

-
- d. Remove the word `static`.
 - e. Remove the word `public`.
 - f. Remove the word `System`.
 - g. Replace `println` with `pintln`.
 - h. Replace `println` with `print`. This one is tricky because it is a logical error, not a syntax error. The statement `System.out.print` is legal, but it may or may not do what you expect.
 - i. Delete one of the parentheses. Add an extra one.

Chapter 2

Variables and types

2.1 More printing

As I mentioned in the last chapter, you can put as many statements as you want in `main`. For example, to print more than one line:

```
class Hello {  
  
    // main: generate some simple output  
  
    public static void main (String[] args) {  
        System.out.println ("Hello, world.");    // print one line  
        System.out.println ("How are you?");    // print another  
    }  
}
```

Also, as you can see, it is legal to put comments at the end of a line, as well as on a line by themselves.

The phrases that appear in quotation marks are called **strings**, because they are made up of a sequence (string) of letters. Actually, strings can contain any combination of letters, numbers, punctuation marks, and other special characters.

`println` is short for “print line,” because after each line it adds a special character, called a **newline**, that causes the cursor to move to the next line of the display. The next time `println` is invoked, the new text appears on the next line.

Often it is useful to display the output from multiple print statements all on one line. You can do this with the `print` command:

```
class Hello {  
  
    // main: generate some simple output
```

```
public static void main (String[] args) {  
    System.out.print ("Goodbye, ");  
    System.out.println ("cruel world!");  
}  
}
```

In this case the output appears on a single line as `Goodbye, cruel world!`. Notice that there is a space between the word “Goodbye” and the second quotation mark. This space appears in the output, so it affects the behavior of the program.

Spaces that appear outside of quotation marks generally do not affect the behavior of the program. For example, I could have written:

```
class Hello {  
public static void main (String[] args) {  
System.out.print ("Goodbye, ");  
System.out.println ("cruel world!");  
}  
}
```

This program would compile and run just as well as the original. The breaks at the ends of lines (newlines) do not affect the program’s behavior either, so I could have written:

```
class Hello { public static void main (String[] args) {  
System.out.print ("Goodbye, "); System.out.println  
("cruel world!");}}
```

That would work, too, although you have probably noticed that the program is getting harder and harder to read. Newlines and spaces are useful for organizing your program visually, making it easier to read the program and locate syntax errors.

2.2 Variables

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a named location that stores a **value**. Values are things that can be printed and stored and (as we’ll see later) operated on. The strings we have been printing (`"Hello, World."`, `"Goodbye, "`, etc.) are values.

In order to store a value, you have to create a variable. Since the values we want to store are strings, we will declare that the new variable is a string:

```
String fred;
```

This statement is a **declaration**, because it declares that the variable named `fred` has the type `String`. Each variable has a type that determines what kind of values it can store. For example, the `int` type can store integers, and it will probably come as no surprise that the `String` type can store strings.

You will notice that some types begin with a capital letter and some with lower-case. We will learn the significance of this distinction later, but for now you should take care to get it right. There is no such type as `Int` or `string`, and the compiler will object if you try to make one up.

To create an integer variable, the syntax is `int bob;`, where `bob` is the arbitrary name you made up for the variable. In general, you will want to make up variable names that indicate what you plan to do with the variable. For example, if you saw these variable declarations:

```
String firstName;  
String lastName;  
int hour, minute;
```

you could probably make a good guess at what values would be stored in them. This example also demonstrates the syntax for declaring multiple variables with the same type: `hour` and `second` are both integers (`int` type).

2.3 Assignment

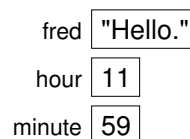
Now that we have created some variables, we would like to store values in them. We do that with an **assignment statement**.

```
fred = "Hello.";    // give fred the value "Hello."  
hour = 11;          // assign the value 11 to hour  
minute = 59;        // set minute to 59
```

This example shows three assignments, and the comments show three different ways people sometimes talk about assignment statements. The vocabulary can be confusing here, but the idea is straightforward:

- When you declare a variable, you create a named storage location.
- When you make an assignment to a variable, you give it a value.

A common way to represent variables on paper is to draw a box with the name of the variable on the outside and the value of the variable on the inside. This figure shows the effect of the three assignment statements:



For each variable, the name of the variable appears outside the box and the value appears inside.

As a general rule, a variable has to have the same type as the value you assign it. You cannot store a `String` in `minute` or an integer in `fred`.

On the other hand, that rule can be confusing, because there are many ways that you can convert values from one type to another, and Java sometimes converts

things automatically. So for now you should remember the general rule, and we'll talk about special cases later.

Another source of confusion is that some strings *look* like integers, but they are not. For example, `fred` can contain the string `"123"`, which is made up of the characters 1, 2 and 3, but that is not the same thing as the *number* 123.

```
fred = "123";    // legal
fred = 123;      // not legal
```

2.4 Printing variables

You can print the value of a variable using the same commands we used to print Strings.

```
class Hello {
    public static void main (String[] args) {
        String firstLine;
        firstLine = "Hello, again!";
        System.out.println (firstLine);
    }
}
```

This program creates a variable named `firstLine`, assigns it the value `"Hello, again!"` and then prints that value. When we talk about “printing a variable,” we mean printing the *value* of the variable. To print the *name* of a variable, you have to put it in quotes. For example: `System.out.println ("firstLine");`

If you want to get a little tricky, you could write

```
String firstLine;
firstLine = "Hello, again!";
System.out.print ("The value of firstLine is ");
System.out.println (firstLine);
```

The output of this program is

The value of firstLine is Hello, again!

