Chapter 3

Input and output

The programs we've looked at so far simply display messages, which doesn't involve a lot of real computation. This chapter will show you how to read input from the keyboard, use that input to calculate a result, and then format that result for output.

3.1 The System class

We have been using `System.out.println` for a while, but you might not have thought about what it means. `System` is a class that provides methods related to the "system" or environment where programs run. It also provides `System.out`, which is a special value that provides methods for displaying output, including `println`.

In fact, we can use `System.out.println` to display the value of `System.out`:

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

The result is:

```
java.io.PrintStream@685d72cd
```

This output indicates that `System.out` is a `PrintStream`, which is defined in a package called `java.io`. A **package** is a collection of related classes; `java.io` contains classes for "I/O" which stands for input and output.

The numbers and letters after the `@` sign are the **address** of `System.out`, represented as a hexadecimal (base 16) number. The address of a value is its location in the computer's memory, which might be different on different computers. In this example the address is `685d72cd`, but if you run the same code you might get something different.

As shown in Figure 3.1, `System` is defined in a file called `System.java`, and `PrintStream` is defined in `PrintStream.java`. These files are part of the Java **library**, which is an extensive collection of classes you can use in your programs.

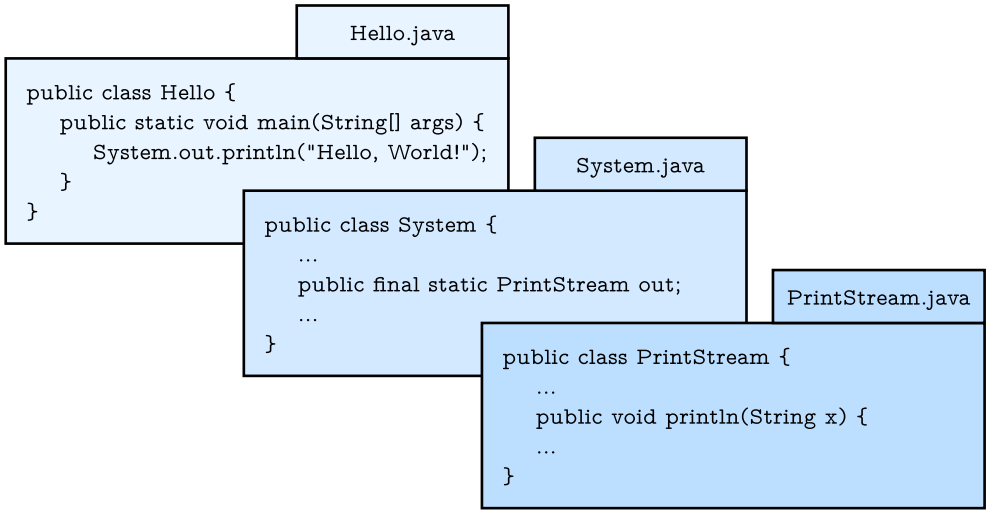

Figure 3.1: `System.out.println` refers to the `out` variable of the `System` class, which is a `PrintStream` that provides a method called `println`.

3.2 The Scanner class

The `System` class also provides the special value `System.in`, which is an `InputStream` that provides methods for reading input from the keyboard. These methods are not easy to use; fortunately, Java provides other classes that make it easier to handle common input tasks.

For example, `Scanner` is a class that provides methods for inputting words, numbers, and other data. `Scanner` is provided by `java.util`, which is a

package that contains classes so useful they are called "utility classes". Before you can use `Scanner`, you have to import it like this:

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

This **import statement** tells the compiler that when you say `Scanner`, you mean the one defined in `java.util`. It's necessary because there might be another class named `Scanner` in another package. Using an import statement makes your code unambiguous.

Import statements can't be inside a class definition. By convention, they are usually at the beginning of the file.

Next you have to create a `Scanner`:

```
Scanner in = new Scanner(System.in);
```

This line declares a `Scanner` variable named `in` and creates a new `Scanner` that takes input from `System.in`.

