Chapter 8 Designing Classes

CHAPTER GOALS

- To learn how to choose appropriate classes to implement
- To understand the concepts of cohesion and coupling
- To minimize the use of side effects
- To document the responsibilities of methods and their callers with preconditions and postconditions
- To understand the difference between instance methods and static methods
- To introduce the concept of static fields
- To understand the scope rules for local variables and instance fields
- To learn about packages
- **T** To learn about unit testing frameworks

In this chapter you will learn more about designing classes. First, we will discuss the process of discovering classes and defining methods. Next, we will discuss how the concepts of pre- and postconditions enable you to specify, implement, and invoke methods correctly. You will also learn about several more technical issues, such as static methods and variables. Finally, you will see how to use packages to organize your classes.

8.1 Choosing 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.

Students who have prior experience with programming in another programming language are used to programming *functions*. A function carries out an action. In object-oriented programming, the actions appear as methods. Each method, however, belongs to a class. Classes are collections of objects, and objects are not actions—they are entities. So you have to start the programming activity by identifying objects and the classes to which they belong.

Remember the rule of thumb from Chapter 2: Class names should be nouns, and method names should be verbs.

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

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

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

336

337

Other classes are abstractions of real-life entities.

- `BankAccount`
- `CashRegister`

For these classes, the properties of a typical object are easy to understand. A `Rectangle` object has a width and height. Given a `BankAccount` object, you can deposit and withdraw money. Generally, concepts from the part of the universe that a program concerns, such as science, business, or a game, make good classes. The name for such a class should be a noun that describes the concept. Some of the standard `Java` class names are a bit strange, such as `Ellipse2D.Double`, but you can choose better names for your own classes.

Another useful category of classes can be described as *actors*. Objects of an actor class do some kinds of work 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 a typical 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, particularly by students who are used to writing programs that consist of functions, is to turn an action into a class. For example, if your homework assignment is to compute a paycheck, you may consider writing a class `ComputePaycheck`. But can you visualize a "ComputePaycheck" object? The fact that "ComputePaycheck" isn't a noun tips you off that you are on the wrong track. On the other hand, a `Paycheck` class makes intuitive sense. The word "paycheck" is a noun. You can visualize a paycheck object. You can then think about useful methods of the `Paycheck` class, such as `computeTaxes`, that help you solve the assignment.

Self Check

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

8.2 Cohesion and Coupling

In this section you will learn two useful criteria for analyzing the quality of the public interface of a class.

A class should represent a single concept. The public methods and constants that the public interface exposes should be *cohesive*. That is, all interface features should be closely related to the single concept that the class represents.

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

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

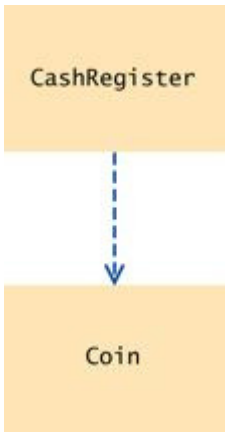
There are really two concepts here: a cash register that holds coins and computes their total, and the values of individual coins. (For simplicity, we assume that the cash register only holds coins, not bills. Exercise P8.1 discusses a more general solution.)

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

```
}
```

Figure 1



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

338

339

Then the `CashRegister` class can be simplified:

```
public class CashRegister
{
   public void enterPayment(int coinCount, Coin
coinType) { . . . }
   . . .
}
```

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.

Many classes need other classes in order to do their jobs. For example, the restructured `CashRegister` class now depends on the `Coin` class to determine the value of the payment.

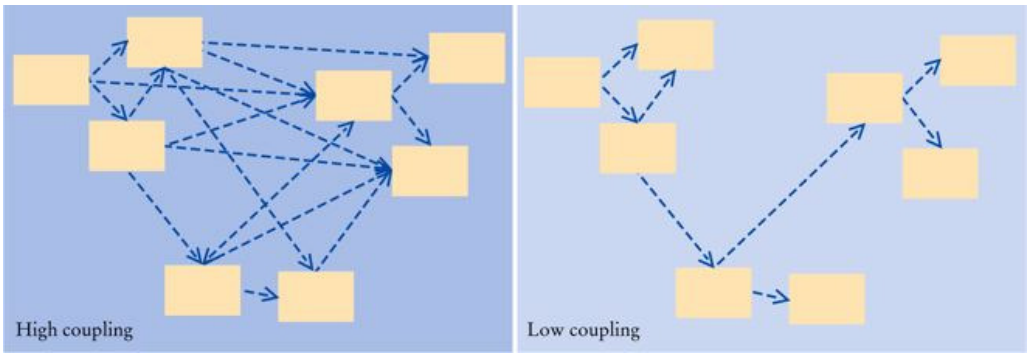
A class depends on another class if it uses objects of that class.

To visualize relationships, such as dependence between classes, 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. 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. Coins have no idea that they are being collected in cash registers, and they can carry out their work without ever calling any method in the `CashRegister` class.

If many classes of a program depend on each other, then we say that the *coupling* between classes is high. Conversely, if there are few dependencies between classes, then we say that the coupling is low (see Figure 2).

Figure 2



High and Low Coupling Between Classes

339

340

Why does coupling matter? If the `Coin` class changes in the next release of the program, all the classes that depend on it may be affected. If the change is drastic, the coupled classes must all be updated. Furthermore, if we would like to use a class in

another program, we have to take with it all the classes on which it depends. Thus, we want to remove unnecessary coupling between classes.

It is a good practice to minimize the coupling (i.e., dependency) between classes.

SELF CHECK

3. Why is the `CashRegister` class from Chapter 4 not cohesive?
4. Why does the `Coin` class not depend on the `CashRegister` class?
5. Why should coupling be minimized between classes?

QUALITY TIP 8.1: Consistency

In this section you learned of two criteria to analyze the quality of the public interface of a class. You should maximize cohesion and remove unnecessary coupling. 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 parameters. 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` parameter? It turns out that the `showMessageDialog` method needs a parameter 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 a fatal flaw, but they are an annoyance, particularly because they can be so easily avoided.

340

8.3 Accessors, Mutators, and Immutable Classes

Recall that a *mutator method* modifies the object on which it is invoked, whereas an *accessor method* merely accesses information without making any modifications. For example, in the `BankAccount` class, the `deposit` and `withdraw` methods are mutator methods. Calling

```
account.deposit(1000);
```

modifies the state of the `account` object, but calling

```
double balance = account.getBalance();
```

does not modify the state of `account`.

You can call an accessor method as many times as you like—you always get the same answer, and it does not change the state of your object. That is clearly a desirable property, because it makes the behavior of such a method very predictable. 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 contents never change. No method in the `String` class can modify the contents of a string. For example, the `toUpperCase` method does not change characters from the original string. Instead, it constructs a *new* string that contains the uppercase characters:

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

An immutable class has no mutator methods.

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. In contrast, if you give out a `BankAccount` reference to any other method, you have to be aware that the state of your object may change—the other method can call the `deposit` and `withdraw` methods on the reference that you gave it.

SELF CHECK

6. Is the `substring` method of the `String` class an accessor or a mutator?
7. Is the `Rectangle` class immutable?

8.4 Side Effects

A mutator method modifies the object on which it is invoked, whereas an accessor method leaves it unchanged. This classification relates only to the object on which the method is invoked.

341

342

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.

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

Here is an example of a method with another kind of side effect, the updating of an explicit parameter:

```
public class BankAccount
{
   /**
      Transfers money from this account to another
account.
      @param amount the amount of money to transfer
      @param other the account into which to
transfer the money
   */
   public void transfer(double amount, BankAccount
other)
   {
      balance = balance - amount;
      other.balance = other.balance + amount;
   }
   . . .
}
```

As a rule of thumb, updating an explicit parameter can be surprising to programmers, and it is best to avoid it whenever possible.

> You should minimize side effects that go beyond modification of the implicit parameter.

Another example of a side effect is output. Consider how we have always printed a bank balance:

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

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

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

That would be more convenient when you actually want to print the value. But, of course, there are cases when you want the value for some other purpose. Thus, you can't simply drop the `getBalance` method in favor of `printBalance`.

More importantly, the `printBalance` method forces strong assumptions on the `BankAccount` class.

- The message is in English—you assume that the user of your software reads English. The majority of people on the planet don't.
- You rely on `System.out`. A method that relies on `System.out` won't work in an embedded system, such as the computer inside an automatic teller machine.

In other words, this design violates the rule of minimizing the coupling of the classes. The `printBalance` method couples the `BankAccount` class with the `System` and `PrintStream` classes. It is best to decouple input/output from the actual work of your classes.

342

Self Check

8. If a refers to a bank account, then the call `a.deposit(100)` modifies the bank account object. Is that a side effect?

9. Consider the `DataSet` class of Chapter 6. Suppose we add a method

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

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

Common Error 8.1: Trying to Modify Primitive Type Parameters

Methods can't update parameters of primitive type (numbers, `char`, and `boolean`). To illustrate this point, let us try to write a method that updates a number parameter:

```
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
   */
   void transfer(double amount, double
otherBalance)
   {
      balance = balance - amount;
      otherBalance = otherBalance + amount;
         // Won't work
   }
   . . .
}
```

This doesn't work. Let's consider a method call.

