

```
4   8   12   16   20   24
```

By now you can probably guess how we are going to print a multiplication table: we'll invoke `printMultiples` repeatedly with different arguments. In fact, we are going to use another loop to iterate through the rows.

```
    int i = 1;
    while (i <= 6) {
        printMultiples (i);
        i = i + 1;
    }
```

First of all, notice how similar this loop is to the one inside `printMultiples`. All I did was replace the print statement with a method invocation.

The output of this program is

```
1   2   3   4   5   6
2   4   6   8  10  12
3   6   9  12  15  18
4   8  12  16  20  24
5  10  15  20  25  30
6  12  18  24  30  36
```

which is a (slightly sloppy) multiplication table. If the sloppiness bothers you, Java provides methods that give you more control over the format of the output, but I'm not going to get into that here.

6.7 Methods

In the last section I mentioned “all the things methods are good for.” About this time, you might be wondering what exactly those things are. Here are some of the reasons methods are useful:

- By giving a name to a sequence of statements, you make your program easier to read and debug.
- Dividing a long program into methods allows you to separate parts of the program, debug them in isolation, and then compose them into a whole.
- Methods facilitate both recursion and iteration.
- Well-designed methods are often useful for many programs. Once you write and debug one, you can reuse it.

6.8 More encapsulation

To demonstrate encapsulation again, I'll take the code from the previous section and wrap it up in a method:

```
public static void printMultTable () {  
    int i = 1;  
    while (i <= 6) {  
        printMultiples (i);  
        i = i + 1;  
    }  
}
```

The process I am demonstrating is a common development plan. You develop code gradually by adding lines to `main` or someplace else, and then when you get it working, you extract it and wrap it up in a method.

The reason this is useful is that you sometimes don't know when you start writing exactly how to divide the program into methods. This approach lets you design as you go along.

6.9 Local variables

About this time, you might be wondering how we can use the same variable `i` in both `printMultiples` and `printMultTable`. Didn't I say that you can only declare a variable once? And doesn't it cause problems when one of the methods changes the value of the variable?

The answer to both questions is "no," because the `i` in `printMultiples` and the `i` in `printMultTable` are *not the same variable*. They have the same name, but they do not refer to the same storage location, and changing the value of one of them has no effect on the other.

Variables that are declared inside a method definition are called **local variables** because they are local to their own methods. You cannot access a local variable from outside its "home" method, and you are free to have multiple variables with the same name, as long as they are not in the same method.

It is often a good idea to use different variable names in different methods, to avoid confusion, but there are good reasons to reuse names. For example, it is common to use the names `i`, `j` and `k` as loop variables. If you avoid using them in one method just because you used them somewhere else, you will probably make the program harder to read.

6.10 More generalization

As another example of generalization, imagine you wanted a program that would print a multiplication table of any size, not just the 6x6 table. You could add a parameter to `printMultTable`:

```
public static void printMultTable (int high) {  
    int i = 1;  
    while (i <= high) {
```

```

        printMultiples (i);
        i = i + 1;
    }
}

```

I replaced the value 6 with the parameter `high`. If I invoke `printMultTable` with the argument 7, I get

```

1  2  3  4  5  6
2  4  6  8 10 12
3  6  9 12 15 18
4  8 12 16 20 24
5 10 15 20 25 30
6 12 18 24 30 36
7 14 21 28 35 42

```

which is fine, except that I probably want the table to be square (same number of rows and columns), which means I have to add another parameter to `printMultiples`, to specify how many columns the table should have.

Just to be annoying, I will also call this parameter `high`, demonstrating that different methods can have parameters with the same name (just like local variables):

```

public static void printMultiples (int n, int high) {
    int i = 1;
    while (i <= high) {
        System.out.print (n*i + "    ");
        i = i + 1;
    }
    newLine ();
}

public static void printMultTable (int high) {
    int i = 1;
    while (i <= high) {
        printMultiples (i, high);
        i = i + 1;
    }
}

```

Notice that when I added a new parameter, I had to change the first line of the method (the interface or prototype), and I also had to change the place where the method is invoked in `printMultTable`. As expected, this program generates a square 7x7 table:

```

1  2  3  4  5  6  7
2  4  6  8 10 12 14
3  6  9 12 15 18 21
4  8 12 16 20 24 28
5 10 15 20 25 30 35
6 12 18 24 30 36 42
7 14 21 28 35 42 49

```

When you generalize a method appropriately, you often find that the resulting program has capabilities you did not intend. For example, you might notice that the multiplication table is symmetric, because $ab = ba$, so all the entries in the table appear twice. You could save ink by printing only half the table. To do that, you only have to change one line of `printMultTable`. Change

```
    printMultiples (i, high);
```

to

```
    printMultiples (i, i);
```

and you get

```
1
2  4
3  6  9
4  8  12 16
5  10 15 20 25
6  12 18 24 30 36
7  14 21 28 35 42 49
```

I'll leave it up to you to figure out how it works.

6.11 Glossary

loop: A statement that executes repeatedly while or until some condition is satisfied.

infinite loop: A loop whose condition is always true.

body: The statements inside the loop.

iteration: One pass through (execution of) the body of the loop, including the evaluation of the condition.

encapsulate: To divide a large complex program into components (like methods) and isolate the components from each other (for example, by using local variables).

local variable: A variable that is declared inside a method and that exists only within that method. Local variables cannot be accessed from outside their home method, and do not interfere with any other methods.

generalize: To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

development plan: A process for developing a program. In this chapter, I demonstrated a style of development based on developing code to do simple, specific things, and then encapsulating and generalizing. In Section 5.2 I demonstrated a technique I called incremental development. In later chapters I will suggest other styles of development.

