

Think Java

How to Think Like a Computer Scientist

Allen Downey and Chris Mayfield

Version 6.0 Draft – August 3, 2015

Copyright © 2016 Allen Downey and Chris Mayfield.

NOTE: This version of the book is a work in progress and won't be completed until February 2016.

Permission is granted to copy, distribute, transmit, and adapt this work under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License: <http://creativecommons.org/licenses/by-nc-sa/3.0/>

The original form of this book is L^AT_EX source code. Compiling the L^AT_EX source has the effect of generating a device-independent representation of the book, which can be converted to other formats and printed.

The L^AT_EX source for this book is available from <http://thinkjava.org>.

Contents

1	The way of the program	1
1.1	What is programming?	1
1.2	What is computer science?	2
1.3	Introduction to Java	3
1.4	Formal languages	5
1.5	The hello world program	7
1.6	Getting started with DrJava	9
1.7	More printing	11
1.8	Working through examples	13
1.9	Vocabulary	14
1.10	Exercises	17
2	Variables and arithmetic	19
2.1	Types of errors	19
2.2	Creating variables	22
2.3	Printing variables	24
2.4	Arithmetic operators	25
2.5	Floating-point numbers	27
2.6	Operators for strings	28
2.7	Order of operations	29

2.8	Composition	29
2.9	Formatting and style	30
2.10	Vocabulary	31
2.11	Exercises	32
3	Input and output	35
3.1	The System class	35
3.2	The Scanner class	36
3.3	Inches to centimeters	39
3.4	Formatting output	40
3.5	Centimeters to inches	42
3.6	Putting it all together	43
3.7	Reading documentation	45
3.8	Writing documentation	46
3.9	Command-line testing	47
3.10	Vocabulary	49
3.11	Exercises	50
4	Void methods	51
4.1	Math methods	51
4.2	Adding new methods	53
4.3	Classes and methods	56
4.4	Parameters and arguments	57
4.5	Stack diagrams and scope	58
4.6	Tracing with a debugger	60
4.7	Multiple parameters	61
4.8	A few more details	62
4.9	Vocabulary	65
4.10	Exercises	66

Chapter 1

The way of the program

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating trade-offs among alternatives. And like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem-solving**. It involves the ability to formulate problems, think creatively about solutions, and express solutions clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to develop problem-solving skills. That's why this chapter is called, "The way of the program."

On one level you will be learning to program, a useful skill by itself. But on another level you will use programming as a means to an end. As we go along, that end will become clearer. Learning how to think in terms of computation is much more valuable than simply learning how to write code.

1.1 What is programming?

A **program** is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, like solving

a system of equations or finding the roots of a polynomial. It can also be a symbolic computation, like searching and replacing text in a document or (strangely enough) compiling a program. The details look different in different languages, but a few basic instructions appear in just about every language.

input: Get data from the keyboard, a file, a sensor, or some other device.

output: Display data on the screen or send data to a file or other device.

math: Perform basic mathematical operations like addition and division.

decisions: Check for certain conditions and execute the appropriate code.

repetition: Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look much like these. So you can think of **programming** as the process of breaking down a large, complex task into smaller and smaller subtasks. The process continues until the subtasks are simple enough to be performed with the basic instructions provided by computer hardware.

1.2 What is computer science?

One of the most interesting aspects of writing programs is deciding how to solve a particular problem, especially when there are multiple solutions. For example, there are numerous ways to sort a list of numbers, and each way has its advantages (see e.g., <http://www.sorting-algorithms.com/>). In order to determine which way is best for a given situation, we need techniques for describing and analyzing solutions formally. That is where computer science comes in.

Put simply, **computer science** is the science of algorithms, including their discovery and analysis. An **algorithm** is a sequence of steps that specify exactly how to solve a problem. Some algorithms are better than others in terms of how long they take or how much memory they use. As you learn to develop algorithms for problems you haven't solved before, you also learn to think like a computer scientist.

Designing algorithms and writing code is difficult and error-prone. For historical reasons, programming errors are called **bugs**, and the process of tracking them down and correcting them is called **debugging**. As you learn to debug your programs, you will develop new problem-solving skills. You will need to think creatively when unexpected errors happen.

Although it can be frustrating, debugging is an intellectually rich, challenging, and interesting part of computer programming. In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see. Thinking about how to correct programs and improve their performance sometimes even leads to the discovery of new algorithms.

1.3 Introduction to Java

The programming language you will learn is Java, which is relatively new (Sun released the first version in May 1995). Java is an example of a **high-level language**. Other high-level languages you may have heard of include C and C++, JavaScript, Python, Ruby, and Visual Basic.

There are also **low-level languages**, sometimes referred to as “machine languages” or “assembly languages.” Loosely speaking, computers can only run programs written in low-level languages. So programs written in a high-level language have to be translated before they can run. This translation takes some time, which is a small disadvantage of high-level languages. But the advantages of high-level languages are enormous. As a result, low-level languages are only used for programs that need to interact directly with hardware.

Due to the advantages, almost all programs are written in high-level languages. First, it is *much* easier to program in a high-level language. Programs take less time to write, are shorter and easier to read, and are more likely to be correct. Second, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can only run on one kind of computer, and have to be rewritten to run on another.

Two kinds of programs translate high-level languages into low-level languages: interpreters and compilers. An **interpreter** reads a high-level program and

executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.

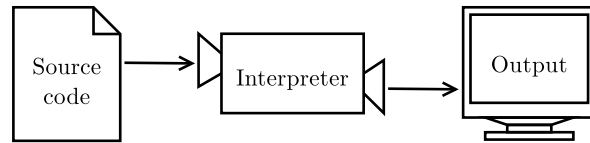


Figure 1.1: How interpreted languages like Python and Ruby are executed.

In contrast, a **compiler** reads the entire program and translates it completely before the program starts running. In this context, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**. Once a program is compiled, you can execute it repeatedly without further translation. As a result, compiled programs often run faster than interpreted programs.

Java is *both* compiled and interpreted. Instead of translating programs directly into machine language, the Java compiler generates **byte code**. Similar to machine language, byte code is easy (and fast) to interpret. But it is also portable, like a high-level language. Thus it is possible to compile a Java program on one machine, transfer the byte code to another machine, and then execute (interpret) the byte code on the other machine.

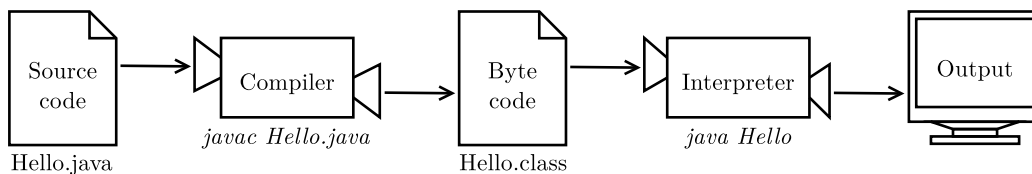


Figure 1.2: The process of editing, compiling, and running a Java program.

Although this process may seem complicated, in most program development environments these steps are automated for you. Usually you will only have to write a program and press a button or type a single command to compile and run it. On the other hand, it is important to know what steps are happening in the background, so if something goes wrong you can figure out what it is.

1.4 Formal languages

Learning a programming language is very different from learning a **natural language** such as English, Spanish, or German. The languages that people speak evolved naturally over time. They were not designed by people, although we try to impose order on them for practical reasons.

In contrast, **formal languages** are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

Programming languages are formal languages that have been designed to express computations.

Formal languages have strict rules about both the **syntax** (structure) and the **semantics** (meaning) of statements. For example, $3 + 3 = 6$ is a syntactically correct mathematical statement, but $3 + = 3 \$ 6$ is not. $1 + 2 = 4$ uses correct syntax, but is semantically incorrect. H_2O is a syntactically correct chemical formula, but $_2Zz$ is not.

1.4.1 Tokens and grammar

Syntax rules come in two flavors, pertaining to tokens and grammar. **Tokens** are the basic elements of the language, like words, numbers, and chemical elements. One of the problems with $3 + = 3 \$ 6$ is that $\$$ is not a legal token in mathematics. Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz .

The second type of syntax rule pertains to the **grammar** of the language, or the way tokens can be arranged. The statement $3 + = 3$ is structurally illegal, even though $+$ and $=$ are legal tokens, because you can't have one right after the other. Similarly, in a chemical formula the subscript comes after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out its structure. This process is called **parsing**, and in

a natural language you learn to do it unconsciously. For example, when you hear the statement “the penny dropped,” you understand that the penny is the subject and dropped is the predicate. After you have parsed the statement, you can begin to figure out what it means.

1.4.2 Reading source code

Although formal and natural languages have features in common—tokens, grammar, and meaning—there are some differences.

ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness: Natural languages are full of idiom and metaphor. When someone says “the penny dropped” there is no penny and nothing dropping. This idiom means that someone finally realized something after a period of confusion. In contrast, formal languages mean exactly what they say.

People who grow up speaking a natural language—that is, everyone—often have a hard time adjusting to formal languages. In some ways, the difference between natural and formal language is like the difference between poetry and prose, but more so.

poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

prose: The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

program: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and grammar.

Remember that formal languages are much more dense than natural languages, so it takes longer to read them. The structure is very important, so it is not always a good idea to read from top to bottom, left to right. Over time you will learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.5 The hello world program

Traditionally, the first program you write when learning a new programming language is called the hello world program. All it does is display the words “Hello, World!” on the screen. In Java, it looks like this:

```
public class Hello {  
  
    public static void main(String[] args) {  
        // generate some simple output  
        System.out.println("Hello, World!");  
    }  
}
```

Note the output of this program does not include the quote marks:

```
Hello, World!
```

Unfortunately in Java, even this simple example requires language features that are difficult to explain to beginners. But it provides a preview of topics that we will see in detail later on. The word `public` means the code can be accessed from other source files. The word `static` means that memory is allocated for the program in advance. We will discuss `void`, `String`, and `args` in the next few chapters. For now, let’s focus on the overall structure.

