Chapter 4

Void methods

So far we've only written short programs that have a single class and a single method (`main`). In this chapter, we'll show you how to organize longer programs into multiple methods and classes. We'll also present the `Math` class, which provides methods for common mathematical operations.

4.1 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. Then you can evaluate the function itself, maybe by punching it into a calculator.

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

The Java library includes a `Math` class that provides common mathematical operations. `Math` is in the `java.lang` package, so you don't have to import it. You can use, or **invoke**, `Math` methods like this:

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

The first line sets `root` to the square root of 17. The third line finds the sine of 1.5 (the value of `angle`).

Arguments of the trigonometric functions – `sin`, `cos`, and `tan` – should be in *radians*. To convert from degrees to radians, you can divide by 180 and multiply by π. Conveniently, the `Math` class provides a constant double named `PI` that contains an approximation of π:

```
double degrees = 90;
double angle = degrees / 180.0 * Math.PI;
```

Notice that `PI` is in capital letters. Java does not recognize `Pi`, `pi`, or `pie`. Also, `PI` is the name of a variable, not a method, so it doesn't have parentheses. The same is true for the constant `Math.E`, which approximates Euler's number.

Converting to and from radians is a common operation, so the `Math` class provides methods that do it for you.

```
double radians = Math.toRadians(180.0);
double degrees = Math.toDegrees(Math.PI);
```

Another useful method is `round`, which rounds a floating-point value to the nearest integer and returns a `long`. A `long` is like an `int`, but bigger. More specifically, an `int` uses 32 bits; the largest value it can hold is $2^{31} - 1$, which is about 2 billion. A `long` uses 64 bits, so the largest value is $2^{63} - 1$, which is about 9 quintillion.

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

The result is 63 (rounded up from 62.8319).

Take a minute to read the documentation for these and other methods in the `Math` class. The easiest way to find documentation for Java classes is to do a web search for "Java" and the name of the class.

4.2 Composition revisited

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

```java
double x = Math.cos(angle + Math.PI / 2.0);
```

This statement divides `Math.PI` by two, adds the result to `angle`, and computes the cosine of the sum. You can also take the result of one method and pass it as an argument to another:

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

In Java, the `log` method 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`.

Some math methods take more than one argument. For example, `Math.pow` takes two arguments and raises the first to the power of the second. This line of code assigns the value `1024.0` to the variable `x`:

```java
double x = Math.pow(2.0, 10.0);
```

When using `Math` methods, it is a common error to forget the `Math`. For example, if you try to invoke `pow(2.0, 10.0)`, you get an error message like:

```
File: Test.java  [line: 5]
Error: cannot find symbol
  symbol:   method pow(double,double)
  location: class Test
```

The message "cannot find symbol" is confusing, but the last line provides a useful hint. The compiler is looking for `pow` in the same class where it is used, which is `Test`. If you don't specify a class name, the compiler looks in the current class.

4.3 Adding new methods

You have probably guessed by now that you can define more than one method in a class. Here's an example:

```
public class NewLine {

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

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

The name of the class is `NewLine`. By convention, class names begin with a capital letter. `NewLine` contains two methods, `newLine` and `main`. Remember that Java is case-sensitive, so `NewLine` and `newLine` are not the same.

Method names should begin with a lowercase letter and use “camel case”, which is a cute name for `jammingWordsTogetherLikeThis`. You can use any name you want for methods, except `main` or any of the Java keywords.

`newLine` and `main` are `public`, which means they can be invoked from other classes. They are both `static`, but we can't explain what that means yet. And they are both `void`, which means that they don't yield a result (unlike the `Math` methods, for example).

The parentheses after the method name contain a list of variables, called **parameters**, where the method stores its arguments. `main` has a single parameter, called `args`, which has type `String[]`. That means that whoever invokes `main` must provide an array of strings (we'll get to arrays in a later chapter).

Since `newLine` has no parameters, it requires no arguments, as shown when it is invoked in `main`. And because `newLine` is in the same class as `main`, we don't have to specify the class name.

The output of this program is:

```
First line.

