CHAPTER 8

DESIGNING CLASSES

© Ivan Stevanovic/iStockphoto.

CHAPTER GOALS

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

CHAPTER CONTENTS

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

8.1 Discovering Classes

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

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`

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

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.

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**.

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)
   . . .
}
```

The members of a cohesive team have a common goal.

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 >-shaped open arrow tip that points to the dependent class. Figure 1 shows a class diagram indicating that the `CashRegister` class depends on the `Coin` class.

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

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

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

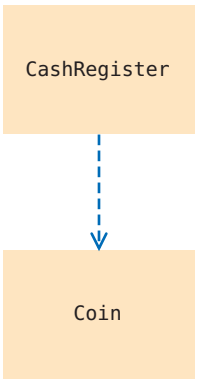


Figure 1
Dependency Relationship Between the `CashRegister` and `Coin` Classes

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

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

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

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

8.2.3 Separating Accessors and Mutators

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

An immutable class has no mutator methods.

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

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

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

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

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

In mutable classes, it is still a good idea to cleanly separate accessors and mutators, in order to avoid accidental mutation. As a rule of thumb, a method that returns a value should not be a mutator. For example, one would not expect that calling `getBalance` on a `BankAccount` object would change the balance. (You would be pretty upset

if your bank charged you a "balance inquiry fee".) If you follow this rule, then all mutators of your class have return type `void`.

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

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

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

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

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

8.2.4 Minimizing Side Effects

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

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

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

```
/**
   Computes the total balance of the given accounts.
   @param accounts a list of bank accounts
*/
   public double getTotalBalance(ArrayList<BankAccount> 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.

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.

When designing methods, minimize side effects.

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.

EXAMPLE CODE See your eText or companion code for a cash register program that uses the `Coin` class.

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

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

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

Other programming languages such as C++ support a mechanism, called "call by reference", that can change the arguments of a method call. You will sometimes read in Java books that "numbers are passed by value, objects are passed by reference". That is technically not