`Scanner` provides a method called `nextLine` that reads a line of input from the keyboard and returns a `String`. The following example reads two lines and repeats them back to the user:

```
import java.util.Scanner;

public class Echo {

    public static void main(String[] args) {
        String line;
        Scanner in = new Scanner(System.in);

        System.out.print("Type something: ");
        line = in.nextLine();
        System.out.println("You said: " + line);

        System.out.print("Type something else: ");
        line = in.nextLine();
        System.out.println("You also said: " + line);
    }
}
```

If you omit the import statement and later refer to `Scanner`, you will get a compiler error like "cannot find symbol". That means the compiler doesn't know what you mean by `Scanner`.

You might wonder why we can use the `System` class without importing it. `System` belongs to the `java.lang` package, which is imported automatically. According to the documentation, `java.lang` "provides classes that are fundamental to the design of the Java programming language." The `String` class is also part of the `java.lang` package.

3.3 Program structure

At this point, we have seen all of the elements that make up Java programs. Figure 3.2 shows these organizational units.

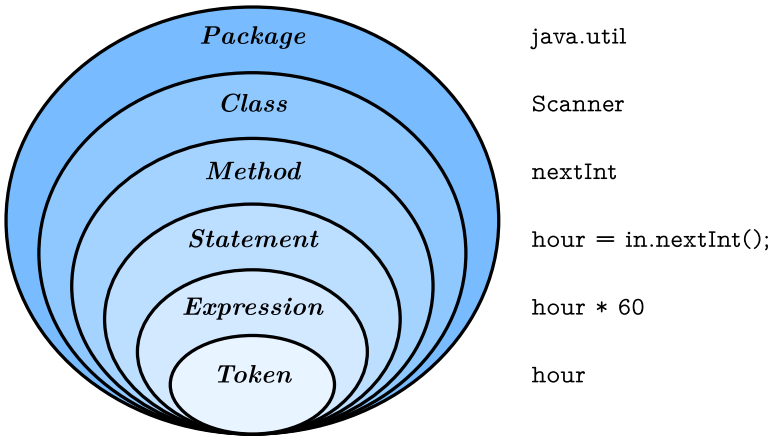

Figure 3.2: Elements of the Java language, from largest to smallest.

To review, a package is a collection of classes, which define methods. Methods contain statements, some of which contain expressions. Expressions are made up of **tokens**, which are the basic elements of a program, including numbers, variable names, operators, keywords, and punctuation like parentheses, braces and semicolons.

The standard edition of Java comes with *several thousand* classes you can `import`, which can be both exciting and intimidating. You can browse this

library at http://docs.oracle.com/javase/8/docs/api/. Most of the Java library itself is written in Java.

Note there is a major difference between the Java *language*, which defines the syntax and meaning of the elements in Figure 3.2, and the Java *library*, which provides the built-in classes.

3.4 Inches to centimeters

Now let's see an example that's a little more useful. Although most of the world has adopted the metric system for weights and measures, some countries are stuck with English units. For example, when talking with friends in Europe about the weather, people in the United States might have to convert from Celsius to Fahrenheit and back. Or they might want to convert height in inches to centimeters.

We can write a program to help. We'll use a `Scanner` to input a measurement in inches, convert to centimeters, and then display the results. The following lines declare the variables and create the `Scanner`:

```
int inch;
double cm;
Scanner in = new Scanner(System.in);
```

The next step is to prompt the user for the input. We'll use `print` instead of `println` so they can enter the input on the same line as the prompt. And we'll use the `Scanner` method `nextInt`, which reads input from the keyboard and converts it to an integer:

```
System.out.print("How many inches? ");
inch = in.nextInt();
```

Next we multiply the number of inches by 2.54, since that's how many centimeters there are per inch, and display the results:

```
cm = inch * 2.54;
System.out.print(inch + " in = ");
System.out.println(cm + " cm");
```

This code works correctly, but it has a minor problem. If another programmer reads this code, they might wonder where 2.54 comes from. For the benefit of others (and yourself in the future), it would be better to assign this value to a variable with a meaningful name. We'll demonstrate in the next section.

3.5 Literals and constants

A value that appears in a program, like 2.54 (or `" in ="`), is called a **literal**. In general, there's nothing wrong with literals. But when numbers like 2.54 appear in an expression with no explanation, they make code hard to read. And if the same value appears many times, and might have to change in the future, it makes code hard to maintain.