Second line.
```

Notice the extra space between the lines. If we wanted more space between them, we could invoke the same method repeatedly:

```java
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 that displays three blank lines:

```java
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 can invoke the same method more than once, and you can have one method invoke another. In this example, `main` invokes `threeLine`, and `threeLine` invokes `newLine`.

Beginners often wonder why it is worth the trouble to create new methods. There are many reasons, but this example demonstrates a few of them:

- Creating a new method gives you an opportunity to give a name to a group of statements, which makes code easier to read and understand.
- Introducing new methods can make a program smaller by eliminating repetitive code. For example, to display nine consecutive new lines, you could invoke `threeLine` three times.
- A common problem solving technique is to break tasks down into sub-problems. Methods allow you to focus on each sub-problem in isolation, and then compose them into a complete solution.

4.4 Flow of execution

Pulling together the code from the previous section, the complete program looks like this:

```
public 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.");
    }
}
```

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 **flow of execution** of the program.

Execution always begins at the first statement of `main`, regardless of where it is in the source file. Statements are executed one at a time, in order, until you reach a method invocation, which you can think of as a detour. Instead of going to the next statement, you jump to the first line of the invoked method, execute all the statements there, and then come back and pick up exactly where you left off.

That sounds simple enough, but remember that one method can invoke another one. In the middle of `main`, we go off to execute the statements in `threeLine`. While we are executing `threeLine`, we go off to execute `newLine`. Then `newLine` invokes `println`, which causes yet another detour.

Fortunately, Java is good at keeping track of which methods are running. So when `println` completes, it picks up where it left off in `newLine`; when `newLine` completes, it goes back to `threeLine`, and when `threeLine` completes, it gets back to `main`.

In summary, when you read a program, don't read from top to bottom. Instead, follow the flow of execution.

4.5 Parameters and arguments

Some of the methods we have used require arguments, which are the values you provide when you invoke the method. For example, to find the sine of a number, you have to provide the number, so `sin` takes a `double` as an argument. To display a message, you have to provide the message, so `println` takes a `String`.

When you use a method, you provide the arguments. When you write a method, you name the parameters. The parameter list indicates what arguments are required. The following class shows an example:

```java
public class PrintTwice {

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

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

`printTwice` has a parameter named `s` with type `String`. When we invoke `printTwice`, we have to provide an argument with type `String`.

Before the method executes, the argument gets assigned to the parameter. In this example, the argument `"Don't make me say this twice!"` gets assigned to the parameter `s`.

This process is called **parameter passing** because the value gets passed from outside the method to the inside. An argument can be any kind of expression, so if you have a `String` variable, you can use it as an argument:

```
String argument = "Never say never.";
printTwice(argument);
```

The value you provide as an argument must have the same type as the parameter. For example, if you try:

```
printTwice(17);  // syntax error
```

You will get an error message like this:

```
File: Test.java  [line: 10]
Error: method printTwice in class Test cannot be applied
       to given types;
  required: java.lang.String
  found: int
  reason: actual argument int cannot be converted to
          java.lang.String by method invocation conversion
```

Sometimes Java can convert an argument from one type to another automatically. For example, `Math.sqrt` requires a `double`, but if you invoke `Math.sqrt(25)`, the integer value 25 is automatically converted to the floating-point value `25.0`. But in the case of `printTwice`, Java can't (or won't) convert the integer `17` to a `String`.

Parameters and other variables only exist inside their own methods. Inside `main`, there is no such thing as `s`. If you try to use it there, you'll get a compiler error. Similarly, inside `printTwice` there is no such thing as `argument`. That variable belongs to `main`.

Because variables only exist inside the methods where they are defined, they are often called **local variables**.

4.6 Multiple parameters

Here is an example of a method that takes two parameters:

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

In the parameter list, it may be tempting to write:

```
public static void printTime(int hour, minute) {
    ...
```

But that format (without the second `int`) is only legal for variable declarations. In parameter lists, you need to specify the type of each variable separately.

To invoke this method, we have to provide two integers as arguments:

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

A common error is to declare the types of the arguments, like this:

```
int hour = 11;
int minute = 59;
printTime(int hour, int minute);  // syntax error
```

That's a syntax error; the compiler sees `int hour` and `int minute` as variable declarations, not expressions. You wouldn't declare the types of the arguments if they were simply integers:

```
printTime(int 11, int 59);  // syntax error
```

4.7 Stack diagrams

Pulling together the code fragments from the previous section, here is a complete class definition:

```java
public class PrintTime {

    public static void printTime(int hour, int minute) {
        System.out.print(hour);
        System.out.print(":");
        System.out.println(minute);
    }

    public static void main(String[] args) {
        int hour = 11;
        int minute = 59;
        printTime(hour, minute);
    }
}
```

`printTime` has two parameters, named `hour` and `minute`. And `main` has two variables, also named `hour` and `minute`. Although they have the same names, these variables are not the same. `hour` in `printTime` and `hour` in `main` refer to different storage locations, and they can have different values.

For example, you could invoke `printTime` like this:

```java
int hour = 11;
int minute = 59;
printTime(hour + 1, 0);
```

Before the method is invoked, Java evaluates the arguments; in this example, the results are `12` and `0`. Then it assigns those values to the parameters. Inside `printTime`, the value of `hour` is `12`, not `11`, and the value of `minute` is `0`, not `59`. Furthermore, if `printTime` modifies one of its parameters, that change has no effect on the variables in `main`.