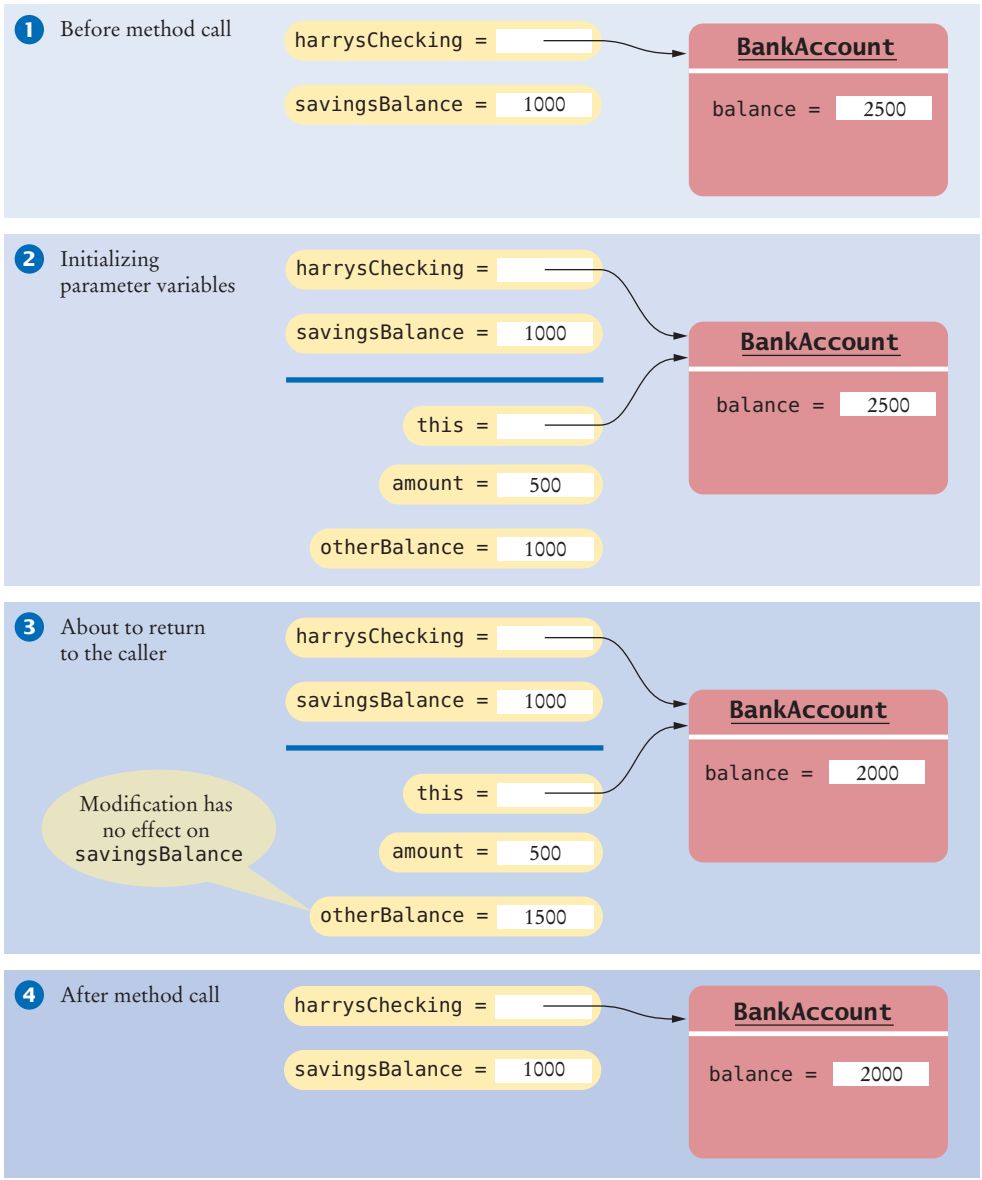


Figure 2 Modifying a Parameter Variable of a Primitive Type Has No Effect on Caller