Java programs are made up of **class** and **method** definitions, which generally have the form:

```
public class CLASSNAME {  
  
    METHOD {  
        STATEMENTS  
    }  
  
    METHOD {  
        STATEMENTS  
    }  
  
}
```

Here `CLASSNAME` indicates the name chosen by the programmer. Java requires the class name to match the source file name. In the hello world example, the file name must be `Hello.java` because the class name is `Hello`.

Classes define a program's methods, or named sequences of **statements**. The `Hello` class has only one method:

```
public static void main(String[] args)
```

The name and format of `main` is special; it marks the place in the class where execution begins. When the program runs, it starts at the first statement in `main` and ends when it finishes the last statement.

Java uses squiggly braces (`{` and `}`) to group things together. In `Hello.java`, the outermost braces (lines 1 and 8) contain the class definition, and the inner braces (lines 3 and 6) contain the definition of `main` method.

The `main` method can have any number of statements, but the `Hello` example has only one. It is a **print statement**, meaning that it displays a message on the screen. Confusingly, `print` can mean both “display something on the screen” and “send something to the printer.” In this book, we’ll do all our printing on the screen. The `print` statement ends with a semicolon (`;`).

Line 4 contains a **comment**, or a bit of English text that explains the code that follows. When the compiler sees `//`, it ignores everything from there until the end of the line. It is a good idea to write a comment before every major block of code so that other programmers (including your future self) can understand what you meant to do.

1.6 Getting started with DrJava

In order to compile Java programs on your own computer, you will need to install the Java Development Kit (JDK). This free software by Oracle includes tools for developing and debugging Java programs. All the examples in this book were developed and tested using Java SE Version 7. Later versions of Java are generally backward compatible, so if you are using a more recent version, the examples in this book should still work.

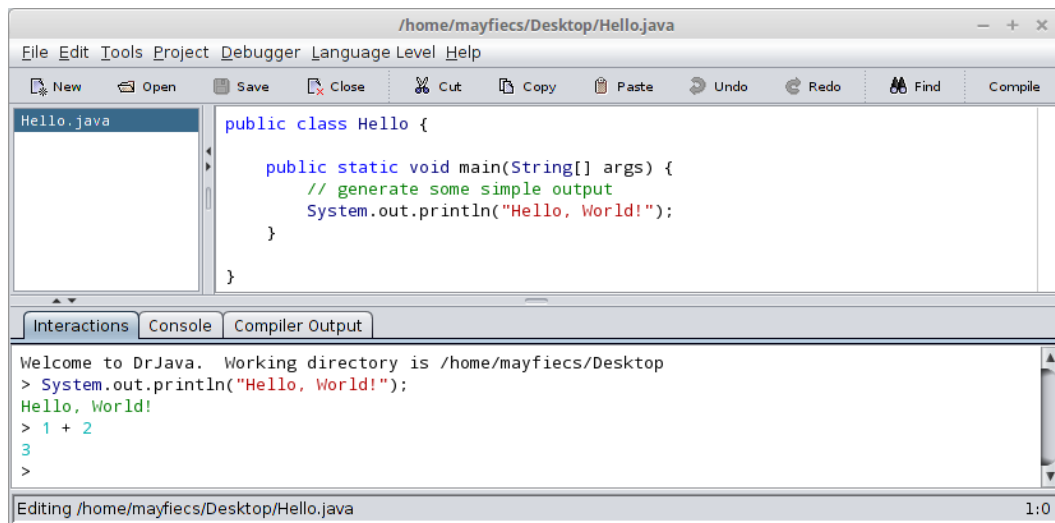


Figure 1.3: Screenshot of DrJava editing the hello world program.

We will use DrJava as the primary development environment throughout the book. A useful feature of DrJava is the Interactions Pane at the bottom of the window. It provides the ability to try out code quickly, without having to write a class definition and save/compile/run the program. Refer to the DrJava documentation (<http://drjava.org/docs/quickstart/>) for more details.

Step-by-step instructions for installing the JDK and configuring DrJava are available on this book's website: <http://thinkjava.org/>

1.6.1 Command-line interface

One of the most powerful and useful skills you can learn as a computer scientist is how to use the **command-line**, also called the *terminal*. The command-line

is a direct interface to the operating system. It allows you to run programs, manage files and directories, and monitor system resources. Many advanced tools, both for software development and general purpose computing, are available only at the command-line.

There are many good tutorials online for learning the command-line for your operating system; just search the web for “command line tutorial.” To get started, you only need to know four commands: how to change the working directory (`cd`), list directory contents (`ls`), compile Java programs (`javac`), and run Java programs (`java`).

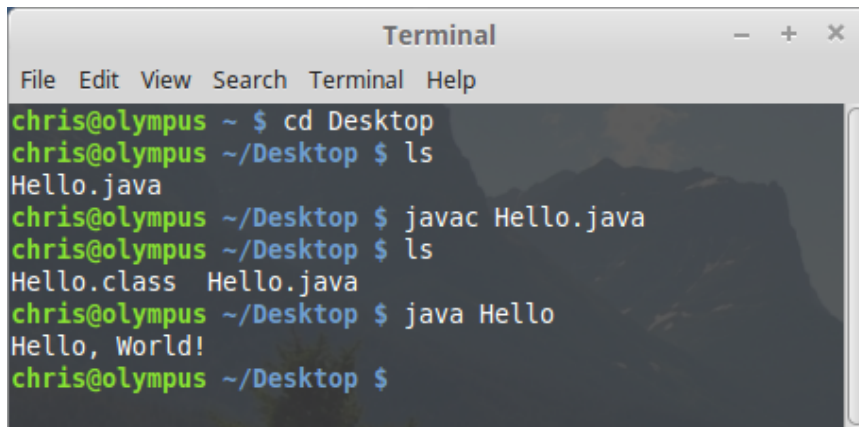
A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal shows a series of commands and their outputs. The user "chris" is at the "olympus" machine. The commands and outputs are:
1. `chris@olympus ~ $ cd Desktop`
2. `chris@olympus ~/Desktop $ ls` followed by the output `Hello.java`
3. `chris@olympus ~/Desktop $ javac Hello.java`
4. `chris@olympus ~/Desktop $ ls` followed by the output `Hello.class Hello.java`
5. `chris@olympus ~/Desktop $ java Hello` followed by the output `Hello, World!`
6. The prompt `chris@olympus ~/Desktop $` is shown again.

Figure 1.4: Compiling and running `Hello.java` from the command-line.

In this example, the `Hello.java` source file is stored in the `Desktop` directory. After changing to that location and listing the files, we use the `javac` command to compile `Hello.java`. Running `ls` again, we see that the compiler generated a new file, `Hello.class`, which contains the byte code. We run the program using the `java` command, which displays the output on the following line.

Note that the `javac` command requires a *file name* (or multiple source files separated by spaces), whereas the `java` command requires a single *class name*. If you use DrJava, it runs these commands for you and displays the output in the Interactions Pane.

Taking time to learn this efficient and elegant way of interacting with your operating system will make you more productive as a computer user. People who don't use the command-line don't know what they're missing.

1.7 More printing

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

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

As this program demonstrates, you can put comments at the end of a line as well as on lines all by themselves.

Phrases that appear in quotation marks are called **strings**, because they contain a sequence of characters strung together. Strings can contain any combination of letters, numbers, punctuation marks, symbols, and even non-printable characters like tab and backspace.

The name `println` is short for “print line.” It appends a special character, called a **newline**, that advances the cursor to the beginning of the next line. To display the output from multiple print statements on one line, use `print`:

```
public class Goodbye {  
  
    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`. 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.

1.7.1 Code formatting

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

```
public class Goodbye {  
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 newlines at the end of each line do not affect the program's behavior either. So we could have also written:

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

It still works, but the program is getting harder and harder to read. Newlines and spaces are important for organizing your program visually, making it easier to understand the program and find errors when they occur.

1.7.2 Escape sequences

It is possible to print multiple lines of output in just one line of code. You simply have to tell Java where to put the newlines.

```
public class Hello {  
  
    public static void main(String[] args) {  
        System.out.print("Hello!\nHow are you doing?\n");  
    }  
}
```

The output is two lines, each ending with a newline character:


```
Hello!  
How are you doing?
```

The code `\n` is an example of an **escape sequence**, which is a sequence of characters in a string that represents a special character. The backslash allows you to “escape” the string’s literal interpretation. Notice there is no space between `\n` and `How`. If you add a space there, there will be a space at the beginning of the second line.

<code>\n</code>	newline
<code>\t</code>	tab
<code>\"</code>	double quote
<code>\\</code>	backslash

Table 1.1: Common escape sequences

Another common use of escape sequences is to have quote marks inside of strings. Since double quotes indicate the beginning and end of strings, you need to escape them with a backslash.

```
System.out.println("She said \"Hello!\" to me.");
```

The result is:

```
She said "Hello!" to me.
```

1.8 Working through examples

It is a good idea to read this book in front of a computer so you can try out the examples as you go. You can run many of the examples directly in DrJava’s Interactions Pane, but if you put the code in a source file, it will be easier to try out variations.