6.12 Exercises

Exercise 6.1

```
public static void main (String[] args) {  
    loop (10);  
}
```

```
public static void loop (int n) {  
    int i = n;  
    while (i > 1) {  
        System.out.println (i);  
        if (i%2 == 0) {  
            i = i/2;  
        } else {  
            i = i+1;  
        }  
    }  
}
```

- Draw a table that shows the value of the variables `i` and `n` during the execution of `loop`. The table should contain one column for each variable and one line for each iteration.
- What is the output of this program?

Exercise 6.2

- Encapsulate the following code fragment, transforming it into a method that takes a `String` as an argument and that returns (and doesn't print) the final value of `count`.
- In a sentence, describe abstractly what the resulting method does.
- Assuming that you have already generalized this method so that it works on any `String`, what else could you do to generalize it more?

```
String s = "((3 + 7) * 2)";  
int len = s.length ();  
  
int i = 0;  
int count = 0;  
  
while (i < len) {  
    char c = s.charAt(i);  
  
    if (c == '(') {  
        count = count + 1;  
    } else if (c == ')') {  
        count = count - 1;  
    }  
    i = i + 1;  
}  
  
System.out.println (count);
```

Exercise 6.3 Let's say you are given a number, a , and you want to find its square root. One way to do that is to start with a very rough guess about the answer, x_0 , and then improve the guess using the following formula:

$$x_1 = (x_0 + a/x_0)/2 \quad (6.2)$$

For example, if we want to find the square root of 9, and we start with $x_0 = 6$, then $x_1 = (6 + 9/6)/2 = 15/4 = 3.75$, which is closer.

We can repeat the procedure, using x_1 to calculate x_2 , and so on. In this case, $x_2 = 3.075$ and $x_3 = 3.00091$. So that is converging very quickly on the right answer (which is 3).

Write a method called `squareRoot` that takes a `double` as a parameter and that returns an approximation of the square root of the parameter, using this algorithm. You may not use the built-in method `Math.sqrt`.

As your initial guess, you should use $a/2$. Your method should iterate until it gets two consecutive estimates that differ by less than 0.0001; in other words, until the absolute value of $x_n - x_{n-1}$ is less than 0.0001. You can use the built-in method `Math.abs` to calculate the absolute value.

Exercise 6.4 In Exercise 5.9 we wrote a recursive version of `power`, which takes a double `x` and an integer `n` and returns x^n . Now write an iterative method to perform the same calculation.

Exercise 6.5 Section 5.10 presents a recursive method that computes the factorial function. Write an iterative version of `factorial`.

Exercise 6.6 One way to calculate e^x is to use the infinite series expansion

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + \dots \quad (6.3)$$

If the loop variable is named `i`, then the i th term is equal to $x^i/i!$.

- Write a method called `myexp` that adds up the first `n` terms of the series shown above. You can use the `factorial` method from Section 5.10 or your iterative version.
- You can make this method much more efficient if you realize that in each iteration the numerator of the term is the same as its predecessor multiplied by `x` and the denominator is the same as its predecessor multiplied by `i`. Use this observation to eliminate the use of `Math.pow` and `factorial`, and check that you still get the same result.
- Write a method called `check` that takes a single parameter, `x`, and that prints the values of `x`, `Math.exp(x)` and `myexp(x)` for various values of `x`. The output should look something like:

```
1.0      2.708333333333333      2.718281828459045
```

HINT: you can use the String `"\t"` to print a tab character between columns of a table.

- d. Vary the number of terms in the series (the second argument that **check** sends to **myexp**) and see the effect on the accuracy of the result. Adjust this value until the estimated value agrees with the “correct” answer when **x** is 1.
- e. Write a loop in **main** that invokes **check** with the values 0.1, 1.0, 10.0, and 100.0. How does the accuracy of the result vary as **x** varies? Compare the number of digits of agreement rather than the difference between the actual and estimated values.
- f. Add a loop in **main** that checks **myexp** with the values -0.1, -1.0, -10.0, and -100.0. Comment on the accuracy.

Exercise 6.7 One way to evaluate e^{-x^2} is to use the infinite series expansion

$$e^{-x^2} = 1 - 2x + 3x^2/2! - 4x^3/3! + 5x^4/4! - \dots \quad (6.4)$$

In other words, we need to add up a series of terms where the i th term is equal to $(-1)^i(i+1)x^i/i!$. Write a method named **gauss** that takes **x** and **n** as arguments and that returns the sum of the first **n** terms of the series. You should not use **factorial** or **pow**.

Chapter 7

Strings and things

7.1 Invoking methods on objects

In Java and other object-oriented languages, objects are collections of related data that come with a set of methods. These methods operate on the objects, performing computations and sometimes modifying the object's data.

Of the types we have seen so far, only **Strings** are objects. Based on the definition of object, you might ask “What is the data contained in a **String** object?” and “What are the methods we can invoke on **String** objects?”

The data contained in a **String** object are the letters of the string. There are quite a few methods that operate on **Strings**, but I will only use a few in this book. The rest are documented at

<http://java.sun.com/j2se/1.4.1/docs/api/java/lang/String.html>

The first method we will look at is **charAt**, which allows you to extract letters from a **String**. In order to store the result, we need a variable type that can store individual letters (as opposed to strings). Individual letters are called characters, and the variable type that stores them is called **char**.

chars work just like the other types we have seen:

```
char fred = 'c';
if (fred == 'c') {
    System.out.println (fred);
}
```

Character values appear in single quotes ('c'). Unlike string values (which appear in double quotes), character values can contain only a single letter or symbol.