I am pleased to report that the syntax for printing a variable is the same regardless of the variable's type.

```
int hour, minute;
hour = 11;
minute = 59;
System.out.print ("The current time is ");
System.out.print (hour);
System.out.print (":");
System.out.print (minute);
System.out.println (".");
```

The output of this program is The current time is 11:59.

WARNING: It is common practice to use several `print` commands followed by a `println`, in order to put multiple values on the same line. But you have

to be careful to remember the `println` at the end. In many environments, the output from `print` is stored without being displayed until the `println` command is invoked, at which point the entire line is displayed at once. If you omit `println`, the program may terminate without ever displaying the stored output!

2.5 Keywords

A few sections ago, I said that you can make up any name you want for your variables, but that's not quite true. There are certain words that are reserved in Java because they are used by the compiler to parse the structure of your program, and if you use them as variable names, it will get confused. These words, called **keywords**, include `public`, `class`, `void`, `int`, and many more.

The complete list is available at

http://java.sun.com/docs/books/jls/second_edition/html/lexical.doc.html

This site, provided by Sun, includes Java documentation I will be referring to throughout the book.

Rather than memorize the list, I would suggest that you take advantage of a feature provided in many Java development environments: code highlighting. As you type, different parts of your program should appear in different colors. For example, keywords might be blue, strings red, and other code black. If you type a variable name and it turns blue, watch out! You might get some strange behavior from the compiler.

2.6 Operators

Operators are special symbols that are used to represent simple computations like addition and multiplication. Most of the operators in Java do exactly what you would expect them to do, because they are common mathematical symbols. For example, the operator for adding two integers is `+`.

The following are all legal Java expressions whose meaning is more or less obvious:

```
1+1          hour-1          hour*60 + minute    minute/60
```

Expressions can contain both variable names and numbers. In each case the name of the variable is replaced with its value before the computation is performed.

Addition, subtraction and multiplication all do what you expect, but you might be surprised by division. For example, the following program:

```
int hour, minute;
hour = 11;
minute = 59;
```

```
System.out.print ("Number of minutes since midnight: ");
System.out.println (hour*60 + minute);
System.out.print ("Fraction of the hour that has passed: ");
System.out.println (minute/60);
```

would generate the following output:

```
Number of minutes since midnight: 719
Fraction of the hour that has passed: 0
```

The first line is what we expected, but the second line is odd. The value of the variable `minute` is 59, and 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that Java is performing **integer division**.

When both of the **operands** are integers (operands are the things operators operate on), the result must also be an integer, and by convention integer division always rounds *down*, even in cases like this where the next integer is so close.

A possible alternative in this case is to calculate a percentage rather than a fraction:

```
System.out.print ("Percentage of the hour that has passed: ");
System.out.println (minute*100/60);
```

The result is:

```
Percentage of the hour that has passed: 98
```

Again the result is rounded down, but at least now the answer is approximately correct. In order to get an even more accurate answer, we could use a different type of variable, called floating-point, that is capable of storing fractional values. We'll get to that in the next chapter.

2.7 Order of operations

When more than one operator appears in an expression the order of evaluation depends on the rules of **precedence**. A complete explanation of precedence can get complicated, but just to get you started:

- Multiplication and division take precedence (happen before) addition and subtraction. So $2*3-1$ yields 5, not 4, and $2/3-1$ yields -1, not 1 (remember that in integer division $2/3$ is 0).
- If the operators have the same precedence they are evaluated from left to right. So in the expression `minute*100/60`, the multiplication happens first, yielding $5900/60$, which in turn yields 98. If the operations had gone from right to left, the result would be $59*1$ which is 59, which is wrong.
- Any time you want to override the rules of precedence (or you are not sure what they are) you can use parentheses. Expressions in parentheses are evaluated first, so $2 * (3-1)$ is 4. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even though it doesn't change the result.

2.8 Operators for Strings

In general you cannot perform mathematical operations on `Strings`, even if the strings look like numbers. The following are illegal (if we know that `fred` has type `String`)

```
fred - 1           "Hello"/123       fred * "Hello"
```

By the way, can you tell by looking at those expressions whether `fred` is an integer or a string? Nope. The only way to tell the type of a variable is to look at the place where it is declared.

Interestingly, the `+` operator *does* work with `Strings`, although it does not do exactly what you might expect. For `Strings`, the `+` operator represents **concatenation**, which means joining up the two operands by linking them end-to-end. So `"Hello, " + "world."` yields the string `"Hello, world."` and `fred + "ism"` adds the suffix *ism* to the end of whatever `fred` is, which is often handy for naming new forms of bigotry.

2.9 Composition

So far we have looked at the elements of a programming language—variables, expressions, and statements—in isolation, without talking about how to combine them.