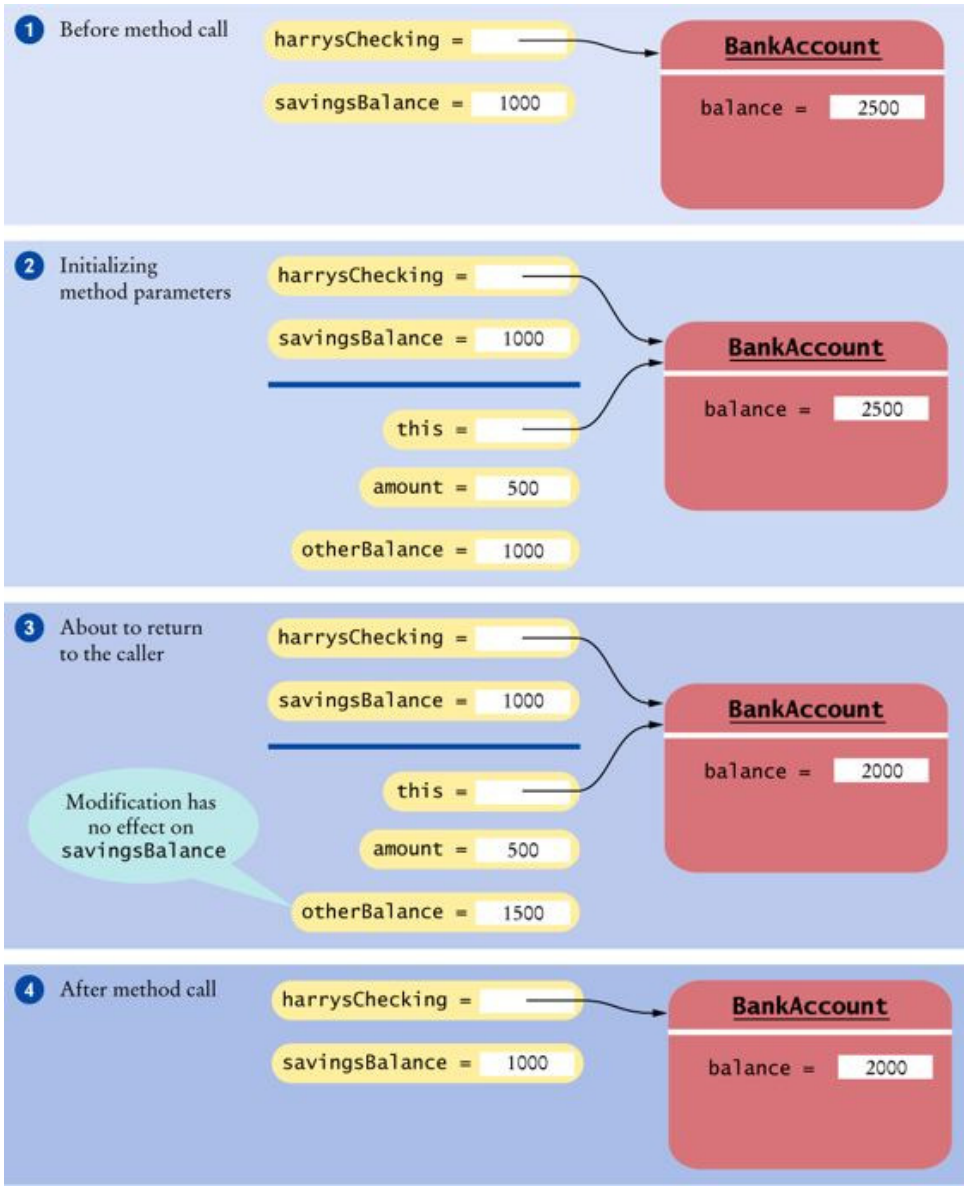
```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance);
System.out.println(savingsBalance);
```

As the method starts, the parameter variable `otherBalance` is set to the same value as `savingsBalance`. Then the value of the `otherBalance` value is modified, but that modification has no effect on `savingsBalance`, because `otherBalance` is a separate variable (see Figure 3). When the method terminates, the `otherBalance` variable dies, and `savingsBalance` isn't increased.

> In Java, a method can never change parameters of primitive type.

Why did the example at the beginning of Section 8.4 work, where the second explicit parameter was a `BankAccount` reference? Then the parameter variable contained a copy of the object reference. Through that reference, the method is able to modify the object.

Figure 3



Modifying a Numeric Parameter Has No Effect on Caller

You already saw this difference between objects and primitive types in Chapter 2. As a consequence, a Java method can *never* modify numbers that are passed to it.

QUALITY TIP 8.2: Minimize Side Effects

In an ideal world, all methods would be accessors that simply return an answer without changing any value at all. (In fact, programs that are written in so-called *functional* programming languages, such as Scheme and ML, come close to this ideal.) Of course, in an object-oriented programming language, we use objects to remember state changes. Therefore, a method that just changes the state of its implicit parameter is certainly acceptable. Although side effects cannot be completely eliminated, they can be the cause of surprises and problems and should be minimized. Here is a classification of method behavior.

- Accessor methods with no changes to any explicit parameters—no side effects. Example:`getBalance`.
- Mutator methods with no changes to any explicit parameters—an acceptable side effect. Example: `withdraw`.
- Methods that change an explicit parameter—a side effect that should be avoided when possible. Example: `transfer`.
- Methods that change another object (such as System.out)—a side effect that should be avoided. Example: `printBalance`.

QUALITY TIP 8.3: Don't Change the Contents of Parameter Variables

As explained in Common Error 8.1 and Advanced Topic 8.1, a method can treat its parameter variables like any other local variables and change their contents. However, that change affects only the parameter variable within the method itself—not any values supplied in the method call. Some programmers take "advantage" of the temporary nature of the parameter variables and use them as "convenient" holders for intermediate results, as in this example:

```
public void deposit(double amount)
{
   // Using the parameter variable to hold an intermediate value
   amount = balance + amount; // Poor style
```

```
      . . .
   }
```

That code would produce errors if another statement in the method referred to amount expecting it to be the value of the parameter, and it will confuse later programmers maintaining this method. You should always treat the parameter variables as if they were constants. Don't assign new values to them. Instead, introduce a new local variable.

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

345

346

ADVANCED TOPIC 8.1: Call by Value and Call by Reference

In Java, method parameters are *copied* into the parameter variables when a method starts. Computer scientists call this call mechanism "call by value". There are some limitations to the "call by value" mechanism. As you saw in Common Error 8.1, it is not possible to implement methods that modify the contents of number variables. Other programming languages such as C++ support an alternate mechanism, called "call by reference". For example, in C++ it would be an easy matter to write a method that modifies a number, by using a so-called *reference parameter*. Here is the C++ code, for those of you who know C++:

```
// This is C++
class BankAccount
{
public:
   void transfer(double amount, double&
otherBalance)
      // otherBalance is a double&, a reference to a double
   {
      balance = balance - amount;
      otherBalance = otherBalance + amount; // Works in
C++
   }
   . . .
```

```
};
```

You will sometimes read in Java books that "numbers are passed by value, objects are passed by reference". That is technically not quite correct. In Java, objects themselves are never passed as parameters; instead, both numbers and *object references* are copied by value. To see this clearly, let us consider another scenario. This method tries to set the `otherAccount` parameter to a new object:

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

In this situation, we are not trying to change the state of the object to which the parameter variable `otherAccount` refers; instead, we are trying to replace the object with a different one (see Modifying an Object Reference Parameter Has No Effect on the Caller). Now the parameter variable `other-Account` is replaced with a reference to a new account. But if you call the method with

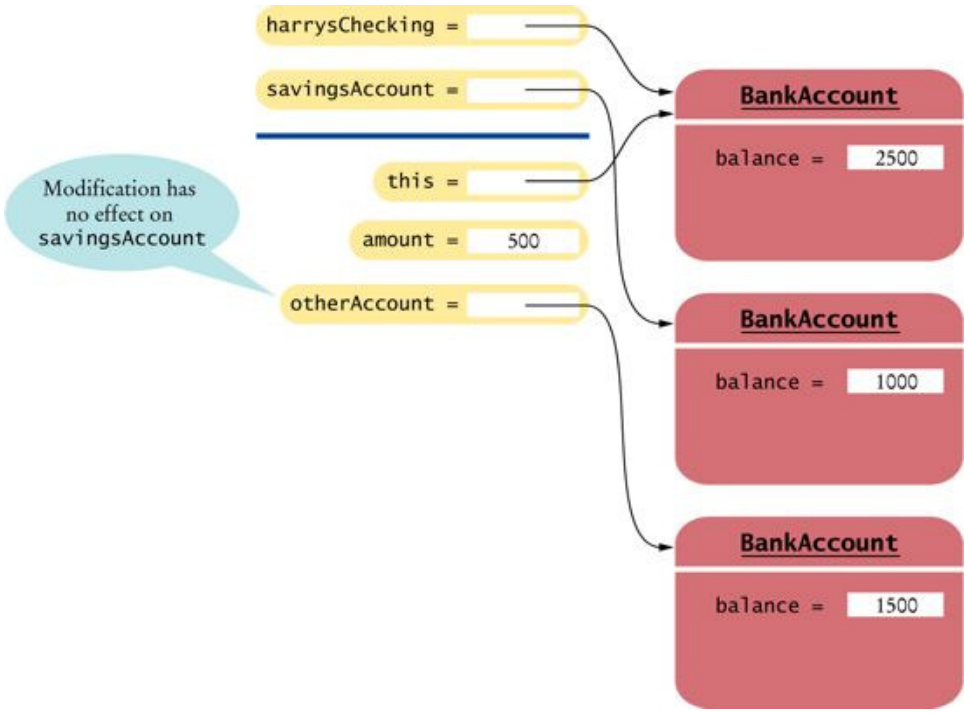
```
harrysChecking.transfer(500, savingsAccount);
```

then that change does not affect the `savingsAccount` variable that is supplied in the call.