One way to keep track of everything is to draw a **stack diagram**, which is a state diagram (see Section 2.3) that shows method invocations. For each method there is a box called a **frame** that contains the method's parameters and variables. The name of the method appears outside the frame; the variables and parameters appear inside.

As with state diagrams, stack diagrams show variables and methods at a particular point in time. Figure 4.1 is a stack diagram at the beginning of the `printTime` method.

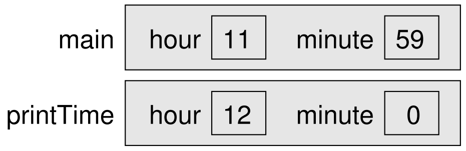
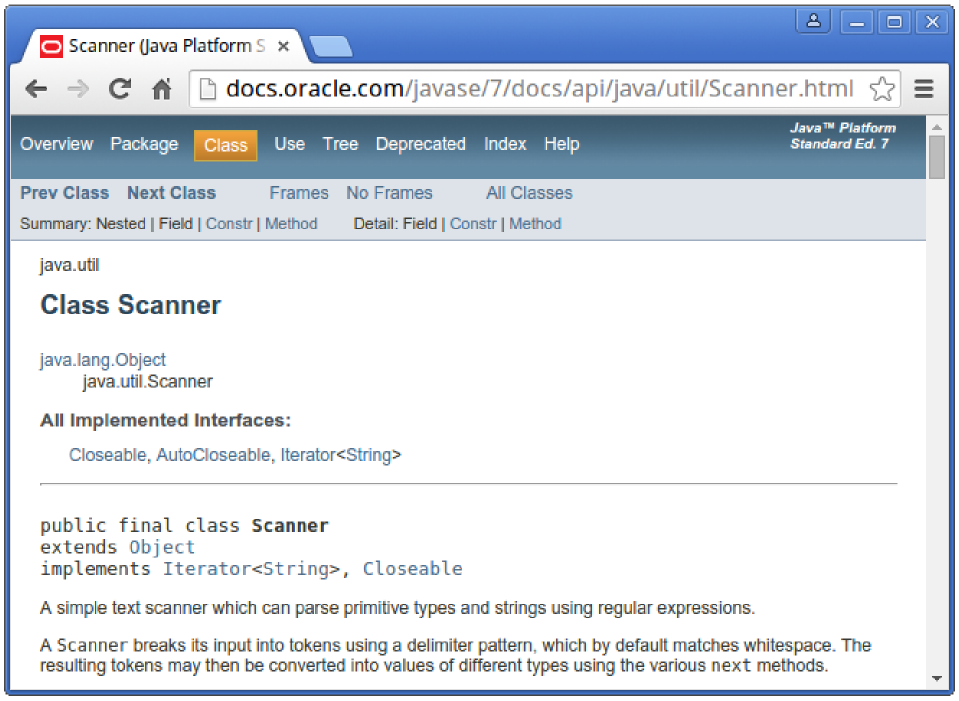


Figure 4.1: Stack diagram for `PrintTime`.

4.8 Reading documentation

One of the nice things about Java is that it comes with an extensive library of classes and methods. But before you use them, you might have to read the documentation. And sometimes that's not easy.

As an example, let's look at the documentation for `Scanner`, which we used in Section 3.2. You can find it by doing a web search for "Java Scanner". Figure 4.2 shows a screenshot of the page.


Scanner (Java Platform S
docs.oracle.com/javase/7/docs/api/java/util/Scanner.html
Overview Package Class Use Tree Deprecated Index Help
Java™ Platform Standard Ed. 7
Prev Class Next Class Frames No Frames All Classes
Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

Class Scanner

java.lang.Object
    java.util.Scanner

All Implemented Interfaces:

Closeable, AutoCloseable, Iterator<String>

```
public final class Scanner
extends Object
implements Iterator<String>, Closeable
```

A simple text scanner which can parse primitive types and strings using regular expressions.

A `Scanner` breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various `next` methods.

Figure 4.2: Screenshot of the documentation for `Scanner`.

Documentation for other classes uses a similar format. The first line is the package that contains the class, such as `java.util`. The second line is the name of the class. The "Implemented Interfaces" section lists some of the things a `Scanner` can do; we won't say more about that for now.

The next section of the documentation is a narrative that explains the purpose of the class and includes examples of how to use it. This text can be difficult to read because it uses terms we have not learned yet. But the examples are often very useful. A good way to get started with a new class is to paste the examples into a test file and see if you can compile and run them.