One of the most useful features of programming languages is their ability to take small building blocks and **compose** them. For example, we know how to multiply numbers and we know how to print; it turns out we can do both at the same time:

```
System.out.println (17 * 3);
```

Actually, I shouldn't say "at the same time," since in reality the multiplication has to happen before the printing, but the point is that any expression, involving numbers, strings, and variables, can be used inside a print statement. We've already seen one example:

```
System.out.println (hour*60 + minute);
```

But you can also put arbitrary expressions on the right-hand side of an assignment statement:

```
int percentage;  
percentage = (minute * 100) / 60;
```

This ability may not seem so impressive now, but we will see other examples where composition makes it possible to express complex computations neatly and concisely.

WARNING: There are limits on where you can use certain expressions; most notably, the left-hand side of an assignment statement has to be a *variable* name, not an expression. That's because the left side indicates the storage location where the result will go. Expressions do not represent storage locations, only values. So the following is illegal: `minute+1 = hour;`

2.10 Glossary

variable: A named storage location for values. All variables have a type, which is declared when the variable is created.

value: A number or string (or other thing to be named later) that can be stored in a variable. Every value belongs to one type.

type: A set of values. The type of a variable determines which values can be stored there. So far, the types we have seen are integers (`int` in Java) and strings (`String` in Java).

keyword: A reserved word that is used by the compiler to parse programs. You cannot use keywords, like `public`, `class` and `void` as variable names.

statement: A line of code that represents a command or action. So far, the statements we have seen are declarations, assignments, and print statements.

declaration: A statement that creates a new variable and determines its type.

assignment: A statement that assigns a value to a variable.

expression: A combination of variables, operators and values that represents a single result value. Expressions also have types, as determined by their operators and operands.

operator: A special symbol that represents a simple computation like addition, multiplication or string concatenation.

operand: One of the values on which an operator operates.

precedence: The order in which operations are evaluated.

concatenate: To join two operands end-to-end.

composition: The ability to combine simple expressions and statements into compound statements and expressions in order to represent complex computations concisely.

2.11 Exercises

Exercise 2.1

- a. Create a new program named `Date.java`. Copy or type in something like the “Hello, World” program and make sure you can compile and run it.
- b. Following the example in Section 2.4, write a program that creates variables named `day`, `date`, `month` and `year`. `day` will contain the day of the week and `date` will contain the day of the month. What type is each variable? Assign values to those variables that represent today’s date.
- c. Print the value of each variable on a line by itself. This is an intermediate step that is useful for checking that everything is working so far.

- d. Modify the program so that it prints the date in standard American form:
`Wednesday, February 17, 1999.`
- e. Modify the program again so that the total output is:
`American format:`
`Wednesday, February 17, 1999`
`European format:`
`Wednesday 17 February, 1999`

The point of this exercise is to use string concatenation to display values with different types (`int` and `String`), and to practice developing programs gradually by adding a few statements at a time.

Exercise 2.2

- a. Create a new program called `Time.java`. From now on, I won't remind you to start with a small, working program, but you should.
- b. Following the example in Section 2.6, create variables named `hour`, `minute` and `second`, and assign them values that are roughly the current time. Use a 24-hour clock, so that at 2pm the value of `hour` is 14.
- c. Make the program calculate and print the number of seconds since midnight.
- d. Make the program calculate and print the number of seconds remaining in the day.
- e. Make the program calculate and print the percentage of the day that has passed.
- f. Change the values of `hour`, `minute` and `second` to reflect the current time (I assume that some time has elapsed), and check to make sure that the program works correctly with different values.

The point of this exercise is to use some of the arithmetic operations, and to start thinking about compound entities like the time of day that are represented with multiple values. Also, you might run into problems computing percentages with `ints`, which is the motivation for floating point numbers in the next chapter.

HINT: you may want to use additional variables to hold values temporarily during the computation. Variables like this, that are used in a computation but never printed, are sometimes called intermediate or temporary variables.

Chapter 3

Methods

3.1 Floating-point

In the last chapter we had some problems dealing with numbers that were not integers. We worked around the problem by measuring percentages instead of fractions, but a more general solution is to use floating-point numbers, which can represent fractions as well as integers. In Java, the floating-point type is called `double`.

You can create floating-point variables and assign values to them using the same syntax we used for the other types. For example:

```
double pi;  
pi = 3.14159;
```

It is also legal to declare a variable and assign a value to it at the same time:

```
int x = 1;  
String empty = "";  
double pi = 3.14159;
```

In fact, this syntax is quite common. A combined declaration and assignment is sometimes called an **initialization**.

Although floating-point numbers are useful, they are often a source of confusion because there seems to be an overlap between integers and floating-point numbers. For example, if you have the value 1, is that an integer, a floating-point number, or both?

Strictly speaking, Java distinguishes the integer value 1 from the floating-point value 1.0, even though they seem to be the same number. They belong to different types, and strictly speaking, you are not allowed to make assignments between types. For example, the following is illegal:

```
int x = 1.1;
```

because the variable on the left is an `int` and the value on the right is a `double`. But it is easy to forget this rule, especially because there are places where Java will automatically convert from one type to another. For example:

```
double y = 1;
```

should technically not be legal, but Java allows it by converting the `int` to a `double` automatically. This leniency is convenient, but it can cause problems; for example:

```
double y = 1 / 3;
```

You might expect the variable `y` to be given the value `0.333333`, which is a legal floating-point value, but in fact it will get the value `0.0`. The reason is that the expression on the right appears to be the ratio of two integers, so Java does *integer* division, which yields the integer value `0`. Converted to floating-point, the result is `0.0`.