Values like that are sometimes called **magic numbers** (with the implication that being "magic" is not a good thing). A good practice is to assign magic numbers to variables with meaningful names, like this:

```
double cmPerInch = 2.54;
cm = inch * cmPerInch;
```

This version is easier to read and less error-prone, but it still has a problem. Variables can vary, but the number of centimeters in an inch does not. Once we assign a value to `cmPerInch`, it should never change. Java provides a language feature that enforces that rule, the keyword `final`.

```
final double CM_PER_INCH = 2.54;
```

Declaring that a variable is `final` means that it cannot be reassigned once it has been initialized. If you try, the compiler reports an error. Variables declared as `final` are called **constants**. By convention, names for constants are all uppercase, with the underscore character (`_`) between words.

3.6 Formatting output

When you output a `double` using `print` or `println`, it displays up to 16 decimal places:

```
System.out.print(4.0 / 3.0);
```

The result is:

```
1.3333333333333333
```

That might be more than you want. `System.out` provides another method, called `printf`, that gives you more control of the format. The "f" in `printf` stands for "formatted". Here's an example:

```
System.out.printf("Four thirds = %.3f", 4.0 / 3.0);
```

The first value in the parentheses is a **format string** that specifies how the output should be displayed. This format string contains ordinary text followed by a **format specifier**, which is a special sequence that starts with a percent sign. The format specifier `%.3f` indicates that the following value should be displayed as floating-point, rounded to three decimal places. The result is:

```
Four thirds = 1.333
```

The format string can contain any number of format specifiers; here's an example with two:

```
int inch = 100;
double cm = inch * CM_PER_INCH;
System.out.printf("%d in = %f cm\n", inch, cm);
```

The result is:

```
100 in = 254.000000 cm
```

Like `print`, `printf` does not append a newline. So format strings often end with a newline character.

The format specifier `%d` displays integer values ("d" stands for "decimal"). The values are matched up with the format specifiers in order, so `inch` is displayed using `%d`, and `cm` is displayed using `%f`.

Learning about format strings is like learning a sub-language within Java. There are many options, and the details can be overwhelming. Table 3.1 lists a few common uses, to give you an idea of how things work. For more details, refer to the documentation of `java.util.Formatter`. The easiest way to find documentation for Java classes is to do a web search for "Java" and the name of the class.

`%d`	decimal integer	12345
`%08d`	padded with zeros, at least 8 digits wide	00012345
`%f`	floating-point	6.789000
`%.2f`	rounded to 2 decimal places	6.79

Table 3.1: Example format specifiers

3.7 Centimeters to inches

Now suppose we have a measurement in centimeters, and we want to round it off to the nearest inch. It is tempting to write:

```
inch = cm / CM_PER_INCH;  // syntax error
```

But the result is an error – you get something like, "Bad types in assignment: from double to int." The problem is that the value on the right is floating-point, and the variable on the left is an integer.

The simplest way to convert a floating-point value to an integer is to use a **type cast**, so called because it molds or "casts" a value from one type to another. The syntax for type casting is to put the name of the type in parentheses and use it as an operator.

```
double pi = 3.14159;
int x = (int) pi;
```

The `(int)` operator has the effect of converting what follows into an integer. In this example, `x` gets the value `3`. Like integer division, converting to an integer always rounds toward zero, even if the fraction part is `0.999999` (or `-0.999999`). In other words, it simply throws away the fractional part.

Type casting takes precedence over arithmetic operations. In this example, the value of `pi` gets converted to an integer before the multiplication. So the result is 60.0, not 62.0.

```
double pi = 3.14159;
double x = (int) pi * 20.0;
```

Keeping that in mind, here's how we can convert a measurement in centimeters to inches:

```
inch = (int) (cm / CM_PER_INCH);
System.out.printf("%f cm = %d in\n", cent, inch);
```

The parentheses after the cast operator require the division to happen before the type cast. And the result is rounded toward zero; we will see in the next chapter how to round floating-point numbers to the closest integer.

3.8 Modulus operator

Let's take the example one step further: suppose you have a measurement in inches and you want to convert to feet and inches. The goal is divide by 12 (the number of inches in a foot) and keep the remainder.

We have already seen the division operator (`/`), which computes the quotient of two numbers. If the numbers are integers, it performs integer division. Java also provides the **modulus** operator (`%`), which divides two numbers and computes the remainder.