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

As you can see, a Java method can update an object's state, but it cannot *replace* the contents of an object reference. This shows that object references are passed by value in Java.

346



Modifying an Object Reference Parameter Has No Effect on the Caller

8.5 Preconditions and Postconditions

A *precondition* is a requirement that the caller of a method must obey. For example, the `deposit` method of the `BankAccount` class has a precondition that the amount to be deposited should not be negative. It is the responsibility of the caller never to call a method if one of its preconditions is violated. If the method is called anyway, it is not responsible for producing a correct result.

A precondition is a requirement that the caller of a method must meet. If a method is called in violation of a precondition, the method is not responsible for computing the correct result.

Therefore, a precondition is an important part of the method, and you must document it. Here we document the precondition that the `amount` parameter must not be negative.

```
/**
   Deposits money into this account.
   @param amount the amount of money to deposit
   (Precondition: amount >= 0)
*/
```

Some `javadoc` extensions support a `@precondition` or `@requires` tag, but it is not a part of the standard `javadoc` program. Because the standard `javadoc` tool skips all unknown tags, we simply add the precondition to the method explanation or the appropriate `@param tag`.

Preconditions are typically provided for one of two reasons:

1. To restrict the parameters of a method
2. To require that a method is only called when it is in the appropriate *state*

For example, once a `Scanner` has run out of input, it is no longer legal to call the `next` method. Thus, a precondition for the `next` method is that the `hasNext` method returns `true`.

A method is responsible for operating correctly only when its caller has fulfilled all preconditions. The method is free to do *anything* if a precondition is not fulfilled. It would be perfectly legal if the method reformatted the hard disk every time it was called with a wrong input. Naturally, that isn't reasonable. What should a method actually do when it is called with inappropriate inputs? For example, what should `account.deposit(-1000)` do? There are two choices.

1. A method can check for the violation and *throw an exception*. Then the method does not return to its caller; instead, control is transferred to an exception handler. If no handler is present, then the program terminates. We will discuss exceptions in Chapter 11.

2. A method can skip the check and work under the assumption that the preconditions are fulfilled. If they aren't, then any data corruption (such as a negative balance) or other failures are the caller's fault.

The first approach can be inefficient, particularly if the same check is carried out many times by several methods. The second approach can be dangerous. The *assertion* mechanism was invented to give you the best of both approaches.

An *assertion* is a condition that you believe to be true at all times in a particular program location. An assertion check tests whether an assertion is true. Here is a typical assertion check that tests a precondition:

An assertion is a logical condition in a program that you believe to be true.

```
public double deposit (double amount)
{
   assert amount >= 0;
   balance = balance + amount;
}
```

In this method, the programmer expects that the quantity `amount` can never be negative. When the assertion is correct, no harm is done, and the program works in the normal way. If, for some reason, the assertion fails, *and assertion checking is enabled*, then the program terminates with an `AssertionError`.

However, if assertion checking is disabled, then the assertion is never checked, and the program runs at full speed. By default, assertion checking is disabled when you execute a program. To execute a program with assertion checking turned on, use this command:

```
java -enableassertions MyProg
```

348

349

SYNTAX 8.1: Assertion

`assert` *condition*;

Example:

```
assert amount >= 0;
```

Purpose:

To assert that a condition is fulfilled. If assertion checking is enabled and the condition is false, an assertion error is thrown.

You can also use the shortcut `-ea` instead of `-enableassertions`. You definitely want to turn assertion checking on during program development and testing.

You don't have to use assertions for checking preconditions—throwing an exception is another reasonable option. But assertions have one advantage: You can turn them off after you have tested your program, so that it runs at maximum speed. That way, you never have to feel bad about putting lots of assertions into your code. You can also use assertions for checking conditions other than preconditions.

Many beginning programmers think that it isn't "nice" to abort the program when a precondition is violated. Why not simply return to the caller instead?

```
public void deposit(double amount)
{
   if (amount < 0)
      return; // Not recommended
   balance = balance + amount;
}
```

That is legal—after all, a method can do anything if its preconditions are violated. But it is not as good as an assertion check. If the program calling the deposit method has a few bugs that cause it to pass a negative amount as an input value, then the version that generates an assertion failure will make the bugs very obvious during testing—it is hard to ignore when the program aborts. The quiet version, on the other hand, will not alert you, and you may not notice that it performs some wrong calculations as a consequence. Think of assertions as the "tough love" approach to precondition checking.

When a method is called in accordance with its preconditions, then the method promises to do its job correctly. A different kind of promise that the method makes is called a *postcondition*. There are two kinds of postconditions:

1. The return value is computed correctly.
2. The object is in a certain state after the method call is completed.

If a method has been called in accordance with its preconditions, then it must ensure that its postconditions are valid.

Here is a postcondition that makes a statement about the object state after the `deposit` method is called.

```
/**
   Deposits money into this account.
   (Postcondition: getBalance() >= 0)
   @param amount the amount of money to deposit
   (Precondition: amount >= 0)
*/
```

As long as the precondition is fulfilled, this method guarantees that the balance after the deposit is not negative.

Some `javadoc` extensions support a `@postcondition` or `@ensures` tag. However, just as with preconditions, we simply add postconditions to the method explanation or the `@return` tag, because the standard `javadoc` program skips all tags that it doesn't know.

Some programmers feel that they must specify a postcondition for every method. When you use `javadoc`, however, you already specify a part of the postcondition in the `@return` tag, and you shouldn't repeat it in a postcondition.

```
// This postcondition statement is overly repetitive.
/**
   Returns the current balance of this account.
   @return the account balance
   (Postcondition: The return value equals the account balance.)
*/
```

Note that we formulate pre- and postconditions only in terms of the *interface* of the class. Thus, we state the precondition of the `withdraw` method as `amount <= getBalance()`, not `amount<= balance`. After all, the caller, which needs to check the precondition, has access only to the public interface, not the private implementation.

Bertrand Meyer [1] compares preconditions and postconditions to contracts. In real life, contracts spell out the obligations of the contracting parties. For example, your mechanic may promise to fix the brakes of your car, and you promise in turn to pay a certain amount of money. If either party breaks the promise, then the other is not bound by the terms of the contract. In the same fashion, pre- and postconditions are contractual terms between a method and its caller. The method promises to fulfill the postcondition for all inputs that fulfill the precondition. The caller promises never to call the method with illegal inputs. If the caller fulfills its promise and gets a wrong answer, it can take the method to "programmer's court". If the caller doesn't fulfill its promise and something terrible happens as a consequence, it has no recourse.

Self Check

10. Why might you want to add a precondition to a method that you provide for other programmers?

11. When you implement a method with a precondition and you notice that the caller did not fulfill the precondition, do you have to notify the caller?

350

351

Advanced Topic 8.2: Class Invariants

Advanced Topic 6.5 introduced the concept of *loop invariants*. A loop invariant is established when the loop is first entered, and it is preserved by all loop iterations. We then know that the loop invariant must be true when the loop exits, and we can use that information to reason about the correctness of a loop.

Class invariants fulfill a similar purpose. A class invariant is a statement about an object that is true after every constructor and that is preserved by every mutator (provided that the caller respects all preconditions). We then know that the class invariant must always be true, and we can use that information to reason about the correctness of our program.

Here is a simple example. Consider a `BankAccount` class with the following preconditions for the constructor and the mutators:

```
public class BankAccount
{
```

```
   /**
      Constructs a bank account with a given balance.
         @param initial Balance the initial balance
         (Precondition: initial Balance >= 0)
   */
   public BankAccount(double initialBalance) {. . .}
   {
      balance = initialBalance;
   }
   /**
      Deposits money into the bank account.
         @param amount the amount to deposit
         (Precondition: amount >= 0)
   */
   public void deposit(double amount) {. . .}
   /**
      Withdraws money from the bank account.
         @param amount the amount to withdraw
         (Precondition: amount <= getBalance())
   */
   public void withdraw(double amount) {. . .}
   . . .
}
```

Now we can formulate the following invariant:

```
getBalance() >= 0
```

To see why this invariant is true, first check the constructor; because the precondition of the constructor is

```
initialBalance >= 0
```

we can prove that the invariant is true after the constructor has set `balance` to `initial Balance`.