One of the examples shows how you can use a `Scanner` to read input from a `String` instead of `System.in`:

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input);
```

After the narrative, code examples, and some other details, you will find the following tables:

Constructor summary: Ways of creating, or "constructing", a `Scanner`.

Method summary: The list of methods that `Scanner` provides.

Constructor detail: More information about how to create a `Scanner`.

Method detail: More information about each method.

As an example, here is the summary information for `nextInt`:

```
public int nextInt()
Scans the next token of the input as an int.
```

The first line is the method's **signature**, which specifies the name of the method, its parameters (none), and what type it returns (`int`). The next line is a short description of what it does.

The "Method detail" explains more:

```
public int nextInt()
Scans the next token of the input as an int.

An invocation of this method of the form nextInt() behaves in
exactly the same way as the invocation nextInt(radix), where
radix is the default radix of this scanner.

Returns:
the int scanned from the input

Throws:
InputMismatchException - if the next token does not match
    the Integer regular expression, or is out of range
NoSuchElementException - if input is exhausted
IllegalStateException - if this scanner is closed
```

The "Returns" section describes the result when the method succeeds. In contrast, the "Throws" section describes possible errors and their resulting exceptions. Exceptions are said to be "thrown", like a referee throwing a flag, or like a toddler throwing a fit.

It might take you some time to get comfortable reading documentation and learning which parts to ignore. But it's worth the effort. Knowing what's available in the library helps you avoid reinventing the wheel. And a little bit of documentation can save you a lot of debugging.

4.9 Writing documentation

As you benefit from reading good documentation, you should "pay it forward" by writing good documentation. A nice feature of the Java language is the ability to embed documentation in your source code. That way, you can write it as you go, and as things change, it is easier to keep the documentation consistent with the code.

If you include documentation in your source code, you can extract it automatically, and generate well-formatted HTML, using a tool called **Javadoc**. This tool is included in standard Java development environments, and it is widely used. In fact, the online documentation of the Java libraries is generated by Javadoc.

Javadoc scans your source files looking for specially-formatted **documentation comments**, also known as “Javadoc comments”. They begin with `/**` (two stars) and end with `*/` (one star). Anything in between is considered part of the documentation.

Here’s a class definition with two Javadoc comments, one for the class and one for the `main` method:

```java
/**
 * Example program that demonstrates print vs println.
 */
public class Goodbye {

    /**
     * Prints a greeting.
     */
    public static void main(String[] args) {
        System.out.print("Goodbye, ");  // note the space
        System.out.println("cruel world");
    }
}
```

The class comment explains the purpose of the class. The method comment explains what the method does.

Notice that this example also includes an inline comment, beginning with `//`. In general, inline comments are short phrases that help explain complex parts of a program. They are intended for other programmers reading and maintaining the source code.

In contrast, Javadoc comments are longer, usually complete sentences. They explain what each method does, but they omit details about how the method works. And they are intended for people who will use the methods without looking at the source code.

Appropriate comments and documentation are essential for making source code readable. And remember that the person most likely to read your code in the future, and appreciate good documentation, is you.

4.10 Vocabulary

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

invoke: To cause a method to execute. Also known as "calling" a method.

parameter: A piece of information that a method requires before it can run. Parameters are variables: they contain values and have types.

flow of execution: The order in which Java executes methods and statements. It may not necessarily be from top to bottom, left to right.

parameter passing: The process of assigning an argument value to a parameter variable.

local variable: A variable declared inside a method. Local variables cannot be accessed from outside their method.

stack diagram: A graphical representation of the variables belonging to each method. The method calls are "stacked" from top to bottom, in the flow of execution.

frame: In a stack diagram, a representation of the variables and parameters for a method, along with their current values.

signature: The first line of a method that defines its name, return type, and parameters.

Javadoc: A tool that reads Java source code and generates documentation in HTML format.

documentation: Comments that describe the technical operation of a class or method.

4.11 Exercises

The code for this chapter is in the `ch04` 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.4, now might be a good time. It describes an efficient way to test programs that take input from the user and display specific output.

Exercise 4.1 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.

1. 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.

2. Draw a stack diagram that shows the state of the program the first time `ping` is invoked.

3. What happens if you invoke `baffle();` at the end of the `ping` method? (We will see why in the next chapter.)

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

1. Write the first line of a method named `zool` that takes three parameters: an `int` and two `String`s.
2. Write a line of code that calls `zool`, passing as arguments the value `11`, the name of your first pet, and the name of the street you grew up on.

Exercise 4.3 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 2.2.

1. Write a method called `printAmerican` that takes the day, date, month and year as parameters and that displays them in American format.
2. 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):

   ```
   Saturday, July 22, 2015
   ```
3. Once you have debugged `printAmerican`, write another method called `printEuropean` that displays the date in European format.