code for a priority queue, we used a `WorkOrder` class. Could we have used strings instead?

```
PriorityQueue<String> q = new PriorityQueue<>();
q.add("3 - Shampoo carpets");
q.add("1 - Fix broken sink");
q.add("2 - Order cleaning supplies");
```

**Practice It** Now you can try these exercises at the end of the chapter: R15.15, E15.8, E15.9.

## 15.6 Stack and Queue Applications

Stacks and queues are, despite their simplicity, very versatile data structures. In the following sections, you will see some of their most useful applications.

### 15.6.1 Balancing Parentheses

A stack can be used to check whether parentheses in an expression are balanced.

In Common Error 4.2, you saw a simple trick for detecting unbalanced parentheses in an expression such as

-(b \* b - (4 \* a \* c ) ) / (2 \* a)  
1      2      10    1      0

Increment a counter when you see a ( and decrement it when you see a ). The counter should never be negative, and it should be zero at the end of the expression.

That works for expressions in Java, but in mathematical notation, one can have more than one kind of parentheses, such as

-{ [b · b - (4 · a · c ) ] / (2 · a) }

To see whether such an expression is correctly formed, place the parentheses on a stack:

**When you see an opening parenthesis, push it on the stack.**

**When you see a closing parenthesis, pop the stack.**

**If the opening and closing parentheses don't match**

**The parentheses are unbalanced. Exit.**

**If at the end the stack is empty**

**The parentheses are balanced.**

**Else**

**The parentheses are not balanced.**



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bje06code](http://wiley.com/go/bje06code) to download a program for checking balanced parentheses.

Here is a walkthrough of the sample expression:

Stack	Unread expression	Comments
Empty	-{ [b * b - (4 * a * c ) ] / (2 * a) }	
{	[b * b - (4 * a * c ) ] / (2 * a) }	
{ [	b * b - (4 * a * c ) ] / (2 * a) }	
{ [ (	4 * a * c ) ] / (2 * a) }	
{ [ ]	] / (2 * a) }	( matches )
{ [ /	/ (2 * a) }	[ matches ]
{ [ 2	2 * a) }	
{ [ }	}	( matches )
Empty	No more input	{ matches }
		The parentheses are balanced

## 15.6.2 Evaluating Reverse Polish Expressions

Use a stack to evaluate expressions in reverse Polish notation.

Consider how you write arithmetic expressions, such as  $(3 + 4) \times 5$ . The parentheses are needed so that 3 and 4 are added before multiplying the result by 5.

However, you can eliminate the parentheses if you write the operators *after* the numbers, like this: 3 4 + 5 × (see Special Topic 15.2 on page 709). To evaluate this expression, apply + to 3 and 4, yielding 7, and then simplify 7 5 × to 35. It gets trickier for complex expressions. For example, 3 4 5 + × means to compute 4 5 + (that is, 9), and then evaluate 3 9 ×. If we evaluate this expression left-to-right, we need to leave the 3 somewhere while we work on 4 5 +. Where? We put it on a stack. The algorithm for evaluating reverse Polish expressions is simple:

```

If you read a number
    Push it on the stack.
Else if you read an operand
    Pop two values off the stack.
    Combine the values with the operand.
    Push the result back onto the stack.
Else if there is no more input
    Pop and display the result.

```

Here is a walkthrough of evaluating the expression 3 4 5 + ×:

Stack	Unread expression	Comments
Empty	3 4 5 + ×	
3	4 5 + ×	Numbers are pushed on the stack
3 4	5 + ×	
3 4 5	+ ×	
3 9	x	Pop 4 and 5, push 4 5 +
27	No more input	Pop 3 and 9, push 3 9 x
Empty		Pop and display the result, 27

The following program simulates a reverse Polish calculator:

### section\_6\_2/Calculator.java

```

1 import java.util.Scanner;
2 import java.util.Stack;
3
4 /**
5  * This calculator uses the reverse Polish notation.
6 */
7 public class Calculator
8 {
9     public static void main(String[] args)
10    {
11        Scanner in = new Scanner(System.in);
12        Stack<Integer> results = new Stack<>();
13        System.out.println("Enter one number or operator per line, Q to quit. ");
14        boolean done = false;

```

```

15  while (!done)
16  {
17      String input = in.nextLine();
18
19      // If the command is an operator, pop the arguments and push the result
20
21      if (input.equals("+"))
22      {
23          results.push(results.pop() + results.pop());
24      }
25      else if (input.equals("-"))
26      {
27          Integer arg2 = results.pop();
28          results.push(results.pop() - arg2);
29      }
30      else if (input.equals("*") || input.equals("x"))
31      {
32          results.push(results.pop() * results.pop());
33      }
34      else if (input.equals("/"))
35      {
36          Integer arg2 = results.pop();
37          results.push(results.pop() / arg2);
38      }
39      else if (input.equals("Q") || input.equals("q"))
40      {
41          done = true;
42      }
43      else
44      {
45          // Not an operator--push the input value
46
47          results.push(Integer.parseInt(input));
48      }
49      System.out.println(results);
50  }
51 }
52 }
```

### 15.6.3 Evaluating Algebraic Expressions

Using two stacks,  
you can evaluate  
expressions in  
standard algebraic  
notation.

In the preceding section, you saw how to evaluate expressions in reverse Polish notation, using a single stack. If you haven't found that notation attractive, you will be glad to know that one can evaluate an expression in the standard algebraic notation using two stacks—one for numbers and one for operators.

*Use two stacks to evaluate algebraic expressions.*



© Jorge Delgado/Stockphoto

First, consider a simple example, the expression  $3 + 4$ . We push the numbers on the number stack and the operators on the operator stack. Then we pop both numbers and the operator, combine the numbers with the operator, and push the result.

	Number stack	Operator stack	Unprocessed input	Comments
	Empty	Empty	$3 + 4$	
1	3		$+ 4$	
2	3	+	4	
3	4 3	+	No more input	Evaluate the top.
4	7			The result is 7.

This operation is fundamental to the algorithm. We call it “evaluating the top”.

In algebraic notation, each operator has a *precedence*. The + and - operators have the lowest precedence, \* and / have a higher (and equal) precedence.

Consider the expression  $3 \times 4 + 5$ . Here are the first processing steps:

	Number stack	Operator stack	Unprocessed input	Comments
	Empty	Empty	$3 \times 4 + 5$	
1	3		$\times 4 + 5$	
2	3	*	$4 + 5$	
3	4 3	*	$+ 5$	Evaluate $\times$ before $+$ .

Because  $\times$  has a higher precedence than  $+$ , we are ready to evaluate the top:

	Number stack	Operator stack	Comments
4	12	+	5
5	5 12	+	No more input
6	17		That is the result.

With the expression,  $3 + 4 \times 5$ , we add  $\times$  to the operator stack because we must first read the next number; then we can evaluate  $\times$  and then the  $+$ :

	Number stack	Operator stack	Unprocessed input	Comments
	Empty	Empty	$3 + 4 \times 5$	
1	3		$+ 4 \times 5$	
2	3	+	$4 \times 5$	

3	<table border="1"><tr><td>4</td></tr><tr><td>3</td></tr></table>	4	3	<table border="1"><tr><td>+</td></tr></table>	+	$\times 5$	Don't evaluate + yet.	
4								
3								
+								
4	<table border="1"><tr><td>4</td></tr><tr><td>3</td></tr></table>	4	3	<table border="1"><tr><td><math>\times</math></td></tr><tr><td>+</td></tr></table>	$\times$	+	5	
4								
3								
$\times$								
+								

In other words, we keep operators on the stack until they are ready to be evaluated. Here is the remainder of the computation:

	Number stack	Operator stack	Comments					
5	<table border="1"><tr><td>5</td></tr><tr><td>4</td></tr><tr><td>3</td></tr></table>	5	4	3	<table border="1"><tr><td><math>\times</math></td></tr><tr><td>+</td></tr></table>	$\times$	+	No more input Evaluate the top.
5								
4								
3								
$\times$								
+								
6	<table border="1"><tr><td>20</td></tr><tr><td>3</td></tr></table>	20	3	<table border="1"><tr><td>+</td></tr></table>	+	Evaluate top again.		
20								
3								
+								
7	<table border="1"><tr><td>23</td></tr></table>	23		That is the result.				
23								

To see how parentheses are handled, consider the expression  $3 \times (4 + 5)$ . A ( is pushed on the operator stack. The + is pushed as well. When we encounter the ), we know that we are ready to evaluate the top until the matching ( reappears:

	Number stack	Operator stack	Unprocessed input	Comments			
	Empty	Empty	$3 \times (4 + 5)$				
1	<table border="1"><tr><td>3</td></tr></table>	3		$\times (4 + 5)$			
3							
2	<table border="1"><tr><td>3</td></tr></table>	3	$\times$	$(4 + 5)$			
3							
3	<table border="1"><tr><td>3</td></tr></table>	3	$($	$4 + 5)$	Don't evaluate $\times$ yet.		
3							
4	<table border="1"><tr><td>4</td></tr><tr><td>3</td></tr></table>	4	3	$($	$+ 5)$		
4							
3							
5	<table border="1"><tr><td>4</td></tr><tr><td>3</td></tr></table>	4	3	$+$	$5)$		
4							
3							
6	<table border="1"><tr><td>5</td></tr><tr><td>4</td></tr><tr><td>3</td></tr></table>	5	4	3	$+$	)	Evaluate the top.
5							
4							
3							
7	<table border="1"><tr><td>9</td></tr><tr><td>3</td></tr></table>	9	3	$($	No more input	Pop (.	
9							
3							
8	<table border="1"><tr><td>9</td></tr><tr><td>3</td></tr></table>	9	3	$\times$		Evaluate top again.	
9							
3							
9	<table border="1"><tr><td>27</td></tr></table>	27			That is the result.		
27							

Here is the algorithm:

```

If you read a number
    Push it on the number stack.
Else if you read a (
    Push it on the operator stack.
Else if you read an operator op
    While the top of the stack has a higher precedence than op
        Evaluate the top.
    Push op on the operator stack.
Else if you read a )
    While the top of the stack is not a (
        Evaluate the top.
    Pop the (.
Else if there is no more input
    While the operator stack is not empty
        Evaluate the top.

```

At the end, the remaining value on the number stack is the value of the expression.

The algorithm makes use of this helper method that evaluates the topmost operator with the topmost numbers:

#### Evaluate the top:

```

Pop two numbers off the number stack.
Pop an operator off the operator stack.
Combine the numbers with that operator.
Push the result on the number stack.

```

**FULL CODE EXAMPLE**  
Go to [wiley.com/go/bje06code](http://wiley.com/go/bje06code) to get the complete code for the expression calculator.

### 15.6.4 Backtracking

Use a stack to remember choices you haven't yet made so that you can backtrack to them.

Suppose you are inside a maze. You need to find the exit. What should you do when you come to an intersection? You can continue exploring one of the paths, but you will want to remember the other ones. If your chosen path didn't work, you can go back to one of the other choices and try again.

Of course, as you go along one path, you may reach further intersections, and you need to remember your choice again. Simply use a stack to remember the paths that still need to be tried. The process of returning to a choice point and trying another choice is called *backtracking*. By using a stack, you return to your more recent choices before you explore the earlier ones.

Figure 11 shows an example. We start at a point in the maze, at position (3, 4). There are four possible paths. We push them all on a stack ❶. We pop off the topmost one, traveling north from (3, 4). Following this path leads to position (1, 4). We now push two choices on the stack, going west or east ❷. Both of them lead to dead ends ❸ ❹.

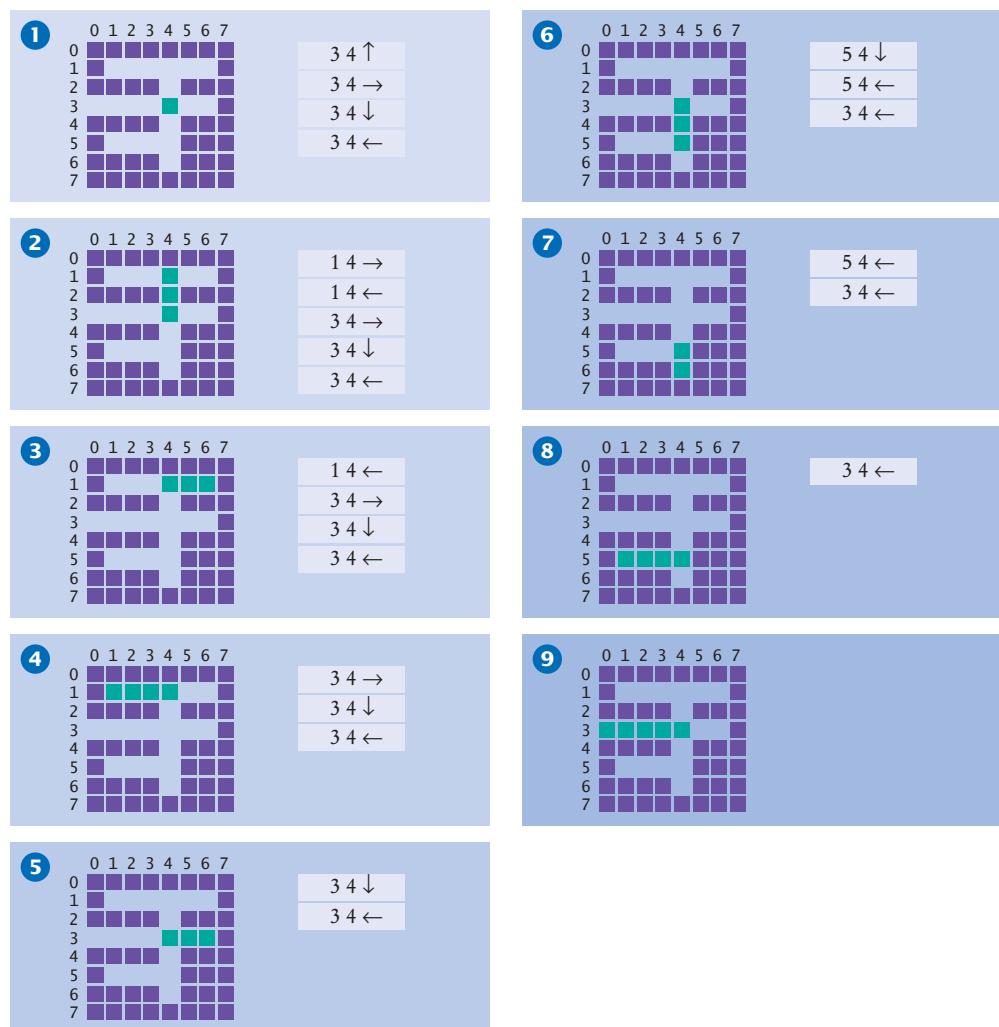
Now we pop off the path from (3, 4) going east. That too is a dead end ❺. Next is the path from (3, 4) going south. At (5, 4), it comes to an intersection. Both choices are pushed on the stack ❻. They both lead to dead ends ❼ ❼.

Finally, the path from (3, 4) going west leads to an exit ❼.



A stack can be used to track positions in a maze.

© Skip O'Donnell/  
iStockphoto.

**Figure 11** Backtracking Through a Maze

Using a stack, we have found a path out of the maze. Here is the pseudocode for our maze-finding algorithm:

```

Push all paths from the point on which you are standing on a stack.
While the stack is not empty
    Pop a path from the stack.
    Follow the path until you reach an exit, intersection, or dead end.
    If you found an exit
        Congratulations!
    Else if you found an intersection
        Push all paths meeting at the intersection, except the current one, onto the stack.

```

This algorithm will find an exit from the maze, provided that the maze has no *cycles*. If it is possible that you can make a circle and return to a previously visited intersection along a different sequence of paths, then you need to work harder—see Exercise E15.21.

How you implement this algorithm depends on the description of the maze. In the example code, we use a two-dimensional array of characters, with spaces for corridors and asterisks for walls, like this:

```
* * * * * * *
*   *   *
* * * *   * *
*   *   *   *
* * * *   * *
*   *   *   *
* * * *   * *
*   *   *   *
```

In the example code, a `Path` object is constructed with a starting position and a direction (North, East, South, or West). The `Maze` class has a method that extends a path until it reaches an intersection or exit, or until it is blocked by a wall, and a method that computes all paths from an intersection point.

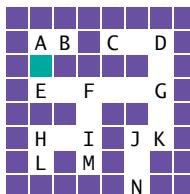
Note that you can use a queue instead of a stack in this algorithm. Then you explore the earlier alternatives before the later ones. This can work just as well for finding an answer, but it isn't very intuitive in the context of exploring a maze—you would have to imagine being teleported back to the initial intersections rather than just walking back to the last one.

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download a complete program demonstrating backtracking.

**SELF CHECK**

26. What is the value of the reverse Polish notation expression  $2\ 3\ 4\ 5\ \times\ +\ \times$ ?
27. Why does the branch for the subtraction operator in the `calculator` program not simply execute  
`results.push(results.pop() - results.pop());`
28. In the evaluation of the expression  $3 - 4 + 5$  with the algorithm of Section 15.6.3, which operator gets evaluated first?
29. In the algorithm of Section 15.6.3, are the operators on the operator stack always in increasing precedence?
30. Consider the following simple maze. Assuming that we start at the marked point and push paths in the order West, South, East, North, in which order are the lettered points visited, using the algorithm of Section 15.6.4?



**Practice It** Now you can try these exercises at the end of the chapter: R15.25, E15.18, E15.20, E15.21, E15.22.

**WORKED EXAMPLE 15.2****Simulating a Queue of Waiting Customers**

Learn how to use a queue to simulate an actual queue of waiting customers. Go to [wiley.com/go/bjeo6examples](http://wiley.com/go/bjeo6examples) and download Worked Example 15.2.



Photodisc/Punchstock.

## Special Topic 15.2

**Reverse Polish Notation**

In the 1920s, the Polish mathematician Jan Łukasiewicz realized that it is possible to dispense with parentheses in arithmetic expressions, provided that you write the operators *before* their arguments, for example,  $+ 3 4$  instead of  $3 + 4$ . Thirty years later, Australian computer scientist Charles Hamblin noted that an even better scheme would be to have the operators *follow* the operands. This was termed **reverse Polish notation** or RPN.

Standard Notation	Reverse Polish Notation
$3 + 4$	$3 4 +$
$3 + 4 \times 5$	$3 4 5 \times +$
$3 \times (4 + 5)$	$3 4 5 + \times$
$(3 + 4) \times (5 + 6)$	$3 4 + 5 6 + \times$
$3 + 4 + 5$	$3 4 + 5 +$

Reverse Polish notation might look strange to you, but that is just an accident of history. Had earlier mathematicians realized its advantages, today's schoolchildren might be using it and not worrying about precedence rules and parentheses.

In 1972, Hewlett-Packard introduced the HP 35 calculator that used reverse Polish notation. The calculator had no keys labeled with parentheses or an equals symbol. There is just a key labeled ENTER to push a number onto a stack. For that reason, Hewlett-Packard's marketing department used to refer to their product as "the calculators that have no equal".

Over time, calculator vendors have adapted to the standard algebraic notation rather than forcing its users to learn a new notation. However, those users who have made the effort to learn reverse Polish notation tend to be fanatic proponents, and to this day, some Hewlett-Packard calculator models still support it.



Courtesy of Nigel Tout.

*The Calculator with No Equal*

## CHAPTER SUMMARY

### Understand the architecture of the Java collections framework.



- A collection groups together elements and allows them to be retrieved later.
- A list is a collection that remembers the order of its elements.
- A set is an unordered collection of unique elements.
- A map keeps associations between key and value objects.

### Understand and use linked lists.



- A linked list consists of a number of nodes, each of which has a reference to the next node.
- Adding and removing elements at a given position in a linked list is efficient.
- Visiting the elements of a linked list in sequential order is efficient, but random access is not.
- You use a list iterator to access elements inside a linked list.

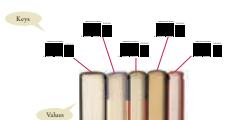
### Choose a set implementation and use it to manage sets of values.



- The HashSet and TreeSet classes both implement the Set interface.
- Set implementations arrange the elements so that they can locate them quickly.
- You can form hash sets holding objects of type String, Integer, Double, Point, Rectangle, or Color.
- You can form tree sets for any class that implements the Comparable interface, such as String or Integer.
- Sets don't have duplicates. Adding a duplicate of an element that is already present is ignored.
- A set iterator visits the elements in the order in which the set implementation keeps them.
- You cannot add an element to a set at an iterator position.



### Use maps to model associations between keys and values.



- The HashMap and TreeMap classes both implement the Map interface.
- To find all keys and values in a map, iterate through the key set and find the values that correspond to the keys.
- A hash function computes an integer value from an object.
- A good hash function minimizes *collisions*—identical hash codes for different objects.
- Override hashCode methods in your own classes by combining the hash codes for the instance variables.
- A class's hashCode method must be compatible with its equals method.



### Use the Java classes for stacks, queues, and priority queues.



- A stack is a collection of elements with “last-in, first-out” retrieval.
- A queue is a collection of elements with “first-in, first-out” retrieval.
- When removing an element from a priority queue, the element with the most urgent priority is retrieved.



### Solve programming problems using stacks and queues.



- A stack can be used to check whether parentheses in an expression are balanced.
- Use a stack to evaluate expressions in reverse Polish notation.
- Using two stacks, you can evaluate expressions in standard algebraic notation.
- Use a stack to remember choices you haven’t yet made so that you can backtrack to them.

### STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

`java.util.Collection<E>`  
`add`  
`contains`  
`iterator`  
`remove`  
`size`  
`java.util.HashMap<K, V>`  
`java.util.HashSet<K, V>`  
`java.util.Iterator<E>`  
`hasNext`  
`next`  
`remove`  
`java.util.LinkedList<E>`  
`addFirst`  
`addLast`  
`getFirst`  
`getLast`  
`removeFirst`  
`removeLast`

`java.util.List<E>`  
`listIterator`  
`java.util.ListIterator<E>`  
`add`  
`hasPrevious`  
`previous`  
`set`  
`java.util.Map<K, V>`  
`get`  
`keySet`  
`put`  
`remove`  
`java.util.Objects`  
`hash`

`java.util.PriorityQueue<E>`  
`remove`  
`java.util.Queue<E>`  
`peek`  
`java.util.Set<E>`  
`java.util.Stack<E>`  
`peek`  
`pop`  
`push`  
`java.util.TreeMap<K, V>`  
`java.util.TreeSet<K, V>`

### REVIEW EXERCISES

- R15.1** An invoice contains a collection of purchased items. Should that collection be implemented as a list or set? Explain your answer.
- R15.2** Consider a program that manages an appointment calendar. Should it place the appointments into a list, stack, queue, or priority queue? Explain your answer.
- R15.3** One way of implementing a calendar is as a map from date objects to event objects. However, that only works if there is a single event for a given date. How can you use another collection type to allow for multiple events on a given date?

■ ■ **R15.4** Look up the descriptions of the methods `addAll`, `removeAll`, `retainAll`, and `containsAll` in the `Collection` interface. Describe how these methods can be used to implement common operations on sets (union, intersection, difference, subset).

■ **R15.5** Explain what the following code prints. Draw a picture of the linked list after each step.

```
LinkedList<String> staff = new LinkedList<>();
staff.addFirst("Harry");
staff.addFirst("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeFirst());
System.out.println(staff.removeFirst());
System.out.println(staff.removeFirst());
```

■ **R15.6** Explain what the following code prints. Draw a picture of the linked list after each step.

```
LinkedList<String> staff = new LinkedList<>();
staff.addFirst("Harry");
staff.addFirst("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeLast());
System.out.println(staff.removeFirst());
System.out.println(staff.removeLast());
```

■ **R15.7** Explain what the following code prints. Draw a picture of the linked list after each step.

```
LinkedList<String> staff = new LinkedList<>();
staff.addFirst("Harry");
staff.addLast("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeLast());
System.out.println(staff.removeFirst());
System.out.println(staff.removeLast());
```

■ **R15.8** Explain what the following code prints. Draw a picture of the linked list and the iterator position after each step.

```
LinkedList<String> staff = new LinkedList<>();
ListIterator<String> iterator = staff.listIterator();
iterator.add("Tom");
iterator.add("Diana");
iterator.add("Harry");
iterator = staff.listIterator();
if (iterator.next().equals("Tom")) { iterator.remove(); }
while (iterator.hasNext()) { System.out.println(iterator.next()); }
```

■ **R15.9** Explain what the following code prints. Draw a picture of the linked list and the iterator position after each step.

```
LinkedList<String> staff = new LinkedList<>();
ListIterator<String> iterator = staff.listIterator();
iterator.add("Tom");
iterator.add("Diana");
iterator.add("Harry");
iterator = staff.listIterator();
iterator.next();
iterator.next();
```

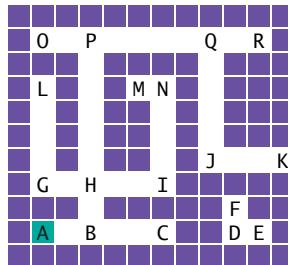
```

iterator.add("Romeo");
iterator.next();
iterator.add("Juliet");
iterator = staff.listIterator();
iterator.next();
iterator.remove();
while (iterator.hasNext()) { System.out.println(iterator.next()); }

```

- R15.10** You are given a linked list of strings. How do you remove all elements with length less than or equal to three?
- R15.11** Repeat Exercise R15.10, using the `removeIf` method. (Read the description in the API of the `Collection` interface.) Use a lambda expression (see Java 8 Note 10.4).
- R15.12** What advantages do linked lists have over arrays? What disadvantages do they have?
- R15.13** Suppose you need to organize a collection of telephone numbers for a company division. There are currently about 6,000 employees, and you know that the phone switch can handle at most 10,000 phone numbers. You expect several hundred look-ups against the collection every day. Would you use an array list or a linked list to store the information?
- R15.14** Suppose you need to keep a collection of appointments. Would you use a linked list or an array list of `Appointment` objects?
- R15.15** Suppose you write a program that models a card deck. Cards are taken from the top of the deck and given out to players. As cards are returned to the deck, they are placed on the bottom of the deck. Would you store the cards in a stack or a queue?
- R15.16** Suppose the strings "A" . . . "Z" are pushed onto a stack. Then they are popped off the stack and pushed onto a second stack. Finally, they are all popped off the second stack and printed. In which order are the strings printed?
- R15.17** What is the difference between a set and a map?
- R15.18** The union of two sets  $A$  and  $B$  is the set of all elements that are contained in  $A$ ,  $B$ , or both. The intersection is the set of all elements that are contained in  $A$  and  $B$ . How can you compute the union and intersection of two sets, using the `add` and `contains` methods, together with an iterator?
- R15.19** How can you compute the union and intersection of two sets, using some of the methods that the `java.util.Set` interface provides, but without using an iterator? (Look up the interface in the API documentation.)
- R15.20** Can a map have two keys with the same value? Two values with the same key?
- R15.21** A map can be implemented as a set of  $(key, value)$  pairs. Explain.
- R15.22** How can you print all key/value pairs of a map, using the `keySet` method? The `entrySet` method? The `forEach` method with a lambda expression? (See Java 8 Note 10.4 on lambda expressions.)
- R15.23** Verify the hash code of the string "Juliet" in Table 6.
- R15.24** Verify that the strings "VII" and "Ugh" have the same hash code.

- **R15.25** Consider the algorithm for traversing a maze from Section 15.6.4 Assume that we start at position A and push in the order West, South, East, and North. In which order will the lettered locations of the sample maze be visited?



- **R15.26** Repeat Exercise R15.25, using a queue instead of a stack.

## PRACTICE EXERCISES

- ■ **E15.1** Write a method

```
public static void downsize(LinkedList<String> employeeNames, int n)
```

that removes every  $n$ th employee from a linked list.

- ■ **E15.2** Write a method

```
public static void reverse(LinkedList<String> strings)
```

that reverses the entries in a linked list.

- ■ **E15.3** Implement the *sieve of Eratosthenes*: a method for computing prime numbers, known to the ancient Greeks. This method will compute all prime numbers up to  $n$ . Choose an  $n$ . First insert all numbers from 2 to  $n$  into a set. Then erase all multiples of 2 (except 2); that is, 4, 6, 8, 10, 12, .... Erase all multiples of 3; that is, 6, 9, 12, 15, .... Go up to  $\sqrt{n}$ . Then print the set.



© martinmcelligott/iStockphoto.

- ■ **E15.4** Write a program that keeps a map in which both keys and values are strings—the names of students and their course grades. Prompt the user of the program to add or remove students, to modify grades, or to print all grades. The printout should be sorted by name and formatted like this:

```
Carl: B+
Joe: C
Sarah: A
```

- ■ ■ **E15.5** Write a program that reads a Java source file and produces an index of all identifiers in the file. For each identifier, print all lines in which it occurs. For simplicity, we will consider each string consisting only of letters, numbers, and underscores an identifier. Declare a Scanner `in` for reading from the source file and call `in.useDelimiter("[^A-Za-z0-9_]+")`. Then each call to `next` returns an identifier.

- ■ ■ **E15.6** Read all words from a file and add them to a map whose keys are the first letters of the words and whose values are sets of words that start with that same letter. Then print out the word sets in alphabetical order.

Provide two versions of your solution, one that uses the `merge` method (see Java 8 Note 15.1) and one that updates the map as in Worked Example 15.1.



- E15.7** Read all words from a file and add them to a map whose keys are word lengths and whose values are comma-separated strings of words of the same length. Then print out those strings, in increasing order by the length of their entries.

Provide two versions of your solution, one that uses the `merge` method (see Java 8 Note 15.1) and one that updates the map as in Worked Example 15.1.

- E15.8** Use a stack to reverse the words of a sentence. Keep reading words until you have a word that ends in a period, adding them onto a stack. When you have a word with a period, pop the words off and print them. Stop when there are no more words in the input. For example, you should turn the input

Mary had a little lamb. Its fleece was white as snow.

into

Lamb little a had mary. Snow as white was fleece its.

Pay attention to capitalization and the placement of the period.

- E15.9** Your task is to break a number into its individual digits, for example, to turn 1729 into 1, 7, 2, and 9. It is easy to get the last digit of a number  $n$  as  $n \% 10$ . But that gets the numbers in reverse order. Solve this problem with a stack. Your program should ask the user for an integer, then print its digits separated by spaces.

- E15.10** A homeowner rents out parking spaces in a driveway during special events. The driveway is a “last-in, first-out” stack. Of course, when a car owner retrieves a vehicle that wasn’t the last one in, the cars blocking it must temporarily move to the street so that the requested vehicle can leave. Write a program that models this behavior, using one stack for the driveway and one stack for the street. Use integers as license plate numbers. Positive numbers add a car, negative numbers remove a car, zero stops the simulation. Print out the stack after each operation is complete.

- E15.11** Implement a to do list. Tasks have a priority between 1 and 9, and a description. When the user enters the command `add priority description`, the program adds a new task. When the user enters `next`, the program removes and prints the most urgent task. The `quit` command quits the program. Use a priority queue in your solution.

- E15.12** Write a program that reads text from a file and breaks it up into individual words. Insert the words into a tree set. At the end of the input file, print all words, followed by the size of the resulting set. This program determines how many unique words a text file has.

- E15.13** Insert all words from a large file (such as the novel “War and Peace”, which is available on the Internet) into a hash set and a tree set. Time the results. Which data structure is more efficient?

- E15.14** Supply compatible `hashCode` and `equals` methods to the `BankAccount` class of Chapter 8. Test the `hashCode` method by printing out hash codes and by adding `BankAccount` objects to a hash set.

- E15.15** 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 `hashCode`

and `equals` methods. Two labeled points are considered the same when they have the same location and label.

- ■ **E15.16** Reimplement the `LabeledPoint` class of Exercise E15.15 by storing the location in a `java.awt.Point` object. Your `hashCode` and `equals` methods should call the `hashCode` and `equals` methods of the `Point` class.
- ■ **E15.17** Modify the `LabeledPoint` class of Exercise E15.15 so that it implements the `Comparable` interface. Sort points first by their *x*-coordinates. If two points have the same *x*-coordinate, sort them by their *y*-coordinates. If two points have the same *x*- and *y*-coordinates, sort them by their label. Write a tester program that checks all cases by inserting points into a `TreeSet`.
- **E15.18** Add a `%` (remainder) operator to the expression calculator of Section 15.6.3.
- ■ **E15.19** Add a `^` (power) operator to the expression calculator of Section 15.6.3. For example,  $2^3$  evaluates to 8. As in mathematics, your power operator should be evaluated from the right. That is,  $2^3^2$  is  $2^{(3^2)}$ , not  $(2^3)^2$ . (That's more useful because you could get the latter as  $2^{(3 \times 2)}$ .)
- **E15.20** Write a program that checks whether a sequence of HTML tags is properly nested. For each opening tag, such as `<p>`, there must be a closing tag `</p>`. A tag such as `<p>` may have other tags inside, for example

```
<p> <ul> <li> </li> </ul> <a> </a> </p>
```

The inner tags must be closed before the outer ones. Your program should process a file containing tags. For simplicity, assume that the tags are separated by spaces, and that there is no text inside the tags.

- **E15.21** Modify the maze solver program of Section 15.6.4 to handle mazes with cycles. Keep a set of visited intersections. When you have previously seen an intersection, treat it as a dead end and do not add paths to the stack.
- ■ ■ **E15.22** In a paint program, a “flood fill” fills all empty pixels of a drawing with a given color, stopping when it reaches occupied pixels. In this exercise, you will implement a simple variation of this algorithm, flood-filling a  $10 \times 10$  array of integers that are initially 0.

**Prompt for the starting row and column.**

**Push the (row, column) pair onto a stack.**

You will need to provide a simple `Pair` class.

Repeat the following operations until the stack is empty.

**Pop off the (row, column) pair from the top of the stack.**

**If it has not yet been filled, fill the corresponding array location with a number 1, 2, 3, and so on  
(to show the order in which the square is filled).**

**Push the coordinates of any unfilled neighbors in the north, east, south, or west direction on the stack.**

When you are done, print the entire array.



## PROGRAMMING PROJECTS

- P15.1** Read all words from a list of words and add them to a map whose keys are the phone keypad spellings of the word, and whose values are sets of words with the same code. For example, 26337 is mapped to the set { "Andes", "coder", "codes", . . . }. Then keep prompting the user for numbers and print out all words in the dictionary that can be spelled with that number. In your solution, use a map that maps letters to digits.



© klenger/Stockphoto

- P15.2** Reimplement Exercise E15.4 so that the keys of the map are objects of class `Student`. A student should have a first name, a last name, and a unique integer ID. For grade changes and removals, lookup should be by ID. The printout should be sorted by last name. If two students have the same last name, then use the first name as a tie breaker. If the first names are also identical, then use the integer ID. *Hint:* Use two maps.

- P15.3** Write a class `Polynomial` that stores a polynomial such as

$$p(x) = 5x^{10} + 9x^7 - x - 10$$

as a linked list of terms. A term contains the coefficient and the power of  $x$ . For example, you would store  $p(x)$  as

$$(5,10), (9,7), (-1,1), (-10,0)$$

Supply methods to add, multiply, and print polynomials. Supply a constructor that makes a polynomial from a single term. For example, the polynomial `p` can be constructed as

```
Polynomial p = new Polynomial(new Term(-10, 0));
p.add(new Polynomial(new Term(-1, 1)));
p.add(new Polynomial(new Term(9, 7)));
p.add(new Polynomial(new Term(5, 10)));
```

Then compute  $p(x) \times p(x)$ .

```
Polynomial q = p.multiply(p);
q.print();
```

- P15.4** Repeat Exercise P15.3, but use a `Map<Integer, Double>` for the coefficients.

- P15.5** Try to find two words with the same hash code in a large file. Keep a `Map<Integer, HashSet<String>>`. When you read in a word, compute its hash code  $h$  and put the word in the set whose key is  $h$ . Then iterate through all keys and print the sets whose size is greater than one.

- P15.6** Supply compatible `hashCode` and `equals` methods to the `Student` class described in Exercise P15.2. Test the hash code by adding `Student` objects to a hash set.

- P15.7** Modify the expression calculator of Section 15.6.3 to convert an expression into reverse Polish notation. *Hint:* Instead of evaluating the top and pushing the result, append the instructions to a string.

- P15.8** Repeat Exercise E15.22, but use a queue instead.

- P15.9** Use a stack to enumerate all permutations of a string. Suppose you want to find all permutations of the string `meat`.

```

Push the string +meat on the stack.
While the stack is not empty
    Pop off the top of the stack.
    If that string ends in a + (such as tame+)
        Remove the + and add the string to the list of permutations.
    Else
        Remove each letter in turn from the right of the +.
        Insert it just before the +.
        Push the resulting string on the stack.

```

For example, after popping `e+mta`, you push `em+ta`, `et+ma`, and `ea+mt`.

- P15.10** Repeat Exercise P15.9, but use a queue instead.

- Business P15.11** An airport has only one runway. When it is busy, planes wishing to take off or land have to wait. Implement a simulation, using two queues, one each for the planes waiting to take off and land. Landing planes get priority. The user enters commands `takeoff flightSymbol`, `land flightSymbol`, `next`, and `quit`. The first two commands place the flight in the appropriate queue. The next command finishes the current takeoff or landing and enables the next one, printing the action (takeoff or land) and the flight symbol.

- Business P15.12** Suppose you buy 100 shares of a stock at \$12 per share, then another 100 at \$10 per share, and then sell 150 shares at \$15. You have to pay taxes on the gain, but exactly what is the gain? In the United States, the FIFO rule holds: You first sell all shares of the first batch for a profit of \$300, then 50 of the shares from the second batch, for a profit of \$250, yielding a total profit of \$550. Write a program that can make these calculations for arbitrary purchases and sales of shares in a single company. The user enters commands `buy quantity price`, `sell quantity` (which causes the gain to be displayed), and `quit`. Hint: Keep a queue of objects of a class `Block` that contains the quantity and price of a block of shares.

- Business P15.13** Extend Exercise P15.12 to a program that can handle shares of multiple companies. The user enters commands `buy symbol quantity price` and `sell symbol quantity`. Hint: Keep a `Map<String, Queue<Block>>` that manages a separate queue for each stock symbol.

- Business P15.14** Consider the problem of finding the least expensive routes to all cities in a network from a given starting point. For example, in the network shown on the map on page 719, the least expensive route from Pendleton to Peoria has cost 8 (going through Pierre and Pueblo).

The following helper class expresses the distance to another city:

```

public class DistanceTo implements Comparable<DistanceTo>
{
    private String target;
    private int distance;

    public DistanceTo(String city, int dist) { target = city; distance = dist; }
    public String getTarget() { return target; }
    public int getDistance() { return distance; }
    public int compareTo(DistanceTo other) { return distance - other.distance; }
}

```

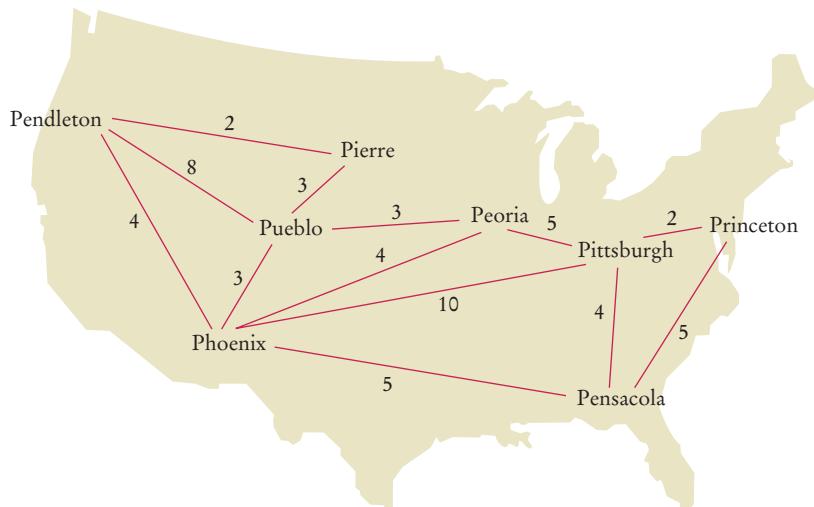
All direct connections between cities are stored in a `Map<String, TreeSet<DistanceTo>>`. The algorithm now proceeds as follows:

```

Let from be the starting point.
Add DistanceTo(from, 0) to a priority queue.
Construct a map shortestKnownDistance from city names to distances.
While the priority queue is not empty
    Get its smallest element.
    If its target is not a key in shortestKnownDistance
        Let d be the distance to that target.
        Put (target, d) into shortestKnownDistance.
        For all cities c that have a direct connection from target
            Add DistanceTo(c, d + distance from target to c) to the priority queue.

```

When the algorithm has finished, `shortestKnownDistance` contains the shortest distance from the starting point to all reachable targets.



Your task is to write a program that implements this algorithm. Your program should read in lines of the form `city1 city2 distance`. The starting point is the first city in the first line. Print the shortest distances to all other cities.

### ANSWERS TO SELF-CHECK QUESTIONS

1. A list is a better choice because the application will want to retain the order in which the quizzes were given.
2. A set is a better choice. There is no intrinsically useful ordering for the students. For example, the registrar's office has little use for a list of all students by their GPA. By storing them in a set, adding, removing, and finding students can be efficient.
3. With a stack, you would always read the latest required reading, and you might never get to the oldest readings.
4. A collection stores elements, but a map stores associations between elements.
5. Yes, for two reasons. A linked list needs to store the neighboring node references, which are not needed in an array. Moreover, there is some overhead for storing an object. In a

linked list, each node is a separate object that incurs this overhead, whereas an array is a single object.

- linked list, each node is a separate object that incurs this overhead, whereas an array is a single object.

6. We can simply access each array element with an integer index.

7. |ABCD  
A|BCD  
AB|CD  
A|CD  
AC|D  
ACE|D  
ACED|  
ACEDF|

8. 

```
ListIterator<String> iter = words.iterator();
while (iter.hasNext())
{
    String str = iter.next();
    if (str.length() < 4) { iter.remove(); }
}
```

9. 

```
ListIterator<String> iter = words.iterator();
while (iter.hasNext())
{
    System.out.println(iter.next());
    if (iter.hasNext())
    {
        iter.next(); // Skip the next element
    }
}
```

10. Adding and removing elements as well as testing for membership is more efficient with sets.

11. Sets do not have an ordering, so it doesn't make sense to add an element at a particular iterator position, or to traverse a set backward.

12. You do not know in which order the set keeps the elements.

13. Here is one possibility:

```
if (s.size() == 3 && s.contains("Tom")
    && s.contains("Diana")
    && s.contains("Harry"))
    . . .
```

14. 

```
for (String str : s)
{
    if (t.contains(str))
    {
        System.out.println(str);
    }
}
```

15. The words would be listed in sorted order.

16. A set stores elements. A map stores associations between keys and values.

17. The ordering does not matter, and you cannot have duplicates.

18. Because it might have duplicates.

19. `Map<String, Integer> wordFrequency;`  
Note that you cannot use a `Map<String, int>` because you cannot use primitive types as type parameters in Java.

20. It associates strings with sets of strings. One application would be a thesaurus that lists synonyms for a given word. For example, the key "improve" might have as its value the set ["ameliorate", "better", "enhance", "enrich", "perfect", "refine"].

21. This way, we can ensure that only queue operations can be invoked on the `q` object.

22. Depending on whether you consider the 0 position the head or the tail of the queue, you would either add or remove elements at that position. Both are inefficient operations because all other elements need to be moved.

23. A B C

24. Stacks use a "last-in, first-out" discipline. If you are the first one to submit a print job and lots of people add print jobs before the printer has a chance to deal with your job, they get their printouts first, and you have to wait until all other jobs are completed.

25. Yes—the smallest string (in lexicographic ordering) is removed first. In the example, that is the string starting with 1, then the string starting with 2, and so on. However, the scheme breaks down if a priority value exceeds 9; a string "10 - Line up braces" comes before "2 - Order supplies" in lexicographic order.

26. 70.