351

352

Next, check the mutators. The precondition of the `deposit` method is

```
amount >= 0
```

We can assume that the invariant condition holds before calling the method. Thus, we know that `balance >= 0` before the method executes. The laws of

mathematics tell us that the sum of two nonnegative numbers is again nonnegative, so we can conclude that `balance >= 0` after the completion of the `deposit`. Thus, the `deposit` method preserves the invariant.

A similar argument shows that the `withdraw` method preserves the invariant.

Because the invariant is a property of the class, you document it with the class description:

```
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
   (Invariant: getBalance() >= 0)
*/
public class BankAccount
{
   . . .
}
```

8.6 Static Methods

Sometimes you need a method that is not invoked on an object. Such a method is called a *static method* or a *class method*. In contrast, the methods that you wrote up to now are often called *instance methods* because they operate on a particular instance of an object.

A static method is not invoked on an object.

A typical example of a static method is the `sqrt` method in the `Math` class. When you call `Math.sqrt(x)`, you don't supply any implicit parameter. (Recall that `Math` is the name of a class, not an object.)

Why would you want to write a method that does not operate on an object? The most common reason is that you want to encapsulate some computation that involves only numbers. Because numbers aren't objects, you can't invoke methods on them. For example, the call `x.sqrt()` can never be legal in Java.

Here is a typical example of a static method that carries out some simple algebra: to compute `p` percent of the amount `a`. Because the parameters are numbers, the method doesn't operate on any objects at all, so we make it into a static method:

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

352

353

You need to find a home for this method. Let us come up with a new class (similar to the `Math` class of the standard Java library). Because the `percentOf` method has to do with financial calculations, we'll design a class `Financial` to hold it. Here is the class:

```
public class Financial
{
   public static double percentOf(double p, double a)
   {
      return (p / 100) * a;
   }
   // More financial methods can be added here.
}
```

When calling a static method, you supply the name of the class containing the method so that the compiler can find it. For example,

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

Note that you do not supply an object of type `Financial` when you call the method.

Now we can tell you why the `main` method is static. When the program starts, there aren't any objects. Therefore, the *first* method in the program must be a static method.

You may well wonder why these methods are called static. The normal meaning of the word *static* (“staying fixed at one place”) does not seem to have anything to do

with what static methods do. Indeed, it's used by accident. Java uses the static keyword because C++ uses it in the same context. C++ uses `static` to denote class methods because the inventors of C++ did not want to invent another keyword. Someone noted that there was a relatively rarely used keyword, `static`, that denotes certain variables that stay in a fixed location for multiple method calls. (Java does not have this feature, nor does it need it.) It turned out that the keyword could be reused to denote class methods without confusing the compiler. The fact that it can confuse humans was apparently not a big concern. You'll just have to live with the fact that "static method" means "class method": a method that does not operate on an object and that has only explicit parameters.

SELF CHECK

12. Suppose Java had no static methods. Then all methods of the `Math` class would be instance methods. How would you compute the square root of *x*?

13. Harry turns in his homework assignment, a program that plays tic-tac-toe. His solution consists of a single class with many static methods. Why is this not an object-oriented solution?

353

354

8.7 Static Fields

Sometimes, you need to store values outside any particular object. You use *static fields* for this purpose. Here is a typical example. We will use a version of our `BankAccount` class in which each bank account object has both a balance and an account number:

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

We want to assign 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. Therefore, we must store the last assigned account number somewhere.

It makes no sense, though, to make this value into an instance field:

```
public class BankAccount
{
   . . .
   private double balance;
   private int accountNumber;
   private int lastAssignedNumber = 1000; // NO—won't
work
}
```

In that case each *instance* of the `BankAccount` class would have its own value of `lastAssignedNumber`.

Instead, we need to have a single value of `lastAssignedNumber` that is the same for the entire *class*. Such a field is called a static field, because you declare it using the `static` keyword.

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

Every `BankAccount` object has its own `balance` and `accountNumber` instance fields, but there is only a single copy of the `lastAssignedNumber` variable (see Figure 4). That field is stored in a separate location, outside any `BankAccount` objects.

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

A static field is sometimes called a *class field* because there is a single field for the entire class.

Every method of a class can access its static fields. Here is the constructor of the `BankAccount` class, which increments the last assigned number and then uses it to initialize the account number of the object to be constructed:

```
public class BankAccount
```

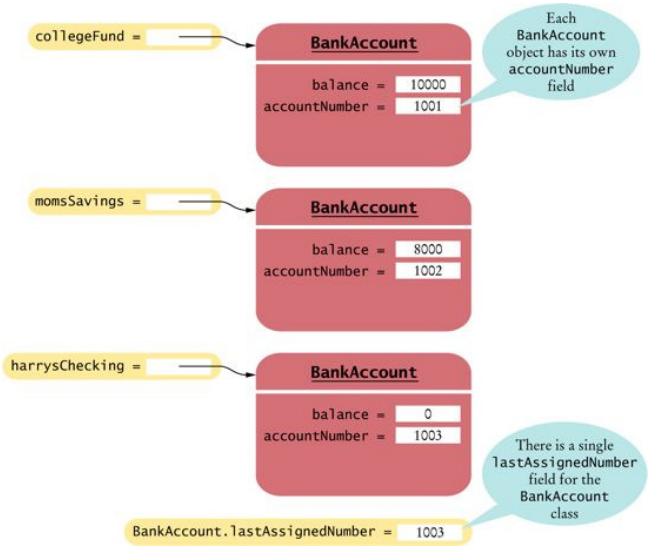
```
{                                                                    354
   public BankAccount()                                              355
   {
      // Generates next account number to be assigned
      lastAssignedNumber++; // Updates the static field
      // Assigns field to account number of this bank account
      accountNumber = lastAssignedNumber; // Sets the
instance field
   }
   . . .
}
```

How do you initialize a static field? You can't set it in the class constructor:

```
public BankAccount()
{
   lastAssignedNumber = 1000; // NO—would reset to 1000 for
each new object
   . . .
}
```

Then the initialization would occur each time a new instance is constructed.

Figure 4



A Static Field and Instance Fields

355

There are three ways to initialize a static field:

1. Do nothing. The static field is then initialized with `0` (for numbers), `false` (for `boolean` values), or `null` (for objects).
2. Use an explicit initializer, such as

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

   The initialization is executed once when the class is loaded.
3. Use a static initialization block (see Advanced Topic 8.3).

Like instance fields, static fields should always be declared as `private` to ensure that methods of other classes do not change their values. The exception to this rule are static *constants*, which may be either private or public. For example, the `BankAccount` class may want to define a public constant value, such as

```
public class BankAccount
{
   . . .
   public static final double OVERDRAFT_FEE = 5;
}
```

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

It makes sense to declare constants as `static`—you wouldn't want every object of the `BankAccount` class to have its own set of variables with these constant values. It is sufficient to have one set of them for the class.

Why are class variables called `static`? As with static methods, the `static` keyword itself is just a meaningless holdover from C++. But static fields and static methods have much in common: They apply to the entire *class*, not to specific instances of the class.

In general, you want to minimize the use of static methods and fields. If you find yourself using lots of static methods, then that's an indication that you may not have found the right classes to solve your problem in an object-oriented way.

SELF CHECK

14. Name two static fields of the `System` class.

15. Harry tells you that he has found a great way to avoid those pesky objects: Put all code into a single class and declare all methods and fields `static`. Then `main` can call the other static methods, and all of them can access the static fields. Will Harry's plan work? Is it a good idea?

356

357

ADVANCED TOPIC 8.3: Alternative Forms of Field Initialization

As you have seen, instance fields 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 for a field. Just as with local variables, you can specify initialization values for fields. 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 definition. All statements in that block are executed whenever an object is being constructed. Here is an example:

```
public class Coin
```

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

For static fields, 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.

357

358

8.8 Scope

8.8.1 Scope of Local Variables

When you have multiple variables or fields with the same name, there is the possibility of conflict. In order to understand the potential problems, you need to know about the *scope* of each variable: the part of the program in which the variable can be accessed.

The scope of a variable is the region of a program in which the variable can be accessed.

The scope of a local variable extends from the point of its declaration to the end of the block that encloses it.

It sometimes happens that the same variable name is used in two methods. Consider the variables `r` in the following example:

```
public class RectangleTester
{
   public static double area(Rectangle rect)
   {
      double r = rect.getWidth() * rect.getHeight();
      return r;
   }
   public static void main(String[] args)
   {
      Rectangle r = new Rectangle(5, 10, 20, 30);
      double a = area(r);
      System.out.println(r);
   }
}
```

These variables are independent from each other, or, in other words, their scopes are disjoint. You can have local variables with the same name `r` in different methods, just as you can have different motels with the same name "Bates Motel" in different cities.

The scope of a local variable cannot contain the definition of another variable with the same name.