One way to solve this problem (once you figure out what it is) is to make the right-hand side a floating-point expression:

```
double y = 1.0 / 3.0;
```

This sets `y` to `0.333333`, as expected.

All the operations we have seen so far—addition, subtraction, multiplication, and division—also work on floating-point values, although you might be interested to know that the underlying mechanism is completely different. In fact, most processors have special hardware just for performing floating-point operations.

3.2 Converting from double to int

As I mentioned, Java converts `ints` to `doubles` automatically if necessary, because no information is lost in the translation. On the other hand, going from a `double` to an `int` requires rounding off. Java doesn't perform this operation automatically, in order to make sure that you, as the programmer, are aware of the loss of the fractional part of the number.

The simplest way to convert a floating-point value to an integer is to use a **typecast**. Typecasting is so called because it allows you to take a value that belongs to one type and “cast” it into another type (in the sense of molding or reforming, not throwing).

Unfortunately, the syntax for typecasting is ugly: you put the name of the type in parentheses and use it as an operator. For example,

```
int x = (int) Math.PI;
```

The `(int)` operator has the effect of converting what follows into an integer, so `x` gets the value `3`.

Typecasting takes precedence over arithmetic operations, so in the following example, the value of `PI` gets converted to an integer first, and the result is `60`, not `62`.

```
int x = (int) Math.PI * 20.0;
```

Converting to an integer always rounds down, even if the fraction part is 0.99999999.

These two properties (precedence and rounding) can make typecasting awkward.

3.3 Math methods

In mathematics, you have probably seen functions like \sin and \log , and you have learned to evaluate expressions like $\sin(\pi/2)$ and $\log(1/x)$. First, you evaluate the expression in parentheses, which is called the **argument** of the function. For example, $\pi/2$ is approximately 1.571, and $1/x$ is 0.1 (assuming that x is 10).

Then you can evaluate the function itself, either by looking it up in a table or by performing various computations. The \sin of 1.571 is 1, and the \log of 0.1 is -1 (assuming that \log indicates the logarithm base 10).

This process can be applied repeatedly to evaluate more complicated expressions like $\log(1/\sin(\pi/2))$. First we evaluate the argument of the innermost function, then evaluate the function, and so on.

Java provides a set of built-in functions that includes most of the mathematical operations you can think of. These functions are called **methods**. Most math methods operate on **doubles**.

The math methods are invoked using a syntax that is similar to the **print** commands we have already seen:

```
double root = Math.sqrt (17.0);
double angle = 1.5;
double height = Math.sin (angle);
```

The first example sets **root** to the square root of 17. The second example finds the sine of 1.5, which is the value of the variable **angle**. Java assumes that the values you use with **sin** and the other trigonometric functions (**cos**, **tan**) are in *radians*. To convert from degrees to radians, you can divide by 360 and multiply by 2π . Conveniently, Java provides π as a built-in value:

```
double degrees = 90;
double angle = degrees * 2 * Math.PI / 360.0;
```

Notice that **PI** is in all capital letters. Java does not recognize **Pi**, **pi**, or **pie**.

Another useful method in the **Math** class is **round**, which rounds a floating-point value off to the nearest integer and returns an **int**.

```
int x = Math.round (Math.PI * 20.0);
```

In this case the multiplication happens first, before the method is invoked. The result is 63 (rounded up from 62.8319).

3.4 Composition

Just as with mathematical functions, Java methods can be **composed**, meaning that you use one expression as part of another. For example, you can use any expression as an argument to a method:

```
double x = Math.cos (angle + Math.PI/2);
```

This statement takes the value `Math.PI`, divides it by two and adds the result to the value of the variable `angle`. The sum is then passed as an argument to the `cos` method. (Notice that `PI` is the name of a variable, not a method, so there are no arguments, not even the empty argument `()`).

You can also take the result of one method and pass it as an argument to another:

```
double x = Math.exp (Math.log (10.0));
```

In Java, the `log` function always uses base e , so this statement finds the log base e of 10 and then raises e to that power. The result gets assigned to `x`; I hope you know what it is.

3.5 Adding new methods

So far we have only been using the methods that are built into Java, but it is also possible to add new methods. Actually, we have already seen one method definition: `main`. The method named `main` is special in that it indicates where the execution of the program begins, but the syntax for `main` is the same as for other method definitions:

```
public static void NAME ( LIST OF PARAMETERS ) {  
    STATEMENTS  
}
```

You can make up any name you want for your method, except that you can't call it `main` or any other Java keyword. The list of parameters specifies what information, if any, you have to provide in order to use (or **invoke**) the new function.

The single parameter for `main` is `String[] args`, which indicates that whoever invokes `main` has to provide an array of Strings (we'll get to arrays in Chapter 10). The first couple of methods we are going to write have no parameters, so the syntax looks like this:

```
public static void newLine () {  
    System.out.println ("");  
}
```

This method is named `newLine`, and the empty parentheses indicate that it takes no parameters. It contains only a single statement, which prints an empty `String`, indicated by `""`. Printing a `String` with no letters in it may not seem all that useful, except remember that `println` skips to the next line after it prints, so this statement has the effect of skipping to the next line.

In `main` we can invoke this new method using syntax that is similar to the way we invoke the built-in Java commands:

```
public static void main (String[] args) {  
    System.out.println ("First line.");  
    newLine ();  
    System.out.println ("Second line.");  
}
```

The output of this program is

First line.