Here's how the **charAt** method is used:

```
String fruit = "banana";
char letter = fruit.charAt(1);
System.out.println (letter);
```

The syntax `fruit.charAt` indicates that I am invoking the `charAt` method on the object named `fruit`. I am passing the argument `1` to this method, which indicates that I would like to know the first letter of the string. The result is a character, which is stored in a `char` named `letter`. When I print the value of `letter`, I get a surprise:

```
a
```

`a` is not the first letter of `"banana"`. Unless you are a computer scientist. For perverse reasons, computer scientists always start counting from zero. The 0th letter (“zeroeth”) of `"banana"` is `b`. The 1th letter (“oneth”) is `a` and the 2th (“twoeth”) letter is `n`.

If you want the zeroeth letter of a string, you have to pass zero as an argument:

```
char letter = fruit.charAt(0);
```

7.2 Length

The second `String` method we’ll look at is `length`, which returns the number of characters in the string. For example:

```
int length = fruit.length();
```

`length` takes no arguments, as indicated by `()`, and returns an integer, in this case `6`. Notice that it is legal to have a variable with the same name as a method (although it can be confusing for human readers).

To find the last letter of a string, you might be tempted to try something like

```
int length = fruit.length();
char last = fruit.charAt (length);           // WRONG!!
```

That won’t work. The reason is that there is no 6th letter in `"banana"`. Since we started counting at `0`, the 6 letters are numbered from `0` to `5`. To get the last character, you have to subtract `1` from `length`.

```
int length = fruit.length();
char last = fruit.charAt (length-1);
```

7.3 Traversal

A common thing to do with a string is start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. A natural way to encode a traversal is with a `while` statement:

```
int index = 0;
while (index < fruit.length()) {
    char letter = fruit.charAt (index);
    System.out.println (letter);
    index = index + 1;
}
```

This loop traverses the string and prints each letter on a line by itself. Notice that the condition is `index < fruit.length()`, which means that when `index` is equal to the length of the string, the condition is false and the body of the loop is not executed. The last character we access is the one with the index `fruit.length()-1`.

The name of the loop variable is `index`. An **index** is a variable or value used to specify one member of an ordered set (in this case the set of characters in the string). The index indicates (hence the name) which one you want. The set has to be ordered so that each letter has an index and each index refers to a single character.

As an exercise, write a method that takes a `String` as an argument and that prints the letters backwards, all on one line.

7.4 Run-time errors

Way back in Section 1.3.2 I talked about run-time errors, which are errors that don't appear until a program has started running. In Java run-time errors are called **exceptions**.

So far, you probably haven't seen many run-time errors, because we haven't been doing many things that can cause one. Well, now we are. If you use the `charAt` command and you provide an index that is negative or greater than `length-1`, you will get an exception: specifically, a `StringIndexOutOfBoundsException`. Try it and see how it looks.

If your program causes an exception, it prints an error message indicating the type of exception and where in the program it occurred. Then the program terminates.

7.5 Reading documentation

If you go to

<http://java.sun.com/j2se/1.4/docs/api/java/lang/String.html>

and click on `charAt`, you get the following documentation (or something like it):

```
public char charAt(int index)
```

Returns the character at the specified index.

An index ranges from 0 to `length() - 1`.

Parameters: `index` - the index of the character.

Returns: the character at the specified index of this string.

The first character is at index 0.

Throws: `StringIndexOutOfBoundsException` if the index is out of range.

The first line is the method's **prototype**, which indicates the name of the method, the type of the parameters, and the return type.

The next line describes what the method does. The following lines explain the parameters and return values. In this case the explanations are a bit redundant, but the documentation is supposed to fit a standard format. The last line explains what exceptions, if any, can be caused by this method.

7.6 The `indexOf` method

In some ways, `indexOf` is the opposite of `charAt`. `charAt` takes an index and returns the character at that index. `indexOf` takes a character and finds the index where that character appears.

`charAt` fails if the index is out of range, and causes an exception. `indexOf` fails if the character does not appear in the string, and returns the value `-1`.

```
String fruit = "banana";
int index = fruit.indexOf('a');
```

This finds the index of the letter 'a' in the string. In this case, the letter appears three times, so it is not obvious what `indexOf` should do. According to the documentation, it returns the index of the *first* appearance.

In order to find subsequent appearances, there is an alternative version of `indexOf` (for an explanation of this kind of overloading, see Section 5.4). It takes a second argument that indicates where in the string to start looking. If we invoke

```
int index = fruit.indexOf('a', 2);
```

it will start at the twoeth letter (the first `n`) and find the second `a`, which is at index 3. If the letter happens to appear at the starting index, the starting index is the answer. Thus,

```
int index = fruit.indexOf('a', 5);
```

returns 5. Based on the documentation, it is a little tricky to figure out what happens if the starting index is out of range:

`indexOf` returns the index of the first occurrence of the character in the character sequence represented by this object that is greater than or equal to `fromIndex`, or `-1` if the character does not occur.

One way to figure out what this means is to try out a couple of cases. Here are the results of my experiments:

- If the starting index is greater than or equal to `length()`, the result is `-1`, indicating that the letter does not appear at any index greater than the starting index.

- If the starting index is negative, the result is 1, indicating the first appearance of the letter at an index greater than the starting index.

If you go back and look at the documentation, you'll see that this behavior is consistent with the definition, even if it was not immediately obvious. Now that we have a better idea how `indexOf` works, we can use it as part of a program.

7.7 Looping and counting

The following program counts the number of times the letter 'a' appears in a string:

```
String fruit = "banana";
int length = fruit.length();
int count = 0;

int index = 0;
while (index < length) {
    if (fruit.charAt(index) == 'a') {
        count = count + 1;
    }
    index = index + 1;
}
System.out.println (count);
```

This program demonstrates a common idiom, called a **counter**. The variable `count` is initialized to zero and then incremented each time we find an 'a' (to **increment** is to increase by one; it is the opposite of **decrement**, and unrelated to **excrement**, which is a noun). When we exit the loop, `count` contains the result: the total number of a's.

As an exercise, encapsulate this code in a method named `countLetters`, and generalize it so that it accepts the string and the letter as arguments.

As a second exercise, rewrite the method so that it uses `indexOf` to locate the a's, rather than checking the characters one by one.

7.8 Increment and decrement operators

Incrementing and decrementing are such common operations that Java provides special operators for them. The `++` operator adds one to the current value of an `int` or `char`. `--` subtracts one. Neither operator works on `doubles`, `booleans` or `Strings`.

Technically, it is legal to increment a variable and use it in an expression at the same time. For example, you might see something like:

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

Looking at this, it is not clear whether the increment will take effect before or after the value is printed. Because expressions like this tend to be confusing, I would discourage you from using them. In fact, to discourage you even more, I'm not going to tell you what the result is. If you really want to know, you can try it.

Using the increment operators, we can rewrite the letter-counter:

```
int index = 0;
while (index < length) {
    if (fruit.charAt(index) == 'a') {
        count++;
    }
    index++;
}
```

It is a common error to write something like

```
index = index++;           // WRONG!!
```

Unfortunately, this is syntactically legal, so the compiler will not warn you. The effect of this statement is to leave the value of `index` unchanged. This is often a difficult bug to track down.

Remember, you can write `index = index + 1;`, or you can write `index++;`, but you shouldn't mix them.

7.9 Strings are immutable

As you look over the documentation of the `String` methods, you might notice `toUpperCase` and `toLowerCase`. These methods are often a source of confusion, because it sounds like they have the effect of changing (or mutating) an existing string. Actually, neither these methods nor any others can change a string, because strings are **immutable**.

When you invoke `toUpperCase` on a `String`, you get a *new* `String` as a return value. For example:

```
String name = "Alan Turing";
String upperName = name.toUpperCase ();
```

After the second line is executed, `upperName` contains the value `"ALAN TURING"`, but `name` still contains `"Alan Turing"`.

7.10 Strings are incomparable

It is often necessary to compare strings to see if they are the same, or to see which comes first in alphabetical order. It would be nice if we could use the comparison operators, like `==` and `>`, but we can't.

In order to compare `Strings`, we have to use the `equals` and `compareTo` methods. For example:

```
String name1 = "Alan Turing";
String name2 = "Ada Lovelace";

if (name1.equals (name2)) {
    System.out.println ("The names are the same.");
}

int flag = name1.compareTo (name2);
if (flag == 0) {
    System.out.println ("The names are the same.");
} else if (flag < 0) {
    System.out.println ("name1 comes before name2.");
} else if (flag > 0) {
    System.out.println ("name2 comes before name1.");
}
```

The syntax here is a little weird. To compare two things, you have to invoke a method on one of them and pass the other as an argument.

The return value from `equals` is straightforward enough; `true` if the strings contain the same characters, and `false` otherwise.

The return value from `compareTo` is a little odd. It is the difference between the first characters in the strings that differ. If the strings are equal, it is 0. If the first string (the one on which the method is invoked) comes first in the alphabet, the difference is negative. Otherwise, the difference is positive. In this case the return value is positive 8, because the second letter of “Ada” comes before the second letter of “Alan” by 8 letters.

Using `compareTo` is often tricky, and I never remember which way is which without looking it up, but the good news is that the interface is pretty standard for comparing many types of objects, so once you get it you are all set.

Just for completeness, I should admit that it is *legal*, but very seldom *correct*, to use the `==` operator with `Strings`. But what that means will not make sense until later, so for now, don’t do it.

7.11 Glossary

object: A collection of related data that comes with a set of methods that operate on it. The only objects we have used so far are `Strings`.

index: A variable or value used to select one of the members of an ordered set, like a character from a string.

traverse: To iterate through all the elements of a set performing a similar operation on each.

counter: A variable used to count something, usually initialized to zero and then incremented.

increment: Increase the value of a variable by one. The increment operator in Java is `++`.

decrement: Decrease the value of a variable by one. The decrement operator in Java is `--`.

exception: A run time error. Exceptions cause the execution of a program to terminate.

7.12 Exercises

Exercise 7.1

The point of this exercise is to try out some of the String operations and fill in some of the details that aren't covered in the chapter.

- Create a new program named `Test.java` and write a `main` method that contains expressions that combine various types using the `+` operator. For example, what happens when you “add” a `String` and a `char`? Does it perform addition or concatenation? What is the type of the result? (How can you determine the type of the result?)
- Make a bigger copy of the following table and fill it in. At the intersection of each pair of types, you should indicate whether it is legal to use the `+` operator with these types, what operation is performed (addition or concatenation), and what the type of the result is.

	boolean	char	int	String
boolean				
char				
int				
String				

- Think about some of the choices the designers of Java made when they filled in this table. How many of the entries seem reasonable, as if there were no other reasonable choice? How many seem like arbitrary choices from several equally reasonable possibilities? How many seem stupid?
- Here's a puzzler: normally, the statement `x++` is exactly equivalent to `x = x + 1`. Unless `x` is a `char`! In that case, `x++` is legal, but `x = x + 1` causes an error. Try it out and see what the error message is, then see if you can figure out what is going on.

Exercise 7.2

What is the output of this program? Describe in a sentence, abstractly, what `bing` does (not how it works).

```
public class Mystery {

    public static String bing (String s) {
        int i = s.length() - 1;
        String total = "";
```



```

        while (i >= 0 ) {
            char ch = s.charAt (i);
            System.out.println (i + "    " + ch);

            total = total + ch;
            i--;
        }
        return total;
    }

    public static void main (String[] args) {
        System.out.println (bing ("Allen"));
    }
}

```

Exercise 7.3 A friend of yours shows you the following method and explains that if `number` is any two-digit number, the program will output the number backwards. He claims that if `number` is 17, the method will output 71.

Is he right? If not, explain what the program actually does and modify it so that it does the right thing.

```

    int number = 17;
    int lastDigit = number%10;
    int firstDigit = number/10;
    System.out.println (lastDigit + firstDigit);

```

Exercise 7.4 What is the output of the following program?

```

public class Rarefy {

    public static void rarefy (int x) {
        if (x == 0) {
            return;
        } else {
            rarefy (x/2);
        }

        System.out.print (x%2);
    }

    public static void main (String[] args) {
        rarefy (5);
        System.out.println ("");
    }
}

```

Explain in 4-5 words what the method `rarefy` really does.

Exercise 7.5

- a. Create a new program named `Palindrome.java`.

- b. Write a method named `first` that takes a `String` and returns the first letter, and one named `last` that returns the last letter.
- c. Write a method named `middle` that takes a `String` and returns a substring that contains everything *except* the first and last characters.
 Hint: read the documentation of the `substring` method in the `String` class. Run a few tests to make sure you understand how `substring` works before you try to write `middle`.
 What happens if you invoke `middle` on a string that has only two letters? One letter? No letters?
- d. The usual definition of a palindrome is a word that reads the same both forward and backward, like “otto” and “palindromeemordnilap.” An alternative way to define a property like this is to specify a way of testing for the property. For example, we might say, “a single letter is a palindrome, and a two-letter word is a palindrome if the letters are the same, and any other word is a palindrome if the first letter is the same as the last and the middle is a palindrome.”
 Write a recursive method named `isPalindrome` that takes a `String` and that returns a boolean indicating whether the word is a palindrome or not.
- e. Once you have a working palindrome checker, look for ways to simplify it by reducing the number of conditions you check. Hint: it might be useful to adopt the definition that the empty string is a palindrome.
- f. On a piece of paper, figure out a strategy for checking palindromes iteratively. There are several possible approaches, so make sure you have a solid plan before you start writing code.
- g. Implement your strategy in a method called `isPalindromeIter`.

Exercise 7.6 A word is said to be “abecedarian” if the letters in the word appear in alphabetical order. For example, the following are all 6-letter English abecedarian words.

abdest, acknow, acorsy, ademtp, adipsy, agnosy, befist, behint, beknow,
 bijoux, biopsy, cestuy, chintz, deflux, dehors, dehort, deinos, diluvy,
 dimpsy

- a. Describe an algorithm for checking whether a given word (`String`) is abecedarian, assuming that the word contains only lower-case letters. Your algorithm can be iterative or recursive.
- b. Implement your algorithm in a method called `isAbecedarian`.

Exercise 7.7 A dupledrome is a word that contains only double letters, like “llaammaa” or “ssaabb”. I conjecture that there are no dupledromes in common English use. To test that conjecture, I would like a program that reads words from the dictionary one at a time and checks them for dupledromity.

Write a method called `isDupledrome` that takes a `String` and returns a boolean indicating whether the word is a dupledrome.

Exercise 7.8

When names are recorded in a computer, they are sometimes written with the first name first, like “Allen Downey,” and sometimes with the last name first and a comma, like “Downey, Allen.” That can make it difficult to compare names and put them in alphabetical order.

A related problem is that some people have names with capital letters in funny places, like my friend Beth DeSombre, or no capitals at all, like the poet e.e. cummings. When computers compare characters, they usually put all the capital letters before the lower-case letters. As a result, computers often put names with capitals in the wrong order.

If you are curious, find a local phone book like a campus directory and see if you can figure out the alphabetizing scheme. Look for multiple-word names like Van Houten and names with nonstandard capitalization, like desJardins. See if you can figure out the sorting rules. If you have access to a European phone book, try that, too, and see if the rules are different.

The result of all this nonstandardization is that it is usually not right to sort names using simple String comparison. A common solution is to keep two versions of each name: the printable version and an internal format used for sorting.

In this exercise, you will write a method that compares two names by converting them to a standard format. We will work from the bottom up, writing some helper methods and then `compareName`.

- a. Create a new program called `Name.java`. In the documentation of String, read about `find` and `toLowerCase` and `compareTo`. Write some simple code to test each of them and make sure you understand how they work.
- b. Write a method called `hasComma` that takes a name as an argument and that returns a boolean indicating whether it contains a comma. If it does, you can assume that it is in last name first format.
- c. Write a method called `convertName` that takes a name as an argument. It should check whether it contains a comma. If it does, it should just return the string. If not, then it should assume that the name is in first name first format, and it should return a new string that contains the name converted to last name first format.
- d. Write a method called `compareName` that takes two names as arguments and that returns -1 if the first comes before the second alphabetically, 0 if the names are equal alphabetically, and 1 otherwise. Your method should be case-insensitive, meaning that it doesn't matter whether the letters are upper- or lower-case.

Exercise 7.9

- a. The Captain Crunch decoder ring works by taking each letter in a string and adding 13 to it. For example, 'a' becomes 'n' and 'b' becomes 'o'. The letters “wrap around” at the end, so 'z' becomes 'm'.

Write a method that takes a String and that returns a new String containing the encoded version. You should assume that the String contains upper and lower case letters, and spaces, but no other punctuation. Lower case letters should be transformed into other lower case letters; upper into upper. You should not encode the spaces.

- b. Generalize the Captain Crunch method so that instead of adding 13 to the letters, it adds any given amount. Now you should be able to encode things by adding 13 and decode them by adding -13. Try it.

Chapter 8

Interesting objects

8.1 What's interesting?

Although `Strings` are objects, they are not very interesting objects, because

- They are immutable.
- They have no instance variables.
- You don't have to use the `new` command to create one.

In this chapter, we are going to use two new object types that are part of the Java language, `Point` and `Rectangle`. Right from the start, I want to make it clear that these points and rectangles are not graphical objects that appear on the screen. They are variables that contain data, just like `ints` and `doubles`. Like other variables, they are used internally to perform computations.

The definitions of the `Point` and `Rectangle` classes are in the `java.awt` package, so we have to import them.

8.2 Packages

The built-in Java classes are divided into a number of **packages**, including `java.lang`, which contains almost all of the classes we have seen so far, and `java.awt`, which contains classes that pertain to the Java **Abstract Window Toolkit** (AWT), which contains classes for windows, buttons, graphics, etc.

In order to use a package, you have to **import** it, which is why the program in Section D.1 starts with `import java.awt.*`. The `*` indicates that we want to import all the classes in the AWT package. If you want, you can name the classes you want to import explicitly, but there is no great advantage. The classes in `java.lang` are imported automatically, which is why most of our programs haven't required an `import` statement.

All `import` statements appear at the beginning of the program, outside the class definition.

8.3 Point objects

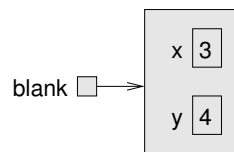
At the most basic level, a point is two numbers (coordinates) that we treat collectively as a single object. In mathematical notation, points are often written in parentheses, with a comma separating the coordinates. For example, $(0,0)$ indicates the origin, and (x,y) indicates the point x units to the right and y units up from the origin.

In Java, a point is represented by a `Point` object. To create a new point, you have to use the `new` command:

```
Point blank;  
blank = new Point (3, 4);
```

The first line is a conventional variable declaration: `blank` has type `Point`. The second line is kind of funny-looking; it invokes the `new` command, specifies the type of the new object, and provides arguments. It will probably not surprise you that the arguments are the coordinates of the new point, $(3,4)$.

The result of the `new` command is a **reference** to the new point. I'll explain references more later; for now the important thing is that the variable `blank` contains a reference to the newly-created object. There is a standard way to diagram this assignment, shown in the figure.



As usual, the name of the variable `blank` appears outside the box and its value appears inside the box. In this case, that value is a reference, which is shown graphically with a dot and an arrow. The arrow points to the object we're referring to.

The big box shows the newly-created object with the two values in it. The names `x` and `y` are the names of the **instance variables**.

Taken together, all the variables, values, and objects in a program are called the **state**. Diagrams like this that show the state of the program are called **state diagrams**. As the program runs, the state changes, so you should think of a state diagram as a snapshot of a particular point in the execution.

8.4 Instance variables

The pieces of data that make up an object are sometimes called components, records, or fields. In Java they are called instance variables because each object, which is an **instance** of its type, has its own copy of the instance variables.

It's like the glove compartment of a car. Each car is an instance of the type "car," and each car has its own glove compartment. If you asked me to get something from the glove compartment of your car, you would have to tell me which car is yours.

Similarly, if you want to read a value from an instance variable, you have to specify the object you want to get it from. In Java this is done using "dot notation."

```
int x = blank.x;
```

The expression `blank.x` means "go to the object `blank` refers to, and get the value of `x`." In this case we assign that value to a local variable named `x`. Notice that there is no conflict between the local variable named `x` and the instance variable named `x`. The purpose of dot notation is to identify *which* variable you are referring to unambiguously.

You can use dot notation as part of any Java expression, so the following are legal.

```
System.out.println (blank.x + ", " + blank.y);
int distance = blank.x * blank.x + blank.y * blank.y;
```

The first line prints 3, 4; the second line calculates the value 25.

8.5 Objects as parameters

You can pass objects as parameters in the usual way. For example

```
public static void printPoint (Point p) {
    System.out.println "(" + p.x + ", " + p.y + ")";
}
```

is a method that takes a point as an argument and prints it in the standard format. If you invoke `printPoint (blank)`, it will print (3, 4). Actually, Java has a built-in method for printing `Points`. If you invoke `System.out.println (blank)`, you get

```
java.awt.Point[x=3,y=4]
```

This is a standard format Java uses for printing objects. It prints the name of the type, followed by the contents of the object, including the names and values of the instance variables.

As a second example, we can rewrite the `distance` method from Section 5.2 so that it takes two `Points` as parameters instead of four doubles.

```
public static double distance (Point p1, Point p2) {
    double dx = (double)(p2.x - p1.x);
    double dy = (double)(p2.y - p1.y);
    return Math.sqrt (dx*dx + dy*dy);
}
```

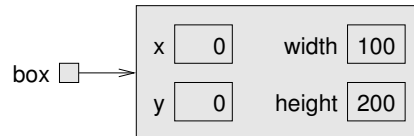
The typecasts are not really necessary; I just added them as a reminder that the instance variables in a `Point` are integers.