In Java, the scope of a local variable can never contain the definition of local variable with the same name. For example, the following is an error:

```
Rectangle r = new Rectangle(5, 10, 20, 30);
if (x >= 0)
{
   double r = Math.sqrt(x);
   // Error—can't declare another variable called r here
```

```
      . . .
   }
```

However, you can have local variables with identical names if their scopes do not overlap, such as

```
if (x >= 0)
{
   double r = Math.sqrt(x);
   . . .                                                    358
}// Scope of r ends here                                    359
else
{
   Rectangle r = new Rectangle(5, 10, 20, 30);
   // OK—it is legal to declare another r here
   . . .
}
```

8.8.2 Scope of Class Members

In this section, we consider the scope of fields and methods of a class. (These are collectively called the *members* of the class.) Private members have *class scope*: You can access all members in any of the methods of the class.

> A qualified name is prefixed by its class name or by an object reference, such as `Math.sqrt` or `other.balance`.

If you want to use a public field or method outside its class, you must *qualify* the name. You qualify a static field or method by specifying the class name, such as `Math.sqrt` or `Math.PI`. You qualify an instance field or method by specifying the object to which the field or method should be applied, such as `harrysChecking.getBalance()`.

> An unqualified instance field or method name refers to the `this` parameter.

Inside a method, you don't need to qualify fields or methods that belong to the same class. Instance fields automatically refer to the implicit parameter of the method, that is, the object on which the method is invoked. For example, consider the `transfer` method:

```
public class BankAccount
{
   public void transfer(double amount, BankAccount
other)
   {
      balance = balance - amount; // i.e., this.balance
      other.balance = other.balance + amount;
   }
   . . .
}
```

Here, the unqualified name `balance` means `this.balance`. (Recall from Chapter 3 that `this` is a reference to the implicit parameter of any method.)

The same rule applies to methods. Thus, another implementation of the `transfer` method is

```
public class BankAccount
{
   public void transfer(double amount, BankAccount
other)
   {
      withdraw(amount); // i.e., this.withdraw(amount);
      other.deposit(amount);
   }
   . . .
}
```

Whenever you see an instance method call without an implicit parameter, then the method is called on the `this` parameter. Such a method call is called a "self-call". 359

360

Similarly, you can use a static field or method of the same class without a qualifier. For example, consider the following version of the `withdraw` method:

```
public class BankAccount
{
   public void withdraw(double amount)
   {
      if (balance < amount) balance = balance -
OVERDRAFT_FEE;
      else . . .
   }
   . . .
```

```
      private static double OVERDRAFT_FEE = 5;
   }
```

Here, the unqualified name `OVERDRAFT_FEE` refers to `BankAccount.OVERDRAFT_FEE`.

8.8.3 Overlapping Scope

Problems arise if you have two identical variable names with overlapping scope. This can never occur with local variables, but the scopes of identically named local variables and instance fields can overlap. Here is a purposefully bad example.

```
public class Coin
{
   . . .
   public double getExchangeValue(double
exchangeRate)
   {
      double value; // Local variable
      . . .
      return value;
   }
   private String name;
   private double value; // Field with the same name
}
```

Inside the `getExchangeValue` method, the variable name `value` could potentially have two meanings: the local variable or the instance shadow a field. The Java language specifies that in this situation the *local* variable wins out. It *shadows* the instance field. This sounds pretty arbitrary, but there is actually a good reason: You can still refer to the instance field as `this.value`.

```
value = this.value * exchangeRate;
```

It isn't necessary to write code like this. You can easily change the name of the local variable to something else, such as `result`.

A local variable can shadow a field with the same name. You can access the shadowed field name by qualifying it with the `this` reference.

However, you should be aware of one common use of the `this` reference. When implementing constructors, many programmers find it tiresome to come up with different names for instance fields and parameters. Using the `this` reference solves that problem. Here is a typical example.

```
public Coin(double value, String name)
{                                                                    360
   this. value = value;                                              361
   this.name = name;
}
```

The expression `this.value` refers to the instance field, but `value` is the parameter. Of course, you can always rename the construction parameters to `aValue` and `aName`, as we have done in this book.

Self Check

16. Consider the `deposit` method of the `BankAccount` class. What is the scope of the variables `amount` and `newBalance`?

17. What is the scope of the balance field of the `BankAccount` class?

Common Error 8.2: Shadowing

Accidentally using the same name for a local variable and an instance field is a surprisingly common error. As you saw in the preceding section, the local variable then *shadows* the instance field. Even though you may have meant to access the instance field, the local variable is quietly accessed. For some reason, this problem is most common in constructors. Look at this example of an incorrect constructor:

```
public class Coin
{
   public Coin(double aValue, String aName)
   {
      value = aValue;
      String name = aName; // Oops. . .
   }
   . . .
   private double value;
```

```
      private String name;
   }
```

The programmer declared a local variable `name` in the constructor. In all likelihood, that was just a typo—the programmer's fingers were on autopilot and typed the keyword `String`, even though the programmer all the time intended to access the instance field. Unfortunately, the compiler gives no warning in this situation and quietly sets the local variable to the value of `aName`. The instance field of the object that is being constructed is never touched, and remains `null`. Some programmers give all instance field names a special prefix to distinguish them from other variables. A common convention is to prefix all instance field names with the prefix `my`, such as `myValue` or `myName`.

361

362

PRODUCTIVITY HINT 8.1: Global Search and Replace

Suppose you chose an unfortunate name for a method—say `perc` instead of `percentOf`—and you regret your choice. Of course, you can locate all occurrences of `perc` in your code and replace them manually. However, most programming editors have a command to search for the `perc`'`s` automatically and replace them with `percentOf`.

You need to specify some details about the search:

- Do you want it to ignore case? That is, should `Perc` be a match? In Java you usually don't want that.
- Do you want it to match whole words only? If not, the `perc` in `superconductor` is also a match. In Java you usually want to match whole words.
- Is this a regular-expression search? No, but regular expressions can make searches even more powerful—see Productivity Hint 8.2.
- Do you want to confirm each replace, or simply go ahead and replace all matches? I usually confirm the first three or four, and when I see that it works as expected, I give the go-ahead to replace the rest. (By the way, a *global* replace means to replace all occurrences in the document.) Good text editors can undo a global replace that has gone awry. Find out whether yours will.

- Do you want the search to go from the point where the cursor is in the file through to the rest of the file, or should it search the currently selected text? Restricting replacement to a portion of the file can be very useful, but in this example you would want to move the cursor to the top of the file and then replace until the end of the file.

Not every editor has all these options. You should investigate what your editor offers.

PRODUCTIVITY HINT 8.2: Regular Expressions

Regular expressions describe character patterns. For example, numbers have a simple form. They contain one or more digits. The regular expression describing numbers is `[0-9]+`. The set `[0-9]` denotes any digit between `0` and `9`, and the `+` means "one or more".

What good is it? Several utility programs use regular expressions to locate matching text. Also, the search commands of some programming editors understand regular expressions. The most popular program that uses regular expressions is *grep* (which stands for "global regular expression print"). You can run grep from a command prompt or from inside some compilation environments. Grep is part of the UNIX operating system, but versions are available for Windows and MacOS. It needs a regular expression and one or more files to search. When grep runs, it displays a set of lines that match the regular expression.

Suppose you want to look for all magic numbers (see Quality Tip 4.1) in a file. The command

```
grep [0-9]+ Homework.java
```

362

363

lists all lines in the file `Homework.java` that contain sequences of digits. That isn't terribly useful; lines with variable names `x1` will be listed. OK, you want sequences of digits that do *not* immediately follow letters:

```
grep [^A-Za-z][0-9]+ Homework.java
```

The set `[^A-Za-z]` denotes any characters that are *not* in the ranges `A` to `Z` and `a` to `z`. This works much better, and it shows only lines that contain actual numbers.

For more information on regular expressions, consult one of the many tutorials on the Internet (such as [2]).

ADVANCED TOPIC 8.4: Static Imports

Starting with Java version 5.0, there is a variant of the `import` directive that lets you use static methods and fields 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.

8.9 Packages

8.9.1 Organizing Related Classes into Packages

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 dozens of packages, some of which are listed in Table 1.

A package is a set of related classes.

363

364

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`
`omg.org.CORBA`	Common Object Request Broker Architecture for distributed objects	`IntHolder`

To put classes in a package, you must place a line

`package` *packageName*;

as the first instruction in the source file containing the classes. A package name consists of one or more identifiers separated by periods. (See Section 8.9.3 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
{
   . . .
}
```

SYNTAX 8.2: Package Specification

`package` *packageName;*

Example:

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

Purpose:

To declare that all classes in this file belong to a particular package

364

365

In addition to the named packages (such as `java.util` or `com.horstmann.bigjava`), there is a special package, called the *default package*, which has no name. If you did not include any `package` statement at the top of your source file, its classes are placed in the default package.

8.9.2 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);
```

Naturally, that is somewhat inconvenient. You can instead *import* a name with an `import` statement:

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

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

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

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

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

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

However, you never need to import the classes in the `java.lang` package explicitly. That is the package containing the most basic Java classes, such as `Math`

and `Object`. These classes are always available to you. In effect, an automatic `import java.lang.*` statement has been placed into every source file.

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

8.9.3 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 Bertha 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.

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`.

> Use a domain name in reverse to construct unambiguous package names.

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 Bertha 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.

8.9.4 How Classes are Located

If the Java compiler is properly set up on your system, and you use only the standard classes, you ordinarily need not worry about the location of class files and can safely skip this section. If you want to add your own packages, however, or if the compiler cannot locate a particular class or package, you need to understand the mechanism.

A package is located in a subdirectory that matches the package name. The parts of the name between periods represent successively nested directories. For example, the package `com.horstmann.bigjava` would be placed in a subdirectory `com/horstmann/bigjava`. If the package is to be used only in conjunction with a single program, then you can place the subdirectory inside the directory holding that program's files. For example, if you do your homework assignments in a *base directory* `/home/walters`, then you can place the class files for the `com.horstmann.bigjava` package into the directory `/home/walters/com/horstmann/bigjava`, as shown in Figure 5. (Here, we are using UNIX-style file names. Under Windows, you might use `c:\home\walters\com\horstmann\bigjava`.)

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

However, if you want to place your programs into many different directories, such as `/home/walters/hw1, /home/walters/hw2, . . .`, then you probably don't want to have lots of identical subdirectories `/home/walters/hw1/com/horstmann/bigjava`, `/home/walters/hw2/com/horstmann/bigjava`, and so on. In that case, you want to make a single directory with a name such as `/home/walters/lib/com/horstmann/bigjava`, place all class files for

the package in that directory, and tell the Java compiler once and for all how to locate the class files.

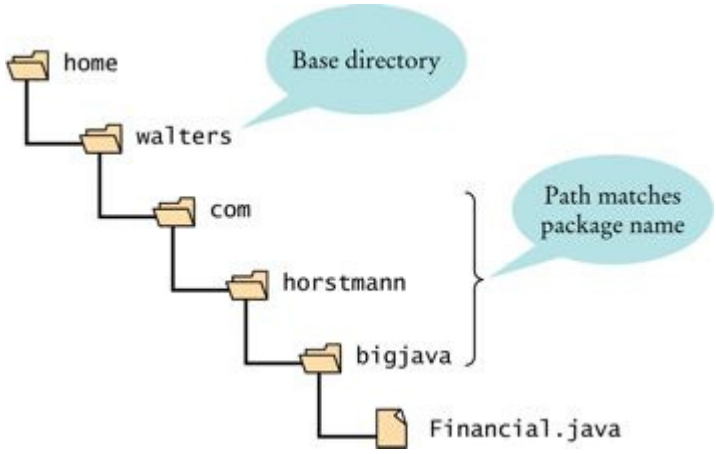
You need to add the directories that might contain packages to the *class path*. In the preceding example, you add the `/home/walters/lib` directory to that class path. The details for doing this depend on your compilation environment; consult the documentation for your compiler, or your instructor. If you use the Sun Java SDK, you need to set the class path. The exact command depends on the operating system. In UNIX, the command might be

366
367

```
export CLASSPATH=/home/walters/lib:.
```

This setting places both the `/home/walters/lib` directory and the current directory onto the class path. (The period denotes the current directory.)

Figure 5



Base Directories and Subdirectories for Packages

A typical example for Windows would be

```
set CLASSPATH=c:\home\walters\lib;.
```

Note that the class path contains the *base directories* that may contain package directories. It is a common error to place the complete package address in the class path. If the class path mistakenly contains `/home/walters/lib/com/horstmann/bigjava`, then the compiler will

attempt to locate the `com.horstmann.bigjava` package in `/home/walters/lib/com/horstmann/bigjava/com/horstmann/bigjava` and won't find the files.

Self Check

18. Which of the following are packages?

a. `java`

b. `java.lang`

c. `java.util`

d. `java.lang.Math`

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

20. Suppose your homework assignments are located in the directory `/home/me/cs101` (`c:\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`?

367

368

Common Error 8.3: 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 [3] 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.

How To 8.1: Programming with Packages

This How To explains in detail how to place your programs into packages. For example, your instructor may ask you to 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.Bank` and `homework2.Bank`).

Step 1 Come up with a package name.

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

Step 2 Pick a *base directory*.

The base directory is the directory that contains the directories for your various packages, for example, `/home/walters` or `c:\cs1`

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 /home/walters/homework1
```

If you have multiple segments, build them up one by one:

```
mkdir c:\cs1\edu
mkdir c:\cs1\edu\sjsu
mkdir c:\cs1\edu\sjsu\cs
mkdir c:\cs1\edu\sjsu\cs\walters
mkdir c:\cs1\edu\sjsu\cs\walters\homework1
```

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/walters/homework1/Tester.java
/home/walters/homework1/Bank.java
```

or

```
c:\cs1\edu\sjsu\cs\walters\homework1\Tester.java
c:\cs1\edu\sjsu\cs\walters\homework1\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;
```

or

```
package edu.sjsu.cs.walters.homework1;
```

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/walters
javac homework1/Tester.java
```

or

```
cd \cs1
java edu\sjsu\cs\walters\homework1\Tester.java
```

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

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

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