Using division and modulus, we can convert to feet and inches like this:

```
quotient = 76 / 12;   // division
remainder = 76 % 12;  // modulus
```

The first line yields 6. The second line, which is pronounced "76 mod 12", yields 4. So 76 inches is 6 feet, 4 inches.

The modulus operator looks like a percent sign, but you might find it helpful to think of it as a division sign (÷) rotated to the left.

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`. You can use modulus to "extract" digits from a number: `x % 10` yields the rightmost digit of `x`, and `x % 100` yields the last two digits. Also, many encryption algorithms use the modulus operator extensively.

3.9 Putting it all together

At this point, you have seen enough Java to write useful programs that solve everyday problems. You can (1) import Java library classes, (2) create a

`Scanner`, (3) get input from the keyboard, (4) format output with `printf`, and (5) divide and mod integers. Now we will put everything together in a complete program:

```
import java.util.Scanner;

/**
 * Converts centimeters to feet and inches.
 */
public class Convert {

    public static void main(String[] args) {
        double cm;
        int feet, inches, remainder;
        final double CM_PER_INCH = 2.54;
        final int IN_PER_FOOT = 12;
        Scanner in = new Scanner(System.in);

        // prompt the user and get the value
        System.out.print("Exactly how many cm? ");
        cm = in.nextDouble();

        // convert and output the result
        inches = (int) (cm / CM_PER_INCH);
        feet = inches / IN_PER_FOOT;
        remainder = inches % IN_PER_FOOT;
        System.out.printf("%.2f cm = %d ft, %d in\n",
                          cm, feet, remainder);
    }
}
```

Although not required, all variables and constants are declared at the top of `main`. This practice makes it easier to find their types later on, and it helps the reader know what data is involved in the algorithm.

For readability, each major step of the algorithm is separated by a blank line and begins with a comment. It also includes a documentation comment (`/**`), which we'll learn more about in the next chapter.

Many algorithms, including the `Convert` program, perform division and modulus together. In both steps, you divide by the same number (`IN_PER_FOOT`).

When statements get long (generally wider than 80 characters), a common style convention is to break them across multiple lines. The reader should never have to scroll horizontally.

3.10 The Scanner bug

Now that you've had some experience with `Scanner`, there is an unexpected behavior we want to warn you about. The following code fragment asks users for their name and age:

```
System.out.print("What is your name? ");
name = in.nextLine();
System.out.print("What is your age? ");
age = in.nextInt();
System.out.printf("Hello %s, age %d\n", name, age);
```

The output might look something like this:

```
Hello Grace Hopper, age 45
```

When you read a `String` followed by an `int`, everything works just fine. But when you read an `int` followed by a `String`, something strange happens.

```
System.out.print("What is your age? ");
age = in.nextInt();
System.out.print("What is your name? ");
name = in.nextLine();
System.out.printf("Hello %s, age %d\n", name, age);
```

Try running this example code. It doesn't let you input your name, and it immediately displays the output:

```
What is your name? Hello , age 45
```

To understand what is happening, you have to understand that the `Scanner` doesn't see input as multiple lines, like we do. Instead, it gets a "stream of characters" as shown in Figure 3.3.

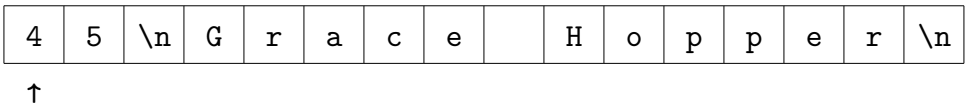

Figure 3.3: A stream of characters as seen by a `Scanner`.

The arrow indicates the next character to be read by `Scanner`. When you call `nextInt`, it reads characters until it gets to a non-digit. Figure 3.4 shows the state of the stream after `nextInt` is invoked.

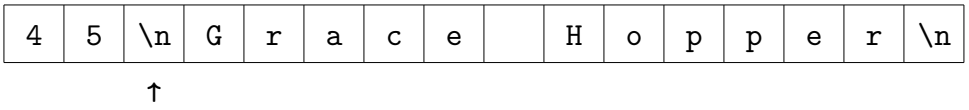

Figure 3.4: A stream of characters after `nextInt` is invoked.