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) ❷
   {
      balance = balance - amount;
      otherAccount.deposit(amount);
   } ❸
}
```

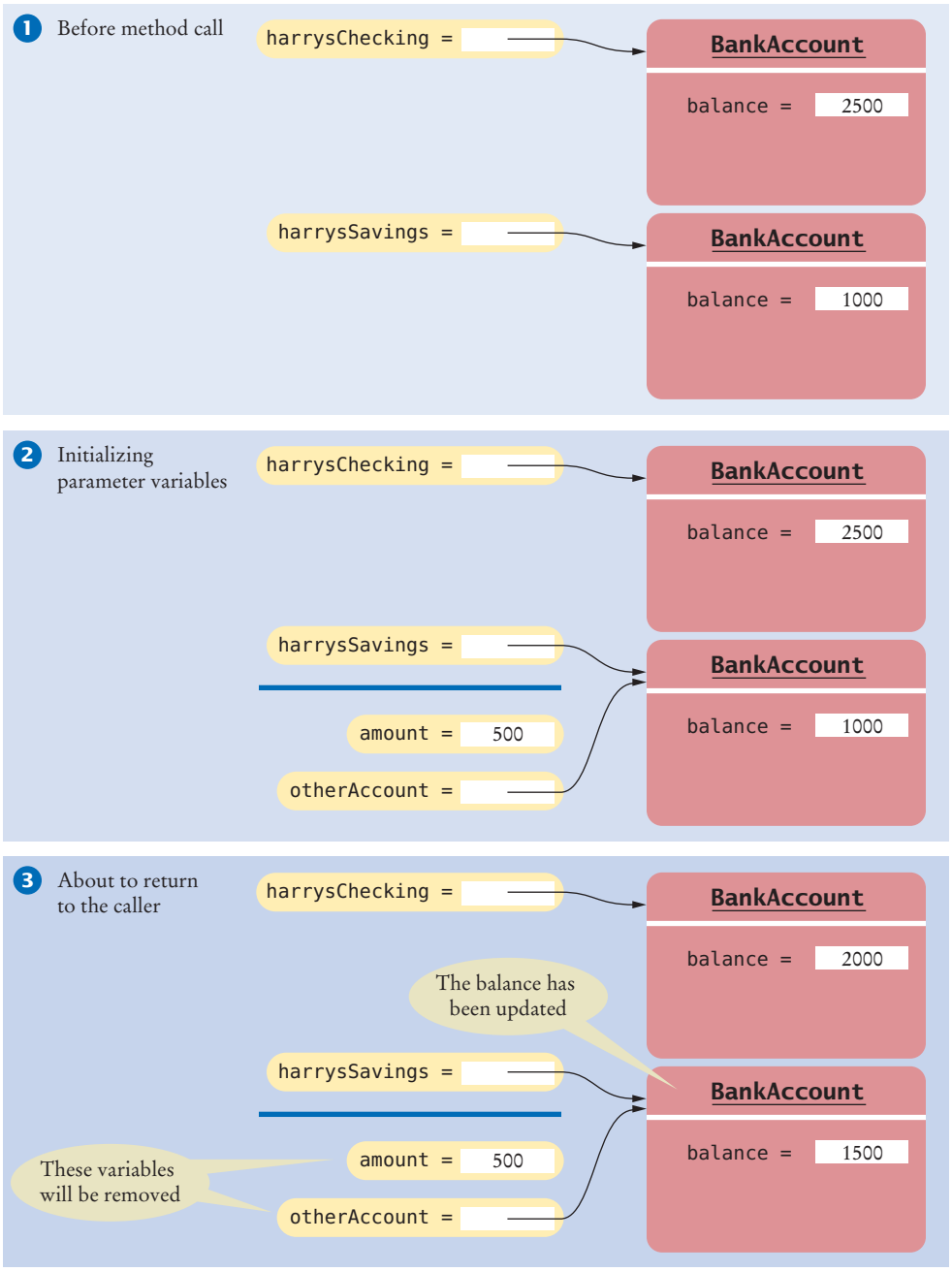


Figure 3 Methods Can Mutate Any Objects to Which They Hold References

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 is means to make a copy of an object reference—you get another reference to the same object. Through that reference, the method is able to modify the object.

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

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

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

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.

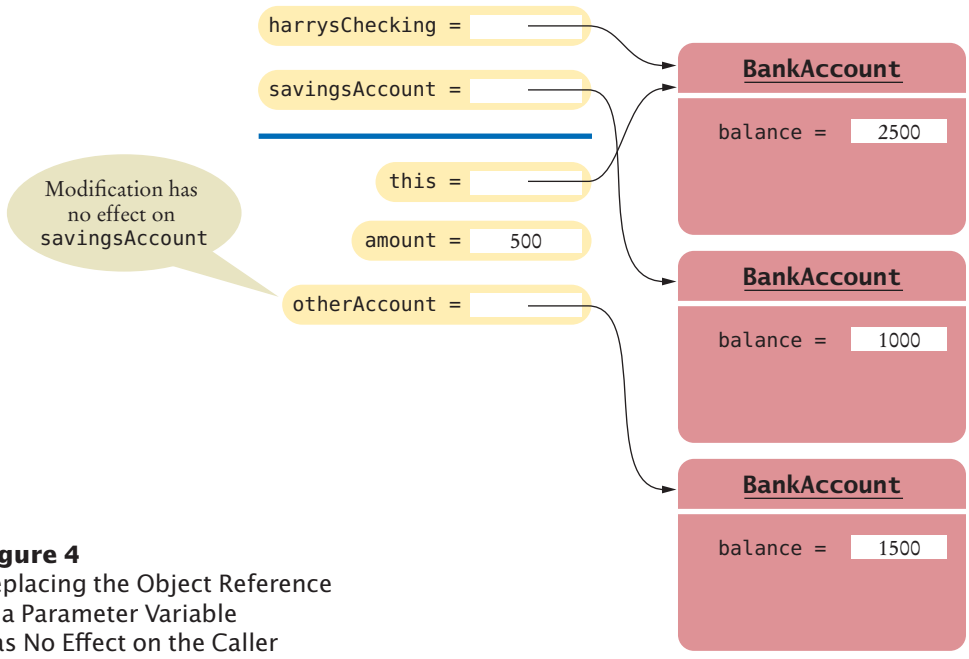


Figure 4
Replacing the Object Reference in a Parameter Variable Has No Effect on the Caller

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.

EXAMPLE CODE See special_topic_1 of your eText or companion code for the program demonstrating call by value.

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

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.

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

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

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.

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

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

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.

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

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

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

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

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

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

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

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

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

8.3.4 Managing Properties of an Object

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

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

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

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

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

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

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

EXAMPLE CODE See sec03_04 of your eText or companion code for a class with getter and setter methods.

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++; }
}
```