```
cd /home/walters
java homework1.Tester
```

or

```
cd \cs1
java edu.sjsu.cs.walters.homework1.Tester
```

369

370

RANDOM FACT 8.1: The Explosive Growth of Personal Computers

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.

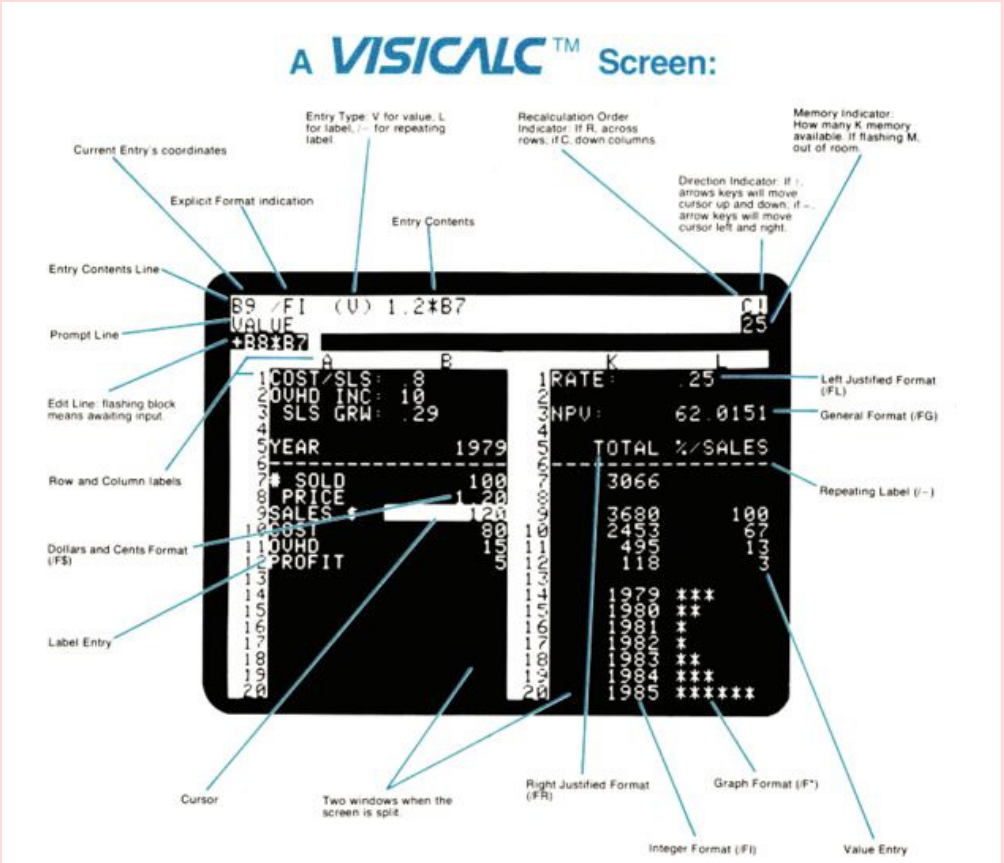
The first big hit was the Apple II. It was a real computer with a keyboard, a monitor, and a floppy disk drive. When it was first released, users had a $3000 machine that could play Space Invaders, run a primitive bookkeeping program, or let users program it in BASIC. The original Apple II did not even 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 VisiCalc Spreadsheet Running on an Apple II). 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. Middle managers in companies, who understood computers and were fed up with having to wait for hours or days to get their data runs back from the computing center, snapped up VisiCalc and the computer that was needed to run it. For them, the computer was a spreadsheet machine.

The next big hit was the IBM Personal Computer, ever after known as the PC. It was the first widely available personal computer that used Intel's 16-bit processor, the 8086, whose successors are still being used in personal computers today. The success of the PC was based not on any engineering breakthroughs but on the fact that it was easy to *clone*. IBM published specifications for plug-in cards, and it went one step further. It published the exact source code of the so-called BIOS (Basic Input/Output System), which controls the keyboard, monitor, ports, and disk drives and must be installed in ROM form in every PC. This allowed third-party vendors of plug-in cards to ensure that the BIOS code, and third-party extensions of it, interacted correctly with the equipment. Of course, the code itself was the property of IBM and could not be copied legally. Perhaps IBM did not foresee that functionally equivalent versions of the BIOS nevertheless could be recreated by others. Compaq, one of the first clone vendors, had fifteen engineers, who certified that they had never seen the original IBM code, write a new version that conformed precisely to the IBM specifications. Other companies did the same, and soon a variety of vendors were selling computers that ran the same software as IBM's PC but distinguished

themselves by a lower price, increased portability, or better performance. In time, IBM lost its dominant position in the PC market. It is now one of many companies producing IBM PC-compatible computers.

IBM never produced an *operating system* for its PCs—that is, the software that organizes the interaction between the user and the computer, starts application programs, and manages disk storage and other resources. Instead, IBM offered customers the option of three separate operating systems. Most customers couldn't care less about the operating system. They chose the system that was able to launch most of the few applications that existed at the time. It happened to be DOS (Disk Operating System) by Microsoft. Microsoft cheerfully licensed the same operating system to other hardware vendors and encouraged software companies to write DOS applications. A huge number of useful application programs for PC-compatible machines was the result.

370
371



The VisiCalc Spreadsheet Running on an Apple II

PC applications were certainly useful, but they were not easy to learn. Every vendor developed a different *user interface*: the collection of keystrokes, menu options, and settings that a user needed to master to use a software package effectively. Data exchange between applications was difficult, because each program used a different data format. The Apple Macintosh changed all that in 1984. The designers of the Macintosh had the vision to supply an intuitive user interface with the computer and to force software developers to adhere to it. It took Microsoft and PC-compatible manufacturers years to catch up.

Accidental Empires [4] is highly recommended for an amusing and irreverent account of the emergence of personal computers.

At the time of this writing, it is estimated that two in three U.S. households own a personal computer. Most personal computers are used for accessing information from online sources, entertainment, word processing, and home finance (banking, budgeting, taxes). Some analysts predict that the personal computer will merge with the television set and cable network into an entertainment and information appliance.

371

372

8.10 Unit Test Frameworks

Up to now, we have used a very simple approach to testing. We provided tester classes whose `main` method computes values and prints actual and expected values. However, that approach has two limitations. It takes some time to inspect the output and decide whether a test has passed. More importantly, the `main` method gets messy if it contains many tests.

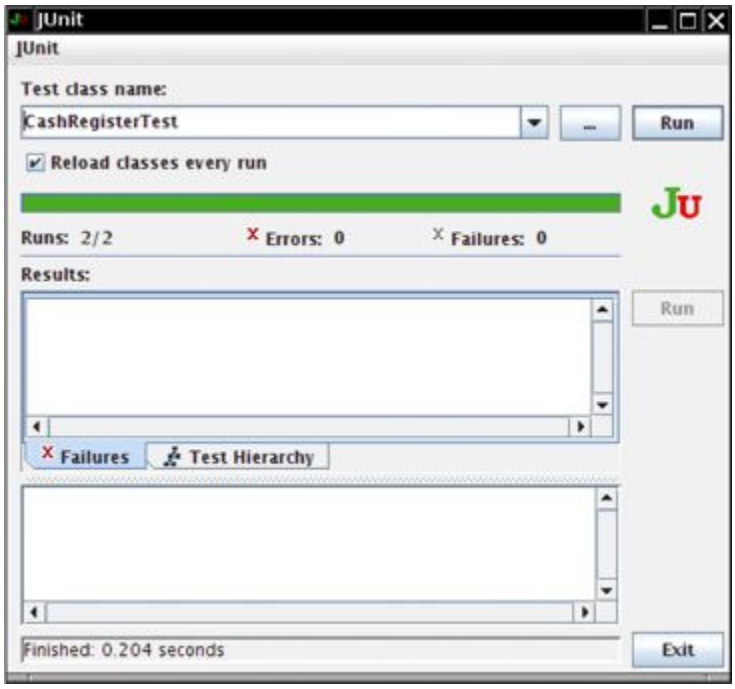
Unit testing frameworks were designed to quickly execute and evaluate test suites, and to make it easy to incrementally add test cases. One of the most popular testing frameworks is JUnit. It is freely available at http://junit.org, and it is also built into a number of development environments, including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

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

When you use JUnit, you design a companion test class for each class that you develop. Two versions of JUnit are currently in common use, 3 and 4. We describe both versions. In JUnit 3, your test class has two essential properties:

- The test class must extend the class `TestCase` from the `junit.framework` package.
- For each test case, you must define a method whose name starts with `test`, such as `testSimpleCase`.

Figure 6



Unit Testing with JUnit

372

373

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 parameters 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`. Consider this example:

```
import junit.framework.TestCase;

public class CashRegisterTest extends TestCase
{
   public void testSimpleCase()
   {
      CashRegister register = new CashRegister();
      register.recordPurchase(0.75);
      register.recordPurchase(1.50);
      register.enterPayment(2, 0, 5, 0, 0);
      double expected = 0.25;
      assertEquals(expected, register.giveChange(),
EPSILON);
   }
   public void testZeroBalance()
   {
      CashRegister register = new CashRegister();
      register.recordPurchase(2.25);
      register.recordPurchase(19.25);
      register.enterPayment(21, 2, 0, 0, 0);
      assertEquals(0, register.giveChange(),
EPSILON);
   }
   // More test cases
   . . .
   private static final double EPSILON = 1E-12;
}
```

If all test cases pass, the JUnit tool shows a green bar (see Figure 6). 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 start with test). These methods typically carry out steps that you want to share among test methods.

JUnit 4 is even simpler. Your test class need not extend any class and you can freely choose names for your test methods. 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.

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

public class CashRegisterTest
{
   @Test public void simpleCase()
   {                                                                    373
      register.recordPurchase(0.75);                                    374
      register.recordPurchase(1.50);
      register.enterPayment(2, 0, 5, 0, 0);
      double expected = 0.25;
      Assert.assert Equals (expected,
register.giveChange(), EPSILON);
   }
   // More test cases
   . . .
}
```

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.

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

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

Self Check

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

22. What is the significance of the `EPSILON` parameter in the `assertEquals` method?

CHAPTER SUMMARY

1. A class should represent a single concept from the problem domain, such as business, science, or mathematics.
2. The public interface of a class is cohesive if all of its features are related to the concept that the class represents.
3. A class depends on another class if it uses objects of that class.
4. It is a good practice to minimize the coupling (i.e., dependency) between classes.
5. An immutable class has no mutator methods.
6. A side effect of a method is any externally observable data modification.
7. You should minimize side effects that go beyond modification of the implicit parameter.
8. In Java, a method can never change parameters of primitive type. *374*
9. In Java, a method can change the state of an object reference parameter, but it cannot replace the object reference with another. *375*
10. A precondition is a requirement that the caller of a method must meet. If a method is called in violation of a precondition, the method is not responsible for computing the correct result.
11. An assertion is a logical condition in a program that you believe to be true.
12. If a method has been called in accordance with its preconditions, then it must ensure that its postconditions are valid.
13. A static method is not invoked on an object.
14. A static field belongs to the class, not to any object of the class.
15. The scope of a variable is the region of a program in which the variable can be accessed.

16. The scope of a local variable cannot contain the definition of another variable with the same name.
17. A qualified name is prefixed by its class name or by an object reference, such as `Math.sqrt` or `other.balance`.
18. An unqualified instance field or method name refers to the `this` parameter.
19. A local variable can shadow a field with the same name. You can access the shadowed field name by qualifying it with the `this` reference.
20. A package is a set of related classes.
21. The `import` directive lets you refer to a class of a package by its class name, without the package prefix.
22. Use a domain name in reverse to construct unambiguous package names.
23. The path of a class file must match its package name.
24. Unit test frameworks simplify the task of writing classes that contain many test cases.
25. The JUnit philosophy is to run all tests whenever you change your code.

FURTHER READING

1. Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1989, Chapter 7.
2. http://www.zvon.org/other/PerlTutorial/Output A dynamic tutorial for regular expressions.
3. http://java.sun.com/docs/books/jls The Java language specification.
4. Robert X Cringely, *Accidental Empires*, Addison-Wesley, 1992.

REVIEW EXERCISES

★★ **Exercise R8.1.** Consider the following problem description:

> Users place coins in a vending machine and select a product by pushing a button. If the inserted coins are sufficient to cover the purchase price of the product, the product is dispensed and change is given. Otherwise, the inserted coins are returned to the user.

What classes should you use to implement it?

★★ **Exercise R8.2.** Consider the following problem description:

> Employees receive their biweekly paychecks. They are paid their hourly rates for each hour worked; however, if they worked more than 40 hours per week, they are paid overtime at 150% of their regular wage.

What classes should you use to implement it?