Second line.

Notice the extra space between the two lines. What if we wanted more space between the lines? We could invoke the same method repeatedly:

```
public static void main (String[] args) {  
    System.out.println ("First line.");  
    newLine ();  
    newLine ();  
    newLine ();  
    System.out.println ("Second line.");  
}
```

Or we could write a new method, named `threeLine`, that prints three new lines:

```
public static void threeLine () {  
    newLine (); newLine (); newLine ();  
}  
  
public static void main (String[] args) {  
    System.out.println ("First line.");  
    threeLine ();  
    System.out.println ("Second line.");  
}
```

You should notice a few things about this program:

- You can invoke the same procedure repeatedly. In fact, it is quite common and useful to do so.
- You can have one method invoke another method. In this case, `main` invokes `threeLine` and `threeLine` invokes `newLine`. Again, this is common and useful.
- In `threeLine` I wrote three statements all on the same line, which is syntactically legal (remember that spaces and new lines usually don't change the meaning of a program). On the other hand, it is usually a better idea to put each statement on a line by itself, to make your program easy to read. I sometimes break that rule in this book to save space.

So far, it may not be clear why it is worth the trouble to create all these new methods. Actually, there are a lot of reasons, but this example only demonstrates two:

1. Creating a new method gives you an opportunity to give a name to a group of statements. Methods can simplify a program by hiding a complex computation behind a single command, and by using English words in place of arcane code. Which is clearer, `newLine` or `System.out.println ("")`?
2. Creating a new method can make a program smaller by eliminating repetitive code. For example, how would you print nine consecutive new lines? You could just invoke `threeLine` three times.

3.6 Classes and methods

Pulling together all the code fragments from the previous section, the whole class definition looks like this:

```
class NewLine {  
  
    public static void newLine () {  
        System.out.println ("");  
    }  
  
    public static void threeLine () {  
        newLine (); newLine (); newLine ();  
    }  
  
    public static void main (String[] args) {  
        System.out.println ("First line.");  
        threeLine ();  
        System.out.println ("Second line.");  
    }  
}
```

The first line indicates that this is the class definition for a new class called `NewLine`. A class is a collection of related methods. In this case, the class named `NewLine` contains three methods, named `newLine`, `threeLine`, and `main`.

The other class we've seen is the `Math` class. It contains methods named `sqrt`, `sin`, and many others. When we invoke a mathematical function, we have to specify the name of the class (`Math`) and the name of the function. That's why the syntax is slightly different for built-in methods and the methods that we write:

```
Math.pow (2.0, 10.0);  
newLine ();
```

The first statement invokes the `pow` method in the `Math` class (which raises the first argument to the power of the second argument). The second statement invokes the `newLine` method, which Java assumes (correctly) is in the `NewLine` class, which is what we are writing.

If you try to invoke a method from the wrong class, the compiler will generate an error. For example, if you type:

```
pow (2.0, 10.0);
```

The compiler will say something like, “Can’t find a method named `pow` in class `NewLine`.” If you have seen this message, you might have wondered why it was looking for `pow` in your class definition. Now you know.

3.7 Programs with multiple methods

When you look at a class definition that contains several methods, it is tempting to read it from top to bottom, but that is likely to be confusing, because that is not the **order of execution** of the program.

Execution always begins at the first statement of `main`, regardless of where it is in the program (in this case I deliberately put it at the bottom). Statements are executed one at a time, in order, until you reach a method invocation. Method invocations are like a detour in the flow of execution. Instead of going to the next statement, you go to the first line of the invoked method, execute all the statements there, and then come back and pick up again where you left off.

That sounds simple enough, except that you have to remember that one method can invoke another. Thus, while we are in the middle of `main`, we might have to go off and execute the statements in `threeLine`. But while we are executing `threeLine`, we get interrupted three times to go off and execute `newLine`.

For its part, `newLine` invokes the built-in method `println`, which causes yet another detour. Fortunately, Java is quite adept at keeping track of where it is, so when `println` completes, it picks up where it left off in `newLine`, and then gets back to `threeLine`, and then finally gets back to `main` so the program can terminate.

Actually, technically, the program does not terminate at the end of `main`. Instead, execution picks up where it left off in the program that invoked `main`, which is the Java interpreter. The Java interpreter takes care of things like deleting windows and general cleanup, and *then* the program terminates.

What’s the moral of this sordid tale? When you read a program, don’t read from top to bottom. Instead, follow the flow of execution.

3.8 Parameters and arguments

Some of the built-in methods we have used have **parameters**, which are values that you provide to let the method do its job. For example, if you want to

find the sine of a number, you have to indicate what the number is. Thus, `sin` takes a `double` value as a parameter. To print a string, you have to provide the string, which is why `println` takes a `String` as a parameter.

Some methods take more than one parameter, like `pow`, which takes two `doubles`, the base and the exponent.

Notice that in each of these cases we have to specify not only how many parameters there are, but also what type they are. So it shouldn't surprise you that when you write a class definition, the parameter list indicates the type of each parameter. For example:

```
public static void printTwice (String phil) {  
    System.out.println (phil);  
    System.out.println (phil);  
}
```

This method takes a single parameter, named `phil`, that has type `String`. Whatever that parameter is (and at this point we have no idea what it is), it gets printed twice. I chose the name `phil` to suggest that the name you give a parameter is up to you, but in general you want to choose something more illustrative than `phil`.