Whenever you are experimenting with a new feature, you should also try to make mistakes. For example, in the hello world program, what happens if you leave out one of the quotation marks? What if you leave out both? What if you spell `println` wrong? This kind of experiment helps you remember what

you read. It also helps with debugging, because you get to know what the error messages mean. It is better to make mistakes now and on purpose than later on and accidentally.

Debugging is like an experimental science. Once you have an idea about what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, “When you have eliminated the impossible, whatever remains, however improbable, must be the truth.” (A. Conan Doyle, *The Sign of Four*.)

Programming and debugging should go hand in hand. Don’t just write a bunch of code and then perform trial and error debugging until it all works. Instead, start with a program that does *something* and make small modifications, debugging them as you go, until the program does what you want. That way you will always have a working program, and it will be easier to isolate errors.

A great example of this principle is the Linux operating system, which contains millions of lines of code. It started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, “One of Linus’s earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux.” (*The Linux Users’ Guide*)

Finally, programming sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent, or embarrassed. Remember that you are not alone, and most if not all programmers have had similar experiences. Don’t hesitate to reach out to a friend and ask questions!

1.9 Vocabulary

problem-solving: The process of formulating a problem, finding a solution, and expressing the solution.

program: A sequence of instructions that specify how to perform tasks on a computer.

programming: The application of problem-solving to creating executable computer programs.

computer science: The scientific and practical approach to computation and its applications.

algorithm: A procedure or formula for solving a problem, with or without a computer.

bug: An error in a program.

debugging: The process of finding and removing any of the three kinds of errors.

high-level language: A programming language that is designed to be easy for humans to read and write.

low-level language: A programming language that is designed to be easy for a computer to run. Also called “machine language” or “assembly language.”

portable: The ability of a program to run on more than one kind of computer.

interpret: To run a program in a high-level language by translating it one line at a time and immediately executing the corresponding instructions.

compile: To translate a program in a high-level language into a low-level language, all at once, in preparation for later execution.

source code: A program in a high-level language, before being compiled.

object code: The output of the compiler, after translating the program.

executable: Another name for object code that is ready to run on specific hardware.

byte code: A special kind of object code used for Java programs. Byte code is similar to a low-level language, but it is portable like a high-level language.

natural language: Any of the languages people speak that have evolved naturally.

formal language: A language people have designed for specific purposes, like representing mathematical ideas or computer programs.

programming language: A formal language that has been designed to express computations.

syntax: The structure of a program.

semantics: The meaning of a program.

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

grammar: A set of rules that determines whether a statement is legal.

parse: To examine a program and analyze the syntactic structure.

statement: A part of a program that specifies a computation.

method: A named sequence of statements.

class: For now, a collection of related methods. (We will see later that there is more to it.)

print statement: A statement that causes output to be displayed on the screen.

comment: A part of a program that contains information about the program but has no effect when the program runs.

command-line: A means of interacting with the computer by issuing commands in the form of successive lines of text.

string: A sequence of characters; the primary data type for text.

newline: A special character signifying the end of a line of text. Also known as line ending, end of line (EOL), or line break.

escape sequence: A sequence of code that represents a special character when used inside a string.

1.10 Exercises

Exercise 1.1 Computer scientists have the annoying habit of using common English words to mean something other than their common English meaning. For example, in English, statements and comments are the same thing, but in programs they are different.

The glossary at the end of each chapter is intended to highlight words and phrases that have special meanings in computer science. When you see familiar words, don't assume that you know what they mean!

1. In computer jargon, what's the difference between a statement and a comment?
2. What does it mean to say that a program is portable?
3. What is an executable?

Exercise 1.2 Before you do anything else, find out how to compile and run a Java program in your environment. Some environments provide sample programs similar to the example in Section 1.5.

1. Type in the "Hello, world" program, then compile and run it.
2. Add a print statement that prints a second message after the "Hello, world!". Say something witty like, "How are you?" Compile and run the program again.
3. Add a comment to the program (anywhere), recompile, and run it again. The new comment should not affect the result.

This exercise may seem trivial, but it is the starting place for many of the programs we will work with. To debug with confidence, you have to have confidence in your programming environment. In some environments, it is easy to lose track of which program is executing. You might find yourself trying to debug one program while you are accidentally running another. Adding (and changing) print statements is a simple way to be sure that the program you are looking at is the program you are running.

Exercise 1.3 It is a good idea to commit as many errors as you can think of, so that you see what error messages the compiler produces. Sometimes the compiler tells you exactly what is wrong, and all you have to do is fix it. But sometimes the error messages are misleading. You will develop a sense for when you can trust the compiler and when you have to figure things out yourself.

1. Remove one of the open squiggly-braces.
2. Remove one of the close squiggly-braces.
3. Instead of `main`, write `mian`.
4. Remove the word `static`.
5. Remove the word `public`.
6. Remove the word `System`.
7. Replace `println` with `Println`.
8. Replace `println` with `print`. This one is tricky because it is a logic error, not a syntax error. The statement `System.out.print` is legal, but it may or may not do what you expect.
9. Delete one of the parentheses. Add an extra one.

Chapter 2

Variables and arithmetic

This chapter is about storing values in computer memory and doing basic arithmetic. More importantly, it discusses how to *compose* statements using smaller building blocks such as variables and operators. We also discuss code quality, which is almost as important as correctness. High quality code is easier to read, which makes it easier to maintain and debug.

2.1 Types of errors

Three kinds of errors can occur in a program: syntax errors, runtime errors, and logic errors. It is useful to distinguish between them in order to track them down more quickly. Regardless of what type of error occurs, remember to *read and think about the error messages carefully*. They will usually point you in the right direction to fix your program.

2.1.1 Syntax errors

The compiler can only translate a program if the syntax is correct; otherwise, it fails and displays an error message. For example, parentheses have to come in matching pairs. So $(1 + 2)$ is legal, but $8)$ is a **syntax error**.

In English, readers can tolerate most syntax errors, which is why we can read the poetry of E. E. Cummings without spewing error messages. Java is not

so forgiving; if there is a single syntax error anywhere in your program, the compiler will display an error message and quit, and you will not be able to run the program.

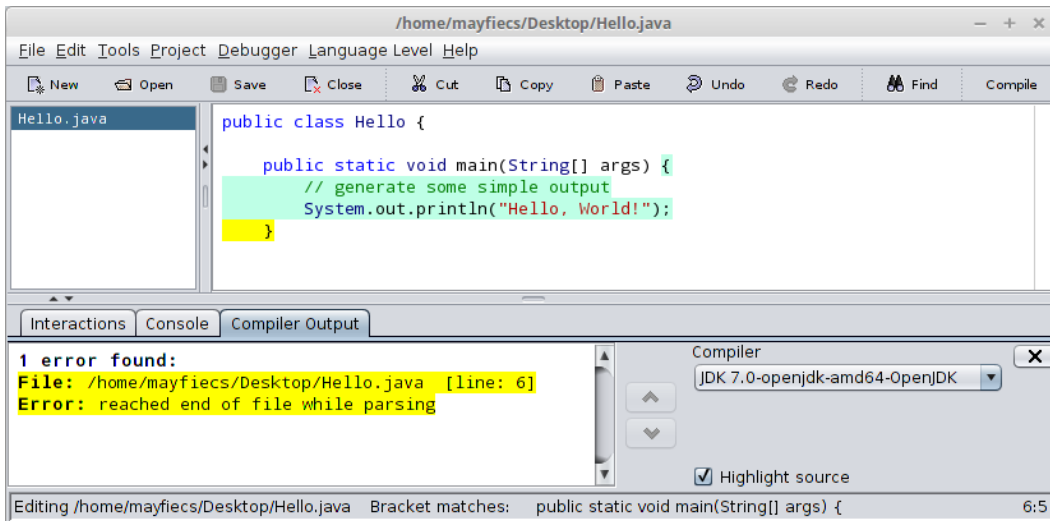


Figure 2.1: A syntax error caused by a missing brace.

To make matters worse, the error messages you get from the compiler are often not very helpful. As shown in Figure 2.1, removing the closing brace on line 8 of the hello world program results in “Error: reached end of file while parsing.” The compiler also reports that the problem was found on line 6, which in this case is not at fault. Since line 8 was deleted, the compiler simply reported the last line of the file.

During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. But as you gain experience, you will make fewer mistakes and find them more quickly.

2.1.2 Runtime errors

The second type of error is a **runtime error**, so called because it does not appear until after the program has started running. In Java, these errors occur when the interpreter is executing the byte code and something goes wrong. Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

These errors are also called *exceptions* because they usually indicate that something exceptional (and bad) has happened. In most environments they appear as windows or dialog boxes that contain information about what happened and what the program was doing when it happened. For example, if you accidentally divide by zero you will get an `ArithmeticException`:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Hello.main(Hello.java:5)
```

This information is useful for debugging. The first line gives a brief description of the error (/ by zero). The subsequent lines report the class and method names (`Hello.main`), along with the file name and line number where the error occurred (`Hello.java:5`). Keep in mind that the line where the program crashed may not be the line that needs to be fixed.

2.1.3 Logic errors

The third type of error is the **logic error**. If there is an error in your program's logic, it will compile and run successfully in the sense that the computer will not generate any error messages. But it will not do the right thing. It will do something else. Specifically, it will do what you told it to do. Here is an example of a logic error in the hello world program:

```
public class Hello {

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

}
```

This program compiles and runs just fine. The problem is that the main method is not the program we intended. The meaning of the program is wrong, because it says goodbye instead of hello. In addition, `world` is not capitalized, and it ends with a period instead of an exclamation point.

Identifying logic errors can be tricky because it requires you to challenge your assumptions, both about the code and the requirements. You will need to work backwards by looking at the output of the program, try to figure out what it is doing, and make sure you understand what it should be doing.

2.2 Creating variables

One of the most powerful features of a programming language is the ability to define and manipulate **variables**. A variable is a named location of computer memory that stores a **value**. Values may be numbers, text, images, sounds, and other types of data. To store a value in memory, you first have to create a variable.

```
String message;
```

This statement is a **declaration**, because it declares that the variable named `message` 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 `char` type can store characters.

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 complain if you make one up.

To declare an integer variable, the syntax is:

```
int x;
```

Note that `x` is an arbitrary name for the variable. In general, you should use names that indicate what the variables mean. For example, if you saw these variable declarations, you could probably guess what values would be stored in them:

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

This example also demonstrates the syntax for declaring multiple variables with the same type: `hour` and `minute` are both integers. Note that each declaration statement ends with a semicolon.

You can use any name you want for a variable. But there are certain words that are reserved in Java, because they are used by the compiler to parse the structure of the program. These **keywords** include `public`, `class`, `static`,

`void`, `int`, and others (there are currently 50 in Java). Search the Internet for “Java keywords” to see the complete list.

Rather than memorize the keywords, you should take advantage of the syntax highlighting provided in many development environments (including DrJava). As you type, the tokens in your program will appear in different colors. For example, keywords might be blue, strings red, comments green, and other code black. If you type a variable name and it turns blue, watch out!

2.2.1 Assignment

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

```
message = "Hello!"; // give message the value "Hello!"
hour = 10;          // assign the value 10 to hour
minute = 59;        // set minute to 59
hour = 11;          // change the hour to 11
```

This example shows four assignments, and the comments illustrate 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.
- If you reassign the variable, its value changes.

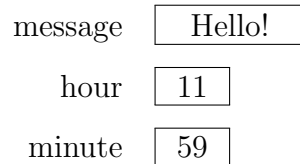
As a general rule, a variable has to have the same type as the value you assign to it. For example, you cannot store a `String` in `minute` or an integer in `message`. We will see some examples that seem to break this rule, but we’ll get to that later.

A common source of confusion is that some strings *look* like integers, but they are not. For example, `message` 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 integer 123.

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

2.2.2 Memory diagrams

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 diagram shows the effect of the four assignments in the previous section:



Each box represents the storage location that holds the variable's value. Once declared, you cannot change the name of a variable, but you can change the value as many times as you like. For example, `hour` was initially 10, then reassigned to 11. The diagram shows only the final value.

2.3 Printing variables

You can display the value of a variable using `print` or `println`. The following program declares a variable named `firstLine`, assigns it the value `"Hello, again!"`, and then prints that value.

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

When we talk about printing a variable, we generally mean printing the *value* of the variable. To print the *name* of a variable, you have to put it in quotes. 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!
```

The syntax for printing a variable is the same regardless of the variable's type. For example:

```
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.
```

To output multiple values on the same line, it's common to use several `print` statements followed by a `println` at the end. **Don't forget the `println`!** On many operating systems, 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 the `println`, the program may display the stored output at unexpected times or even terminate without displaying anything.

2.4 Arithmetic operators

Operators are symbols that represent computations like addition and multiplication. For example, the operator for addition is `+`, subtraction is `-`, multiplication is `*`, and division is `/`.

The following program converts the time of day to minutes:

```
int hour, minute;
hour = 11;
minute = 59;
System.out.print("Number of minutes since midnight: ");
System.out.println(hour * 60 + minute);
```

In this program, `hour * 60 + minute` is an **expression**, which represents a single value to be computed. When the program runs, each variable is replaced by its current value, and then the operators are applied. The result is:

```
Number of minutes since midnight: 719
```

Expressions are generally a combination of numbers, variables, and operators. When compiled and executed, they become a single value:

<code>1 + 1</code>	<code>hour - 1</code>	<code>hour * 60 + minute</code>	<code>minute / 60</code>
2	11 - 1	11 * 60 + 59	59 / 60
	10	660 + 59	0
		719	

Addition, subtraction, and multiplication all do what you expect, but you might be surprised by division. For example, the following lines try to compute the fraction of an hour that has elapsed:

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

But the program outputs:

```
Fraction of the hour that has passed: 0
```

This result often confuses people. After all, the value of `minute` is 59, and 59 divided by 60 should be 0.98333, not 0. The problem here is that Java performs *integer division*. When the values being divided are integers, the result is also an integer. Computer hardware is designed so that integer division always rounds *down*, even in cases like this one where the next integer is close.

One solution is to calculate a percentage rather than a fraction:

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

The new output is:

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

Again the result is rounded down, but at least now it's approximately correct.

2.5 Floating-point numbers

A more general solution is to use **floating-point** numbers, which can represent fractions as well as integers.

In Java, the default floating-point type is called `double`, which is short for double-precision. You can create `double` variables and assign values to them using the same syntax we used for the other types:

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

Although floating-point numbers are useful, they can be a source of confusion. For example, 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 data types, and strictly speaking, you are not allowed to make assignments between types.

The following is illegal because the variable on the left is an `int` and the value on the right is a `double`.

```
int x;  
x = 1.1; // syntax error
```

But it is easy to forget this rule because in many cases Java *automatically* converts from one type to another:

```
double y;  
y = 1; // bad style
```

The above example should be illegal, but Java allows it by converting the `int` value 1 to the `double` value 1.0 automatically. This leniency is convenient, but it often causes problems for beginners. For example:

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

You might expect the variable `y` to get the value 0.333333, which is a legal floating-point value. But instead it gets the value 0.0. The reason is that the expression on the right divides two integers. So Java does *integer division*, which yields the `int` value 0. Converted to `double`, the final result is 0.0.

One way to solve this problem (after you finally discover that bug) is to make the right-hand side a floating-point expression. The following initializes `y` to 0.333333, as expected:

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

As a matter of style, you should always assign floating-point values to floating-point variables. The compiler won't make you do it, but you never know when a bug like this one will come back and haunt you.

2.6 Operators for strings

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following expressions are illegal:

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

The `+` operator does work with strings, but it might not do what you expect. For strings, the `+` operator performs **concatenation**, which means joining each part end-to-end. So `"Hello, " + "world!"` yields the string `"Hello, world!"`. Likewise, the expression `"Hello, " + name` adds the value of `name` to the hello string, which is handy for creating a personalized greeting.

Since addition is defined for both numbers and strings, Java performs automatic conversions you may not expect:

```
System.out.println(1 + 2 + "Hello");  
// the output is 3Hello  
  
System.out.println("Hello" + 1 + 2);  
// the output is Hello12
```

Java executes these operations from left to right. In the first line, `1 + 2` is the value 3, and `3 + "Hello"` is the value `"3Hello"`. But in the second line, `"Hello" + 1` is `"Hello1"`, and `"Hello1" + 2` is `"Hello12"`. The difference is when the conversion from integer to string actually takes place.

2.7 Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the rules of **precedence**. Generally speaking, Java executes individual operations from left to right (as was the case in the previous section). But for numeric operators, Java follows mathematical conventions:

- 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 because of 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 these same operations had gone from right to left, the result would have been $59 * 1$, which is incorrect.
- 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 actual result.

Don't work too hard to remember all the rules of precedence, especially for other operators. If it's not obvious by looking at the expression, use parentheses to make it more clear.

2.8 Composition

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

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 these operations into 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);
```

You can also put arbitrary expressions on the right side of an assignment:

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

The left side of an assignment must be a *variable name*, not an expression. That's because the left side indicates where the result will be stored, and expressions do not represent storage locations.

```
hour = minute + 1; // correct  
minute + 1 = hour; // syntax error
```

The ability to compose operations may not seem that impressive now, but we will see examples later on that allow us to write complex computations neatly and concisely.

Before you get too carried away with composition, keep in mind that other people will be reading your source code. In practice, software developers spend the vast majority of their time *understanding* and *modifying* existing code. Thus it's far more important to write code that is readable than to write code that is (or appears to be) optimal. In general, each line of code should be a single step of the algorithm.

2.9 Formatting and style

Recall from Section 1.7.1 that the compiler generally ignores **whitespace**, i.e., newlines, tab characters, and other spaces. Programmers have a lot of freedom in how they *format* their code in terms of indenting, blank lines, spaces around operators, etc. However with that freedom comes responsibility, both to yourself (when you look at the code in the future) and to others who will be reading, understanding, and modifying your code.

Virtually every organization that does a lot of software development has strict guidelines on how to format source code. For example, Google published its Java coding standards for use in open-source projects: <http://google.github.io/styleguide/javaguide.html> It is easier to understand a large codebase when all the source code is formatted consistently.

Style rules can be difficult to learn, especially for beginners who haven't yet seen many of the language features discussed in them. Fortunately there are many tools that help programmers find and correct formatting errors. One prominent example is Checkstyle, which has the built-in ability to enforce most of Google's coding standards: <http://checkstyle.sourceforge.net/>

Checkstyle is primarily a command-line tool. Instructions for downloading and running Checkstyle are available on our website: <http://thinkjava.org/>

There are limits to what automatic style checkers can do. In particular, they can't evaluate the *quality* of your comments, the *meaning* of your variable names, or the *structure* of your algorithms. Good comments make it easier for experienced developers to identify errors in your code. Good variable names communicate the intent of your program and how the data is organized. And good programs are designed to be efficient and demonstrably correct.

2.10 Vocabulary

syntax error: An error in a program that makes it impossible to parse (and therefore impossible to compile).

runtime error: An error in a program that makes it impossible to execute completely. In Java, they are “exceptions” that terminate the program.

logic error: An error in a program that makes it do something other than what the programmer intended.

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 that can be stored in a variable. Every value belongs to a type (for example, `int` or `String`).

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

type: Mathematically speaking, a set of values. The type of a variable determines which values it can have.

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

assignment: A statement that stores a value in a memory location.

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

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

floating-point: A data type that represents decimal numbers (numbers that have an integer part and a fractional part). In Java, the default floating-point type is `double`.

concatenate: To join two values end-to-end. For string values, concatenation means to append.

precedence: The order in which operations are evaluated.

composition: The ability to combine simple expressions and statements into compound expressions and statements, making it possible to represent complex computations in a concise manner.

whitespace: Newlines, tab characters, and other spaces in a source program. Its purpose in the Java language is to separate tokens.

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 The point of this exercise is (1) to use string concatenation to display values with different types (`int` and `String`), and (2) to practice developing programs gradually by adding a few statements at a time.

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 2.3, 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 format, for example: `Thursday, July 16, 2015`.
5. Modify the program again so that the total output is:

```
American format:
Thursday, July 16, 2015
European format:
Thursday 16 July, 2015
```

HINT: You should be able to copy, paste, and modify the code from Step 4 when completing Step 5.

Exercise 2.3 The point of this exercise is (1) to use some of the arithmetic operators, and (2) to start thinking about compound entities (like time of day) that are represented with multiple values.

1. Create a new program called `Time.java`. From now on, we won't remind you to start with a small, working program, but you should.

2. Following the example program in Section 2.3, create variables named `hour`, `minute`, and `second`. Assign values that are roughly the current time. Use a 24-hour clock, i.e., so that at 2pm the value of `hour` is 14.
3. Make the program calculate and print the number of seconds since the most recent 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. You might run into problems when computing percentages with integers, so consider using floating-point.
6. Change the values of `hour`, `minute`, and `second` to reflect the current time. Check that the program works correctly each time you run it.

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

Chapter 3

Input and output

The programs we've looked at so far just 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

`System.out.println` can display the value of any type of variable. You can even use `println` to print the value of `System.out`:

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

The result is:

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

From this output we can see 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.

After the `@` sign is the location of the object in memory, which is called its **address**. In this example the address is `685d72cd`, but if you run the same code you will likely get something different.

`System.out` is an **object**, which means that it is a special value that provides methods. Specifically, `System.out` provides methods for displaying output, including `print` and `println`. Numbers with type `int` and `double` are not objects because they provide no methods. Strings are objects; we will see some of their methods soon.

The `System` class 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.

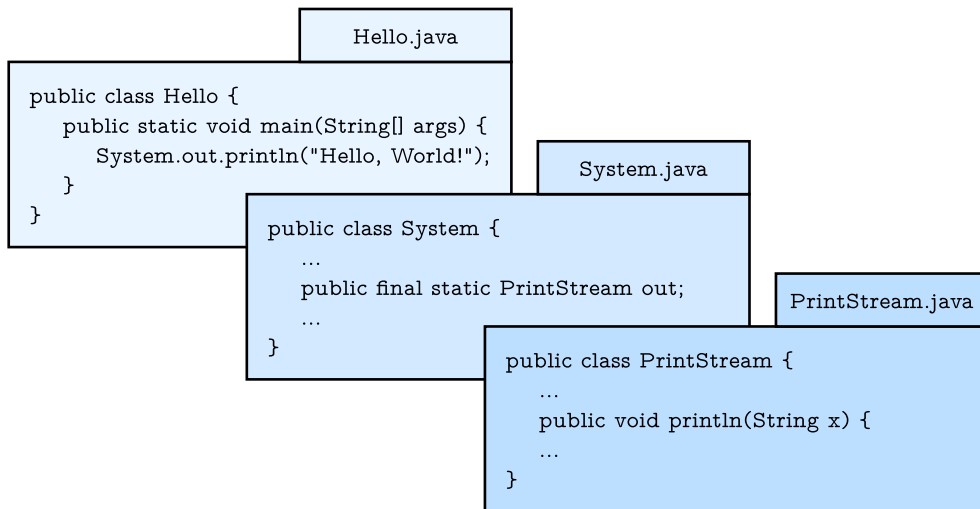


Figure 3.1: `System.out.println` refers to the `out` variable of the `System` class, which is a `PrintStream` that provides a method called `println`.

3.2 The Scanner class

The `System` class also provides an object named `in`, which is an `InputStream` that provides methods for reading input from the keyboard. These methods perform simple operations, but they 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 at the top of your source file:


```
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.

Next you have to create a **Scanner** object using the keyword `new`. The following code declares a **Scanner** variable and then creates a **Scanner** object:

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

Although **Scanner** is not a method, the syntax is the same as a method call. We pass `System.in` as an argument, which specifies that we are planning to input values from the keyboard. Alternatively, you can declare the variable and assign it using one line of code.

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

The new **Scanner** object (stored in the variable `in`) 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.

3.2.1 Structure of Java programs

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

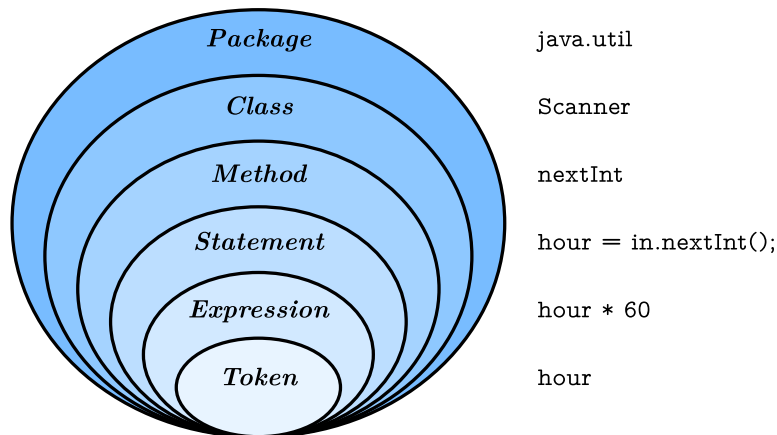


Figure 3.2: Elements of the Java language, from largest to smallest.

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 include variable names, numbers, operators, keywords, and punctuation like braces and semi-colons.

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 on Oracle’s website: <http://docs.oracle.com/javase/7/docs/api/> Note there is a major difference between the Java *language*, which deals with syntax and grammar, and the Java *library*, which provides the built-in classes. In face, most of the Java library itself is written in Java.

3.3 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 may have to convert from Celsius to Fahrenheit and back. Or you might want to convert your height in inches to centimeters.

We can write a program to help. We can use a `Scanner` to input a measurement in inches, convert to centimeters, and then print the results. These lines declare the variables and create the `Scanner`:

```
int inch; // the input
double cm; // the output
Scanner in = new Scanner(System.in);
```

The first 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.

```
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. Finally we display the results on one line, but use two print statements since it's easier to read.

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

This code works, but it has a problem. If another programmer reads this code, they might wonder where 2.54 comes from. Numbers that appear in an expression with no explanation are called **magic numbers**. For the benefit of other programmers (and yourself in the future), it is helpful to assign magic numbers to variables with informative names:

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

3.3.1 Literals and constants

Parts of the output we have been printing (`" in = "`, `" cm"`, etc.) are *literal* string values. **Literals** are data embedded directly into programs. In contrast, variables (`inch`, `cm`, etc.) are the names of values stored in memory. The English word *variable* implies that this data is subject to change.

As we saw with `cmPerInch`, it's often useful to give names to literals that will be used throughout the program. But when doing so, we don't want those variables to be changed accidentally because of an unexpected bug in the code. As another example, consider the famous card game that comes with most versions of Microsoft Windows.

```
final String title = "Solitaire";  
final int deckSize = 52;
```

The keyword `final` indicates these variables are **constants** and therefore may be assigned only one time. Constants are generally declared and **initialized** on the same line of code. If you attempt to assign `title` or `deckSize` later in the program, the compiler will report an error. This feature helps prevent you from making mistakes.

It's good practice to create final variables for constant values, rather than repeat literal values again and again. For one, it makes the program easier to understand. When looking at the code, it may not be obvious what the number 52 means. The name `deckSize` explains the programmer's intent. Second, named constants make the program easier to maintain. If we want to change the title from `"Solitaire"` to `"Klondike"`, we would only need to change one line of code (as opposed to every line where that title is used).

3.4 Formatting output

When printing floating-point numbers, Java automatically decides how many decimal places to display.

```
System.out.print(7.0 / 3.0);  
// prints 2.3333333333333335    note: 5 is a rounding error
```

`System.out` provides another method called `printf`, where the “f” stands for “formatted”. The first argument of `printf` is a *format string* that specifies how values should be displayed. The other arguments are the values themselves.

```
System.out.printf("Seven thirds = %.3f", 7.0 / 3.0);
// prints Seven thirds = 2.333
```

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 value should be displayed as floating-point with three decimal places. Here’s an example that contains two format specifiers:

```
inch = 100;
cm = inch * cmPerInch;
System.out.printf("%d in = %f cm\n", inch, cm);
// prints 100 in = 254.000000 cm
```

The values are matched up with the format specifiers in order, so `inch` is displayed as an integer (“d” stands for “decimal”) and `cm` is displayed as a floating-point number. Format strings often end with a newline character (`\n`), since `printf` does not append a newline like `println` does.

Learning `printf` is like learning a sub-language within Java. There are many options, and the details can be overwhelming. But here are some common uses, to give you an idea of how it works:

<code>%d</code>	decimal integer	12345
<code>%,d</code>	decimal integer with comma separators	12,345
<code>%08d</code>	padded with zeros, at least 8 digits wide	00012345
<code>%f</code>	floating-point	6.789000
<code>%.2f</code>	floating-point <i>rounded</i> to 2 decimal places	6.79

Table 3.1: Example format specifiers

For more details, refer to the documentation of `java.util.Formatter` or search the web for “java formatting.”

3.5 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 / centPerInch; // 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 shape 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. Converting to an integer always rounds down (toward zero), even if the fraction part is 0.999999.

Type casting takes precedence over arithmetic operations. In this example, the value of `pi` gets converted to an integer first. So the result is 60.0, not 62.

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

The parentheses after the cast operator require the division to come before the type cast. This result will be rounded down, but we will learn in the next chapter how to round floating-point numbers to the closest integer.

3.5.1 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.

Assuming the following variables are integers, we can convert 76 inches 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.

Although the modulus operator is a percent sign, you might find it helpful to think of it as a division sign (\div) rotated to the left.

Integer division 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 also 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 are based on modular arithmetic.

3.6 Putting it all together

At this point you know enough Java to write useful programs that solve everyday problems. You've seen how to 1) import Java library classes, 2) initialize a `Scanner` object, 3) get input from the keyboard, 4) format output with `printf`, and 5) divide and mod integers. Now we can put them 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;
        final double centPerInch = 2.54;
        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 / centPerInch);
        feet = inches / 12;
        inches = inches % 12;
        System.out.printf("%.2f cm = %d ft, %d in\n",
                           cm, feet, inches);
    }
}
```

- 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 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.
- Integer division and modulus often go together. Notice how `inches` gets reassigned (which replaces its value) just before the `printf`.
- 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.

3.7 Reading documentation

Now would be a good time to take a look at the documentation for **Scanner**. You can find it in the Java library (see the link in Section 3.2.1) or simply do a web search for “java scanner.” The latter method is more useful in the long run, especially as Oracle releases new versions of Java. Either way, you should get something like this:

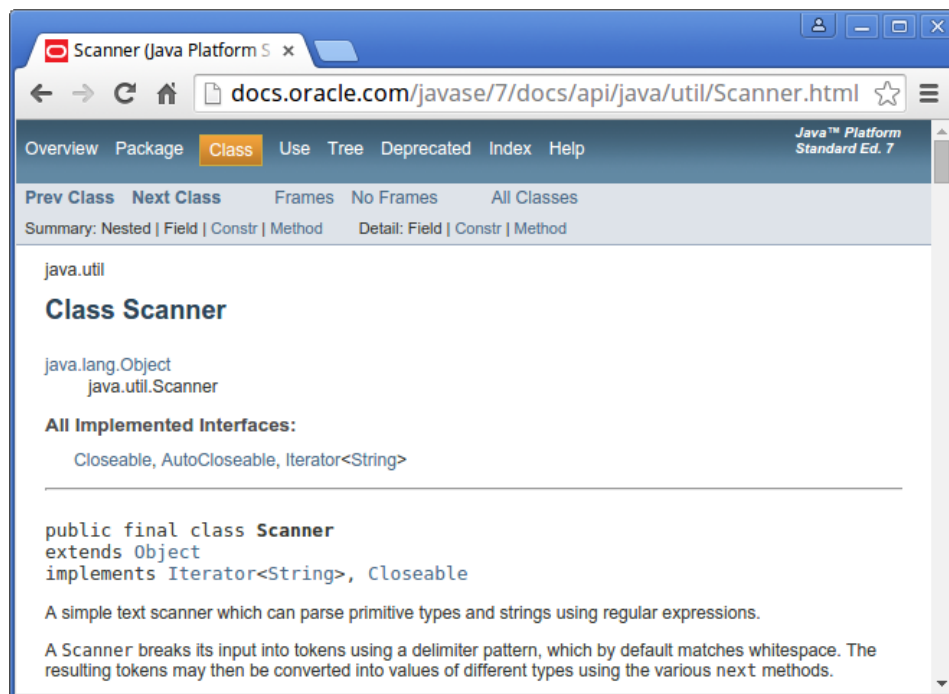


Figure 3.3: Screenshot of the documentation for **Scanner** on Oracle’s website.

Scroll down to the “Method Summary” section. As you can see, the **Scanner** class provides quite a few methods. In this chapter we focused on the “next” methods. Click on the link for `nextInt`.

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

The first line is the method’s **prototype**, which specifies the name of the method and its return type. In this example, `nextInt` returns an `int`. The next line describes what the method does. The subsequent lines (not shown) explain the parameters and return values. Explanations are often redundant, but the documentation is supposed to fit this standard format.

It might take some time to get comfortable reading this kind of information, but it's well worth the effort. Knowing what methods a class provides helps you avoid reinventing the wheel. Whenever you learn about a new class, you should take a quick look at its documentation. On that note, take a few minutes to review the documentation for `System` and `String`.

3.8 Writing documentation

A nice feature of the Java language is the ability to write documentation at the same time you are writing the source code. That way, the documentation stays in sync with the classes and methods themselves. In fact, the HTML pages you browsed in the previous section were automatically generated using a tool called **Javadoc**. This tool is part of the standard JDK, and you can run it directly from DrJava by pressing the **Javadoc** button on the toolbar.

Javadoc parses your source files for **documentation comments** and extracts other relevant information about your class and method definitions. Given the prevalence of this tool, people sometimes refer to documentation as “Javadoc comments.” In contrast to inline comments that begin with `//`, documentation comments begin with `/**` (two stars) and end with `*/` (one star). Anything in between these two tokens becomes part of the documentation.

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

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

This example has perhaps too many comments, since all the program does is print a message. But it illustrates the differences between inline and documentation comments:

- Inline comments tend to be short phrases that help explain complex parts of a method. Documentation comments are typically complete sentences that begin with a capital letter and end with a period.
- Documentation comments often span multiple lines. By convention, each line begins with a `*` that is aligned vertically with the start and end of the comment.
- Some development environments (e.g., Eclipse and NetBeans) automatically display documentation comments when you hover your mouse over the name of a class or method.

Writing documentation and inline comments is essential for making source code readable. As we discussed in the last chapter, people spend the majority of their development time understanding and modifying existing code. You should not only write good comments for others, but for yourself as well. When you haven't looked at your own code for a while, it takes a long time to remember how it works (or what you were trying to do) if there's no comments.

3.9 Command-line testing

You should review the advice in Section 1.8, now that you've written some more substantial programs. Remember, it's more effective to program and debug your code little by little than to attempt writing everything at once. And once you've completed programming an algorithm, it's important to test that it works correctly on a variety of inputs.

Throughout the book, we will illustrate techniques for testing your programs. Most if not all testing is based on a simple idea: does the program do what we expect it to do? For simple programs, it's not difficult to run them several times and see what happens. But at some point, you will get tired of typing the same test cases over and over.

We can automate the process of entering input and comparing *expected output* with *actual output* using the command-line. The basic idea is to store the test

cases in plain text files and trick Java into thinking they are coming from the keyboard. Here are step by step instructions.

1. Make sure you can compile and run the `Convert.java` example in the previous section.
2. In the same directory as `Convert.java`, create a plain text file named `test.in` (“in” is for input). Enter the following line and save the file.

```
193.04
```

3. Create a second plain text file named `test.exp` (“exp” is for expected). Enter the following line and save the file.

```
193.04 cm = 6 ft, 4 in
```

4. Open a command-line, and change to the directory with these files. Run the following command to test the program.

```
java Convert < test.in > test.out
```

On the command-line, `<` and `>` are **redirection operators**. The first one redirects the contents of `test.in` to `System.in`, as if it were entered from the keyboard. The second one redirects the contents of `System.out` to a new file `test.out`, much like a screen capture. In other words, the `test.out` file contains the output of your program.

By the way, it’s perfectly okay to compile your programs in DrJava (or some other environment) and run them from the command-line. Knowing both techniques allows you to use the right tool for the job.

At this point, we just need to compare the contents `test.out` with `test.exp`. If the files are the same, then the program outputted what we expected it to output. If not, then we found a bug, and we can use the output to begin debugging our program. Fortunately, there’s a simple way to compare files on the command-line:

```
diff test.exp test.out
```

The `diff` utility summarizes the differences between two files. If there are no differences, then it prints nothing, which in our case is what we want. If the expected output differs from the actual output, then we need to debug our program. Or in some cases, we need to debug our test cases. There’s always a chance we have a correct program and a typo in the expected output.

3.10 Vocabulary

package: A group of classes that are related to each other. Java classes are organized into packages.

address: The storage location of a variable or object in memory. Addresses are integers encoded in hexadecimal (base 16).

object: An abstract entity that represents data and performs actions. In Java, objects are stored in memory and referenced by variables.

library: A collection of packages and classes that are available for use in other programs. Libraries are often distributed in `.jar` (Java Archive) files.

utility class: A class that provides commonly needed functionality.

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

magic number: A unique value with unexplained meaning or multiple occurrences. They should generally be replaced with named constants.

literal: A constant value written directly in the source code. For example, `"Hello"` is a string literal and `74` is an integer literal.

constant: A variable that can only be assigned one time. Once initialized, its value cannot be changed.

initialize: To assign an initial value to a variable.

format specifier: A special code beginning with percent sign and ending with a single letter that stands for the data type.

type cast: An operation that explicitly converts one data type into another, sometimes with loss of information. 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 (e.g., `5 % 2` is 1).

prototype: The signature of a method that defines its name and what type it returns.

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.

redirection operator: A command-line feature that substitutes `System.in` and/or `System.out` with a plain text file.

3.11 Exercises

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

Exercise 3.2 TODO

Exercise 3.3 TODO

Chapter 4

Void methods

So far we've only written short programs that have a single class with a `main` method. In this chapter, we'll show you how to organize longer programs into multiple methods and classes.

At a conceptual level, a **method** represents a mathematical *function* or a general *procedure*. Some methods perform a computation and return a result. For example, `Math.sqrt(25)` returns the value 5.0. Other methods (including `main`) carry out a sequence of actions without returning a result. Java uses the keyword `void` to declare such methods. Regardless whether they return a value or not, methods enable you to break down a complex program into smaller blocks of code.

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 most common mathematical operations. `Math` is in the `java.lang` package, so you don't have to import it. You can 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 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 `final 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.

It turns out that converting to/from radians is a common operation, so the `Math` class provides methods for that.

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

If you haven't already, take a look at the documentation for `Math` so you know what methods are provided. For example, another useful method is `round`, which rounds a floating-point value to the nearest integer and returns a `long`.

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

In Java, `int` values are stored using 32 bits (4 bytes) of memory, whereas `long` values are stored in 64 bits (8 bytes). As a result, `long` variables can represent much larger integers. In the above example, the multiplication happens first, before the method is invoked. The result is 63 (rounded up from 62.8319).

4.1.1 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:

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

This statement takes the value `Math.PI`, divides it by two, and then adds the result to the value of the variable `angle`. The sum is then passed as an argument to `cos`. 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` 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`.

When using `Math` methods, it is a common error to forget to specify the class name `Math`. For example, `Math.pow` takes two arguments and raises the first argument to the power of the second.

```
pow(2.0, 10.0); // syntax error
```

If you try to invoke `Math.pow` this way, the compiler will say it “cannot find symbol” (i.e., there is no method named `pow` in the current class).

4.2 Adding new methods

Like the `Math` class, you can write your own set of methods for use in other programs. Let’s revisit the method definition for `main`:

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

The first line contains information about the method: `main` is a **public** method, which means it can be invoked from other classes; it is a **static**

method, but we're not going to explain what that means yet; and it is a `void` method, which means that it doesn't yield a result (unlike the `Math` methods).

The statement in parentheses declares a parameter named `args`. A **parameter** is a variable that stores an argument. This parameter has type `String[]`, which means that whoever invokes `main` must provide an array of Strings (we'll get to arrays in a later chapter).

You can define other methods using syntax is similar to `main`:

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

By convention, methods start with a lower case letter and use “camel case,” which is a cute name for `jammingWordsTogetherLikeThis`. You can use any name you want for your method, except `main` or any of the Java keywords.

The list of parameters specifies what values, if any, you have to provide in order to invoke the new method. The first methods we are going to write have no parameters, so the parameter list is empty. Here's an example:

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

The name of this method is `newLine`. It contains only one statement, which prints a blank line. In `main`, we can invoke the new method like this:

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

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:

```
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 prints three blank 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 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`.

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

1. 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.
2. Creating a new method can make a program smaller by eliminating repetitive code. For example, to print nine consecutive new lines, you could invoke `threeLine` three times.
3. A common problem-solving technique is to break things down into sub-problems. Methods allow you to focus on each sub-problem in isolation, and then compose them into a complete solution.

4.3 Classes and methods

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.");  
    }  
}
```

The first line is the class definition. Class names should be capitalized; this convention helps readers tell the difference between classes and methods in your source code. The `NewLine` class contains three `void` methods: `newLine`, `threeLine`, and `main`. Java is a case-sensitive language, so `NewLine` and `newLine` are considered different.

4.3.1 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 source file. In the previous example, we deliberately put `main` at the

bottom of the program. Statements are executed one at a time, in order, until you reach a method invocation. Think of method invocations as 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 the statements there, and then come back and pick up again 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`. But while we are executing `threeLine`, we go off to execute `newLine`. Then `newLine` invokes `println`, which causes yet another detour.

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

4.4 Parameters and arguments

Some of the methods we have used require arguments, which are values that 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 print a message, you have to provide the string. So `println` takes a `String` as an argument. Some methods take more than one argument. For example, `Math.pow` takes two `doubles`: the base and the exponent.

When you use a method, you provide the arguments. When you write a method, you list the parameters. The parameter list indicates what arguments are required. Here's a method that takes a string and prints it twice:

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

`printTwice` has a parameter named `s` with type `String`. The parameter name hints that it is a `String`, but you could use any legal variable name. 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`.

```
printTwice("Don't make me say this twice!");
```

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 the compiler error “cannot find symbol,” which might be confusing. Java is looking for a method named `printTwice` that can take an integer. Since there isn’t one in the current class, Java can’t find such a “symbol.”

There are some apparent exceptions to this rule, because sometimes Java converts from one type to another automatically. For example, `Math.sqrt` requires a `double` value. If you run `Math.sqrt(25)`, the integer value 25 is automatically converted to the floating-point value 25.0.

`System.out.println` can accept any type as an argument. But generally speaking, that is an exception; most methods are not so accommodating.

4.5 Stack diagrams and scope

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

```
public class PrintTwice {  
  
    public static void printTwice(String s) {  
        System.out.println(s);  
        System.out.println(s);  
    }  
  
    public static void main(String[] args) {  
        String argument = "Never say never.";  
        printTwice(argument);  
    }  
}
```

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

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

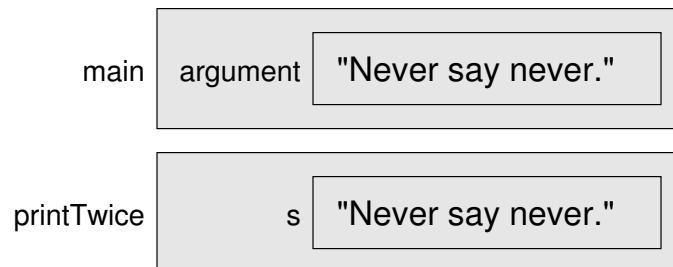


Figure 4.1: Stack diagram for the `PrintTwice` program.

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 value of each variable is drawn inside a box with the name of the variable beside it. (Note for people who know too much: this diagram leaves out a detail we will explain later.)

Stack diagrams help you to visualize the **scope** of a variable, which is the area of a program where a variable exists. It's possible to have two variables

with the same name in two different methods. Each only exists within its own method, so they don't interfere with each other.

4.6 Tracing with a debugger

Keeping track of variables and methods on paper is a useful skill, and you should practice drawing stack diagrams. Another way to visualize the scope of variables and the flow of execution is to use a **debugger**. The overall process is the same, regardless which development environment you use:

1. Set **breakpoints** on lines where you want the program to pause.
2. Step through the code one line at a time and watch what it does.
3. Check the values of variables and see when they change.

For example, open any program in DrJava and move the cursor to the first line of `main`. Press `Ctrl+B` to toggle a breakpoint on the current line; it should now be highlighted in red. Press `Ctrl+Shift+D` to turn on Debug Mode; a new pane should appear at the bottom of the window. (These commands are also available from the *Debugger* menu, in case you forget the shortcut keys.)

When you the your program, execution pauses at the first breakpoint. The debug pane shows the **call stack**, with the current method on top of the stack. You might be surprised to see how many methods were called before the `main` method! To the right are several buttons that allow you to step through the code at your own pace. You can also press “Automatic Trace” to watch DrJava run your code one line at a time.

When the program is paused, you can examine (or even change) the value of any variable using the Interactions Pane. This feature allows you to verify your assumptions about how data is passed from one method to another. You can edit your code while debugging it, but the changes won't take effect until after you compile. The result can be confusing, so we don't recommend it.

Using a debugger is like having the computer proofread your code out loud. You might expect the code do one thing, and then the debugger shows it doing something else. At that moment, you gain insight about what may be wrong with the code.

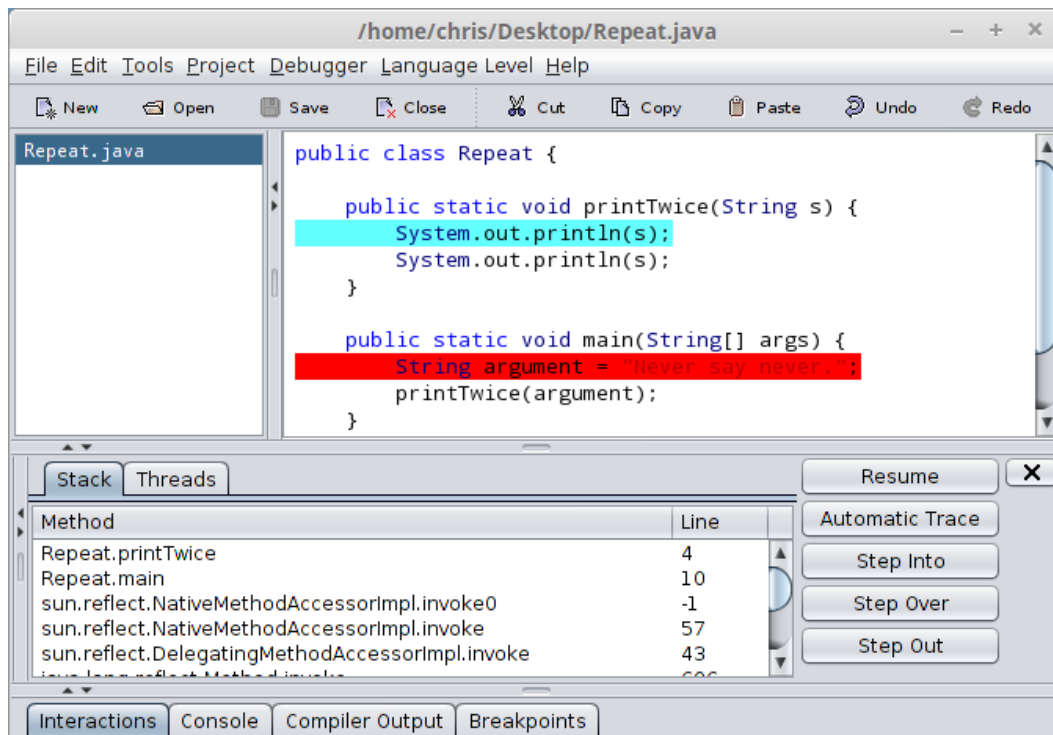


Figure 4.2: Screenshot of the DrJava debugger. Execution is currently paused on the first line of `printTwice`. There is a breakpoint on the first line of `main`.

4.7 Multiple parameters

The syntax for declaring and invoking methods with multiple parameters is a common source of confusion. For one, you have to declare the type of every parameter separately.

```
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 parameter lists. Another common source of confusion is that you do not have to declare the types of arguments. The following is incorrect:

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

In this case, Java can tell the type of `hour` and `minute` by looking at their declarations. It is unnecessary (and therefore not allowed) to include the type when you pass them as arguments. The correct syntax is:

```
printTime(hour, minute);
```

4.8 A few more details

Now that you've had some experience with `double` and `Scanner`, there are a few unexpected behaviors we want to warn you about.

4.8.1 Rounding errors

On hardware, floating-point numbers are only approximately correct. Some numbers, like reasonably-sized integers, can be represented exactly. But repeating fractions, like $1/3$, and irrational numbers, like π , cannot. To represent these numbers, computers have to round off to the nearest floating-point number. The difference between the number we want and the floating-point number we get is called **rounding error**.

For example, the following two statements should be equivalent:

```
System.out.println(0.1 * 10);
System.out.println(0.1 + 0.1 + 0.1 + 0.1 + 0.1
                  + 0.1 + 0.1 + 0.1 + 0.1 + 0.1);
```

But on many machines, the output is:

```
1.0
0.9999999999999999
```

The problem is that 0.1 , which is a terminating fraction in base 10, is a repeating fraction in base 2. So its floating-point representation is only approximate. When we add up the approximations, the rounding errors accumulate.

For many applications, like computer graphics, encryption, statistical analysis, and multimedia rendering, floating-point arithmetic has benefits that outweigh the costs. But if you need *absolute* precision, use integers instead. For example, consider a bank account with a balance of \$123.45:

```
double balance = 123.45; // potential rounding error
```

In this example, balances will become inaccurate over time as the variable is used in arithmetic operations like deposits and withdrawals. The result would be angry customers and potential law suits. You can avoid the problem by representing the balance as an integer:

```
int balance = 12345; // total number of cents
```

This solution works as long as the number of cents doesn't exceed the largest integer, which is about 2 billion. If necessary you can use `long` instead, which has a max value of $2^{63} - 1$ (about 92 quadrillion dollars). Hopefully nobody will ever need that much money!

4.8.2 The Scanner bug

Consider a simple program that asks users for their name and age. Somewhere in the middle of the code, we have the following lines:

```
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 of the `printf` statement looks something like this:

```
Hello Darth Vader, 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 the above example. It doesn't let you input your name and immediately displays the output:

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

To understand what is happening, recall that computers do not *see* input as multiple lines as we do. Instead, the operating system simply forwards a stream of characters to your program via `System.in`:

4	5	\n	D	a	r	t	h		V	a	d	e	r	\n
↑														

The up-arrow represents the next character to be read by `Scanner`. When you call `nextInt`, it will read characters until a non-digit is found.

4	5	\n	D	a	r	t	h		V	a	d	e	r	\n
		↑												

At this point, `nextInt` returns the `int` value 45. The program then asks "What is your name? " and calls `nextLine`. `Scanner` will read characters until a newline is found. Since the next character to be read already is a newline, `nextLine` returns the empty string "".

To solve this problem, you need to add an extra call to `nextLine` after you call `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, then you read the rest of the line (which is just a newline character). Note that you do not have to assign the return value of `nextLine` to a variable in that case; you can simply ignore it.

4.9 Vocabulary

method: A named sequence of statements that performs a procedure or function. Methods may or may not take parameters, and may or may not return a value.

invoke: To call a method, i.e., cause it to execute.

parameter: A piece of information a method requires before it can run. Parameters are variables: 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.

composition: The ability to combine simple expressions and statements into compound expressions and statements, making it possible to use intermediate computations as arguments.

order 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.

stack diagram: A memory diagram that shows which variables belong to which methods at a certain point in the program. The methods calls are “stacked” from top to bottom, in the order of execution.

frame: A structure (represented by a box in stack diagrams) that contains a method’s parameters and variables.

scope: The area of a program where a variable exists.

debugger: A tool that allows you to run one statement at a time and see the contents of variables.

breakpoint: A line of code where the debugger will pause a running program.

call stack: The history of method calls and where to resume execution after each method returns.

rounding error: The small difference between a floating-point number and its actual representation on computer hardware.

4.10 Exercises

Exercise 4.1 What is the difference between a variable and a method? In terms of their syntax, how does the Java compiler tell the difference between the two?

Exercise 4.2 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.

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.

```
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 4.3 Draw a stack diagram that shows the state of the program in Section 4.7 when `main` invokes `printTime` with the arguments 11 and 59.

Exercise 4.4 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 `zoo1` that takes three parameters: an `int` and two `Strings`.
2. Write a line of code that invokes `zoo1`, passing as arguments the value 11, the name of your first pet, and the name of the street you grew up on.

Exercise 4.5 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 prints 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 16, 2011
```

3. Once you have debugged `printAmerican`, write another method called `printEuropean` that prints the date in European format.

Index

- address, 35, 49
- algorithm, 2, 15
- ambiguity, 6
- argument, 51, 57, 65
- arithmetic
 - floating-point, 62
- assignment, 23, 32
- braces, 8
- breakpoint, 60, 65
- bug, 2, 15
- byte code, 4, 15
- call stack, 60, 65
- case-sensitive, 56
- Checkstyle, 31
- class, 16, 56
 - definition, 7
 - Math, 51
 - name, 8
 - Scanner, 36
 - System, 35
 - Time, 61
 - utility, 36
- command-line, 9, 16
- comment, 16
- comments
 - documentation, 46
 - inline, 8
- compile, 15
- compiler, 4
- composition, 29, 32, 53, 65
- computer science, 2, 14
- concatenate, 28, 32
- constant, 40, 49
- debugger, 60, 65
- debugging, 2, 15
 - experimental, 14
- declaration, 22, 31
- degrees, 52
- diagram
 - stack, 58
- divisible, 43
- division
 - integer, 26, 27
- documentation, 45, 50
- documentation comments, 46
- double (floating-point), 27
- Doyle, Arthur Conan, 14
- DrJava, 9
- error
 - logic, 21
 - message, 19
 - runtime, 20
 - syntax, 19
- escape sequence, 13, 16
- exception, 20
- executable, 4, 15
- experimental debugging, 14
- expression, 25, 32, 51, 53
- extract digits, 43

- final, 40
- floating-point, 27, 32
- formal language, 5, 15
- format specifier, 41, 49
- frame, 59, 65
- Google style, 30
- grammar, 5, 16
- Greenfield, Larry, 14
- hello world, 7
- high-level language, 3, 15
- Holmes, Sherlock, 14
- import, 37, 49
- initialize, 40, 49
- integer division, 26, 27
- interpret, 15
- interpreter, 3
- invoke, 65
- Javadoc, 46, 49
- JDK, 9
- keyword, 32
- language
 - formal, 5
 - high-level, 3
 - low-level, 3
 - natural, 5
 - programming, 5
- library, 36, 49
- Linux, 14
- literal, 40, 49
- literalness, 6
- logic error, 21, 31
- low-level language, 3, 15
- magic number, 39, 49
- main, 8, 53
- Math class, 51
- memory diagram, 24
- method, 16, 51, 56, 65
 - definition, 7, 53
 - main, 53
 - multiple parameter, 61
- modulus, 43, 49
- natural language, 5, 15
- newline, 11, 16
- object, 35, 49
- object code, 4, 15
- operator, 25, 32
 - cast, 42
 - modulus, 43
 - string, 28
- order of execution, 56, 65
- order of operations, 29
- package, 35, 49
- parameter, 54, 57, 65
 - multiple, 61
- parameter passing, 58, 65
- parse, 5, 16
- poetry, 6
- portable, 3, 15
- precedence, 29, 32
- print, 11
- print statement, 16
- printf, 40
- println, 8
- problem-solving, 1, 14
- program, 1, 6, 14
- programming, 2, 14
- programming language, 5, 15
- prose, 6
- prototype, 45, 49
- public, 7, 53

-
- radians, 52
 - readability, 30
 - redirection operator, 48, 50
 - redundancy, 6
 - rounding error, 62, 65
 - runtime error, 20, 31
 - Scanner class, 36
 - scope, 65
 - semantics, 5, 16
 - source code, 4, 15
 - squiggly braces, 8
 - stack diagram, 58, 65
 - statement, 8, 16
 - assignment, 23
 - comment, 8
 - declaration, 22
 - import, 37
 - print, 8, 11
 - static, 7
 - String, 11
 - string, 16
 - string operator, 28
 - syntax, 5, 16
 - syntax error, 19, 31
 - System class, 35
 - terminal, 9
 - token, 5, 16
 - Torvalds, Linus, 14
 - type, 32
 - char, 22
 - double, 27
 - int, 22
 - long, 52
 - String, 11, 22
 - void, 53
 - type cast, 42, 49
 - utility class, 36, 49
 - value, 22, 31
 - variable, 22, 31
 - void, 53
 - whitespace, 30, 32