© John Alexander/iStockphoto.

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

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.
   }
   . . .
}
```

EXAMPLE CODE See sec03_05 of your eText or companion code for the `Fish` class.

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;
```

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:

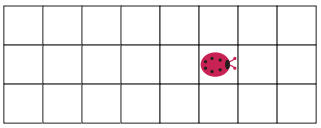
```
private double zPosition;
private double zVelocity;
```

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--; }
}
```



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

Exercise ••• Science P8.11 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.

EXAMPLE CODE See sec03_06 of your eText or companion code for two classes that update position data.

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.

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.*

© Diane Diederich/iStockphoto.

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.

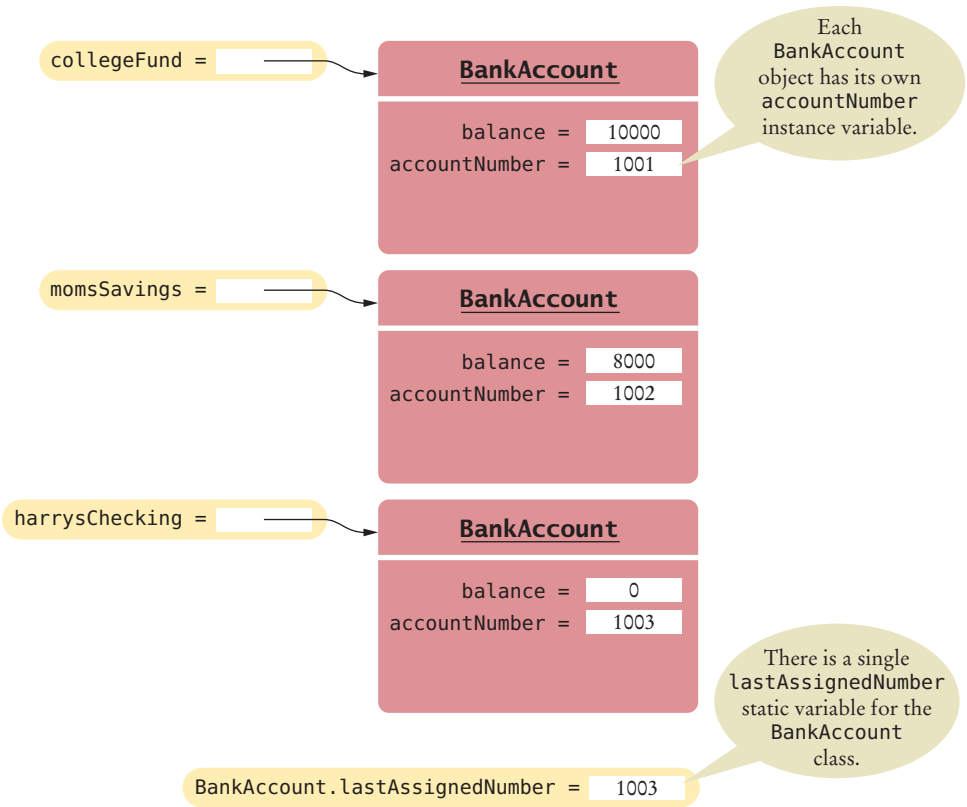


Figure 5 A Static Variable and Instance Variables

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`.

A static method is not invoked on an object.

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

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

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

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

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

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

EXAMPLE CODE See sec04 of your eText or companion code for a bank account class that includes static variables and uses a static method.

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.

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 ••• E8.9).

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.

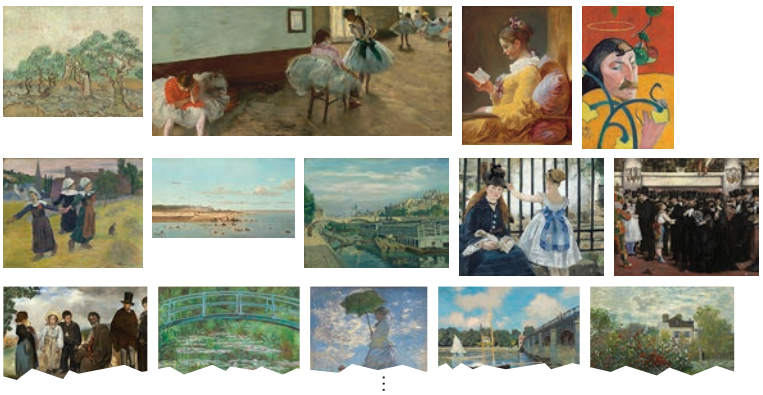
As you learn more about programming, the complexity of the tasks that you are asked to solve will increase. When you face a complex task, you should apply an important skill: simplifying the problem, and solving the simpler problem first.

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.

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

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 Credits page for details.)

A `Picture` class is given to you. It has methods