In order to invoke this method, we have to provide a `String`. For example, we might have a `main` method like this:

```
public static void main (String[] args) {  
    printTwice ("Don't make me say this twice!");  
}
```

The string you provide is called an **argument**, and we say that the argument is **passed** to the method. In this case we are creating a string value that contains the text "Don't make me say this twice!" and passing that string as an argument to `printTwice` where, contrary to its wishes, it will get printed twice.

Alternatively, if we had a `String` variable, we could use it as an argument instead:

```
public static void main (String[] args) {  
    String argument = "Never say never.";  
    printTwice (argument);  
}
```

Notice something very important here: the name of the variable we pass as an argument (**argument**) has nothing to do with the name of the parameter (`phil`). Let me say that again:

The name of the variable we pass as an argument has nothing to do with the name of the parameter.

They can be the same or they can be different, but it is important to realize that they are not the same thing, except that they happen to have the same value (in this case the string "Never say never.").

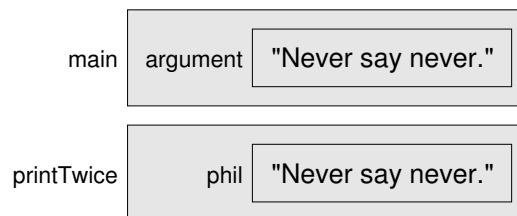
The value you provide as an argument must have the same type as the parameter of the method you invoke. This rule is very important, but it often gets complicated in Java for two reasons:

- There are some methods that can accept arguments with many different types. For example, you can send *any* type to `print` and `println`, and it will do the right thing no matter what. This sort of thing is an exception, though.
- If you violate this rule, the compiler often generates a confusing error message. Instead of saying something like, “You are passing the wrong kind of argument to this method,” it will probably say something to the effect that it could not find a method with that name that would accept an argument with that type. Once you have seen this error message a few times, though, you will figure out how to interpret it.

3.9 Stack diagrams

Parameters and other variables only exist inside their own methods. Within the confines of `main`, there is no such thing as `phil`. If you try to use it, the compiler will complain. Similarly, inside `printTwice` there is no such thing as `argument`.

One way to keep track of where each variable is defined is with a **stack diagram**. The stack diagram for the previous example looks like this:



For each method there is a gray box called a **frame** that contains the methods parameters and local variables. The name of the method appears outside the frame. As usual, the value of each variable is drawn inside a box with the name of the variable beside it.

3.10 Methods with multiple parameters

The syntax for declaring and invoking methods with multiple parameters is a common source of errors. First, remember that you have to declare the type of every parameter. For example

```
public static void printTime (int hour, int minute) {
    System.out.print (hour);
```

```
        System.out.print (":");  
        System.out.println (minute);  
    }
```

It might be tempting to write `int hour, minute`, but that format is only legal for variable declarations, not for parameters.

Another common source of confusion is that you do not have to declare the types of arguments. The following is wrong!

```
    int hour = 11;  
    int minute = 59;  
    printTime (int hour, int minute);    // WRONG!
```

In this case, Java can tell the type of `hour` and `minute` by looking at their declarations. It is unnecessary and illegal to include the type when you pass them as arguments. The correct syntax is `printTime (hour, minute)`.

Exercise 3.1 Draw a stack frame that shows the state of the program when `main` invokes `printTime` with the arguments 11 and 59.

3.11 Methods with results

You might have noticed by now that some of the methods we are using, like the `Math` methods, yield results. Other methods, like `println` and `newLine`, perform some action but they don't return a value. That raises some questions:

- What happens if you invoke a method and you don't do anything with the result (i.e. you don't assign it to a variable or use it as part of a larger expression)?
- What happens if you use a `print` method as part of an expression, like `System.out.println ("boo!") + 7`?
- Can we write methods that yield results, or are we stuck with things like `newLine` and `printTwice`?

The answer to the third question is “yes, you can write methods that return values,” and we'll do it in a couple of chapters. I will leave it up to you to answer the other two questions by trying them out. In fact, any time you have a question about what is legal or illegal in Java, a good way to find out is to ask the compiler.

3.12 Glossary

floating-point: A type of variable (or value) that can contain fractions as well as integers. In Java this type is called `double`.

class: A named collection of methods. So far, we have used the `Math` class and the `System` class, and we have written classes named `Hello` and `NewLine`.

method: A named sequence of statements that performs some useful function. Methods may or may not take parameters, and may or may not produce a result.

parameter: A piece of information you provide in order to invoke a method. Parameters are like variables in the sense that they contain values and have types.

argument: A value that you provide when you invoke a method. This value must have the same type as the corresponding parameter.

invoke: Cause a method to be executed.

3.13 Exercises

Exercise 3.2

The point of this exercise is to practice reading code and to make sure that you understand the flow of execution through a program with multiple methods.

- a. What is the output of the following program? Be precise about where there are spaces and where there are newlines.
HINT: Start by describing in words what **ping** and **baffle** do when they are invoked.
- b. Draw a stack diagram that shows the state of the program the first time **ping** is invoked.

```
public static void zoop () {
    baffle ();
    System.out.print ("You wugga ");
    baffle ();
}

public static void main (String[] args) {
    System.out.print ("No, I ");
    zoop ();
    System.out.print ("I ");
    baffle ();
}

public static void baffle () {
    System.out.print ("wug");
    ping ();
}

public static void ping () {
    System.out.println (".");
}
```

Exercise 3.3 The point of this exercise is to make sure you understand how to write and invoke methods that take parameters.