★★ **Exercise R8.3.** Consider the following problem description:

> Customers order products from a store. Invoices are generated to list the items and quantities ordered, payments received, and amounts still due. Products are shipped to the shipping address of the customer, and invoices are sent to the billing address.

What classes should you use to implement it?

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

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

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

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

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

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

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

376

377

★ **Exercise R8.11.** Which of the following classes are immutable?

a. `Rectangle`

b. `String`

c. `Random`

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

a. `PrintStream`

b. `Date`

c. `Integer`

★★ **Exercise R8.13.** 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;
      }
      . . .
   }
```

★★★ **Exercise R8.14.** 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?

★★ **Exercise R8.15.** Write preconditions for the following methods. Do not implement the methods.

a. `public static double sqrt(double x)`

b. `public static String romanNumeral (int n)`

c. `public static double slope(Line2D.Double a)`

d. `public static String weekday (int day)`

★★ **Exercise R8.16.** What preconditions do the following methods from the standard Java library have?

a. `Math.sqrt`

b. `Math.tan`

c. `Math.log`

d. `Math.exp`

e. `Math.pow`

f. `Math.abs`

★★ **Exercise R8.17.** What preconditions do the following methods from the standard Java library have?

a. `Integer.parseInt(String s)`

b. `StringTokenizer. nextToken()`

c. `Random. nextInt(int n)`

d. `String.substring(int m, int n)`

★★★ **Exercise R8.18.** When a method is called with parameters that violate its precondition(s), it can terminate (by throwing an exception or an assertion error), or it can return to its caller. Give two examples of library methods (standard or the library methods used in this book) that return some result to their callers when called with invalid parameters, and give two examples of library methods that terminate.

★★ **Exercise R8.19.** Consider a `CashRegister` class with methods

- `public void enterPayment(int coinCount, Coin coinType)`
- `public double getTotalPayment()`

Give a reasonable postcondition of the `enterPayment` method. What preconditions would you need so that the `CashRegister` class can ensure that postcondition?

★★ **Exercise R8.20.** Consider the following method that is intended to swap the values of two floating-point numbers:

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

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

★★★ **Exercise R8.21.** How can you write a method that swaps two floating-point numbers? *Hint*: `Point2D.Double`.

★★ **Exercise R8.22.** 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;
}
```

378

379

★ **Exercise R8.23.** Consider an enhancement of the `Die` class of Chapter 6 with a static field

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

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` fields.

★ **Exercise R8.24.** 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);
   }
}
```

★ **Exercise R8.25.** Look at the methods in the `Integer` class. Which are static? Why?

★★ **Exercise R8.26.** Look at the methods in the `String` class (but ignore the ones that take a parameter of type `char[]`). Which are static? Why?

★★ **Exercise R8.27.** The `in` and `out` fields of the `System` class are public static fields of the `System` class. Is that good design? If not, how could you improve on it?

★★ **Exercise R8.28.** In the following class, the variable `n` occurs in multiple scopes. Which declarations of `n` are legal and which are illegal?

```
public class X
{
   public int f()
   {
      int n = 1;
      return n;
   }
   public int g(int k)
   {
      int a;
      for (int n = 1; n <= k; n++)
         a = a + n;
      return a;
   }
   public int h(int n)
   {
      int b;
      for (int n = 1; n <= 10; n++)
         b = b + n;
      return b + n;
   }
```

379

380

```
   public int k(int n)
   {
      if (n < 0)
      {
         int k = -n;
         int n = (int) (Math.sqrt(k));
         return n;
      }
      else return n;
   }
   public int m(int k)
   {
      int a;
      for (int n = 1; n <= k; n++)
         a = a + n;
      for (int n = k; n >= 1; n++)
         a = a + n;
      return a;
   }
   private int n;
}
```

★ **Exercise R8.29.** What is a qualified name? What is an unqualified name?

★★ **Exercise R8.30.** When you access an unqualified name in a method, what does that access mean? Discuss both instance and static features.

★★ **Exercise R8.31.** Every Java program can be rewritten to avoid `import` statements. Explain how, and rewrite `RectangleComponent.java` from Chapter 2 to avoid `import` statements.

★ **Exercise R8.32.** What is the default package? Have you used it before this chapter in your programming?

★★T **Exercise R8.33.** What does JUnit do when a test method throws an exception? Try it out and report your findings.

Additional review exercises are available in WileyPLUS.

PROGRAMMING EXERCISES

★★ **Exercise P8.1.** 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 enterPayment(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.

★★ **Exercise P8.2.** 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.

★ **Exercise P8.3.** 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.

★ **Exercise P8.4.** Enhance the `BankAccount` class by adding preconditions for the constructor and the deposit method that require the `amount` parameter to be at least zero, and a precondition for the `withdraw` method that requires `amount` to be a value between `0` and the current balance. Use assertions to test the preconditions.

★★ **Exercise P8.5.** Write static methods

- `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 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.

★★ **Exercise P8.6.** Solve Exercise P8.5 by implementing classes `Sphere`, `Cylinder`, and `Cone`. Which approach is more object-oriented?

★★ **Exercise P8.7.** 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?

381

382

★★ **Exercise P8.8.** 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?

★★ **Exercise P8.9.** 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`.

★ **Exercise P8.10.** 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`.

★★ **Exercise P8.11.** 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 $x^n = x^{n-1} \cdot x$. Implement a static method `double intPower(double x, int n)` that uses this algorithm. Add it to a class called `Numeric`.

★★ **Exercise P8.12.** Improve the `Needle` class of Chapter 6. Turn the `generator` field into a static field so that all needles share a single random number generator.

★★ **Exercise P8.13.** Implement a `Coin` and `CashRegister` class as described in Exercise P8.1. Place the classes into a package called `money`. Keep the `CashRegister`Tester class in the default package.

★ **Exercise P8.14.** Place a `BankAccount` class in a package whose name is derived from your e-mail address, as described in Section 8.9. Keep the `BankAccountTester` class in the default package.

★★T **Exercise P8.15.** Provide a JUnit test class `BankTest` with three test methods, each of which tests a different method of the `Bank` class in Chapter 7.

★★T **Exercise P8.16.** Provide JUnit test class `TaxReturnTest` with three test methods that test different tax situations for the `Tax` class in Chapter 5.

★G **Exercise P8.17.** Write methods

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

382

383

- `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 on 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`.

★★G **Exercise P8.18.** Repeat Exercise P8.15 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?

Additional programming exercises are available in WileyPLUS.

PROGRAMMING PROJECTS

★★★ **Project 8.1.** 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.

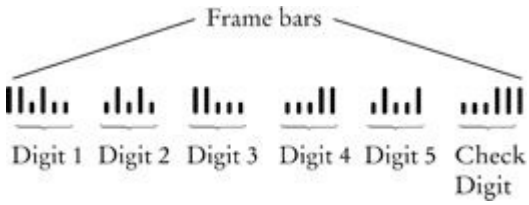
★★★ **Project 8.2.** For faster sorting of letters, the United States Postal Service encourages companies that send large volumes of mail to use a bar code denoting the ZIP code (see Figure 7).

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

Figure 7

A Postal Bar Code

Figure 8



Encoding for Five-Digit Bar Codes

383

384

Each digit of the ZIP code, and the check digit, is encoded according to the following table:

	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

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

$$0.7 + 1.4 + 1.2 + 0.1 + 0.0 = 6$$

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

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

| | : | : : : | : | : | | : : : : : : : | | : | : : | : : : | | |

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

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

ANSWERS TO SELF-CHECK QUESTIONS

1. Look for nouns in the problem description.
2. Yes (`ChessBoard`) and no (`MovePiece`).
3. Some of its features deal with payments, others with coin values.

384

4. None of the coin operations require the `CashRegister` class.

385

5. If a class doesn't depend on another, it is not affected by interface changes in the other class.
6. It is an accessor—calling `substring` doesn't modify the string on which the method is invoked. In fact, all methods of the `String` class are accessors.
7. No—`translate` is a mutator.
8. It is a side effect; this kind of side effect is common in object-oriented programming.
9. Yes—the method affects the state of the `Scanner` parameter.
10. Then you don't have to worry about checking for invalid values—it becomes the caller's responsibility.
11. No—you can take any action that is convenient for you.
12. `Math m = new Math(); y = m.sqrt(x);`
13. In an object-oriented solution, the `main` method would construct objects of classes `Game`, `Player`, and the like. Most methods would be instance methods that depend on the state of these objects.

14. `System.in` and `System.out`.

15. Yes, it works. Static methods can access static fields of the same class. But it is a terrible idea. As your programming tasks get more complex, you will want to use objects and classes to organize your programs.

16. The scope of `amount` is the entire `deposit` method. The scope of `newBalance` starts at the point at which the variable is defined and extends to the end of the method.

17. It starts at the beginning of the class and ends at the end of the class.

18. (a) No; (b) Yes; (c) Yes; (d) No

19. No—you simply use fully qualified names for all other classes, such as `java.util.Random and java.awt.Rectangle`.

20. `/home/me/cs101/hw1/problem1` or, on Windows, `c:\me\cs101\hw1\problem1`.

21. Here is one possible answer, using the JUnit 4 style.

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

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