At this point, `nextInt` returns `45`. The program then displays the prompt `"What is your name? "` and calls `nextLine`, which reads characters until it gets to a newline. But since the next character is already a newline, `nextLine` returns the empty string `""`.

To solve this problem, you need an extra `nextLine` after `nextInt`.

```
System.out.print("What is your age? ");
age = in.nextInt();
in.nextLine();  // read the newline
System.out.print("What is your name? ");
name = in.nextLine();
System.out.printf("Hello %s, age %d\n", name, age);
```

This technique is common when reading `int` or `double` values that appear on their own line. First you read the number, and then you read the rest of the line, which is just a newline character.

3.11 Vocabulary

package: A group of classes that are related to each other.

address: The location of a value in computer memory, often represented as a hexadecimal integer.

library: A collection of packages and classes that are available for use in other programs.

import statement: A statement that allows programs to use classes defined in other packages.

token: A basic element of a program, such as a word, space, symbol, or number.

literal: A value that appears in source code. For example, `"Hello"` is a string literal and `74` is an integer literal.

magic number: A number that appears without explanation as part of an expression. It should generally be replaced with a constant.

constant: A variable, declared `final`, whose value cannot be changed.

format string: A string passed to `printf` to specify the format of the output.

format specifier: A special code that begins with a percent sign and specifies the data type and format of the corresponding value.

type cast: An operation that explicitly converts one data type into another. In Java it appears as a type name in parentheses, like `(int)`.

modulus: An operator that yields the remainder when one integer is divided by another. In Java, it is denoted with a percent sign; for example, `5 % 2` is `1`.

3.12 Exercises

The code for this chapter is in the `ch03` directory of `ThinkJavaCode`. See page xv for instructions on how to download the repository. Before you start the exercises, we recommend that you compile and run the examples.

If you have not already read Appendix A.3, now might be a good time. It describes the command-line interface, which is a powerful and efficient way to interact with your computer.

Exercise 3.1 When you use `printf`, the Java compiler does not check your format string. See what happens if you try to display a value with type `int` using `%f`. And what happens if you display a `double` using `%d`? What if you use two format specifiers, but then only provide one value?

Exercise 3.2 Write a program that converts a temperature from Celsius to Fahrenheit. It should (1) prompt the user for input, (2) read a `double` value from the keyboard, (3) calculate the result, and (4) format the output to one decimal place. For example, it should display `"24.0 C = 75.2 F"`.

Here is the formula. Be careful not to use integer division!

$$F = C \times \frac{9}{5} + 32$$

Exercise 3.3 Write a program that converts a total number of seconds to hours, minutes, and seconds. It should (1) prompt the user for input, (2) read an integer from the keyboard, (3) calculate the result, and (4) use `printf` to display the output. For example, `"5000 seconds = 1 hours, 23 minutes, and 20 seconds"`.

Hint: Use the modulus operator.

Exercise 3.4 The goal of this exercise is to program a "Guess My Number" game. When it's finished, it will work like this:

```
I'm thinking of a number between 1 and 100
(including both). Can you guess what it is?
Type a number: 45
Your guess is: 45
The number I was thinking of is: 14
You were off by: 31
```

To choose a random number, you can use the `Random` class in `java.util`. Here's how it works:

```
import java.util.Random;

public class GuessStarter {

    public static void main(String[] args) {
        // pick a random number
        Random random = new Random();
        int number = random.nextInt(100) + 1;
        System.out.println(number);
    }
}
```

Like the `Scanner` class we saw in this chapter, `Random` has to be imported before we can use it. And as we saw with `Scanner`, we have to use the `new` operator to create a `Random` (number generator).

Then we can use the method `nextInt` to generate a random number. In this example, the result of `nextInt(100)` will be between 0 and 99, including both. Adding 1 yields a number between 1 and 100, including both.

1. The definition of `GuessStarter` is in a file called `GuessStarter.java`, in the directory called `ch03`, in the repository for this book.

2. Compile and run this program.

3. Modify the program to prompt the user, then use a `Scanner` to read a line of user input. Compile and test the program.

4. Read the user input as an integer and display the result. Again, compile and test.

5. Compute and display the difference between the user's guess and the number that was generated.