- a. Write the first line of a method named `zool` that takes three parameters: an `int` and two `Strings`.
- b. Write a line of code that invokes `zool`, passing as arguments the value 11, the name of your first pet, and the name of the street you grew up on.

Exercise 3.4

The purpose of this exercise is to take code from a previous exercise and encapsulate it in a method that takes parameters. You should start with a working solution to Exercise 2.1.

- a. Write a method called `printAmerican` that takes the day, date, month and year as parameters and that prints them in American format.
- b. Test your method by invoking it from `main` and passing appropriate arguments. The output should look something like this (except that the date might be different):
`Wednesday, September 29, 1999`
- c. Once you have debugged `printAmerican`, write another method called `printEuropean` that prints the date in European format.

Exercise 3.5

Many computations can be expressed concisely using the “multadd” operation, which takes three operands and computes `a*b + c`. Some processors even provide a hardware implementation of this operation for floating-point numbers.

- a. Create a new program called `Multadd.java`.
- b. Write a method called `multadd` that takes three `doubles` as parameters and that prints their multadditionization.
- c. Write a `main` method that tests `multadd` by invoking it with a few simple parameters, like 1.0, 2.0, 3.0, and then prints the result, which should be 5.0.
- d. Also in `main`, use `multadd` to compute the following values:

$$\sin \frac{\pi}{4} + \frac{\cos \frac{\pi}{4}}{2}$$

$$\log 10 + \log 20$$

- e. Write a method called `yikes` that takes a double as a parameter and that uses `multadd` to calculate and print

$$xe^{-x} + \sqrt{1 - e^{-x}}$$

HINT: the `Math` method for raising e to a power is `Math.exp`.

In the last part, you get a chance to write a method that invokes a method you wrote. Whenever you do that, it is a good idea to test the first method carefully before you start working on the second. Otherwise, you might find yourself debugging two methods at the same time, which can be very difficult.

One of the purposes of this exercise is to practice pattern-matching: the ability to recognize a specific problem as an instance of a general category of problems.

Chapter 4

Conditionals and recursion

4.1 The modulus operator

The modulus operator works on integers (and integer expressions) and yields the *remainder* when the first operand is divided by the second. In Java, the modulus operator is a percent sign, `%`. The syntax is exactly the same as for other operators:

```
int quotient = 7 / 3;
int remainder = 7 % 3;
```

The first operator, integer division, yields 2. The second operator yields 1. Thus, 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another: if `x % y` is zero, then `x` is divisible by `y`.

Also, you can use the modulus operator to extract the rightmost digit or digits from a number. For example, `x % 10` yields the rightmost digit of `x` (in base 10). Similarly `x % 100` yields the last two digits.

4.2 Conditional execution

In order to write useful programs, we almost always need the ability to check certain conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the **if** statement:

```
if (x > 0) {
    System.out.println ("x is positive");
}
```

The expression in parentheses is called the condition. If it is true, then the statements in brackets get executed. If the condition is not true, nothing happens.

The condition can contain any of the comparison operators, sometimes called **relational operators**:

```
x == y          // x equals y
x != y          // x is not equal to y
x > y           // x is greater than y
x < y           // x is less than y
x >= y          // x is greater than or equal to y
x <= y          // x is less than or equal to y
```

Although these operations are probably familiar to you, the syntax Java uses is a little different from mathematical symbols like $=$, \neq and \leq . A common error is to use a single $=$ instead of a double $==$. Remember that $=$ is the assignment operator, and $==$ is a comparison operator. Also, there is no such thing as $=<$ or $=>$.

The two sides of a condition operator have to be the same type. You can only compare `ints` to `ints` and `doubles` to `doubles`. Unfortunately, at this point you can't compare `Strings` at all! There is a way to compare `Strings`, but we won't get to it for a couple of chapters.

4.3 Alternative execution

A second form of conditional execution is alternative execution, in which there are two possibilities, and the condition determines which one gets executed. The syntax looks like:

```
if (x%2 == 0) {
    System.out.println ("x is even");
} else {
    System.out.println ("x is odd");
}
```

If the remainder when `x` is divided by 2 is zero, then we know that `x` is even, and this code prints a message to that effect. If the condition is false, the second print statement is executed. Since the condition must be true or false, exactly one of the alternatives will be executed.

As an aside, if you think you might want to check the parity (evenness or oddness) of numbers often, you might want to “wrap” this code up in a method, as follows:

```
public static void printParity (int x) {
    if (x%2 == 0) {
        System.out.println ("x is even");
    } else {
        System.out.println ("x is odd");
    }
}
```

Now you have a method named `printParity` that will print an appropriate message for any integer you care to provide. In `main` you would invoke this method as follows:

```
printParity (17);
```

Always remember that when you *invoke* a method, you do not have to declare the types of the arguments you provide. Java can figure out what type they are. You should resist the temptation to write things like:

```
int number = 17;
printParity (int number);           // WRONG!!!
```

4.4 Chained conditionals

Sometimes you want to check for a number of related conditions and choose one of several actions. One way to do this is by **chaining** a series of `ifs` and `elses`:

```
if (x > 0) {
    System.out.println ("x is positive");
} else if (x < 0) {
    System.out.println ("x is negative");
} else {
    System.out.println ("x is zero");
}
```

These chains can be as long as you want, although they can be difficult to read if they get out of hand. One way to make them easier to read is to use standard indentation, as demonstrated in these examples. If you keep all the statements and squiggly-brackets lined up, you are less likely to make syntax errors and you can find them more quickly if you do.