27. It would then subtract the first argument from the second. Consider the input 5 3 -. The stack contains 5 and 3, with the 3 on the top. Then `results.pop() - results.pop()` computes  $3 - 5$ .

28. The `-` gets executed first because `+` doesn't have a higher precedence.

29. No, because there may be parentheses on the stack. The parentheses separate groups of operators, each of which is in increasing precedence.

30. A B E F G D C K J N



## WORKED EXAMPLE 15.1



## Word Frequency

**Problem Statement** Write a program that reads a text file and prints a list of all words in the file in alphabetical order, together with a count that indicates how often each word occurred in the file.

For example, the following is the beginning of the output that results from processing the book *Alice in Wonderland*:

a	653
abide	1
able	1
about	97
above	4
absence	1
absurd	2

**Step 1** Determine how you access the values.

In our case, the values are the word frequencies. We have a frequency value for every word. That is, we want to use a map that maps words to frequencies.

**Step 2** Determine the element types of keys and values.

Each word is a String and each frequency is an Integer. (You cannot use an int as a type parameter because it is a primitive type.) Therefore, we need a Map<String, Integer>.

**Step 3** Determine whether element or key order matters.

We are supposed to print the words in sorted order, so we will use a TreeMap.

**Step 4** For a collection, determine which operations must be efficient.

We skip this step because we use a map, not a collection.

**Step 5** For hash sets and maps, decide what to do about the equals and hashCode methods.

We skip this step because we use a tree map.

**Step 6** If you use a tree, decide whether to supply a comparator.

The key type for our tree map is String, which implements the Comparable interface. Therefore, we need to do nothing further.

We have now chosen our collection. The program for completing our task is fairly simple. Here is the pseudocode:

```

For each word in the input file
  Remove non-letters (such as punctuation marks) from the word.
  If the word is already present in the frequencies map
    Increment the frequency.
  Else
    Set the frequency to 1.
  
```

Here is the program code:

### worked\_example\_1/WordFrequency.java

```

1 import java.util.Map;
2 import java.util.Scanner;
3 import java.util.TreeMap;
4 import java.io.File;
5 import java.io.FileNotFoundException;
  
```

```
6  /**
7  * This program prints the frequencies of all words in “Alice in Wonderland”.
8  */
9  public class WordFrequency
10 {
11     public static void main(String[] args)
12         throws FileNotFoundException
13     {
14         Map<String, Integer> frequencies = new TreeMap<>();
15         Scanner in = new Scanner(new File("alice30.txt"));
16         while (in.hasNext())
17         {
18             String word = clean(in.next());
19
20             // Get the old frequency count
21
22             Integer count = frequencies.get(word);
23
24             // If there was none, put 1; otherwise, increment the count
25
26             if (count == null) { count = 1; }
27             else { count = count + 1; }
28
29             frequencies.put(word, count);
30         }
31
32         // Print all words and counts
33
34         for (String key : frequencies.keySet())
35         {
36             System.out.printf("%-20s%10d\n", key, frequencies.get(key));
37         }
38     }
39
40     /**
41      * Removes characters from a string that are not letters.
42      * @param s a string
43      * @return a string with all the letters from s
44      */
45     public static String clean(String s)
46     {
47         String r = "";
48         for (int i = 0; i < s.length(); i++)
49         {
50             char c = s.charAt(i);
51             if (Character.isLetter(c))
52             {
53                 r = r + c;
54             }
55         }
56         return r.toLowerCase();
57     }
58 }
59 }
```



## WORKED EXAMPLE 15.2

**Simulating a Queue of Waiting Customers**

A good application of object-oriented programming is simulation. In fact, the first object-oriented language, Simula, was designed with this application in mind. One can simulate the activities of air molecules around an aircraft wing, of customers in a supermarket, or of vehicles on a road system. The goal of a simulation is to observe how changes in the design affect the behavior of a system. Modifying the shape of a wing, the location and staffing of cash registers, or the synchronization of traffic lights has an effect on turbulence in the air stream, customer satisfaction, or traffic throughput. Modeling these systems in the computer is far cheaper than running actual experiments.

**Kinds of Simulation**

Simulations fall into two broad categories. A *continuous simulation* constantly updates all objects in a system. A simulated clock advances in seconds or some other suitable constant time interval. At every clock tick, each object is moved or updated in some way. Consider the simulation of traffic along a road. Each car has some position, velocity, and acceleration. Its position needs to be updated with every clock tick. If the car gets too close to an obstacle, it must decelerate. The new position may be displayed on the screen.

In contrast, in a *discrete event simulation*, time advances in chunks. All interesting events are kept in a priority queue, sorted by the time in which they are to happen. As soon as one event has completed, the clock jumps to the time of the next event to be executed.

To see the contrast between these two simulation styles, consider the updating of a traffic light. Suppose the traffic light just turned red, and it will turn green again in 30 seconds. In a continuous model, the traffic light is visited every second, and a counter variable is decremented. Once the counter reaches 0, the color changes. In a discrete model, the traffic light schedules an event to be notified 30 seconds from now. For 29 seconds, the traffic light is not bothered at all, and then it receives a message to change its state. Discrete event simulation avoids “busy waiting”.

In this Worked Example, you will see how to use queues and priority queues in a discrete event simulation of customers at a bank. The simulation makes use of two generic classes, `Event` and `Simulation`, that are useful for any discrete event simulation. We use inheritance to extend these classes to make classes that simulate the bank.

**Events**

A discrete event simulation generates, stores, and processes events. Each event has a time stamp indicating when it is to be executed. Each event has some action associated with it that must be carried out at that time. Beyond these properties, the scheduler has no concept of what an event represents. Of course, actual events must carry with them some information. For example, the event notifying a traffic light of a state change must know which traffic light to notify.

To do so, we will have all events extend a common superclass, `Event`. An `Event` object has an instance variable to indicate at which time it should be processed. When that time has arrived, the event’s `process` method is called. This method may move objects around, update information, and schedule additional events.

The `Event` class also implements the `Comparable` interface. An event is considered more urgent than another if its processing time is earlier.

```
public class Event implements Comparable<Event>
{
    private double time;

    public Event(double eventTime)
    {
```

```

        time = eventTime;
    }

    public void process(Simulation sim) {}
    public double getTime() { return time; }

    public int compareTo(Event other)
    {
        if (time < other.time) { return -1; }
        else if (time > other.time) { return 1; }
        else { return 0; }
    }
}

```

### The Simulation Class

In any discrete event simulation, events are kept in a priority queue. After initialization, the simulation enters an event loop in which events are retrieved from the priority queue in the order specified by their time stamp. The simulated time is advanced to the time stamp of the event, and the event is processed according to its `process` method. To simulate a specific activity, such as customer activity in a bank, extend the `Simulation` class and provide methods for displaying the current state after each event, and a summary after the completion of the simulation.

```

public class Simulation
{
    private PriorityQueue<Event> eventQueue;
    private double currentTime;
    . .
    public void display() {}
    public void displaySummary() {}
    . .
}

```

Here is the event loop in the `Simulation` class:

```

public void run(double startTime, double endTime)
{
    currentTime = startTime;

    while (eventQueue.size() > 0 && currentTime <= endTime)
    {
        Event event = eventQueue.remove();
        currentTime = event.getTime();
        event.process(this);
        display();
    }
    displaySummary();
}

```

In the `Simulation` class, we provide a utility method for generating reasonable random values for the time between two independent events. These random time differences can be modeled with an “exponential distribution”, as follows: Let  $m$  be the mean time between arrivals. Let  $u$  be a random value that can, with equal probability, assume any floating-point value between 0 inclusive and 1 exclusive. Then inter-arrival times can be generated as

$$a = -m \log(1 - u)$$

where  $\log$  is the natural logarithm. The utility method `expdist` computes these random values:

```

public static double expdist(double mean)
{
    return -mean * Math.log(1 - Math.random());
}

```

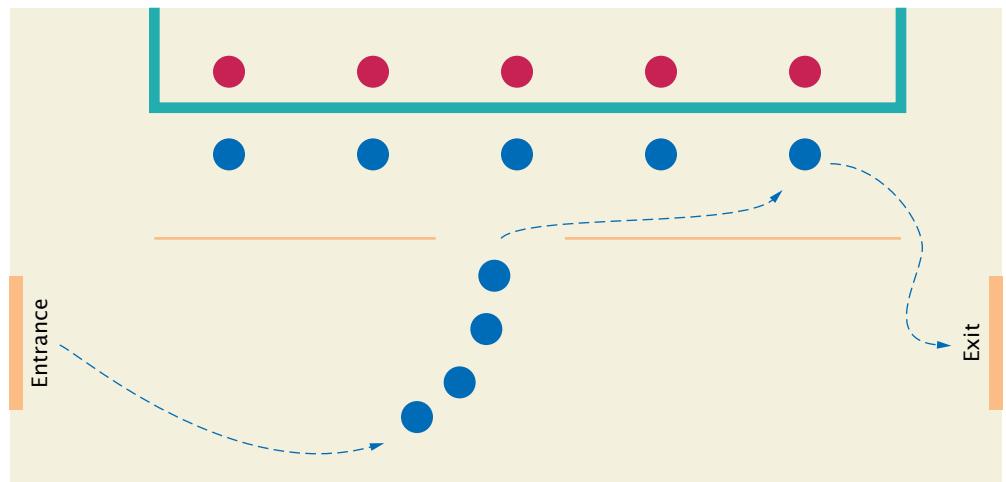
```
}
```

If a customer arrives at time  $t$ , the program can schedule the next customer arrival at  $t + \text{expdist}(m)$ .

Processing time is also exponentially distributed, with a different average. In this simulation we assume that, on average, one minute elapses between customer arrivals, and customer transactions require an average of five minutes.

### The Bank

The following figure shows the layout of the bank. Customers enter the bank. If there is a queue, they join the queue; otherwise they move up to a teller. When a customer has completed a teller transaction, the time spent in the bank is logged, the customer is removed, and the next customer in the queue moves up to the teller.



The `BankSimulation` class keeps an array of tellers as well as a queue to hold waiting customers. The queue is not a priority queue but a regular FIFO (first-in, first-out) queue:

```
public class BankSimulation extends Simulation
{
    private Customer[] tellers;
    private Queue<Customer> custQueue;

    private int totalCustomers;
    private double totalTime;

    private static final double INTERARRIVAL = 1;
        // average of 1 minute between customer arrivals
    private static final double PROCESSING = 5;
        // average of 5 minutes processing time per customer
    . .
}
```

It also keeps track of the total number of customers that have been serviced, and the total amount of time they spent in the bank (both in the waiting queue and in front of a teller.)

Teller  $i$  is busy if `tellers[i]` holds a reference to a `Customer` object and available if it is `null`.

When a customer is added to the bank, the program first checks whether a teller is available to handle the customer. If not, the customer is added to the waiting queue:

```

public void add(Customer c)
{
    boolean addedToTeller = false;
    for (int i = 0; !addedToTeller && i < tellers.length; i++)
    {
        if (tellers[i] == null)
        {
            addToTeller(i, c);
            addedToTeller = true;
        }
    }
    if (!addedToTeller) { custQueue.add(c); }

    addEvent(new Arrival(getCurrentTime() + expdist(INTERARRIVAL)));
}

```

In addition, the simulation must ensure that customers keep coming. We know the next customer will arrive in about one minute, but it may be a bit earlier or, occasionally, a lot later. To obtain a random time, we call `expdist(INTERARRIVAL)`. Of course, we cannot wait around for that to happen, because other events will be going on in the meantime. Therefore when a customer is added, another arrival event is scheduled to occur when this random time has elapsed.

Similarly, when a customer steps up to a teller, the average transaction will be five minutes. We need to schedule a departure event that removes the customer from the bank. This happens in the `addToTeller` method:

```

private void addToTeller(int i, Customer c)
{
    tellers[i] = c;
    addEvent(new Departure(getCurrentTime() + expdist(PRECESSING), i));
}

```

When the departure event is processed, it will notify the bank to remove the customer. The bank simulation removes the customer and keeps track of the total amount of time the customer spent in the waiting queue and with the teller. This makes the teller available to service the next customer from the waiting queue. If there is a queue, we add the first customer to this teller:

```

public void remove(int i)
{
    Customer c = tellers[i];
    tellers[i] = null;

    // Update statistics
    totalCustomers++;
    totalTime = totalTime + getCurrentTime() - c.getArrivalTime();

    if (custQueue.size() > 0)
    {
        addToTeller(i, custQueue.remove());
    }
}

```

## Event Classes

The classes `Arrival` and `Departure` are subclasses of `Event`.

When a new customer is to arrive at the bank, an arrival event is processed. The processing action of that event has the responsibility of making a customer and adding it to the bank.

```

public class Arrival extends Event
{

```

```

public Arrival(double time) { super(time); }

public void process(Simulation sim)
{
    double now = sim.getCurrentTime();
    BankSimulation bank = (BankSimulation) sim;
    Customer c = new Customer(now);
    bank.add(c);
}
}

```

Departures remember not only the departure time but also the teller from whom a customer is to depart. To process a departure event, we remove the customer from the teller.

```

public class Departure extends Event
{
    private int teller;

    public Departure(double time, int teller)
    {
        super(time);
        this.teller = teller;
    }
    public void process(Simulation sim)
    {
        BankSimulation bank = (BankSimulation) sim;
        bank.remove(teller);
    }
}

```

## Running the Simulation

To run the simulation, we first construct a `BankSimulation` object with five tellers. The most important task in setting up the simulation is to get the flow of events going. At the outset, the event queue is empty. We will schedule the arrival of a customer at the start time (9 A.M.). Because the processing of an arrival event schedules the arrival of each successor, the insertion of the arrival event for the first customer takes care of the generation of all arrivals. Once customers arrive at the bank, they are added to tellers, and departure events are generated.

Here is the `main` method:

```

public static void main(String[] args)
{
    final double START_TIME = 9 * 60; // 9 A.M.
    final double END_TIME = 17 * 60; // 5 P.M.

    final int NTELLERS = 5;

    Simulation sim = new BankSimulation(NTELLERS);
    sim.addEvent(new Arrival(START_TIME));
    sim.run(START_TIME, END_TIME);
}

```

Here is a typical program run. The bank starts out with empty tellers, and customers start dropping in:

```

.....<
C....<
CC...<
CCC..<
CCCC.<
C.CC.<

```

## WE8 Chapter 15 The Java Collections Framework

```
CCCC.<
CCCCC<
CCCCC<C
CCCCC<
C.CCC<
```

Due to the random fluctuations of customer arrival and processing, the queue can get quite long:

```
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCC
```

```
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCC
```

At other times, the bank is empty again:

```
CCC.C<
CCC..<
CC...<
.C...<
....<
C....<
```

This particular run of the simulation ends up with the following statistics:

```
457 customers. Average time 15.28 minutes.
```

If you are the bank manager, this result is quite depressing. You hired enough tellers to take care of all customers. (Every hour, you need to serve, on average, 60 customers. Their transactions take an average of 5 minutes each; that is 300 teller-minutes, or 5 teller-hours. Hence, hiring five tellers should be just right.) Yet the average customer had to wait in line more than 10 minutes, twice as long as their transaction time. This is an average, so some customers had to wait even longer. If disgruntled customers hurt your business, you may have to hire more tellers and pay them for being idle some of the time.

(See the ch15/worked\_example\_2 folder in your companion code for the complete bank simulation program.)

### worked\_example\_2/BankSimulation.java

```
1 import java.util.LinkedList;
2 import java.util.Queue;
3
4 /**
5  * Simulation of customer traffic in a bank.
6 */
7 public class BankSimulation extends Simulation
8 {
9     private Customer[] tellers;
10    private Queue<Customer> custQueue;
11
12    private int totalCustomers;
13    private double totalTime;
14
15    private static final double INTERARRIVAL = 1;
16        // average of 1 minute between customer arrivals
```

```

17     private static final double PROCESSING = 5;
18     // average of 5 minutes processing time per customer
19
20     public BankSimulation(int numberOfTellers)
21     {
22         tellers = new Customer[numberOfTellers];
23         custQueue = new LinkedList<>();
24         totalCustomers = 0;
25         totalTime = 0;
26     }
27
28     /**
29      * Adds a customer to the bank.
30      * @param c the customer
31     */
32     public void add(Customer c)
33     {
34         boolean addedToTeller = false;
35         for (int i = 0; !addedToTeller && i < tellers.length; i++)
36         {
37             if (tellers[i] == null)
38             {
39                 addToTeller(i, c);
40                 addedToTeller = true;
41             }
42         }
43         if (!addedToTeller) { custQueue.add(c); }
44
45         addEvent(new Arrival(getCurrentTime() + expdist(INTERARRIVAL)));
46     }
47
48     /**
49      * Adds a customer to a teller and schedules the departure event.
50      * @param i the teller number
51      * @param c the customer
52     */
53     private void addToTeller(int i, Customer c)
54     {
55         tellers[i] = c;
56         addEvent(new Departure(getCurrentTime() + expdist(PROCESSING), i));
57     }
58
59     /**
60      * Removes a customer from a teller.
61      * @param i teller position
62     */
63     public void remove(int i)
64     {
65         Customer c = tellers[i];
66         tellers[i] = null;
67
68         // Update statistics
69         totalCustomers++;
70         totalTime = totalTime + getCurrentTime() - c.getArrivalTime();
71
72         if (custQueue.size() > 0)
73         {
74             addToTeller(i, custQueue.remove());
75         }
76     }

```

```
77  /**
78   * Displays tellers and queue.
79   */
80  public void display()
81  {
82      for (int i = 0; i < tellers.length; i++)
83      {
84          if (tellers[i] == null)
85          {
86              System.out.print(".");
87          }
88          else
89          {
90              System.out.print("C");
91          }
92      }
93      System.out.print("<");
94      int q = custQueue.size();
95      for (int j = 1; j <= q; j++) { System.out.print("C"); }
96      System.out.println();
97  }
98
99
100 /**
101  * Displays a summary of the gathered statistics.
102 */
103 public void displaySummary()
104 {
105     double averageTime = 0;
106     if (totalCustomers > 0)
107     {
108         averageTime = totalTime / totalCustomers;
109     }
110     System.out.println(totalCustomers + " customers. Average time "
111                     + averageTime + " minutes.");
112 }
113 }
```

# BASIC DATA STRUCTURES

## CHAPTER GOALS

To understand the implementation of linked lists and array lists

To analyze the efficiency of fundamental operations of lists and arrays

To implement the stack and queue data types

To implement a hash table and understand the efficiency of its operations



© andrea laurita/iStockphoto.

## CHAPTER CONTENTS

### 16.1 IMPLEMENTING LINKED LISTS 722

**ST1** Static Classes 736

**WE1** Implementing a Doubly-Linked List

### 16.2 IMPLEMENTING ARRAY LISTS 737

### 16.3 IMPLEMENTING STACKS AND QUEUES 741

### 16.4 IMPLEMENTING A HASH TABLE 747

**ST2** Open Addressing 755



© andrea laurita/iStockphoto.

In the preceding chapter, you learned how to use the collection classes in the Java library. In this and the next chapter, we will study how these classes are implemented. This chapter deals with simple data structures in which elements are arranged in a linear sequence. By investigating how these data structures add, remove, and locate elements, you will gain valuable experience in designing algorithms and estimating their efficiency.

## 16.1 Implementing Linked Lists

In Chapter 15 you saw how to use the linked list class supplied by the Java library. Now we will look at the implementation of a simplified version of this class. This will show you how the list operations manipulate the links as the list is modified.

To keep this sample code simple, we will not implement all methods of the linked list class. We will implement only a singly-linked list, and the list class will supply direct access only to the first list element, not the last one. (A worked example and several exercises explore additional implementation options.) Our list will not use a type parameter. We will simply store raw `Object` values and insert casts when retrieving them. (You will see how to use type parameters in Chapter 18.) The result will be a fully functional list class that shows how the links are updated when elements are added or removed, and how the iterator traverses the list.

### 16.1.1 The Node Class

A linked list stores elements in a sequence of nodes. We need a class to represent the nodes. In a singly-linked list, a `Node` object stores an element and a reference to the next node.

Because the methods of both the linked list class and the iterator class have frequent access to the `Node` instance variables, we do not make the instance variables of the `Node` class private. Instead, we make `Node` a **private inner class** of the `LinkedList` class. An inner class is a class that is defined inside another class. The methods of the outer class can access the public features of the inner class. However, because the inner class is private, it cannot be accessed anywhere other than from the outer class.

```
public class LinkedList
{
    . . .
    class Node
    {
        public Object data;
        public Node next;
    }
}
```

A linked list object holds a reference to the first node, and each node object holds a reference to the next node.

Our `LinkedList` class holds a reference `first` to the first node (or `null`, if the list is completely empty):

```
public class LinkedList
{
    private Node first;
```

```

public LinkedList() { first = null; }

public Object getFirst()
{
    if (first == null) { throw new NoSuchElementException(); }
    return first.data;
}

```

### 16.1.2 Adding and Removing the First Element

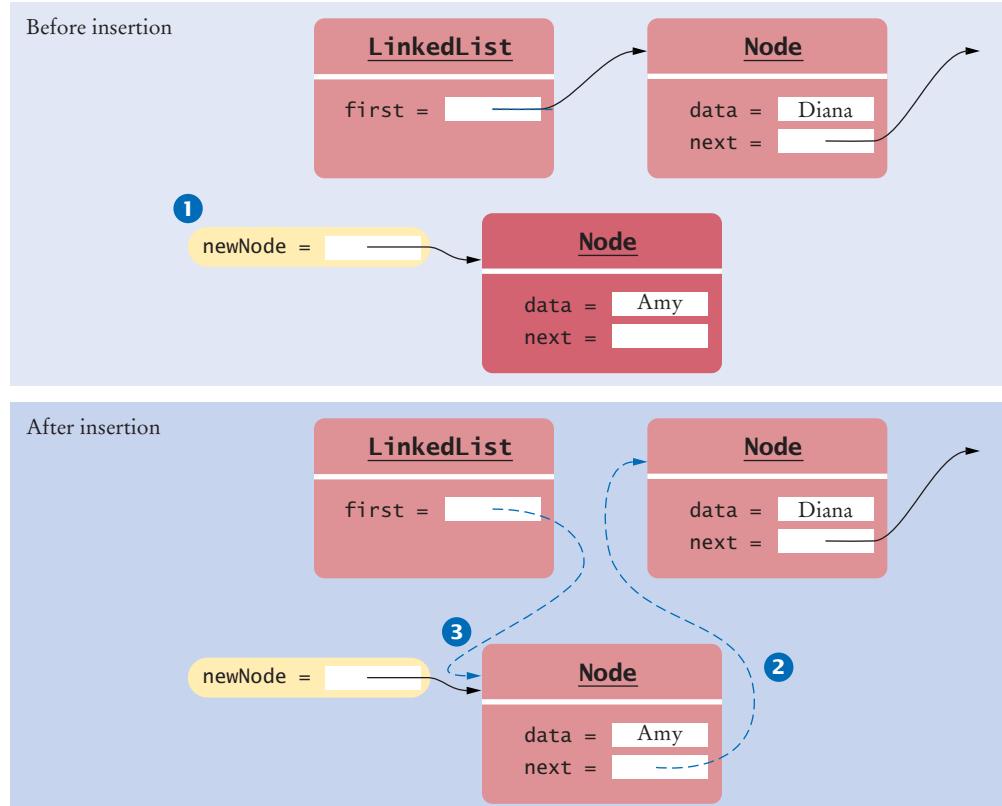
When adding or removing the first element, the reference to the first node must be updated.

Figure 1 shows the `addFirst` method in action. When a new node is added, it becomes the head of the list, and the node that was the old list head becomes its next node:

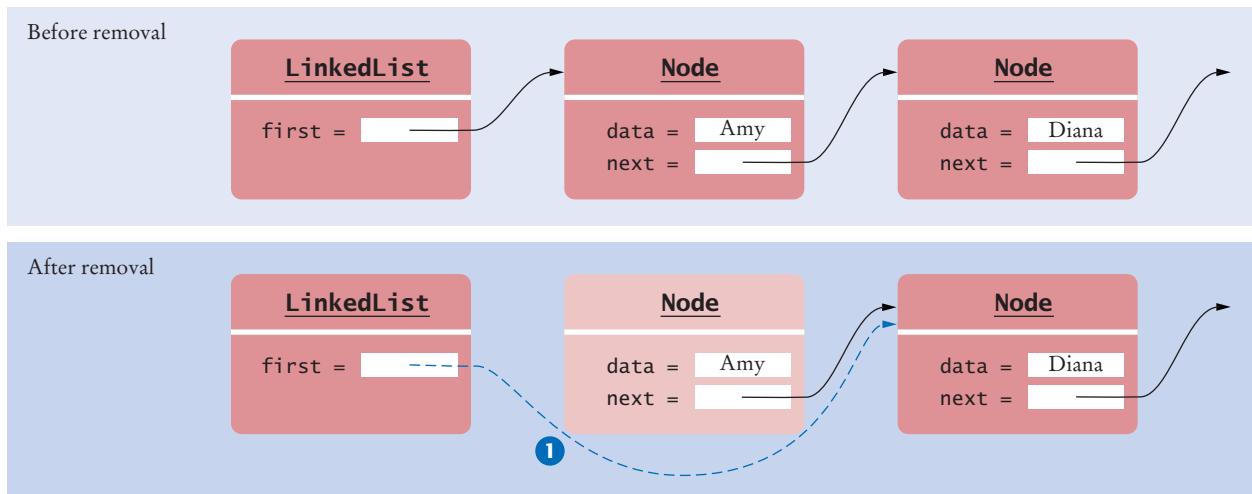
```

public class LinkedList
{
    ...
    public void addFirst(Object element)
    {
        Node newNode = new Node(); ①
        newNode.data = element;
        newNode.next = first; ②
        first = newNode; ③
    }
    ...
}

```



**Figure 1**  
Adding a Node  
to the Head of a  
Linked List



**Figure 2** Removing the First Node from a Linked List

Removing the first element of the list works as follows. The data of the first node are saved and later returned as the method result. The successor of the first node becomes the first node of the shorter list (see Figure 2). Then there are no further references to the old node, and the garbage collector will eventually recycle it.

```
public class LinkedList
{
    . . .
    public Object removeFirst()
    {
        if (first == null) { throw new NoSuchElementException(); }
        Object element = first.data;
        first = first.next; ①
        return element;
    }
    . . .
}
```

### 16.1.3 The Iterator Class

The `ListIterator` interface in the standard library declares nine methods. Our simplified `ListIterator` interface omits four of them (the methods that move the iterator backward and the methods that report an integer index of the iterator). Our interface requires us to implement list iterator methods `next`, `hasNext`, `remove`, `add`, and `set`.

Our `LinkedList` class declares a private inner class `LinkedListIterator`, which implements our simplified `ListIterator` interface. Because `LinkedListIterator` is an inner class, it has access to the private features of the `LinkedList` class—in particular, the instance variable `first` and the private `Node` class.

Note that clients of the `LinkedList` class don't actually know the name of the iterator class. They only know it is a class that implements the `ListIterator` interface.

Each iterator object has a reference, `position`, to the currently visited node. We also store a reference to the last node before that, `previous`. We will need that reference to adjust the links properly in the `remove` method. Finally, because calls to `remove` and `set`

A list iterator object has a reference to the last visited node.

are only valid after a call to `next`, we use the `isAfterNext` flag to track when the `next` method has been called.

```
public class LinkedList
{
    . .
    public ListIterator listIterator()
    {
        return new LinkedListIterator();
    }

    class LinkedListIterator implements ListIterator
    {
        private Node position;
        private Node previous;
        private boolean isAfterNext;

        public LinkedListIterator()
        {
            position = null;
            previous = null;
            isAfterNext = false;
        }
        . .
    }
}
```

### 16.1.4 Advancing an Iterator

To advance an iterator, update the position and remember the old position for the remove method.

When advancing an iterator with the `next` method, the `position` reference is updated to `position.next`, and the old position is remembered in `previous`. The previous position is used for just one purpose: to remove the element if the `remove` method is called after the `next` method.

There is a special case, however—if the iterator points before the first element of the list, then the old position is `null`, and `position` must be set to `first`:

```
class LinkedListIterator implements ListIterator
{
    . .
    public Object next()
    {
        if (!hasNext()) { throw new NoSuchElementException(); }
        previous = position; // Remember for remove
        isAfterNext = true;

        if (position == null)
        {
            position = first;
        }
        else
        {
            position = position.next;
        }

        return position.data;
    }
    . .
}
```

The `next` method is supposed to be called only when the iterator is not yet at the end of the list, so we declare the `hasNext` method accordingly. The iterator is at the end if the list is empty (that is, `first == null`) or if there is no element after the current position (`position.next == null`):

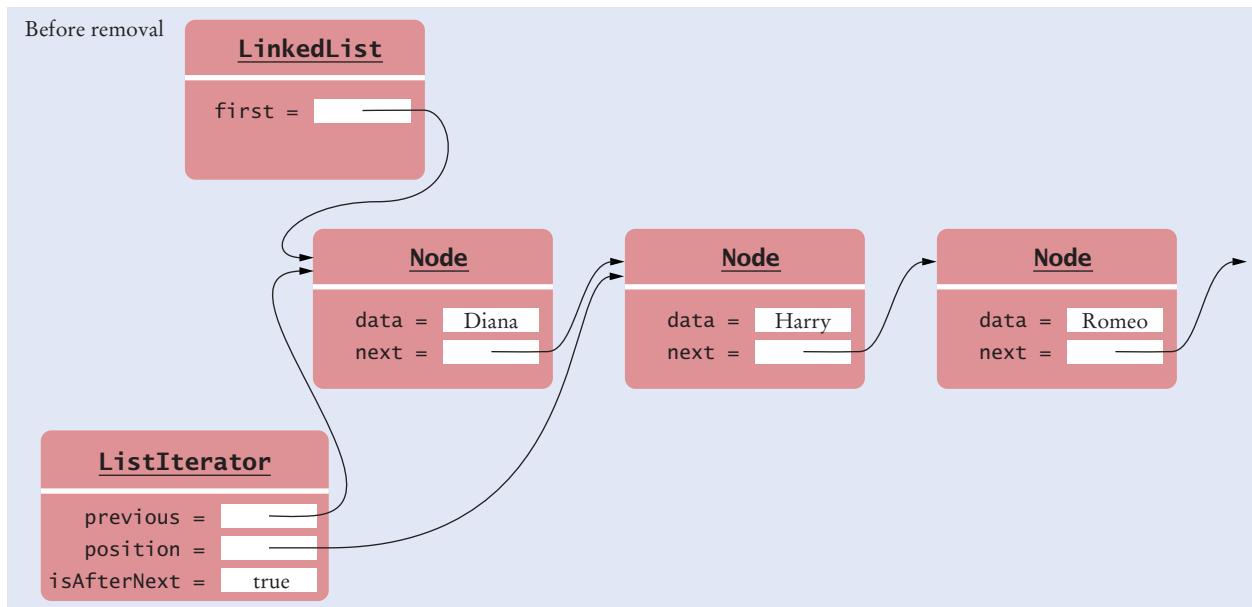
```
class LinkedListIterator implements ListIterator
{
    . .
    public boolean hasNext()
    {
        if (position == null)
        {
            return first != null;
        }
        else
        {
            return position.next != null;
        }
    }
    . .
}
```

### 16.1.5 Removing an Element

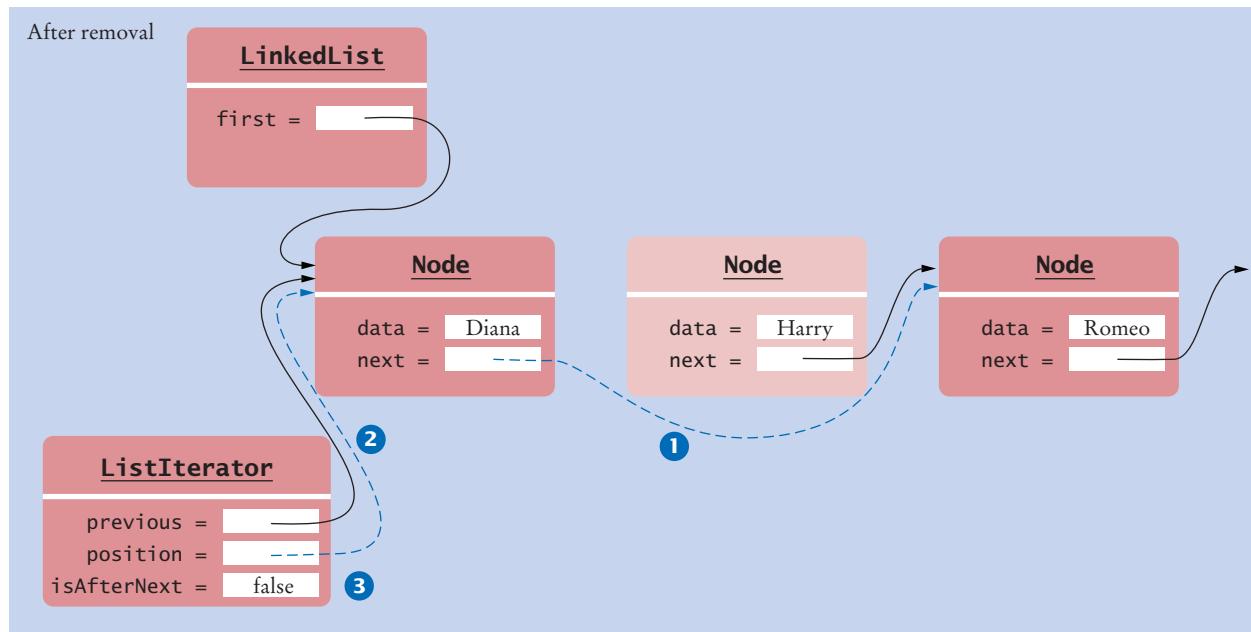
Next, we implement the `remove` method of the list iterator. Recall that, in order to remove an element, one must first call `next` and then call `remove` on the iterator.

If the element to be removed is the first element, we just call `removeFirst`. Otherwise, an element in the middle of the list must be removed, and the node preceding it needs to have its `next` reference updated to skip the removed element (see Figure 3).

We also need to update the `position` reference so that a subsequent call to the `next` method skips over the element after the removed one.



**Figure 3** Removing a Node from the Middle of a Linked List



**Figure 3 (continued)** Removing a Node from the Middle of a Linked List

According to the specification of the `remove` method, it is illegal to call `remove` twice in a row. Our implementation handles this situation correctly. After completion of the `remove` method, the `isAfterNext` flag is set to `false`. An exception occurs if `remove` is called again without another call to `next`.

```
class LinkedListIterator implements ListIterator
{
    ...
    public void remove()
    {
        if (!isAfterNext) { throw new IllegalStateException(); }

        if (position == first)
        {
            removeFirst();
        }
        else
        {
            previous.next = position.next; ①
        }
        position = previous; ②

        isAfterNext = false; ③
    }
    ...
}
```

There is a good reason for disallowing `remove` twice in a row. After the first call to `remove`, the current position reverts to the predecessor of the removed element. Its predecessor is no longer known, which makes it impossible to efficiently remove the current element.

### 16.1.6 Adding an Element

The add method of the iterator inserts the new node after the last visited node (see Figure 4).

After adding the new element, we set the `isAfterNext` flag to `false`, in order to disallow a subsequent call to the `remove` or `set` method.

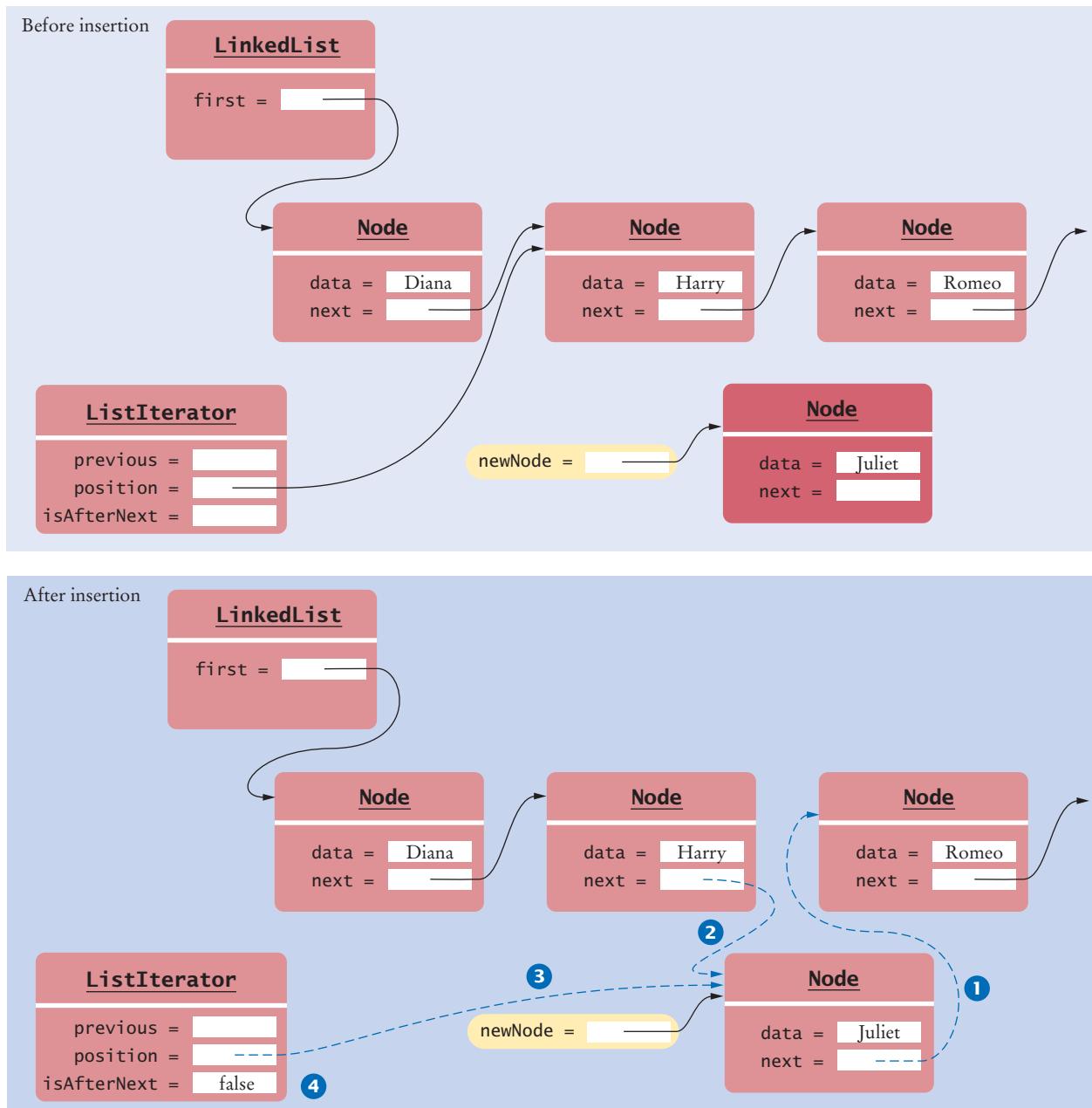


Figure 4 Adding a Node to the Middle of a Linked List

```

class LinkedListIterator implements ListIterator
{
    . . .
    public void add(Object element)
    {
        if (position == null)
        {
            addFirst(element);
            position = first;
        }
        else
        {
            Node newNode = new Node();
            newNode.data = element;
            newNode.next = position.next; ①
            position.next = newNode; ②
            position = newNode; ③
        }
        isAfterNext = false; ④
    }
    . . .
}

```

### 16.1.7 Setting an Element to a Different Value

The `set` method changes the data stored in the previously visited element:

```

public void set(Object element)
{
    if (!isAfterNext) { throw new IllegalStateException(); }
    position.data = element;
}

```

As with the `remove` method, a call to `set` is only valid if it was preceded by a call to the `next` method. We throw an exception if we find that there was a call to `add` or `remove` immediately before calling `set`.

You will find the complete implementation of our `LinkedList` class after the next section.

### 16.1.8 Efficiency of Linked List Operations

In a doubly-linked list, accessing an element is an  $O(n)$  operation; adding and removing an element is  $O(1)$ .

Now that you have seen how linked list operations are implemented, we can determine their efficiency.

Consider first the cost of accessing an element. To get the  $k$ th element of a linked list, you start at the beginning of the list and advance the iterator  $k$  times. Suppose it takes an amount of time  $T$  to advance the iterator once. This quantity is independent of the iterator position—advancing an iterator does some checking and then it follows the next reference of the current node (see Section 16.1.4).

Therefore, advancing the iterator to the  $k$ th element consumes  $kT$  time. If the linked list has  $n$  elements and  $k$  is chosen at random, then  $k$  will average out to be  $n/2$ , and  $kT$  is on average  $nT/2$ . Because  $T/2$  is a constant, this is an  $O(n)$  expression. We have determined that accessing an element in a linked list of length  $n$  is an  $O(n)$  operation.

Now consider the cost of adding an element at a given position, assuming that we already have an iterator to the position. Look at the implementation of the `add`



To get to the  $k$ th node of a linked list, one must skip over the preceding nodes.

method in Section 16.1.6. To add an element, one updates a couple of references in the neighboring nodes and the iterator. This operation requires a constant number of steps, independent of the size of the linked list.

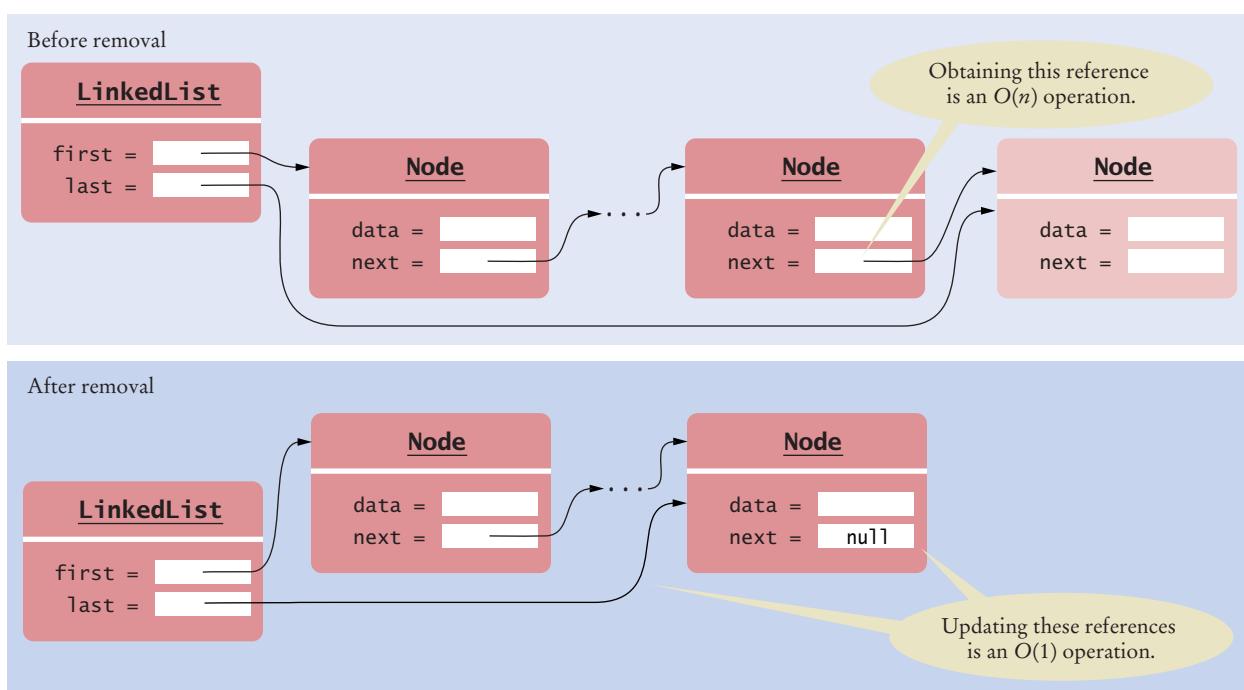
Using the big-Oh notation, an operation that requires a bounded amount of time, regardless of the total number of elements in the structure, is denoted as  $O(1)$ . Adding an element to a linked list takes  $O(1)$  time.

Similar reasoning shows that removing an element at a given position is an  $O(1)$  operation.

Now consider the task of adding an element at the end of the list. We first need to get to the end, at a cost of  $O(n)$ . Then it takes  $O(1)$  time to add the element. However, we can improve on this performance if we add a reference to the last node to the `LinkedList` class:

```
public class LinkedList
{
    private Node first;
    private Node last;
    . . .
}
```

Of course, this reference must be updated when the last node changes, as elements are added or removed. In order to keep the code as simple as possible, our implementation does not have a reference to the last node. However, we will always assume that a linked list implementation can access the last element in constant time. This is the case for the `LinkedList` class in the standard Java library, and it is an easy enhancement to our implementation. Worked Example 16.1 shows how to add the `last` reference, update it as necessary, and provide an `addLast` method for adding an element at the end.



**Figure 5** Removing the Last Element of a Singly-Linked List

The code for the `addLast` method is very similar to the `addFirst` method in Section 16.1.2. It too requires constant time, independent of the length of the list. We conclude that, with an appropriate implementation, adding an element at the end of a linked list is an  $O(1)$  operation.

How about removing the last element? We need a reference to the next-to-last element, so that we can set its `next` reference to `null`. (See Figure 5.)

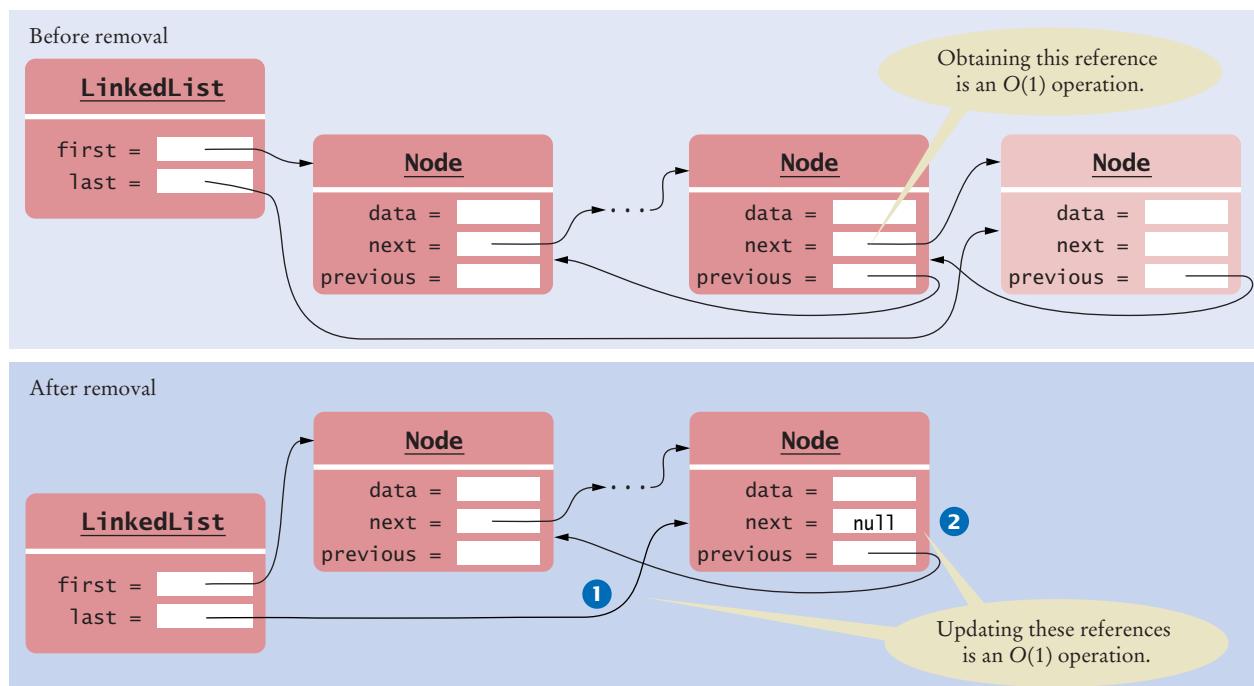
We also need to update the `last` reference and set it to the next-to-last reference. But how can we get that next-to-last reference? It takes  $n - 1$  iterations to obtain it, starting at the beginning of the list. Thus, removing an element from the back of a singly-linked list is an  $O(n)$  operation.

We can do better in a doubly-linked list, such as the one in the standard Java library. In a doubly-linked list, each node has a reference to the previous node in addition to the next one (see Figure 6).

```
public class LinkedList
{
    ...
    class Node
    {
        public Object data;
        public Node next;
        public Node previous;
    }
}
```

In that case, removal of the last element takes a constant number of steps:

```
last = last.previous; ①
last.next = null; ②
```



**Figure 6** Removing the Last Element of a Doubly-Linked List

Therefore, removing an element from the end of a doubly-linked list is also an  $O(1)$  operation. Worked Example 16.1 contains a full implementation.

Table 1 summarizes the efficiency of linked list operations.

**Table 1 Efficiency of Linked List Operations**

Operation	Singly-Linked List	Doubly-Linked List
Access an element.	$O(n)$	$O(n)$
Add/remove at an iterator position.	$O(1)$	$O(1)$
Add/remove first element.	$O(1)$	$O(1)$
Add last element.	$O(1)$	$O(1)$
Remove last element.	$O(n)$	$O(1)$

### section\_1/LinkedList.java

```

1 import java.util.NoSuchElementException;
2
3 /**
4  * A linked list is a sequence of nodes with efficient
5  * element insertion and removal. This class
6  * contains a subset of the methods of the standard
7  * java.util.LinkedList class.
8 */
9 public class LinkedList
10 {
11     private Node first;
12
13     /**
14      * Constructs an empty linked list.
15     */
16     public LinkedList()
17     {
18         first = null;
19     }
20
21     /**
22      * Returns the first element in the linked list.
23      * @return the first element in the linked list
24     */
25     public Object getFirst()
26     {
27         if (first == null) { throw new NoSuchElementException(); }
28         return first.data;
29     }
30
31     /**
32      * Removes the first element in the linked list.
33      * @return the removed element
34     */
35     public Object removeFirst()
36     {
37         if (first == null) { throw new NoSuchElementException(); }
38         Object element = first.data;

```

```
39         first = first.next;
40         return element;
41     }
42
43     /**
44      Adds an element to the front of the linked list.
45      @param element the element to add
46     */
47     public void addFirst(Object element)
48     {
49         Node newNode = new Node();
50         newNode.data = element;
51         newNode.next = first;
52         first = newNode;
53     }
54
55     /**
56      Returns an iterator for iterating through this list.
57      @return an iterator for iterating through this list
58     */
59     public ListIterator listIterator()
60     {
61         return new LinkedListIterator();
62     }
63
64     class Node
65     {
66         public Object data;
67         public Node next;
68     }
69
70     class LinkedListIterator implements ListIterator
71     {
72         private Node position;
73         private Node previous;
74         private boolean isAfterNext;
75
76         /**
77          Constructs an iterator that points to the front
78          of the linked list.
79         */
80         public LinkedListIterator()
81         {
82             position = null;
83             previous = null;
84             isAfterNext = false;
85         }
86
87         /**
88          Moves the iterator past the next element.
89          @return the traversed element
90         */
91         public Object next()
92         {
93             if (!hasNext()) { throw new NoSuchElementException(); }
94             previous = position; // Remember for remove
95             isAfterNext = true;
96
97             if (position == null)
98             {
```

```

    position = first;
}
else
{
    position = position.next;
}

return position.data;
}

<**
   Tests if there is an element after the iterator position.
   @return true if there is an element after the iterator position
*/
public boolean hasNext()
{
    if (position == null)
    {
        return first != null;
    }
    else
    {
        return position.next != null;
    }
}

<**
   Adds an element before the iterator position
   and moves the iterator past the inserted element.
   @param element the element to add
*/
public void add(Object element)
{
    if (position == null)
    {
        addFirst(element);
        position = first;
    }
    else
    {
        Node newNode = new Node();
        newNode.data = element;
        newNode.next = position.next;
        position.next = newNode;
        position = newNode;
    }

    isAfterNext = false;
}

<**
   Removes the last traversed element. This method may
   only be called after a call to the next method.
*/
public void remove()
{
    if (!isAfterNext) { throw new IllegalStateException(); }

    if (position == first)
    {

```

```

159             removeFirst();
160         }
161     else
162     {
163         previous.next = position.next;
164     }
165     position = previous;
166     isAfterNext = false;
167 }
168 /**
169 * Sets the last traversed element to a different value.
170 * @param element the element to set
171 */
172 public void set(Object element)
173 {
174     if (!isAfterNext) { throw new IllegalStateException(); }
175     position.data = element;
176 }
177 }
178 }
179 }
```

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download a program that demonstrates linked list operations.

**section\_1/ListIterator.java**

```

1 /**
2  * A list iterator allows access to a position in a linked list.
3  * This interface contains a subset of the methods of the
4  * standard java.util.ListIterator interface. The methods for
5  * backward traversal are not included.
6 */
7 public interface ListIterator
8 {
9     /**
10      * Moves the iterator past the next element.
11      * @return the traversed element
12      */
13     Object next();
14
15     /**
16      * Tests if there is an element after the iterator position.
17      * @return true if there is an element after the iterator position
18      */
19     boolean hasNext();
20
21     /**
22      * Adds an element before the iterator position
23      * and moves the iterator past the inserted element.
24      * @param element the element to add
25      */
26     void add(Object element);
27
28     /**
29      * Removes the last traversed element. This method may
30      * only be called after a call to the next method.
31      */
32     void remove();
33
34     /**
35      * Sets the last traversed element to a different value.
36      * @param element the element to set
37     }
```

```

37     */
38     void set(Object element);
39 }

```

**SELF CHECK**

1. Trace through the `addFirst` method when adding an element to an empty list.
2. Conceptually, an iterator is located between two elements (see Figure 9 in Chapter 15). Does the `position` instance variable refer to the element to the left or the element to the right?
3. Why does the `add` method have two separate cases?
4. Assume that a `last` reference is added to the `LinkedList` class, as described in Section 16.1.8. How does the `add` method of the `ListIterator` need to change?
5. Provide an implementation of an `addLast` method for the `LinkedList` class, assuming that there is no `last` reference.
6. Expressed in big-Oh notation, what is the efficiency of the `addFirst` method of the `LinkedList` class? What is the efficiency of the `addLast` method of Self Check 5?
7. How much slower is the binary search algorithm for a linked list compared to the linear search algorithm?

**Practice It** Now you can try these exercises at the end of the chapter: R16.1, E16.2, E16.4, E16.6.

**Special Topic 16.1****Static Classes**

You first saw the use of inner classes for event handlers in Chapter 10. Inner classes are useful in that context, because their methods have the privilege of accessing private instance variables of outer-class objects. The same is true for the `LinkedListIterator` inner class in the sample code for this section. The iterator needs to access the `first` instance variable of its linked list.

However, there is a cost for this feature. Every object of the inner class has a reference to the object of the enclosing class that constructed it. If an inner class has no need to access the enclosing class, you can declare the class as static and eliminate the reference to the enclosing class. This is the case with the `Node` class.

You can declare it as follows:

```

public class LinkedList
{
    ...
    static class Node
    {
        ...
    }
}

```

However, the `LinkedListIterator` class cannot be a static class. Its methods must access the `first` element of the enclosing `LinkedList`.

**WORKED EXAMPLE 16.1****Implementing a Doubly-Linked List**

Learn how to modify a singly-linked list to implement a doubly-linked list. Go to [wiley.com/go/bjeo6examples](http://wiley.com/go/bjeo6examples) and download Worked Example 16.1.



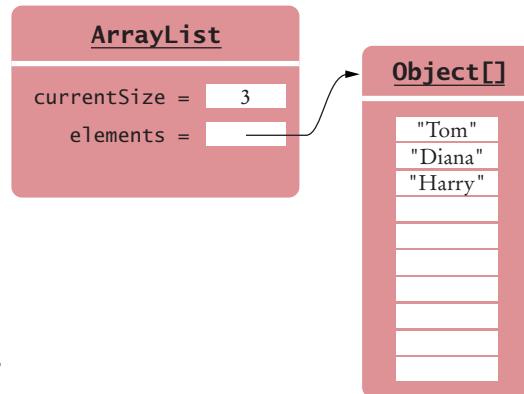
## 16.2 Implementing Array Lists

Array lists were introduced in Chapter 7. They are conceptually similar to linked lists, allowing you to add and remove elements at any position. In the following sections, we will develop an implementation of an array list, study the efficiency of operations on array lists, and compare them with the equivalent operations on linked lists.

### 16.2.1 Getting and Setting Elements

An array list maintains a reference to an array of elements. The array is large enough to hold all elements in the collection—in fact, it is usually larger to allow for adding additional elements. When the array gets full, it is replaced by a larger one. We discuss that process in Section 16.2.3.

In addition to the internal array of elements, an array list has an instance field that stores the current number of elements (see Figure 7).



**Figure 7**  
An ArrayList Stores Its  
Elements in an Array

For simplicity, our `ArrayList` implementation does not work with arbitrary element types, but it simply manages elements of type `Object`. (Chapter 18 shows how to implement classes with type parameters.)

```
public class ArrayList
{
    private Object[] elements;
    private int currentSize;

    public ArrayList()
    {
        final int INITIAL_SIZE = 10;
        elements = new Object[INITIAL_SIZE];
        currentSize = 0;
    }

    public int size() { return currentSize; }
    . .
}
```

To access array list elements, we provide `get` and `set` methods. These methods simply check for valid positions and access the internal array at the given position:

```

private void checkBounds(int n)
{
    if (n < 0 || n >= currentSize)
    {
        throw new IndexOutOfBoundsException();
    }
}

public Object get(int pos)
{
    checkBounds(pos);
    return element[pos];
}

public void set(int pos, Object element)
{
    checkBounds(pos);
    elements[pos] = element;
}

```

Getting or setting an array list element is an  $O(1)$  operation.

As you can see, getting and setting an element can be carried out with a bounded set of instructions, independent of the size of the array list. These are  $O(1)$  operations.

## 16.2.2 Removing or Adding Elements

When removing an element at position  $k$ , the elements with higher index values need to move (see Figure 8). Here is the implementation, following Section 7.3.6:

```

public Object remove(int pos)
{
    checkBounds(pos);

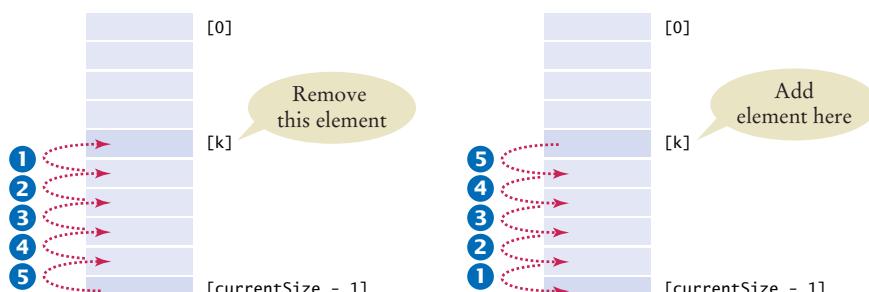
    Object removed = elements[pos];

    for (int i = pos + 1; i < currentSize; i++)
    {
        elements[i - 1] = elements[i];
    }

    currentSize--;
    return removed;
}

```

How many elements are affected? If we assume that removal happens at random locations, then on average, each removal moves  $n / 2$  elements, where  $n$  is the size of the array list.



**Figure 8**  
Removing and  
Adding Elements

Inserting or removing an array list element is an  $O(n)$  operation.

The same argument holds for inserting an element. On average,  $n / 2$  elements need to be moved. Therefore, we say that adding and removing elements are  $O(n)$  operations.

There is one situation where adding an element to an array list isn't so costly: when the insertion happens *after* the last element. If the current size is less than the length of the array, the size is incremented and the new element is simply stored in the array. This is an  $O(1)$  operation.

```
public boolean addLast(Object newElement)
{
    growIfNecessary();
    currentSize++;

    elements[currentSize - 1] = newElement;
    return true;
}
```

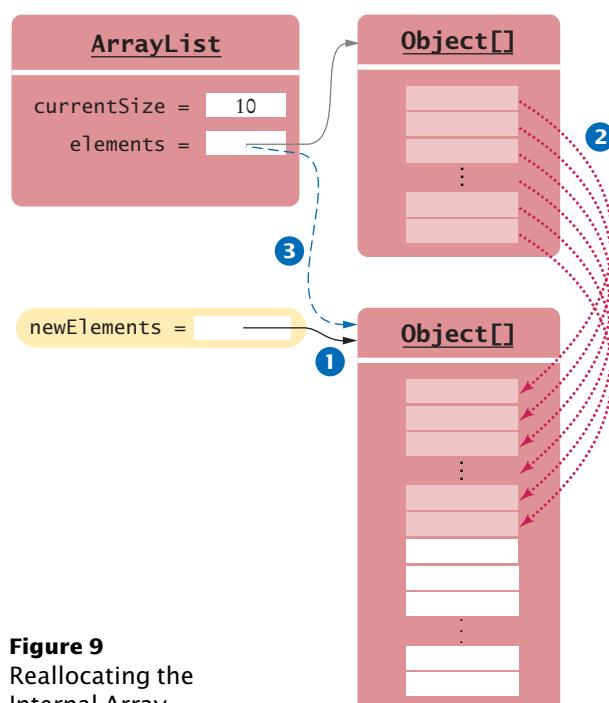
One issue remains: If there is no more room in the internal array, then we need to grow it. That is the topic of the next section.

### 16.2.3 Growing the Internal Array

Before inserting an element into an internal array that is completely full, we must replace the array with a bigger one. This new array is typically twice the size of the current array. (See Figure 9.) The existing elements are then copied into the new array. Reallocation is an  $O(n)$  operation because all elements need to be copied to the new array.



© Craig Dingle/Stockphoto.



When an array list is completely full, we must move the contents to a larger array.

```
private void growIfNecessary()
{
    if (currentSize == elements.length)
    {
        Object[] newElements =
            new Object[2 * elements.length]; 1
        for (int i = 0; i < elements.length; i++)
        {
            newElements[i] = elements[i]; 2
        }
        elements = newElements; 3
    }
}
```

**Figure 9**  
Reallocating the Internal Array

If we carefully analyze the total cost of a sequence of `addLast` operations, it turns out that these reallocations are not as expensive as they first appear. The key observation is that array growth does not happen very often. Suppose we start with an array list of capacity 10 and double the size with each reallocation. We must reallocate the array of elements when it reaches sizes 10, 20, 40, 80, 160, 320, 640, 1280, and so on.

Let us assume that one insertion without reallocation takes time  $T_1$  and that reallocation of  $k$  elements takes time  $kT_2$ . What is the cost of 1280 `addLast` operations?

Of course, we pay  $1280 \cdot T_1$  for the insertions. The reallocation cost is

$$\begin{aligned} 10T_2 + 20T_2 + 40T_2 + \cdots + 1280T_2 &= (1 + 2 + 4 + \cdots + 128) \cdot 10 \cdot T_2 \\ &= 255 \cdot 10 \cdot T_2 \\ &< 256 \cdot 10 \cdot T_2 \\ &= 1280 \cdot 2 \cdot T_2 \end{aligned}$$

Therefore, the total cost is a bit less than

$$1280 \cdot (T_1 + 2T_2)$$

In general, the total cost of  $n$  `addLast` operations is less than  $n \cdot (T_1 + 2T_2)$ . Because the second factor is a constant, we conclude that  $n$  `addLast` operations take  $O(n)$  time.

We know that it isn't quite true that an individual `addLast` operation takes  $O(1)$  time. After all, occasionally a call to `addLast` is unlucky and must reallocate the elements array.

But if the cost of that reallocation is distributed over the preceding `addLast` operations, then the surcharge for each of them is still a constant amount. We say that `addLast` takes *amortized*  $O(1)$  time, which is written as  $O(1)+$ . (Accountants say that a cost is amortized when it is distributed over multiple periods.)

In our implementation, we do not shrink the array when elements are removed. However, it turns out that you can (occasionally) shrink the array and still have  $O(1)+$  performance for removing the last element (see Exercise E16.10).

Adding or removing the last element in an array list takes amortized  $O(1)$  time.

#### FULL CODE EXAMPLE

Go to [wiley.com/go/bjeo6code](http://wiley.com/go/bjeo6code) to download a program that demonstrates this array list implementation.

**Table 2** Efficiency of Array List and Linked List Operations

Operation	Array List	Doubly-Linked List
Add/remove element at end.	$O(1)+$	$O(1)$
Add/remove element in the middle.	$O(n)$	$O(1)$
Get $k$ th element.	$O(1)$	$O(k)$

#### SELF CHECK



8. Why is it much more expensive to get the  $k$ th element in a linked list than in an array list?
9. Why is it much more expensive to insert an element at the beginning of an array list than at the beginning of a linked list?
10. What is the efficiency of adding an element exactly in the middle of a linked list? An array list?
11. Suppose we insert an element at the beginning of an array list, and the internal array must be grown to hold the new element. What is the efficiency of the `add` operation in this situation?

- 12.** Using big-Oh notation, what is the cost of adding an element to an array list as the second-to-last element?

**Practice It** Now you can try these exercises at the end of the chapter: R16.9, R16.10, R16.11.

## 16.3 Implementing Stacks and Queues

In Section 15.5, we introduced the stack and queue data types. Stacks and queues are very simple. Elements are added and retrieved, either in *last-in, first-out* order or in *first-in, first-out* order.

Stacks and queues are examples of **abstract data types**. We only specify how the operations must behave, not how they are implemented. In the following sections, we will study several implementations of stacks and queues and determine how efficient they are.

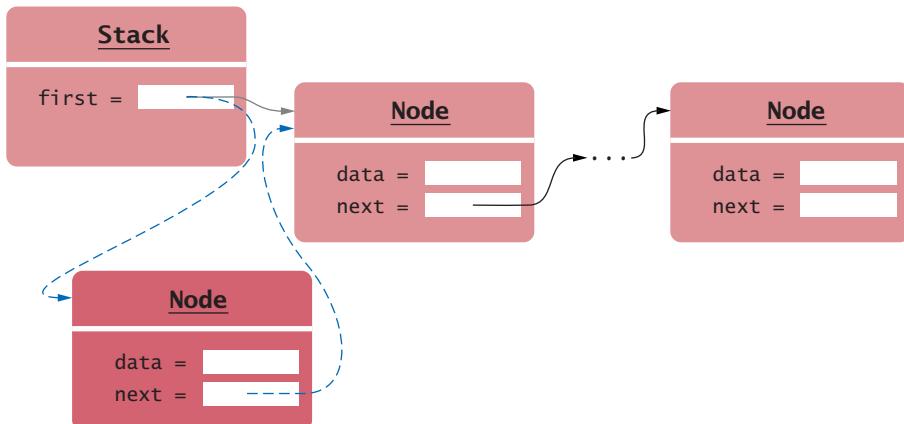
### 16.3.1 Stacks as Linked Lists

A stack can be implemented as a linked list, adding and removing elements at the front.

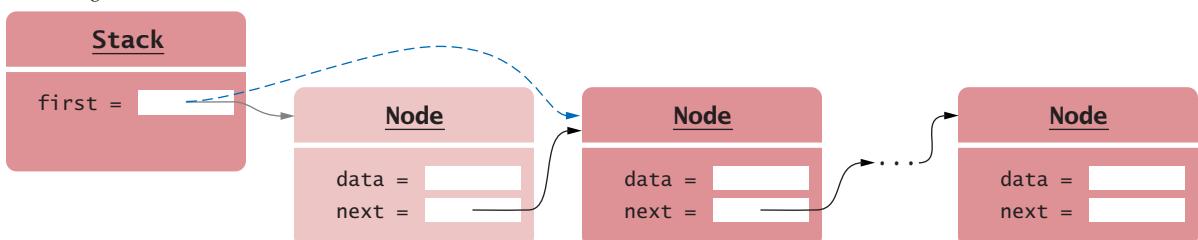
Let us first implement a stack as a sequence of nodes. New elements are added (or “pushed”) to an end of the sequence, and they are removed (or “popped”) from the same end.

Which end? It is up to us to choose, and we will make the least expensive choice: to add and remove elements at the front (see Figure 10).

Adding an element



Removing an element



**Figure 10** Push and Pop for a Stack Implemented as a Linked List

The push and pop operations are identical to the addFirst and removeFirst operations from Section 16.1.2. They are both  $O(1)$  operations.

Here is the complete implementation:

### section\_3\_1/LinkedListStack.java

```
1 import java.util.NoSuchElementException;
2
3 /**
4  * An implementation of a stack as a sequence of nodes.
5 */
6 public class LinkedListStack
7 {
8     private Node first;
9
10    /**
11     * Constructs an empty stack.
12     */
13    public LinkedListStack()
14    {
15        first = null;
16    }
17
18    /**
19     * Adds an element to the top of the stack.
20     * @param element the element to add
21     */
22    public void push(Object element)
23    {
24        Node newNode = new Node();
25        newNode.data = element;
26        newNode.next = first;
27        first = newNode;
28    }
29
30    /**
31     * Removes the element from the top of the stack.
32     * @return the removed element
33     */
34    public Object pop()
35    {
36        if (first == null) { throw new NoSuchElementException(); }
37        Object element = first.data;
38        first = first.next;
39        return element;
40    }
41
42    /**
43     * Checks whether this stack is empty.
44     * @return true if the stack is empty
45     */
46    public boolean empty()
47    {
48        return first == null;
49    }
50
51    class Node
52    {
53        public Object data;
54        public Node next;
```

```
55 }
56 }
```

### 16.3.2 Stacks as Arrays

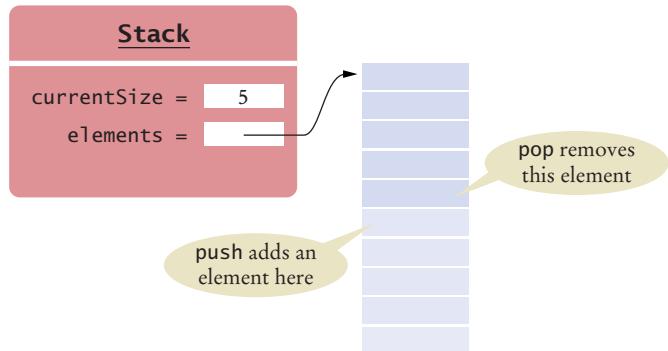
When implementing a stack as an array list, add and remove elements at the back.

In the preceding section, you saw how a list was implemented as a sequence of nodes. In this section, we will instead store the values in an array, thus saving the storage of the node references.

Again, it is up to us at which end of the array we place new elements. This time, it is better to add and remove elements at the back of the array (see Figure 11).

Of course, an array may eventually fill up as more elements are pushed on the stack. As with the `ArrayList` implementation of Section 16.2, the array must grow when it gets full.

The push and pop operations are identical to the `addLast` and `removeLast` operations of an array list. They are both  $O(1)$ + operations.



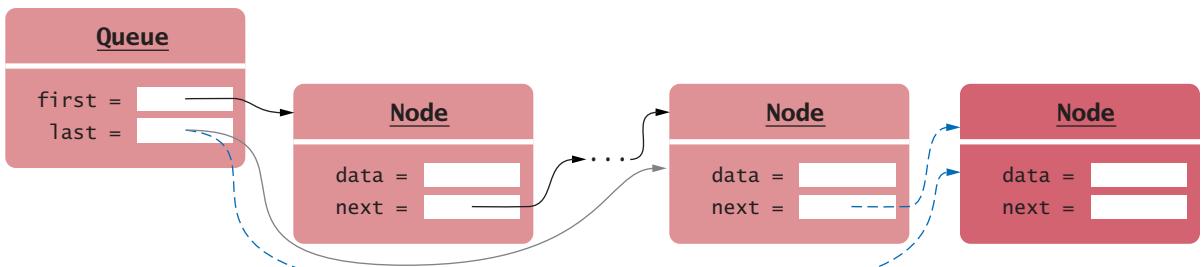
**Figure 11**  
A Stack Implemented  
as an Array

### 16.3.3 Queues as Linked Lists

A queue can be implemented as a linked list, adding elements at the back and removing them at the front.

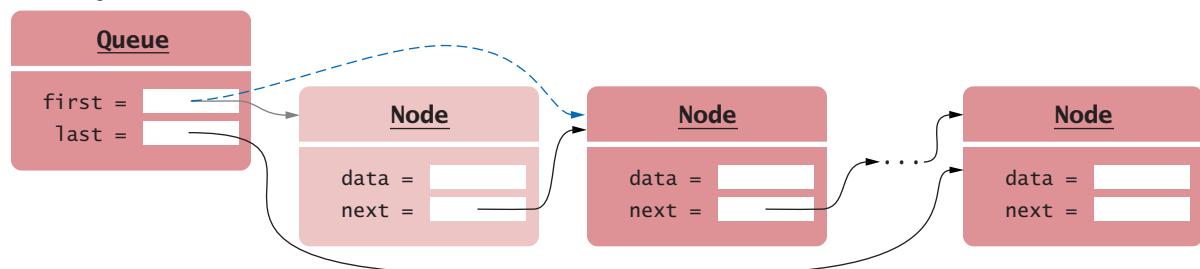
We now turn to the implementation of a queue. When implementing a queue as a sequence of nodes, we add nodes at one end and remove them at the other. As we discussed in Section 16.1.8, a singly-linked node sequence is not able to remove the last node in  $O(1)$  time. Therefore, it is best to remove elements at the front and add them at the back (see Figure 12).

Adding an element



**Figure 12** A Queue Implemented as a Linked List

Removing an element



**Figure 12 (continued)** A Queue Implemented as a Linked List

The add and remove operations of a queue are  $O(1)$  operations because they are the same as the `addLast` and `removeFirst` operations of a doubly-linked list. Note that we need a reference to the last node so that we can efficiently add elements.

### 16.3.4 Queues as Circular Arrays

When storing queue elements in an array, we have a problem: elements get added at one end of the array and removed at the other. But adding or removing the first element of an array is an  $O(n)$  operation, so it seems that we cannot avoid this expensive operation, no matter which end we choose for adding elements and which for removing them.



© ihsanyildiz/iStockphoto.

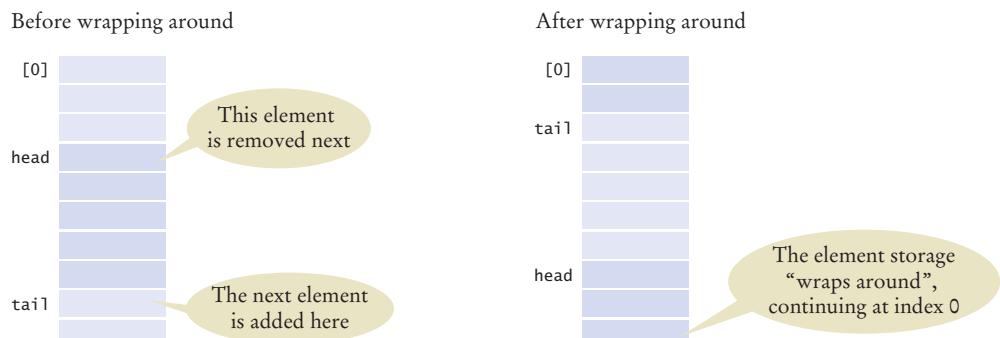
In a circular array implementation of a queue, element locations wrap from the end of the array to the beginning.

However, we can solve this problem with a trick. We add elements at the end, but when we remove them, we don't actually move the remaining elements. Instead, we increment the index at which the head of the queue is located (see Figure 13).

After adding sufficiently many elements, the last element of the array will be filled. However, if there were also a few calls to remove, then there is additional room in the front of the array. Then we “wrap around” and start storing elements again at index 0—see part 2 of Figure 13. For that reason, the array is called “circular”.

Eventually, of course, the tail reaches the head, and a larger array must be allocated.

As you can see from the source code that follows, adding or removing an element requires a bounded set of operations, independent of the queue size, except for array



**Figure 13** Queue Elements in a Circular Array

reallocation. However, as discussed in Section 16.2.3, reallocation happens rarely enough that the total cost is still amortized constant time,  $O(1)+$ .

**Table 3 Efficiency of Stack and Queue Operations**

	Stack as Linked List	Stack as Array	Queue as Linked List	Queue as Circular Array
Add an element.	$O(1)$	$O(1)+$	$O(1)$	$O(1)+$
Remove an element.	$O(1)$	$O(1)+$	$O(1)$	$O(1)+$

### section\_3\_4/CircularArrayQueue.java

```

1  import java.util.NoSuchElementException;
2
3  /**
4   * An implementation of a queue as a circular array.
5  */
6  public class CircularArrayQueue
7  {
8      private Object[] elements;
9      private int currentSize;
10     private int head;
11     private int tail;
12
13     /**
14      Constructs an empty queue.
15     */
16     public CircularArrayQueue()
17     {
18         final int INITIAL_SIZE = 10;
19         elements = new Object[INITIAL_SIZE];
20         currentSize = 0;
21         head = 0;
22         tail = 0;
23     }
24
25     /**
26      Checks whether this queue is empty.
27      @return true if this queue is empty
28     */
29     public boolean empty() { return currentSize == 0; }
30
31     /**
32      Adds an element to the tail of this queue.
33      @param newElement the element to add
34     */
35     public void add(Object newElement)
36     {
37         growIfNecessary();
38         currentSize++;
39         elements[tail] = newElement;
40         tail = (tail + 1) % elements.length;
41     }
42
43     /**
44      Removes an element from the head of this queue.
45      @return the removed element

```

```

46  */
47  public Object remove()
48  {
49      if (currentSize == 0) { throw new NoSuchElementException(); }
50      Object removed = elements[head];
51      head = (head + 1) % elements.length;
52      currentSize--;
53      return removed;
54  }
55
56 /**
57     Grows the element array if the current size equals the capacity.
58 */
59 private void growIfNecessary()
60 {
61     if (currentSize == elements.length)
62     {
63         Object[] newElements = new Object[2 * elements.length];
64         for (int i = 0; i < elements.length; i++)
65         {
66             newElements[i] = elements[(head + i) % elements.length];
67         }
68         elements = newElements;
69         head = 0;
70         tail = currentSize;
71     }
72 }
73 }
```

**SELF CHECK**

13. Add a method `peek` to the Stack implementation in Section 16.3.1 that returns the top of the stack without removing it.
14. When implementing a stack as a sequence of nodes, why isn't it a good idea to push and pop elements at the back end?
15. When implementing a stack as an array, why isn't it a good idea to push and pop elements at index 0?
16. What is wrong with this implementation of the `empty` method for the circular array queue?

```
public boolean empty()
{
    return head == 0 && tail == 0;
}
```
17. What is wrong with this implementation of the `empty` method for the circular array queue?

```
public boolean empty()
{
    return head == tail;
}
```
18. Have a look at the `growIfNecessary` method of the `CircularArrayQueue` class. Why isn't the loop simply

```
for (int i = 0; i < elements.length; i++)
{
    newElements[i] = elements[i];
}
```

**Practice It** Now you can try these exercises at the end of the chapter: R16.20, R16.23, E16.11, E16.12.

## 16.4 Implementing a Hash Table

In Section 15.3, you were introduced to the set data structure and its two implementations in the Java collections framework, hash sets and tree sets. In these sections, you will see how hash sets are implemented and how efficient their operations are.

### 16.4.1 Hash Codes

A good hash function minimizes *collisions*—identical hash codes for different objects.

The basic idea behind hashing is to place objects into an array, at a location that can be determined from the object itself. Each object has a **hash code**, an integer value that is computed from an object in such a way that different objects are likely to yield different hash codes.

Table 4 shows some examples of strings and their hash codes. Special Topic 15.1 shows how these values are computed.

It is possible for two or more distinct objects to have the same hash code; this is called a *collision*. For example, the strings "VII" and "Ugh" happen to have the same hash code.

**Table 4** Sample Strings and Their Hash Codes

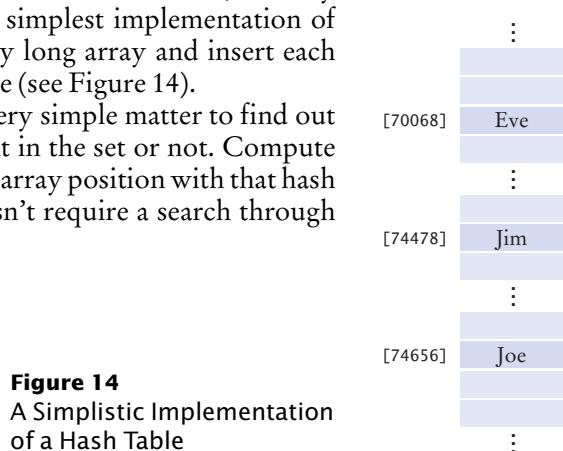
String	Hash Code	String	Hash Code
"Adam"	2035631	"Juliet"	-2065036585
"Eve"	70068	"Katherine"	2079199209
"Harry"	69496448	"Sue"	83491
"Jim"	74478	"Ugh"	84982
"Joe"	74656	"VII"	84982

### 16.4.2 Hash Tables

A hash table uses the hash code to determine where to store each element.

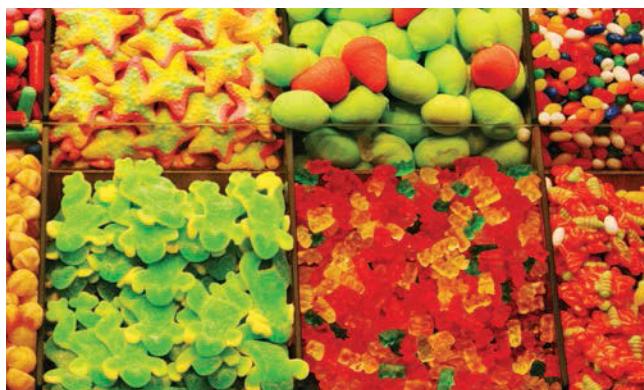
A hash code is used as an array index into a **hash table**, an array that stores the set elements. In the simplest implementation of a hash table, you could make a very long array and insert each object at the location of its hash code (see Figure 14).

If there are no collisions, it is a very simple matter to find out whether an object is already present in the set or not. Compute its hash code and check whether the array position with that hash code is already occupied. This doesn't require a search through the entire array!



**Figure 14**  
A Simplistic Implementation  
of a Hash Table

*Elements with the same hash code are placed in the same bucket.*



© Neil Kurtzman/iStockphoto.

Of course, it is not feasible to allocate an array that is large enough to hold all possible integer index positions. Therefore, we must pick an array of some reasonable size and then “compress” the hash code to become a valid array index. Compression can be easily achieved by using the remainder operation:

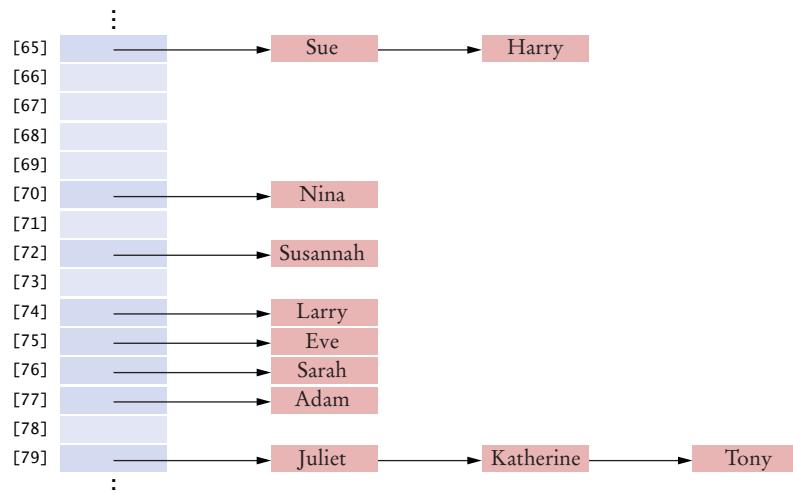
```
int h = x.hashCode();
if (h < 0) { h = -h; }
position = h % arrayLength;
```

See Exercise E16.20 for an alternative compression technique.

A hash table can be implemented as an array of *buckets*—sequences of nodes that hold elements with the same hash code.

After compressing the hash code, it becomes more likely that several objects will collide. There are several techniques for handling collisions. The most common one is called *separate chaining*. All colliding elements are collected in a linked list of elements with the same position value (see Figure 15). Such a list is called a “bucket”. Special Topic 16.2 discusses *open addressing*, in which colliding elements are placed in empty locations of the hash table.

In the following, we will use the first technique. Each entry of the hash table points to a sequence of nodes containing elements with the same (compressed) hash code.



**Figure 15** A Hash Table with Buckets to Store Elements with the Same Hash Code

### 16.4.3 Finding an Element

Let's assume that our hash table has been filled with a number of elements. Now we want to find out whether a given element is already present.

Here is the algorithm for finding an object `obj` in a hash table:

1. Compute the hash code and compress it. This gives an index `h` into the hash table.
2. Iterate through the elements of the bucket at position `h`. For each element of the bucket, check whether it is equal to `obj`.
3. If a match is found among the elements of that bucket, then `obj` is in the set. Otherwise, it is not.

If there are no or only a few collisions, then adding, locating, and removing hash table elements takes constant or  $O(1)$  time.

How efficient is this operation? It depends on the hash code computation. In the best case, in which there are no collisions, all buckets either are empty or have a single element.

But in practice, some collisions will occur. We need to make some assumptions that are reasonable in practice.

First, we assume that the hash code does a good job scattering the elements into different buckets. In practice, the hash functions described in Special Topic 15.1 work well.

Next, we assume that the table is large enough. This is measured by the *load factor*  $F = n / L$ , where  $n$  is the number of elements and  $L$  the table length. For example, if the table is an array of length 1,000, and it has 700 elements, then the load factor is 0.7.

If the load factor gets too large, the elements should be moved into a larger table. The hash table in the standard Java library reallocates the table when the load factor exceeds 0.75.

Under these assumptions, each bucket can be expected to have, on average,  $F$  elements.

Finally, we assume that the hash code, its compression, and the `equals` method can be computed in bounded time, independent of the size of the set.

Now let us compute the cost of finding an element. Computing the array index takes constant time, due to our last assumption. Now we traverse a chain of buckets, which on average has a bounded length  $F$ . Finally, we invoke the `equals` method on each bucket element, which we also assume to be  $O(1)$ . The entire operation takes constant or  $O(1)$  time.

### 16.4.4 Adding and Removing Elements

Adding an element is an extension of the algorithm for finding an object. First compute the hash code to locate the bucket in which the element should be inserted:

1. Compute the compressed hash code `h`.
2. Iterate through the elements of the bucket at position `h`. For each element of the bucket, check whether it is equal to `obj` (using the `equals` method of the element type).
3. If a match is found among the elements of that bucket, then exit.
4. Otherwise, add a node containing `obj` to the beginning of the node sequence.
5. If the load factor exceeds a fixed threshold, reallocate the table.

As described in the preceding section, the first three steps are  $O(1)$ . Inserting at the beginning of a node sequence is also  $O(1)$ . As with array lists, we can choose the new table to be twice the size of the old table, and amortize the cost of reallocation over the preceding insertions. That is, adding an element to a hash table is  $O(1)+$ .

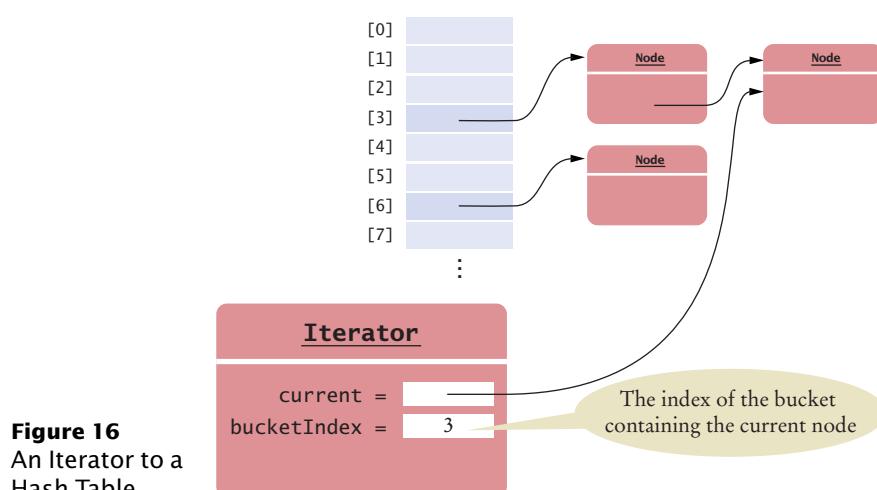
Removing an element is equally simple. First compute the hash code to locate the bucket in which the element should be inserted. Try finding the object in that bucket. If it is present, remove it. Otherwise, do nothing. Again, this is a constant time operation. If we shrink a table that becomes too sparse, the cost is  $O(1)+$ .

### 16.4.5 Iterating over a Hash Table

An iterator for a linked list points to the current node in a list. A hash table has multiple node chains. When we are at the end of one chain, we need to move to the start of the next one. Therefore, the iterator also needs to store the bucket number (see Figure 16).

When the iterator points into the middle of a node chain, then it is easy to advance it to the next element. However, when the iterator points to the last node in a chain, then we must skip past all empty buckets. When we find a non-empty bucket, we advance the iterator to its first node:

```
if (current != null && current.next != null)
{
    current = current.next; // Move to next element in bucket
}
else // Move to next bucket
{
    do
    {
        bucketIndex++;
        if (bucketIndex == buckets.length)
        {
            throw new NoSuchElementException();
        }
        current = buckets[bucketIndex];
    }
    while (current == null);
}
```



**Figure 16**  
An Iterator to a  
Hash Table

As you can see, the cost of iterating over all elements of a hash table is proportional to the table length. Note that the table length could be in excess of  $O(n)$  if the table is sparsely filled. This can be avoided if we shrink the table when the load factor gets too small. In that case, iterating over the entire table is  $O(n)$ , and each iteration step is  $O(1)$ .

Table 5 summarizes the efficiency of the operations on a hash table.

Table 5 Hash Table Efficiency	
Operation	Hash Table
Find an element.	$O(1)$
Add/remove an element.	$O(1) +$
Iterate through all elements.	$O(n)$

Here is an implementation of a hash set. For simplicity, we do not reallocate the table when it grows or shrinks, and we do not support the `remove` operation on iterators. Exercises E16.18 and E16.19 ask you to provide these enhancements.

### section\_4/HashSet.java

```

1 import java.util.Iterator;
2 import java.util.NoSuchElementException;
3
4 /**
5  * This class implements a hash set using separate chaining.
6 */
7 public class HashSet
8 {
9     private Node[] buckets;
10    private int currentSize;
11
12    /**
13     * Constructs a hash table.
14     * @param bucketsLength the length of the buckets array
15     */
16    public HashSet(int bucketsLength)
17    {
18        buckets = new Node[bucketsLength];
19        currentSize = 0;
20    }
21
22    /**
23     * Tests for set membership.
24     * @param x an object
25     * @return true if x is an element of this set
26     */
27    public boolean contains(Object x)
28    {
29        int h = x.hashCode();
30        if (h < 0) { h = -h; }
31        h = h % buckets.length;
32    }

```

```

33     Node current = buckets[h];
34     while (current != null)
35     {
36         if (current.data.equals(x)) { return true; }
37         current = current.next;
38     }
39     return false;
40 }
41
42 /**
43  * Adds an element to this set.
44  * @param x an object
45  * @return true if x is a new object, false if x was
46  * already in the set
47 */
48 public boolean add(Object x)
49 {
50     int h = x.hashCode();
51     if (h < 0) { h = -h; }
52     h = h % buckets.length;
53
54     Node current = buckets[h];
55     while (current != null)
56     {
57         if (current.data.equals(x)) { return false; }
58         // Already in the set
59         current = current.next;
60     }
61     Node newNode = new Node();
62     newNode.data = x;
63     newNode.next = buckets[h];
64     buckets[h] = newNode;
65     currentSize++;
66     return true;
67 }
68
69 /**
70  * Removes an object from this set.
71  * @param x an object
72  * @return true if x was removed from this set, false
73  * if x was not an element of this set
74 */
75 public boolean remove(Object x)
76 {
77     int h = x.hashCode();
78     if (h < 0) { h = -h; }
79     h = h % buckets.length;
80
81     Node current = buckets[h];
82     Node previous = null;
83     while (current != null)
84     {
85         if (current.data.equals(x))
86         {
87             if (previous == null) { buckets[h] = current.next; }
88             else { previous.next = current.next; }
89             currentSize--;
90             return true;
91         }
92         previous = current;
}

```

```

93         current = current.next;
94     }
95     return false;
96 }
97 /**
98  * Returns an iterator that traverses the elements of this set.
99  * @return a hash set iterator
100 */
101 public Iterator iterator()
102 {
103     return new HashSetIterator();
104 }
105 /**
106  * Gets the number of elements in this set.
107  * @return the number of elements
108 */
109 public int size()
110 {
111     return currentSize;
112 }
113
114 class Node
115 {
116     public Object data;
117     public Node next;
118 }
119
120 class HashSetIterator implements Iterator
121 {
122     private int bucketIndex;
123     private Node current;
124
125     /**
126      * Constructs a hash set iterator that points to the
127      * first element of the hash set.
128     */
129     public HashSetIterator()
130     {
131         current = null;
132         bucketIndex = -1;
133     }
134
135     public boolean hasNext()
136     {
137         if (current != null && current.next != null) { return true; }
138         for (int b = bucketIndex + 1; b < buckets.length; b++)
139         {
140             if (buckets[b] != null) { return true; }
141         }
142         return false;
143     }
144
145     public Object next()
146     {
147         if (current != null && current.next != null)
148         {
149             current = current.next; // Move to next element in bucket
150         }
151     }
152 }
```

```

153     else // Move to next bucket
154     {
155         do
156         {
157             bucketIndex++;
158             if (bucketIndex == buckets.length)
159             {
160                 throw new NoSuchElementException();
161             }
162             current = buckets[bucketIndex];
163             }
164             while (current == null);
165         }
166         return current.data;
167     }
168
169     public void remove()
170     {
171         throw new UnsupportedOperationException();
172     }
173 }
174 }
```

**section\_4/HashSetDemo.java**

```

1 import java.util.Iterator;
2
3 /**
4  * This program demonstrates the hash set class.
5 */
6 public class HashSetDemo
7 {
8     public static void main(String[] args)
9     {
10         HashSet names = new HashSet(101);
11
12         names.add("Harry");
13         names.add("Sue");
14         names.add("Nina");
15         names.add("Susannah");
16         names.add("Larry");
17         names.add("Eve");
18         names.add("Sarah");
19         names.add("Adam");
20         names.add("Tony");
21         names.add("Katherine");
22         names.add("Juliet");
23         names.add("Romeo");
24         names.remove("Romeo");
25         names.remove("George");
26
27         Iterator iter = names.iterator();
28         while (iter.hasNext())
29         {
30             System.out.println(iter.next());
31         }
32     }
33 }
```

### Program Run

```
Harry
Sue
Nina
Susannah
Larry
Eve
Sarah
Adam
Juliet
Katherine
Tony
```

#### SELF CHECK



19. If a hash function returns 0 for all values, will the hash table work correctly?
20. If a hash table has size 1, will it work correctly?
21. Suppose you have two hash tables, each with  $n$  elements. To find the elements that are in both tables, you iterate over the first table, and for each element, check whether it is contained in the second table. What is the big-Oh efficiency of this algorithm?
22. In which order does the iterator visit the elements of the hash table?
23. What does the `hasNext` method of the `HashSetIterator` do when it has reached the end of a bucket?
24. Why doesn't the iterator have an `add` method?

**Practice It** Now you can try these exercises at the end of the chapter: E16.18, E16.20, E16.21.

#### Special Topic 16.2



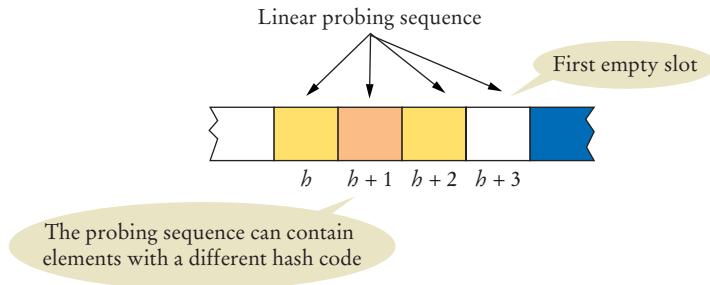
### Open Addressing

In the preceding sections, you studied a hash table implementation that uses separate chaining for collision handling, placing all elements with the same hash code in a bucket. This implementation is fast and easy to understand, but it requires storage for the links to the nodes. If one places the elements directly into the hash table, then one doesn't need to store any links. This alternative technique is called *open addressing*. It can be beneficial if one must minimize the memory usage of a hash table.

Of course, open addressing makes collision handling more complicated. If you have two elements with (compressed) hash code  $b$ , and the first one is placed at index  $b$ , then the second must be placed in another location.

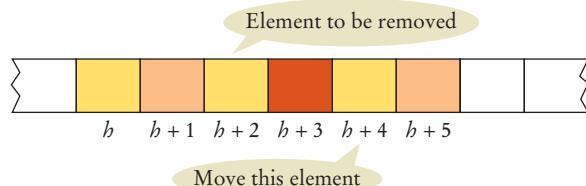
There are different techniques for placing colliding elements. The simplest is *linear probing*. If possible, place the colliding element at index  $b + 1$ . If that slot is occupied, try  $b + 2, b + 3$ , and so on, wrapping around to 0, 1, 2, and so on, if necessary. This sequence of index values is called the *probing sequence*. (You can see other probing sequences in Exercises P16.15 and P16.16.) If the probing sequence contains no empty slots, one must reallocate to a larger table.

How do we find an element in such a hash table? We compute the hash code and traverse the probing sequence until we either find a match or an empty slot. As long as the hash table is not too full, this is still an  $O(1)$  operation, but it may require more comparisons than with separate chaining. With separate chaining, we only compare objects with the same hash code. With open addressing, there may be some objects with different hash codes that happen to lie on the probing sequence.



Adding an element is similar. Try finding the element first. If it is not present, add it in the first empty slot in the probing sequence.

Removing an element is trickier. You cannot simply empty the slot at which you find the element. Instead, you must traverse the probing sequence, look for the last element with the same hash code, and move that element into the slot of the removed element (Exercise P16.14).



Alternatively, you can replace the removed element with a special “inactive” marker that, unlike an empty slot, does not indicate the end of a probing sequence. When adding another element, you can overwrite an inactive slot (Exercise P16.17).

## CHAPTER SUMMARY

### Describe the implementation and efficiency of linked list operations.



- A linked list object holds a reference to the first node object, and each node holds a reference to the next node.
- When adding or removing the first element, the reference to the first node must be updated.
- A list iterator object has a reference to the last visited node.
- To advance an iterator, update the position and remember the old position for the `remove` method.
- In a doubly-linked list, accessing an element is an  $O(n)$  operation; adding and removing an element is  $O(1)$ .

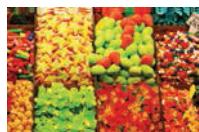
### Understand the implementation and efficiency of array list operations.



- Getting or setting an array list element is an  $O(1)$  operation.
- Inserting or removing an array list element is an  $O(n)$  operation.
- Adding or removing the last element in an array list takes amortized  $O(1)$  time.

**Compare different implementations of stacks and queues.**

- A stack can be implemented as a linked list, adding and removing elements at the front.
- When implementing a stack as an array list, add and remove elements at the back.
- A queue can be implemented as a linked list, adding elements at the back and removing them at the front.
- In a circular array implementation of a queue, element locations wrap from the end of the array to the beginning.

**Understand the implementation of hash tables and the efficiencies of its operations.**

- A good hash function minimizes *collisions*—identical hash codes for different objects.
- A hash table uses the hash code to determine where to store each element.
- A hash table can be implemented as an array of *buckets*—sequences of nodes that hold elements with the same hash code.
- If there are no or only a few collisions, then adding, locating, and removing hash table elements takes constant or  $O(1)$  time.

**REVIEW EXERCISES**

- **R16.1** The linked list class in the Java library supports operations `addLast` and `removeLast`. To carry out these operations efficiently, the `LinkedList` class has an added reference `last` to the last node in the linked list. Draw a “before/after” diagram of the changes to the links in a linked list when the `addLast` method is executed.
- **R16.2** The linked list class in the Java library supports bidirectional iterators. To go backward efficiently, each `Node` has an added reference, `previous`, to the predecessor node in the linked list. Draw a “before/after” diagram of the changes to the links in a linked list when the `addFirst` and `removeFirst` methods execute. The diagram should show how the `previous` references need to be updated.
- **R16.3** What is the big-Oh efficiency of replacing all negative values in a linked list of `Integer` objects with zeroes? Of removing all negative values?
- **R16.4** What is the big-Oh efficiency of replacing all negative values in an array list of `Integer` objects with zeroes? Of removing all negative values?
- **R16.5** In the `LinkedList` implementation of Section 16.1, we use a flag `isAfterNext` to ensure that calls to the `remove` and `set` methods occur only when they are allowed. It is not actually necessary to introduce a new instance variable for this check. Instead, one can set the `previous` instance variable to a special value at the end of every call to `add` or `remove`. With that change, how should the `remove` and `set` methods check whether they are allowed?
- **R16.6** What is the big-Oh efficiency of the `size` method of Exercise E16.4?
- **R16.7** Show that the introduction of the `size` method in Exercise E16.6 does not affect the big-Oh efficiency of the other list operations.

- R16.8** Given the `size` method of Exercise E16.6 and the `get` method of Exercise P16.1, what is the big-Oh efficiency of this loop?

```
for (int i = 0; i < myList.size(); i++) { System.out.println(myList.get(i)); }
```

- R16.9** Given the `size` method of Exercise E16.6 and the `get` method of Exercise P16.3, what is the big-Oh efficiency of this loop?

```
for (int i = 0; i < myList.size(); i++) { System.out.println(myList.get(i)); }
```

- R16.10** It is not safe to remove the first element of a linked list with the `removeFirst` method when an iterator has just traversed the first element. Explain the problem by tracing the code and drawing a diagram.

- R16.11** Continue Exercise R16.10 by providing a code example demonstrating the problem.

- R16.12** It is not safe to simultaneously modify a linked list using two iterators. Find a situation where two iterators refer to the same linked list, and when you add an element with one iterator and remove an element with the other, the result is incorrect. Explain the problem by tracing the code and drawing a diagram.

- R16.13** Continue Exercise R16.12 by providing a code example demonstrating the problem.

- R16.14** In the implementation of the `LinkedList` class of the standard Java library, the problem described in Exercises R16.10 and R16.12 results in a `ConcurrentModificationException`. Describe how the `LinkedList` class and the iterator classes can discover that a list was modified through multiple sources. *Hint:* Count mutating operations. Where are the counts stored? Where are they updated? Where are they checked?

- R16.15** Consider the efficiency of locating the  $k$ th element in a doubly-linked list of length  $n$ . If  $k > n / 2$ , it is more efficient to start at the end of the list and move the iterator to the previous element. Why doesn't this increase in efficiency improve the big-Oh estimate of element access in a doubly-linked list?

- R16.16** A linked list implementor, hoping to improve the speed of accessing elements, provides an array of `Node` references, pointing to every tenth node. Then the operation `get(n)` looks up the reference at position  $n - n \% 10$  and follows  $n \% 10$  links.

- a.** With this implementation, what is the efficiency of the `get` operation?
- b.** What is the disadvantage of this implementation?

- R16.17** Suppose an array list implementation were to add ten elements at each reallocation instead of doubling the capacity. Show that the `addLast` operation no longer has amortized constant time.

- R16.18** Consider an array list implementation with a `removeLast` method that shrinks the internal array to half of its size when it is at most half full. Give a sequence of `addLast` and `removeLast` calls that does not have amortized  $O(1)$  efficiency.

- R16.19** Suppose the `ArrayList` implementation of Section 16.2 had a `removeLast` method that shrinks the internal array by 50 percent when it is less than 25 percent full. Show that any sequence of `addLast` and `removeLast` calls has amortized  $O(1)$  efficiency.

- R16.20** Given a queue with  $O(1)$  methods `add`, `remove`, and `size`, what is the big-Oh efficiency of moving the element at the head of the queue to the tail? Of moving the element at the tail of the queue to the head? (The order of the other queue elements should be unchanged.)

- R16.21** A deque (double-ended queue) is a data structure with operations `addFirst`, `removeFirst`, `addLast`, and `removeLast`. What is the  $O(1)$  efficiency of these operations if the deque is implemented as
- a singly-linked list?
  - a doubly-linked list?
  - a circular array?
- R16.22** In our circular array implementation of a queue, can you compute the value of the `currentSize` from the values of the `head` and `tail` fields? Why or why not?
- R16.23** Draw the contents of a circular array implementation of a queue `q`, with an initial array size of 10, after each of the following loops:
- `for (int i = 1; i <= 5; i++) { q.add(i); }`
  - `for (int i = 1; i <= 3; i++) { q.remove(); }`
  - `for (int i = 1; i <= 10; i++) { q.add(i); }`
  - `for (int i = 1; i <= 8; i++) { q.remove(); }`
- R16.24** Suppose you are stranded on a desert island on which stacks are plentiful, but you need a queue. How can you implement a queue using two stacks? What is the big-Oh running time of the queue operations?
- R16.25** Suppose you are stranded on a desert island on which queues are plentiful, but you need a stack. How can you implement a stack using two queues? What is the big-Oh running time of the stack operations?
- R16.26** Craig Coder doesn't like the fact that he has to implement a hash function for the objects that he wants to collect in a hash table. "Why not assign a unique ID to each object?" he asks. What is wrong with his idea?



© Philip Dyer/iStockphoto.

## PRACTICE EXERCISES

- E16.1** Add a method `reverse` to our `LinkedList` implementation that reverses the links in a list. Implement this method by directly rerouting the links, not by using an iterator.
- E16.2** Consider a version of the `LinkedList` class of Section 16.1 in which the `addFirst` method has been replaced with the following faulty version:

```
public void addFirst(Object element)
{
    Node newNode = new Node();
    first = newNode;
    newNode.data = element;
    newNode.next = first;
}
```

Develop a program `ListTest` with a test case that shows the error. That is, the program should print a failure message with this implementation but not with the correct implementation.

- E16.3** Consider a version of the `LinkedList` class of Section 16.1 in which the iterator's `hasNext` method has been replaced with the following faulty version:

```
public boolean hasNext() { return position != null; }
```

Develop a program `ListTest` with a test case that shows the error. The program should print a failure message with this implementation but not with the correct one.

- **E16.4** Add a method `size` to our implementation of the `LinkedList` class that computes the number of elements in the list by following links and counting the elements until the end of the list is reached.

- **E16.5** Solve Exercise E16.4 recursively by calling a recursive helper method

```
private static int size(Node start)
```

*Hint:* If `start` is `null`, then the size is 0. Otherwise, it is one larger than the size of `start.next`.

- **E16.6** Add an instance variable `currentSize` to our implementation of the `LinkedList` class. Modify the `add`, `addLast`, and `remove` methods of both the linked list and the list iterator to update the `currentSize` variable so that it always contains the correct size. Change the `size` method of Exercise E16.4 so that it simply returns the value of `currentSize`.

- **E16.7** Reimplement the `LinkedList` class of Section 16.1 so that the `Node` and `LinkedListIterator` classes are not inner classes.

- **E16.8** Reimplement the `LinkedList` class of Section 16.1 so that it implements the `java.util.LinkedList` interface. *Hint:* Extend the `java.util.AbstractList` class.

- **E16.9** Provide a `listIterator` method for the `ArrayList` implementation in Section 16.2. Your method should return an object of a class implementing `java.util.ListIterator`. Also have the `ArrayList` class implement the `Iterable` interface type and provide a test program that demonstrates that your array list can be used in an enhanced `for` loop.

- **E16.10** Provide a `removeLast` method for the `ArrayList` implementation in Section 16.2 that shrinks the internal array by 50 percent when it is less than 25 percent full.

- **E16.11** Complete the implementation of a stack in Section 16.3.2, using an array for storing the elements.

- **E16.12** Complete the implementation of a queue in Section 16.3.3, using a sequence of nodes for storing the elements.

- **E16.13** Add a method `firstToLast` to the implementation of a queue in Exercise E16.12. The method moves the element at the head of the queue to the tail of the queue. The element that was second in line will now be at the head.

- **E16.14** Add a method `lastToFirst` to the implementation of a queue in Exercise E16.12. The method moves the element at the tail of the queue to the head.

- **E16.15** Add a method `firstToLast`, as described in Exercise E16.13, to the circular array implementation of a queue.

- **E16.16** Add a method `lastToFirst`, as described in Exercise E16.14, to the circular array implementation of a queue.

- **E16.17** The `hasNext` method of the hash set implementation in Section 16.4 finds the location of the next element, but when `next` is called, the same search happens again. Improve the efficiency of these methods so that `next` (or a repeated call to `hasNext`) uses the position located by a preceding call to `hasNext`.

- **E16.18** Relocate the buckets of the hash set implementation in Section 16.4 when the load factor is greater than 1.0 or less than 0.5, doubling or halving its size. Note that you need to recompute the hash values of all elements.

- **E16.19** Implement the `remove` operation for iterators on the hash set in Section 16.4.
- **E16.20** Implement the hash set in Section 16.4, using the “MAD (multiply-add-divide) method” for hash code compression. For that method, you choose a prime number  $p$  larger than the length  $L$  of the hash table and two values  $a$  and  $b$  between 1 and  $p - 1$ . Then reduce  $b$  to  $((a \cdot b + b) \% p) \% L$ .
- **E16.21** Add methods to count collisions to the hash set in Section 16.4 and the one in Exercise E16.20. Insert all words from a dictionary (in `/usr/share/dict/words` or in `words.txt` in your companion code) into both hash set implementations. Does the MAD method reduce collisions? (Use a table size that equals the number of words in the file. Choose  $p$  to be the next prime greater than  $L$ ,  $a = 3$ , and  $b = 5$ .)

## PROGRAMMING PROJECTS

- **P16.1** Add methods `Object get(int n)` and `void set(int n, Object newElement)` to the `LinkedList` class. Use a helper method that starts at `first` and follows  $n$  links:

```
private static Node getNode(int n)
```

- **P16.2** Solve Exercise P16.1 by using a recursive helper method

```
private static Node getNode(Node start, int distance)
```

- **P16.3** Improve the efficiency of the `get` and `set` methods of Exercise P16.1 by storing (or “caching”) the last known (node, index) pair. If  $n$  is larger than the last known index, start from the corresponding node instead of the front of the list. Be sure to discard the last known pair when it is no longer accurate. (This can happen when another method edits the list).

- **P16.4** Add a method `boolean contains(Object obj)` that checks whether our `LinkedList` implementation contains a given object. Implement this method by directly traversing the links, not by using an iterator.

Use the `equals` method to determine whether `obj` equals `node.data` for a given node.

- **P16.5** Solve Exercise P16.4 recursively, by calling a recursive helper method

```
private static boolean contains(Node start, Object obj)
```

*Hint:* If `start` is `null`, then it can't contain the object. Otherwise, check `start.data` before recursively moving on to `start.next`.

- **P16.6** A linked list class with an  $O(1)$  `addLast` method needs an efficient mechanism to get to the end of the list, for example by setting an instance variable to the last element. It is then possible to remove the reference to the first node if one makes the `next` reference of the last node point to the first node, so that all nodes form a cycle. Such an implementation is called a circular linked list. Turn the linked list implementation of Section 16.1 into a circular singly-linked list.

- **P16.7** In a circular doubly-linked list, the previous reference of the first node points to the last node, and the next reference of the last node points to the first node. Change the doubly-linked list implementation of Worked Example 16.1 into a circular list. You should remove the `last` instance variable because you can reach the last element as `first.previous`.

- **P16.8** Modify the insertion sort algorithm of Special Topic 14.2 to sort a linked list.

- P16.9** The LISP language, created in 1960, implements linked lists in a very elegant way. You will explore a Java analog in this set of exercises. Conceptually, the *tail* of a list—that is, the list with its head node removed—is also a list. The tail of that list is again a list, and so on, until you reach the empty list. Here is a Java interface for such a list:

```
public interface LispList
{
    boolean empty();
    Object head();
    LispList tail();
    ...
}
```

There are two kinds of lists, empty lists and nonempty lists:

```
public class EmptyList implements LispList { ... }
public class NonEmptyList implements LispList { ... }
```

These classes are quite trivial. The `EmptyList` class has no instance variables. Its `head` and `tail` methods simply throw an `UnsupportedOperationException`, and its `empty` method returns true. The `NonEmptyList` class has instance variables for the head and tail.

Here is one way of making a LISP list with three elements:

```
LispList list = new NonEmptyList("A", new NonEmptyList("B",
    new NonEmptyList("C", new EmptyList())));
```

This is a bit tedious, and it is a good idea to supply a convenience method `cons` that calls the constructor, as well as a static variable `NIL` that is an instance of an empty list. Then our list construction becomes

```
LispList list = LispList.NIL.cons("C").cons("B").cons("A");
```

Note that you need to build up the list starting from the (empty) tail.

To see the elegance of this approach, consider the implementation of a `toString` method that produces a string containing all list elements. The method must be implemented by both classes:

```
public class EmptyList implements LispList
{
    ...
    public String toString() { return ""; }

    public class NonEmptyList implements LispList
    {
        ...
        public String toString() { return head() + " " + tail().toString(); }
    }
}
```

Note that no `if` statement is required. A list is either empty or nonempty, and the correct `toString` method is invoked due to polymorphism.

In this exercise, complete the `LispList` interface and the `EmptyList` and `NonEmptyList` classes. Write a test program that constructs a list and prints it.

- P16.10** Add a method `length` to the `LispList` interface of Exercise P16.9 that returns the length of the list. Implement the method in the `EmptyList` and `NonEmptyList` classes.

- P16.11** Add a method

```
LispList merge(LispList other)
```

to the `LispList` interface of Exercise P16.9. Implement the method in the `EmptyList` and `NonEmptyList` classes. When merging two lists, alternate between the elements, then add the remainder of the longer list. For example, merging the lists with elements 1 2 3 4 and 5 6 yields 1 5 2 6 3 4.

**•• P16.12** Add a method

```
boolean contains(Object obj)
```

to the `LispList` interface of Exercise P16.9 that returns true if the list contains an element that equals `obj`.

**•• P16.13** A deque (double-ended queue) is a data structure with operations `addFirst`, `removeFirst`, `addLast`, `removeLast`, and `size`. Implement a deque as a circular array, so that these operations have amortized constant time.

**••• P16.14** Implement a hash table with open addressing. When removing an element that is followed by other elements with the same hash code, replace it with the last such element.

**••• P16.15** Modify Exercise P16.14 to use *quadratic probing*. The  $i$ th index in the probing sequence is computed as  $(h + i^2) \% L$ .

**••• P16.16** Modify Exercise P16.14 to use *double hashing*. The  $i$ th index in the probing sequence is computed as  $(h + i h_2(k)) \% L$ , where  $k$  is the original hash key before compression and  $h_2$  is a function mapping integers to non-zero values. A common choice is  $h_2(k) = 1 + k \% q$  for a prime  $q$  less than  $L$ .

**••• P16.17** Modify Exercise P16.14 so that you mark removed elements with an “inactive” element. You can’t use `null`—that is already used for empty elements. Instead, declare a static variable

```
private static final Object INACTIVE = new Object();
```

Use the test `if (table[i] == INACTIVE)` to check whether a table entry is inactive.

## ANSWERS TO SELF-CHECK QUESTIONS

- When the list is empty, `first` is `null`. A new `Node` is allocated. Its data instance variable is set to the element that is being added. Its `next` instance variable is set to `null` because `first` is `null`. The `first` instance variable is set to the new node. The result is a linked list of length 1.
- It refers to the element to the left. You can see that by tracing out the first call to `next`. It leaves `position` to refer to the first node.
- If `position` is `null`, we must be at the head of the list, and inserting an element requires updating the `first` reference. If we are in the middle of the list, the `first` reference should not be changed.

- If an element is added after the last one, then the `last` reference must be updated to point to the new element. After

```
position.next = newNode;
```

add

```
if (position == last) { last = newNode; }
```

- public void addLast(Object element)

```
{
```

```
if (first == null) { addFirst(element); }
```

```
else
```

```
{
```

```
Node last = first;
```

```
while (last.next != null)
```

```
{
```

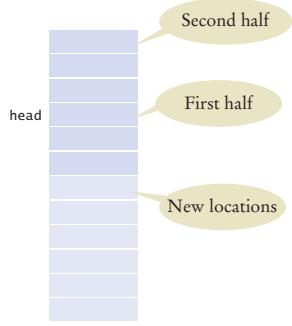
```
last = last.next;
```

```

    }
    last.next = new Node();
    last.next.data = element;
}
}

```

6.  $O(1)$  and  $O(n)$ .
7. To locate the middle element takes  $n / 2$  steps. To locate the middle of the subinterval to the left or right takes another  $n / 4$  steps. The next lookup takes  $n / 8$  steps. Thus, we expect almost  $n$  steps to locate an element. At this point, you are better off just making a linear search that, on average, takes  $n / 2$  steps.
8. In a linked list, one must follow  $k$  links to get to the  $k$ th elements. In an array list, one can reach the  $k$ th element directly as `elements[k]`.
9. In a linked list, one merely updates references to the first and second node—a constant cost that is independent of the number of elements that follow. In an array list of size  $n$ , inserting an element at the beginning requires us to move all  $n$  elements.
10. It is  $O(n)$  in both cases. In the case of the linked list, it costs  $O(n)$  steps to move an iterator to the middle.
11. It is still  $O(n)$ . Reallocating the array is an  $O(n)$  operation, and moving the array elements also requires  $O(n)$  time.
12.  $O(1)+$ . The cost of moving one element is  $O(1)$ , but every so often one has to pay for a reallocation.
13. 

```
public Object peek()
{
    if (first == null)
    {
        throw new NoSuchElementException();
    }
    return first.data;
}
```
14. Removing an element from a singly-linked list is  $O(n)$ .
15. Adding and removing an element at index 0 is  $O(n)$ .
16. The queue can be empty when the head and tail are at a position other than zero. For example, after the calls `q.add(obj)` and `q.remove()`, the queue is empty, but `head` and `tail` are 1.
17. Indeed, if the queue is empty, then the head and tail are equal. But that situation also occurs when the array is completely full.
18. Then the circular wrapping wouldn't work. If we simply added new elements without reordering the existing ones, the new array layout would be

19. Yes, the hash set will work correctly. All elements will be inserted into a single bucket.
20. Yes, but there will be a single bucket containing all elements. Finding, adding, and removing elements is  $O(n)$ .
21. The iteration takes  $O(n)$  steps. Each step makes an  $O(1)$  containment check. Therefore, the total cost is  $O(n)$ .
22. Elements are visited by increasing (compressed) hash code. This ordering will appear random to users of the hash table.
23. It locates the next bucket in the bucket array and points to its first element.
24. In a set, it doesn't make sense to add an element at a specific position.



## WORKED EXAMPLE 16.1



## Implementing a Doubly-Linked List

**Problem Statement** Provide two enhancements to the linked list implementation from Section 16.1 so that it is a doubly-linked list.

In a doubly-linked list, each node has a reference to the node preceding it, so we will add an instance variable `previous`:

```
class Node
{
    public Object data;
    public Node next;
    public Node previous;
}
```

We will also add a reference to the last node, which speeds up adding and removing elements at the end of the list:

```
public class LinkedList
{
    private Node first;
    private Node last;
    ...
}
```

We need to revisit all methods of the `LinkedList` and `ListIterator` classes to make sure that these instance variables are properly updated. We will also add methods to add, remove, and get the last element.

### Changes in the `LinkedList` Class

In the constructor, we simply add an initialization of the `last` instance variable:

```
public LinkedList()
{
    first = null;
    last = null;
}
```

The `getFirst` method is unchanged. However, in the `removeFirst` method, we need to update the `previous` reference of the node following the one that is being removed.

Moreover, we need to take into account the possibility that the list contains a single element before removal. When that element is removed, then the `last` reference needs to be set to `null`:

```
public Object removeFirst()
{
    if (first == null) { throw new NoSuchElementException(); }
    Object element = first.data;
    first = first.next;
    if (first == null) { last = null; } // List is now empty
    else { first.previous = null; }
    return element;
}
```

In the `addFirst` method, we also need to update the `previous` reference of the node following the added node. Moreover, if the list was previously empty, the new node becomes both the first and the last node:

```
public void addFirst(Object element)
{
```

```

        Node newNode = new Node();
        newNode.data = element;
        newNode.next = first;
        newNode.previous = null;
        if (first == null) { last = newNode; }
        else { first.previous = newNode; }
        first = newNode;
    }
}

```

### New Methods for Accessing the Last Element of the List

The `getLast`, `removeLast`, and `addLast` methods are the mirror opposites of the `getFirst`, `removeFirst`, and `addFirst` methods, where the roles of `first/last` and `next/previous` are switched.

```

public Object getLast()
{
    if (last == null) { throw new NoSuchElementException(); }
    return last.data;
}

public Object removeLast()
{
    if (last == null) { throw new NoSuchElementException(); }
    Object element = last.data;
    last = last.previous;
    if (last == null) { first = null; } // List is now empty
    else { last.next = null; }
    return element;
}

public void addLast(Object element)
{
    Node newNode = new Node();
    newNode.data = element;
    newNode.next = null;
    newNode.previous = last;
    if (last == null) { first = newNode; }
    else { last.next = newNode; }
    last = newNode;
}

```

Compare `removeLast/addLast` with the `removeFirst/addFirst` methods given above and pay attention to the `first/last` and `next/previous` references!

### The Bidirectional Iterator

In the `ListIterator` class, we no longer need to store the `previous` reference because we can reach the preceding node as `position.previous`. We can simply remove it from the constructor and the `next` method. (Recall that this reference was required to support the iterator's `remove` operation.)

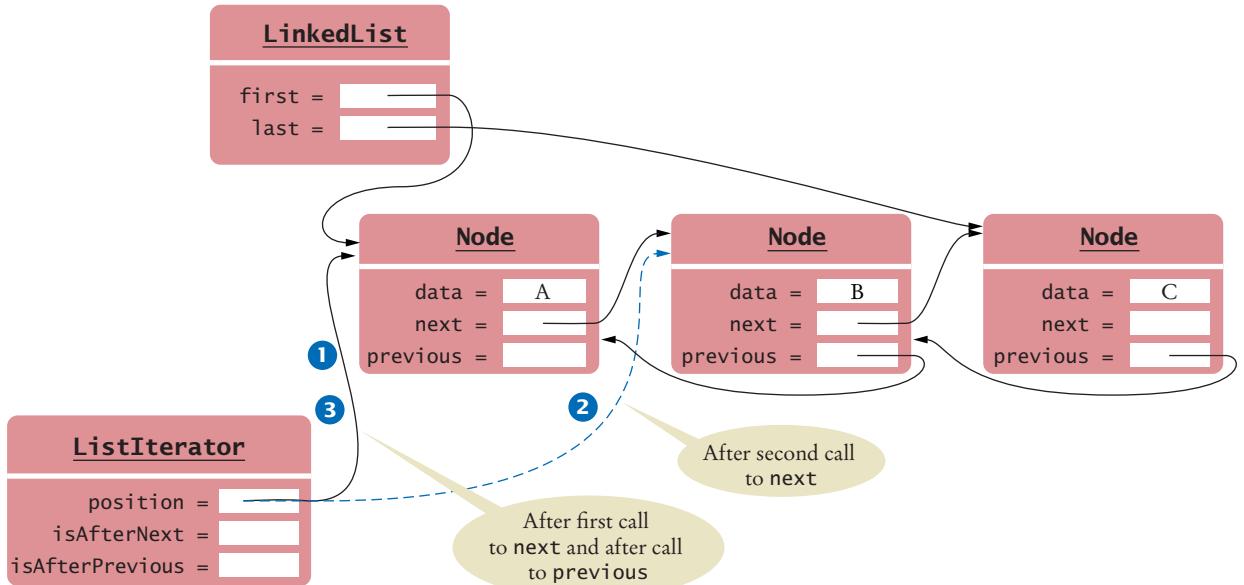
In a doubly-linked list, the iterator can move forward and backward. For example,

```

LinkedList lst = new LinkedList();
lst.addLast("A");
lst.addLast("B");
lst.addLast("C");
ListIterator iter = lst.listIterator(); // The iterator is before the first element |ABC
iter.next(); // Returns "A"; the iterator is after the first element A|BC ①
iter.next(); // Returns "B"; the iterator is after the second element AB|C ②
iter.previous(); // Returns "B"; the iterator is after the first element A|BC ③

```

The previous method is similar to the next method. However, it returns the value *after* the iterator position. That is perhaps not so intuitive, and it is best to draw a diagram to verify the point. In the figure below, we show two calls to next, followed by a call to previous, as in the code example above. Recall that an iterator conceptually points *between* elements, like the cursor of a word processor, and that the position reference of the iterator points to the element *to the left* (or to null when it is at the beginning of the list).



As you can see, a call to previous moves the iterator backward, and the element that is returned is the one to which it pointed before being moved:

```
public Object previous()
{
    if (!hasPrevious()) { throw new NoSuchElementException(); }
    isAfterNext = false;
    isAfterPrevious = true;

    Object result = position.data;
    position = position.previous;
    return result;
}
```

### Removing and Setting Elements Through an Iterator

Note the `isAfterNext` and `isAfterPrevious` variables in the previous method. They track whether the iterator just carried out a `next` or `previous` call (or neither of the two). This information is needed for implementing the `remove` and `set` methods.

These methods remove or set the element that the iterator just traversed, which is positioned after a call to `next` or `position.next` after a call to `previous`. (If calling `previous` sets `position` to `null` because we reached the front of the list, then we remove or set `first`.) The following helper method computes this node:

```
private Node lastPosition()
{
    if (isAfterNext)
    {
```

```

        return position;
    }
    else if (isAfterPrevious)
    {
        if (position == null)
        {
            return first;
        }
        else
        {
            return position.next;
        }
    }
    else { throw new IllegalStateException(); }
}

```

With this helper method, the set method is simple:

```

public void set(Object element)
{
    Node positionToSet = lastPosition();
    positionToSet.data = element;
}

```

The remove method also uses the lastPosition helper method. To ensure that the first and last references are properly updated, we have separate cases for removing the first or last element. Note that the iterator moves one step back when calling remove after next, and it stays at the same position when calling remove after previous.

```

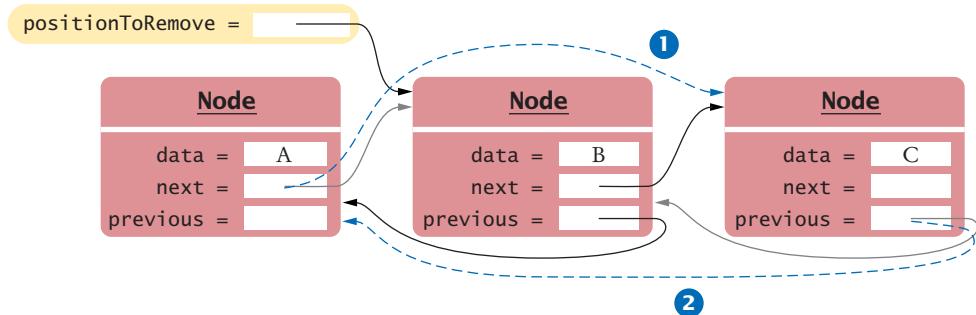
public void remove()
{
    Node positionToRemove = lastPosition();
    if (positionToRemove == first)
    {
        removeFirst();
    }
    else if (positionToRemove == last)
    {
        removeLast();
    }
    else
    {
        positionToRemove.previous.next = positionToRemove.next; ❶
        positionToRemove.next.previous = positionToRemove.previous; ❷
    }

    if (isAfterNext)
    {
        position = position.previous;
    }

    isAfterNext = false;
    isAfterPrevious = false;
}

```

The most complex part of this method is the routing of the next and previous references around the removed elements, which is highlighted above. We know that `positionToRemove.previous` and `positionToRemove.next` are not `null` because we don't remove the first or last element. The following figure shows how the references are updated.



### Testing the Implementation

This implementation is so complex that it is unlikely to be implemented correctly at first try. (In fact, I made several errors when I wrote this section.) It is essential to provide a suite of test cases that checks the integrity of all references after every operation, and to test adding and removing elements at either end and in the middle.

Suppose we have a list of strings that should contain nodes for "A", "B", "C", and "D". We can test the first and last references by verifying that `getFirst` and `getLast` return "A" and "D". To check the next references of all nodes, we can get an iterator and call the `next` method four times, checking that we get "A", "B", "C", and "D". Then we call `hasNext`, expecting `false`, to check for a `null` in the `next` instance variable of the last node. To check the previous references, call `previous` four times on the same iterator and check for "D", "C", "B", and "A". Finally, check that `hasPrevious` returns `false`. These checks ensure that all references are intact.

We provide a test method `check` for this purpose. For example,

```
LinkedList lst = new LinkedList();
check("", lst, "Constructing empty list");
lst.addLast("A");
check("A", lst, "Adding last to empty list");
lst.addLast("B");
check("AB", lst, "Adding last to non-empty list");
```

The `check` method has three arguments: the expected contents (as a string—we assume each node contains a string of length 1), the list, and a string describing the test. The strings are used to print messages such as

```
Passed "Constructing empty list".
Passed "Adding last to empty list".
Passed "Adding last to non-empty list".
```

When implementing the `check` method, we use a helper method `assertEquals` that checks whether an expected value equals an actual one. If it doesn't, an exception is thrown. For example,

```
assertEquals(expected.substring(0, 1), actual.getFirst());
```

You can find the implementation of the `check` and `assertEquals` methods and the provided test cases in the `LinkedListTest` class at the end of this example.

### [worked\\_example\\_1/LinkedList.java](#)

```
1 import java.util.NoSuchElementException;
2
3 /**
4  * An implementation of a doubly-linked list.
5 */
```

## WE6 Chapter 16 Basic Data Structures

```
6  public class LinkedList
7  {
8      private Node first;
9      private Node last;
10
11     /**
12      Constructs an empty linked list.
13     */
14     public LinkedList()
15     {
16         first = null;
17         last = null;
18     }
19
20    /**
21     Returns the first element in the linked list.
22     @return the first element in the linked list
23     */
24     public Object getFirst()
25     {
26         if (first == null) { throw new NoSuchElementException(); }
27         return first.data;
28     }
29
30    /**
31     Removes the first element in the linked list.
32     @return the removed element
33     */
34     public Object removeFirst()
35     {
36         if (first == null) { throw new NoSuchElementException(); }
37         Object element = first.data;
38         first = first.next;
39         if (first == null) { last = null; } // List is now empty
40         else { first.previous = null; }
41         return element;
42     }
43
44    /**
45     Adds an element to the front of the linked list.
46     @param element the element to add
47     */
48     public void addFirst(Object element)
49     {
50         Node newNode = new Node();
51         newNode.data = element;
52         newNode.next = first;
53         newNode.previous = null;
54         if (first == null) { last = newNode; }
55         else { first.previous = newNode; }
56         first = newNode;
57     }
58
59    /**
60     Returns the last element in the linked list.
61     @return the last element in the linked list
62     */
63     public Object getLast()
64     {
65         if (last == null) { throw new NoSuchElementException(); }
```