```
public void load(String source)
public int getWidth()
public int getHeight()
public void move(int dx, int dy)
```

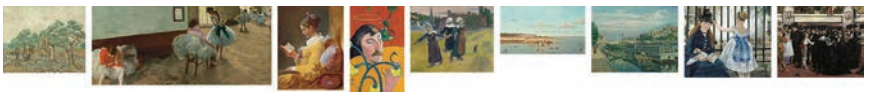
Instead of tackling the entire assignment at once, here is a plan that solves a series of simpler problems.

1. Draw one picture.

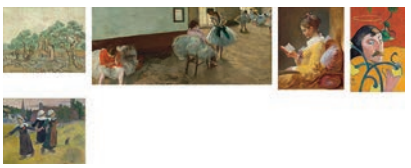
2. Draw two pictures next to each other.

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();
      pic.load("picture1.jpg");
   }
}
```

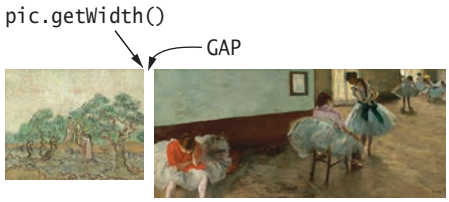
That's enough to show the picture.

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();
pic.load("picture1.jpg");
Picture pic2 = new Picture();
```

```
pic2.load("picture2.jpg");
pic2.move(pic.getWidth(), 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();
pic.load("picture1.jpg");
Picture pic2 = new Picture();
pic2.load("picture2.jpg");
int x = pic.getWidth() + 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();
pic.load("picture1.jpg");
int x = 0;
for (int i = 2; i <= PICTURES; i++)
{
   Picture previous = pic;
   pic = new Picture();
   pic.load("picture" + i + ".jpg");
   x = x + previous.getWidth() + 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`.

```
x = x + previous.getWidth() + GAP;
if (x + pic.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.getHeight());
```

The following statement places a picture on the next row:

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

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.

EXAMPLE CODE See your companion code for the listings of all preliminary stages of the gallery program.

