Chapter 2

Variables and types

2.1 More printing

You can put as many statements as you want in `main`; for example, to print more than one line:

```
class Hello {

  // Generates 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
  }
}
```

As this example demonstrates, you can 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 characters. 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 moves the cursor to the next line of the

display. The next time `println` is invoked, the new text appears on the next line.

To display the output from multiple print statements all on one line, use `print`:

```
class Hello {

  // Generates some simple output.

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

The output appears on a single line as `Goodbye, cruel world!`. 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, but 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 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, stored and (as we'll see later) operated on. The strings we have been printing (`"Hello, World."`, `"Goodbye, "`, etc.) are values.

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

```
String bob;
```

This statement is a **declaration**, because it declares that the variable named `bob` 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 the `String` type can store strings.

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 guess 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

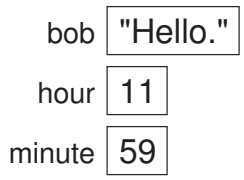
Now that we have created variables, we want to store values. We do that with an **assignment statement**.

```
bob = "Hello.";      // give bob 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:



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 `bob`.

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. For now you should remember the general rule, and we'll talk about exceptions later.

Another source of confusion is that some strings *look* like integers, but they are not. For example, `bob` 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`.

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

2.4 Printing variables

You can print the value of a variable using `println` or `print`:

```
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");`

For example, you can 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 happy 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: To put multiple values on the same line, is common to use several `print` statements followed by a `println`. But you have to remember the `println` at the end. In many environments, the output from `print` is stored without being displayed until `println` is invoked, at which point

the entire line is displayed at once. If you omit `println`, the program may terminate without 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://download.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html. This site, provided by Oracle, includes Java documentation I refer to throughout the book.

Rather than memorize the list, I suggest you take advantage of a feature provided in many Java development environments: code highlighting. As you type, 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 symbols used to represent computations like addition and multiplication. Most operators in Java do what you expect them to do because they are common mathematical symbols. For example, the operator for addition is `+`. Subtraction is `-`, multiplication is `*`, and division is `/`.

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

Expressions can contain both variable names and numbers. Variables are replaced with their values before the computation is performed.

Addition, subtraction and multiplication all do what you expect, but you might be surprised by division. For example, this 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);
```

generates this output:

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

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

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

An alternative 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. To get a more accurate answer, we can use a different type of variable, called floating-point, that can store 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 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 `String`s

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

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

By the way, can you tell by looking at those expressions whether `bob` 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 `String`s, but it might not do what you expect. For `String`s, 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 `bob + "ism"` adds the suffix *ism* to the end of whatever `bob` is, which is 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 combine them in a single statement:

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

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 impressive now, but we will see examples where composition expresses complex computations neatly and concisely.

WARNING: The left side of an assignment 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 a type.

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

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

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 value. Expressions also have types, as determined by their operators and operands.

operator: A symbol that represents a 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 to represent complex computations concisely.

2.11 Exercises

Exercise 2.1. If you are using this book in a class, you might enjoy this exercise: find a partner and play "Stump the Chump":

Start with a program that compiles and runs correctly. One player turns away while the other player adds an error to the program. Then the first player tries to find and fix the error. You get two points if you find the error without compiling the program, one point if you find it using the compiler, and your opponent gets a point if you don't find it.

Exercise 2.2.

1. 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.

2. 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.

3. 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.

4. Modify the program so that it prints the date in standard American form: `Saturday, July 16, 2011`.

5. Modify the program again so that the total output is:

```
American format:
Saturday, July 16, 2011
European format:
Saturday 16 July, 2011
```

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 2.3. 1. 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.

2. 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.

3. Make the program calculate and print the number of seconds since midnight.

4. Make the program calculate and print the number of seconds remaining in the day.

5. Make the program calculate and print the percentage of the day that has passed.

6. 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 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.