8.6 Rectangles

Rectangles are similar to points, except that they have four instance variables, named **x**, **y**, **width** and **height**. Other than that, everything is pretty much the same.

```
Rectangle box = new Rectangle (0, 0, 100, 200);
```

creates a new **Rectangle** object and makes **box** refer to it. The figure shows the effect of this assignment.



If you print **box**, you get

```
java.awt.Rectangle[x=0,y=0,width=100,height=200]
```

Again, this is the result of a built-in Java method that knows how to print **Rectangle** objects.

8.7 Objects as return types

You can write methods that return objects. For example, **findCenter** takes a **Rectangle** as an argument and returns a **Point** that contains the coordinates of the center of the **Rectangle**:

```
public static Point findCenter (Rectangle box) {  
    int x = box.x + box.width/2;  
    int y = box.y + box.height/2;  
    return new Point (x, y);  
}
```

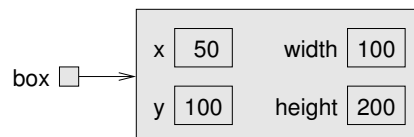
Notice that you can use **new** to create a new object, and then immediately use the result as a return value.

8.8 Objects are mutable

You can change the contents of an object by making an assignment to one of its instance variables. For example, to “move” a rectangle without changing its size, you could modify the **x** and **y** values:

```
box.x = box.x + 50;  
box.y = box.y + 100;
```

The result is shown in the figure:



We could take this code and encapsulate it in a method, and generalize it to move the rectangle by any amount:

```
public static void moveRect (Rectangle box, int dx, int dy) {
    box.x = box.x + dx;
    box.y = box.y + dy;
}
```

The variables `dx` and `dy` indicate how far to move the rectangle in each direction. Invoking this method has the effect of modifying the `Rectangle` that is passed as an argument.

```
Rectangle box = new Rectangle (0, 0, 100, 200);
moveRect (box, 50, 100);
System.out.println (box);
```

prints `java.awt.Rectangle[x=50,y=100,width=100,height=200]`.

Modifying objects by passing them as arguments to methods can be useful, but it can also make debugging more difficult because it is not always clear which method invocations do or do not modify their arguments. Later, I will discuss some pros and cons of this programming style.

In the meantime, we can enjoy the luxury of Java's built-in methods, which include `translate`, which does exactly the same thing as `moveRect`, although the syntax for invoking it is a little different. Instead of passing the `Rectangle` as an argument, we invoke `translate` on the `Rectangle` and pass only `dx` and `dy` as arguments.

```
box.translate (50, 100);
```

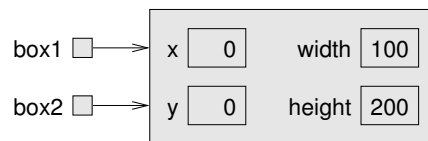
The effect is exactly the same.

8.9 Aliasing

Remember that when you make an assignment to an object variable, you are assigning a *reference* to an object. It is possible to have multiple variables that refer to the same object. For example, this code:

```
Rectangle box1 = new Rectangle (0, 0, 100, 200);
Rectangle box2 = box1;
```

generates a state diagram that looks like this:

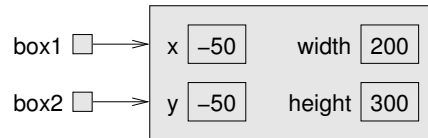


Both `box1` and `box2` refer or “point” to the same object. In other words, this object has two names, `box1` and `box2`. When a person uses two names, it’s called **aliasing**. Same thing with objects.

When two variables are aliased, any changes that affect one variable also affect the other. For example:

```
System.out.println (box2.width);
box1.grow (50, 50);
System.out.println (box2.width);
```

The first line prints 100, which is the width of the `Rectangle` referred to by `box2`. The second line invokes the `grow` method on `box1`, which expands the `Rectangle` by 50 pixels in every direction (see the documentation for more details). The effect is shown in the figure:



As should be clear from this figure, whatever changes are made to `box1` also apply to `box2`. Thus, the value printed by the third line is 200, the width of the expanded rectangle. (As an aside, it is perfectly legal for the coordinates of a `Rectangle` to be negative.)

As you can tell even from this simple example, code that involves aliasing can get confusing fast, and it can be very difficult to debug. In general, aliasing should be avoided or used with care.

8.10 null

When you create an object variable, remember that you are creating a *reference* to an object. Until you make the variable point to an object, the value of the variable is `null`. `null` is a special value in Java (and a Java keyword) that is used to mean “no object.”

The declaration `Point blank;` is equivalent to this initialization

```
Point blank = null;
```

and is shown in the following state diagram:

`blank`

The value `null` is represented by a dot with no arrow.

If you try to use a null object, either by accessing an instance variable or invoking a method, you will get a `NullPointerException`. The system will print an error message and terminate the program.

```
Point blank = null;
int x = blank.x;           // NullPointerException
blank.translate (50, 50);   // NullPointerException
```

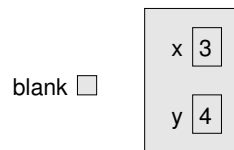
On the other hand, it is legal to pass a null object as an argument or receive one as a return value. In fact, it is common to do so, for example to represent an empty set or indicate an error condition.

8.11 Garbage collection

In Section 8.9 we talked about what happens when more than one variable refers to the same object. What happens when *no* variable refers to an object? For example:

```
Point blank = new Point (3, 4);  
blank = null;
```

The first line creates a new `Point` object and makes `blank` refer to it. The second line changes `blank` so that instead of referring to the object, it refers to nothing (the null object).



If no one refers to an object, then no one can read or write any of its values, or invoke a method on it. In effect, it ceases to exist. We could keep the object in memory, but it would only waste space, so periodically as your program runs, the Java system looks for stranded objects and reclaims them, in a process called **garbage collection**. Later, the memory space occupied by the object will be available to be used as part of a new object.

You don't have to do anything to make garbage collection work, and in general you will not be aware of it.

8.12 Objects and primitives

There are two kinds of types in Java, primitive types and object types. Primitives, like `int` and `boolean` begin with lower-case letters; object types begin with upper-case letters. This distinction is useful because it reminds us of some of the differences between them:

- When you declare a primitive variable, you get storage space for a primitive value. When you declare an object variable, you get a space for a reference to an object. In order to get space for the object itself, you have to use the `new` command.
- If you don't initialize a primitive type, it is given a default value that depends on the type. For example, 0 for `ints` and `true` for `booleans`. The default value for object types is `null`, which indicates no object.
- Primitive variables are well isolated in the sense that there is nothing you can do in one method that will affect a variable in another method. Object variables can be tricky to work with because they are not as well isolated. If you pass a reference to an object as an argument, the method you invoke might modify the object, in which case you will see the effect. The same

is true when you invoke a method on an object. Of course, that can be a good thing, but you have to be aware of it.

There is one other difference between primitives and object types. You cannot add new primitives to the Java language (unless you get yourself on the standards committee), but you can create new object types! We'll see how in the next chapter.

8.13 Glossary

package: A collection of classes. The built-in Java classes are organized in packages.

AWT: The Abstract Window Toolkit, one of the biggest and most commonly-used Java packages.

instance: An example from a category. My cat is an instance of the category “feline things.” Every object is an instance of some class.

instance variable: One of the named data items that make up an object. Each object (instance) has its own copy of the instance variables for its class.

reference: A value that indicates an object. In a state diagram, a reference appears as an arrow.

aliasing: The condition when two or more variables refer to the same object.

garbage collection: The process of finding objects that have no references and reclaiming their storage space.

state: A complete description of all the variables and objects and their values, at a given point during the execution of a program.

state diagram: A snapshot of the state of a program, shown graphically.

8.14 Exercises

Exercise 8.1

- For the following program, draw a stack diagram showing the local variables and parameters of `main` and `fred`, and show any objects those variables refer to.
- What is the output of this program?

```
public static void main (String[] args)
{
    int x = 5;
    Point blank = new Point (1, 2);

    System.out.println (fred (x, blank));
    System.out.println (x);
}
```

```

        System.out.println (blank.x);
        System.out.println (blank.y);
    }

    public static int fred (int x, Point p)
    {
        x = x + 7;
        return x + p.x + p.y;
    }

```

The point of this exercise is to make sure you understand the mechanism for passing Objects as parameters.

Exercise 8.2

- a. For the following program, draw a stack diagram showing the state of the program just before `distance` returns. Include all variables and parameters and the objects those variables refer to.
- b. What is the output of this program?

```

public static double distance (Point p1, Point p2) {
    int dx = p1.x - p2.x;
    int dy = p1.y - p2.y;
    return Math.sqrt (dx*dx + dy*dy);
}

public static Point findCenter (Rectangle box) {
    int x = box.x + box.width/2;
    int y = box.y + box.height/2;
    return new Point (x, y);
}

public static void main (String[] args) {
    Point blank = new Point (5, 8);

    Rectangle rect = new Rectangle (0, 2, 4, 4);
    Point center = findCenter (rect);

    double dist = distance (center, blank);

    System.out.println (dist);
}

```

Exercise 8.3 The method `grow` is part of the built-in `Rectangle` class. Here is the documentation of it (from the Sun web page):

```
public void grow(int h, int v)
```

Grows the rectangle both horizontally and vertically.

This method modifies the rectangle so that it is `h` units larger on both the left and right side, and `v` units larger at both the top and

bottom.

The new rectangle has $(x - h, y - v)$ as its top-left corner, a width of $\text{width} + 2h$, and a height of $\text{height} + 2v$.

If negative values are supplied for h and v , the size of the rectangle decreases accordingly. The `grow` method does not check whether the resulting values of width and height are non-negative.

- a. What is the output of the following program?
- b. Draw a state diagram that shows the state of the program just before the end of `main`. Include all local variables and the objects they refer to.
- c. At the end of `main`, are `p1` and `p2` aliased? Why or why not?

```
public static void printPoint (Point p) {
    System.out.println "(" + p.x + ", " + p.y + ")";
}

public static Point findCenter (Rectangle box) {
    int x = box.x + box.width/2;
    int y = box.y + box.height/2;
    return new Point (x, y);
}

public static void main (String[] args) {

    Rectangle box1 = new Rectangle (2, 4, 7, 9);
    Point p1 = findCenter (box1);
    printPoint (p1);

    box1.grow (1, 1);
    Point p2 = findCenter (box1);
    printPoint (p2);
}
```

Exercise 8.4 You are probably getting sick of the factorial method by now, but we're going to do one more version.

- a. Create a new program called `Big.java` and start by writing an iterative version of `factorial`.
 - b. Print a table of the integers from 0 to 30 along with their factorials. At some point around 15, you will probably see that the answers are not right any more. Why not?
 - c. `BigInteger` are built-in objects that can represent arbitrarily big integers. There is no upper bound except the limitations of memory size and processing speed. Print the documentation for the `BigInteger` class in the `java.math` package, and read it.
 - d. There are several ways to create a new `BigInteger`, but the one I recommend uses `valueOf`. The following code converts an integer to a `BigInteger`:
- ```
int x = 17;
BigInteger big = BigInteger.valueOf (x);
```