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

sec05/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();
10       pic.load("picture1.jpg");
11       int x = 0;
12       int y = 0;
13       int maxY = 0;
14
15       for (int i = 2; i <= PICTURES; i++)
16       {
17          maxY = Math.max(maxY, y + pic.getHeight());
18          Picture previous = pic;
19          pic = new Picture();
20          pic.load("picture" + i + ".jpg");
21          x = x + previous.getWidth() + GAP;
22          if (x + pic.getWidth() >= MAX_WIDTH)
23          {
24             x = 0;
25             y = maxY + GAP;
```

```
26          }
27          pic.move(x, y);
28       }
29    }
30 }
```

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 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.

In Java, related classes are grouped into packages.

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`.

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.

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` 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`.)

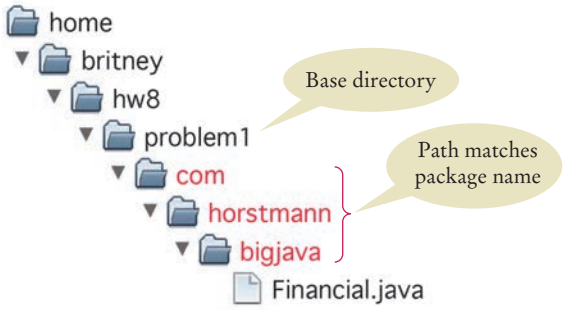


Figure 6 Base Directories and Subdirectories for Packages

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.

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;
   . . .
}
```

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.

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/iStockphoto.

Step 1 Come up with a package name.

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

Step 2 Pick a *base directory*.

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

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

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

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

or

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

Step 4 Place your source files into the package subdirectory.

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

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

or

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

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

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

```
package homework1.problem2;
```

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

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

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

or

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

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

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

Unlike the Java compiler, the Java interpreter needs the *class name (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

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

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

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

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

In each test case, you make some computations and then compute some condition that you believe to be true. You then pass the result to a method that communicates a test result to the framework, most commonly the `assertEquals` method.

The `assertEquals` method takes as arguments the expected and actual values and, for floating-point numbers, a tolerance value.

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

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

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

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

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

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

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

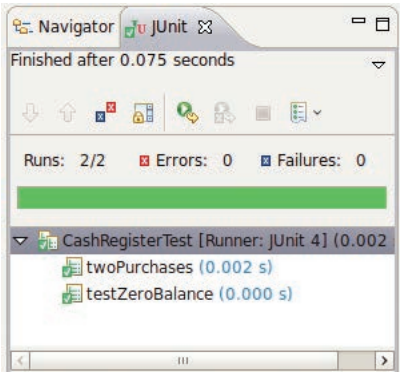


Figure 7 Unit Testing with JUnit

Computing & Society 8.1 Personal Computing

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

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

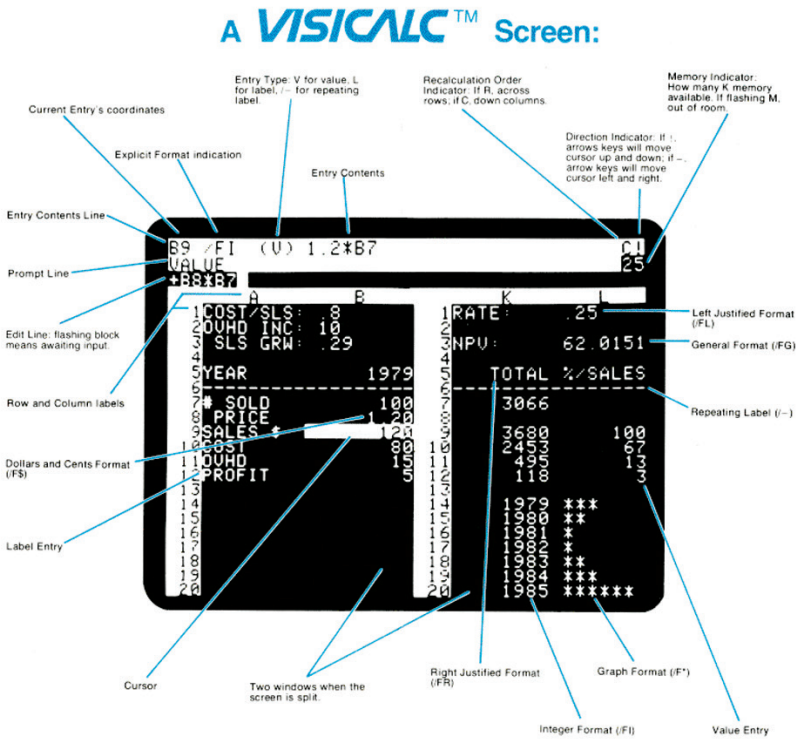
The first big hit was the Apple II. It was a real computer with a keyboard, a monitor, and a floppy disk drive. When it was first released, users had a $3,000 machine that could play Space Invaders, run a primitive bookkeeping program, or let users program it in BASIC. The original Apple II did not even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

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

Personal computers have been with us ever since, and countless users have tinkered with their hardware and software. 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. 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.

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?



Reprinted Courtesy of International Business Machines Corporation, © International Business Machines Corporation/IBM Corporation.

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:

a. `Invoice`

b. `InvoicePrinter`

c. `PrintInvoice`

d. `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 •• Science P8.13 a mutable or immutable class? Why?

■ **R8.14** Which of the following classes are immutable?

a. `Rectangle`
b. `String`
c. `Random`

■ **R8.15** Which of the following classes are immutable?

a. `PrintStream`
b. `Date`
c. `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** 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?

■■ **R8.27** 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?

■■ **R8.28** 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.

■ **R8.29** Is a Java program without `import` statements limited to using the default and `java.lang` packages?

■ **R8.30** 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`?

■■ **R8.31** 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.

■■ **R8.32** 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?

■■ **R8.33** 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.

■ **R8.34** What is the default package? Have you used it before this chapter in your programming?

■■ **Testing R8.35** 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** As pointed out in Section 8.2.3, the `Scanner.next` method is a mutator that returns a value. Implement a class `Reader` that reads from `System.in` and does not suffer from that shortcoming. Provide methods

```
public boolean hasMoreElements()   // Checks whether there is another element
public String getCurrent()   // Yields the current element without consuming it
public void next()   // Consumes the current element
```

■■ **E8.5** Reimplement the `BankAccount` class so that it is immutable. The `deposit` and `withdraw` methods need to return new `BankAccount` objects with the appropriate balance.

■ **E8.6** Reimplement the `Day` class of Worked Example 2.1 to be mutable. Change the methods `addDays`, `nextDay`, and `previousDay` to mutate the implicit parameter and to return `void`. Also change the demonstration program.

■■ **E8.7** Write static methods

© DNY59/iStockphoto.

- `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.

■■ **E8.8** Solve Exercise •• E8.7 by implementing classes `Cube`, `Sphere`, `Cylinder`, and `Cone`. Which approach is more object-oriented?

■■■ **E8.9** 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.

■■■ **E8.10** Repeat Exercise ••• E8.9, using multiple classes. Provide a `GradeBook` class that collects objects of type `Student`.

■■■ **E8.11** 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?

■■ **E8.12** 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?

■■■ **E8.13** 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`.

■■ **E8.14** 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.

■■ **E8.15** 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`.

■ **E8.16** 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`.

■■ **E8.17** 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`.

■■ **E8.18** Improve the `Die` class of Chapter 6. Turn the `generator` variable into a static variable so that all dice share a single random number generator.

■■ **E8.19** 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.

■ **E8.20** 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 E8.21** 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 E8.22** Provide JUnit test class `TaxReturnTest` with three test methods that test different tax situations for the `TaxReturn` class in Chapter 5.

■ **Graphics E8.23** 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 E8.24** Repeat Exercise • Graphics E8.23 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?

■■ **E8.25** 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.

■■ **E8.26** 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.

■■ **E8.27** 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

© pixhook/iStockphoto.

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.

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

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** In a tile matching game, tiles are arranged on a grid with rows and columns. Two tiles are turned over at a time, and if they match, the player earns a point. If they are adjacent, the player earns an additional point. Implement such a game, using classes `Tile`, `Location` (encapsulating a row and column), `Grid`, and `MatchingGame`. You can implement the game using pictures, as in Section 8.5, or tiles with words. When designing your classes, pay attention to cohesion and coupling.

■■ **P8.4** 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.

■■ **P8.5** In Exercise •• P8.4, 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.

■■ **P8.6** 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.

■■■ **P8.7** 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 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.

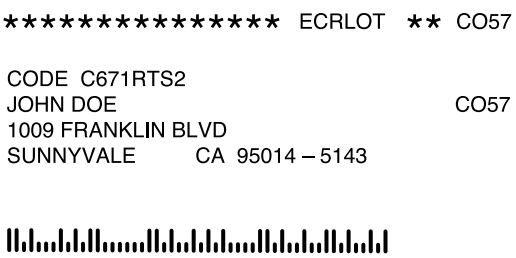


Figure 8 A Postal Bar Code

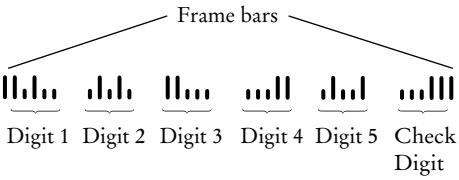


Figure 9 Encoding for Five-Digit Bar Codes

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.

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

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

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

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

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

■■■ **Business P8.8** 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 P8.9 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 P8.10 The Downtown Marketing Association wants to promote downtown shopping with a loyalty program similar to the one in Exercise •• Business P8.9. 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 P8.11 Design a class `Cannonball` to model a cannonball that is fired into the air. A ball has

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

Supply the following methods:

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

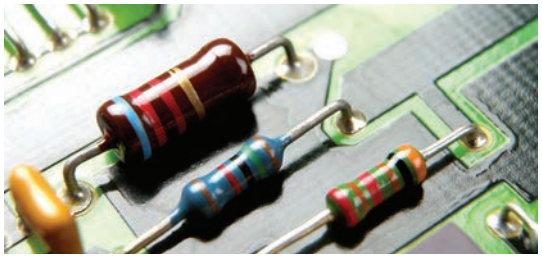
Use this class in a program that prompts the user for the starting angle and the initial velocity. Then call `shoot` and print the locations.

■ Graphics P8.12 Continue Exercise ••• Science P8.11, and draw the trajectory of the cannonball.

■■ Science P8.13 The colored bands on the top-most resistor shown in the photo below indicate a resistance of 6.2 kΩ ±5 percent. The resistor tolerance of ±5 percent indicates the acceptable variation in the resistance. A 6.2 kΩ ±5 percent resistor could have a resistance as small as 5.89 kΩ or as large as 6.51 kΩ. We say that 6.2 kΩ is the *nominal*

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

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

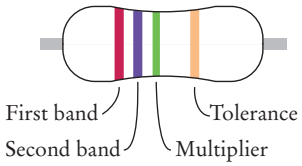


© Maria Toutoudaki/iStockphoto.

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

▪▪ Science P8.14 In the `Resistor` class from Exercise •• Science P8.13, supply a method that returns a description of the "color bands" for the resistance and tolerance. A resistor has four color bands:

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



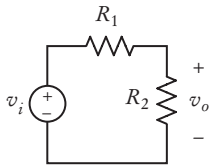
Color	DIgit	Multiplier	Tolerance
Black	0	$\times 10^0$	—
Brown	1	$\times 10^1$	±1%
Red	2	$\times 10^2$	±2%
Orange	3	$\times 10^3$	—
Yellow	4	$\times 10^4$	—
Green	5	$\times 10^5$	±0.5%
Blue	6	$\times 10^6$	±0.25%
Violet	7	$\times 10^7$	±0.1%
Gray	8	$\times 10^8$	±0.05%
White	9	$\times 10^9$	—
Gold	—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

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

▪▪ Science P8.15 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 •• Science P8.13. 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 •• Science P8.13. 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 $R_2 = 750\ \Omega$.