4.5 Nested conditionals

In addition to chaining, you can also nest one conditional within another. We could have written the previous example as:

```
if (x == 0) {
    System.out.println ("x is zero");
} else {
    if (x > 0) {
        System.out.println ("x is positive");
    } else {
        System.out.println ("x is negative");
    }
}
```

There is now an outer conditional that contains two branches. The first branch contains a simple `print` statement, but the second branch contains another conditional statement, which has two branches of its own. Fortunately, those two

branches are both `print` statements, although they could have been conditional statements as well.

Notice again that indentation helps make the structure apparent, but nevertheless, nested conditionals get difficult to read very quickly. In general, it is a good idea to avoid them when you can.

On the other hand, this kind of **nested structure** is common, and we will see it again, so you better get used to it.

4.6 The return statement

The `return` statement allows you to terminate the execution of a method before you reach the end. One reason to use it is if you detect an error condition:

```
public static void printLogarithm (double x) {
    if (x <= 0.0) {
        System.out.println ("Positive numbers only, please.");
        return;
    }

    double result = Math.log (x);
    System.out.println ("The log of x is " + result);
}
```

This defines a method named `printLogarithm` that takes a `double` named `x` as a parameter. The first thing it does is check whether `x` is less than or equal to zero, in which case it prints an error message and then uses `return` to exit the method. The flow of execution immediately returns to the caller and the remaining lines of the method are not executed.

I used a floating-point value on the right side of the condition because there is a floating-point variable on the left.

4.7 Type conversion

You might wonder how you can get away with an expression like `"The log of x is " + result`, since one of the operands is a `String` and the other is a `double`. Well, in this case Java is being smart on our behalf, by automatically converting the `double` to a `String` before it does the string concatenation.

This kind of feature is an example of a common problem in designing a programming language, which is that there is a conflict between *formalism*, which is the requirement that formal languages should have simple rules with few exceptions, and *convenience*, which is the requirement that programming languages be easy to use in practice.

More often than not, convenience wins, which is usually good for expert programmers (who are spared from rigorous but unwieldy formalism), but bad for

beginning programmers, who are often baffled by the complexity of the rules and the number of exceptions. In this book I have tried to simplify things by emphasizing the rules and omitting many of the exceptions.

Nevertheless, it is handy to know that whenever you try to “add” two expressions, if one of them is a **String**, then Java will convert the other to a **String** and then perform string concatenation. What do you think happens if you perform an operation between an integer and a floating-point value?

4.8 Recursion

I mentioned in the last chapter that it is legal for one method to call another, and we have seen several examples of that. I neglected to mention that it is also legal for a method to invoke itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical and interesting things a program can do.

For example, look at the following method:

```
public static void countdown (int n) {  
    if (n == 0) {  
        System.out.println ("Blastoff!");  
    } else {  
        System.out.println (n);  
        countdown (n-1);  
    }  
}
```

The name of the method is **countdown** and it takes a single integer as a parameter. If the parameter is zero, it prints the word “Blastoff.” Otherwise, it prints the number and then invokes a method named **countdown**—itself—passing **n-1** as an argument.

What happens if we invoke this method, in **main**, like this:

```
countdown (3);
```

The execution of **countdown** begins with **n=3**, and since **n** is not zero, it prints the value 3, and then invokes itself..

The execution of **countdown** begins with **n=2**, and since **n** is not zero, it prints the value 2, and then invokes itself..

The execution of **countdown** begins with **n=1**, and since **n** is not zero, it prints the value 1, and then invokes itself..

The execution of **countdown** begins with **n=0**, and since **n** is zero, it prints the word “Blastoff!” and then returns.

The countdown that got **n=1** returns.

The countdown that got **n=2** returns.

The countdown that got `n=3` returns.

And then you're back in `main` (what a trip). So the total output looks like:

```
3
2
1
Blastoff!
```

As a second example, let's look again at the methods `newLine` and `threeLine`.

```
public static void newLine () {
    System.out.println ("");
}

public static void threeLine () {
    newLine (); newLine (); newLine ();
}
```

Although these work, they would not be much help if I wanted to print 2 newlines, or 106. A better alternative would be

```
public static void nLines (int n) {
    if (n > 0) {
        System.out.println ("");
        nLines (n-1);
    }
}
```

This program is very similar; as long as `n` is greater than zero, it prints one newline, and then invokes itself to print `n-1` additional newlines. Thus, the total number of newlines that get printed is $1 + (n-1)$, which usually comes out to roughly `n`.

The process of a method invoking itself is called **recursion**, and such methods are said to be **recursive**.

4.9 Stack diagrams for recursive methods

In the previous chapter we used a stack diagram to represent the state of a program during a method call. The same kind of diagram can make it easier to interpret a recursive method.

Remember that every time a method gets called it creates a new instance of the method that contains a new version of the method's local variables and parameters.

The following figure is a stack diagram for countdown, called with `n = 3`: