

Notices

This is the free HTML version of Think [Java](#) 2e. Print and electronic versions are available from [Bookshop.org](#) and [Amazon](#) (affiliate links).

We are working on making this site accessible in compliance with [Web Content Accessibility Guidelines](#). If you have suggestions for improvements, please [file an issue](#).

Are you using one of our books in a [class](#)? We'd like to [know](#) about it. Please consider filling out [this short survey](#).

2 Variables and Operators

This chapter describes how to [write](#) statements using *variables*, which store [values](#) like numbers and words, and *operators*, which are symbols that perform a computation. We also explain three kinds of programming [errors](#) and offer additional debugging advice.

To run the examples in this chapter, you will [need](#) to [create](#) a new [Java class](#) with a [main](#) method (see Section [1.3](#)). Throughout the book, we often omit [class](#) and method definitions to [keep](#) the examples concise.

2.1 Declaring 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 in memory that stores a **value**. [Values](#) [may](#) be numbers, text, images, sounds, and other types of data. To store a value, you first have to declare a variable:

```
String message;
```

This statement is called a **declaration**, because it declares that the variable `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 like `1` and `-5`, and the `char` type can store characters like `'A'` and `'z'`.

Some types begin with a capital letter and some with lowercase. You 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`.

To declare an integer variable named `x`, you simply type this:

```
int x;
```

Note that `x` is an arbitrary name for the variable. In general, you [should](#) use names that indicate what the variables mean:

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

This example declares two variables with type `String` and two with type `int`. The last line shows how to declare multiple variables with the same type: `hour` and `minute` are both integers. **Note** that each declaration statement ends with a semicolon (`;`).

Variable names usually begin with a lowercase letter, in contrast to **class names** (like `Hello`) that **start with a capital letter**. **When** a variable name contains more than one word (like `firstName`), it is conventional to capitalize the first letter of each subsequent word. Variable names are case-sensitive, so `firstName` is **not** the same as `firstname` or `FirstName`.

You can use any name you want for a variable. **But** there are about 50 reserved words, called **keywords**, that you are **not** allowed to use as variable names. These words **include** `public`, `class`, `static`, `void`, and `int`, which are used by the compiler to analyze the structure of the program.

You can see the full **list** of keywords (<https://thinkjava.org/keywords>), **but** you don't have to memorize them. Most programming editors provide "syntax highlighting", which makes different parts of the program appear in different colors. And the compiler will complain even **if** one **does** sneak past you and your editor.

2.2 Assigning Variables

Now that we have declared some variables, we can use them to store **values**. We do that with an **assignment** statement:

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

This example shows three assignments, and the comments 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 update its value.

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 show 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
```

Variables **must** be **initialized** (assigned for the first **time**) before they can be used. You can declare a variable and then assign a value later, as in the previous example. You can also declare and initialize on the same line:

```
String message = "Hello!";
int hour = 11;
int minute = 59;
```

2.3 Memory Diagrams

Because **Java** uses the `=` symbol for **assignment**, it is tempting to interpret the statement `a = b` as a statement of equality. It is **not**!

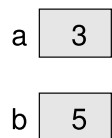
Equality is commutative, and **assignment** is **not**. For example, in mathematics if $a = 7$, then $7 = a$. In **Java** `a = 7`; is a legal **assignment** statement, **but** `7 = a`; is **not**. The left side of an **assignment** statement has to be a variable name (storage location).

Also, in mathematics, a statement of equality is true for all **time**. If $a = b$ now, a is **always** equal to b . In **Java**, an **assignment** statement can make two variables equal, **but** they don't have to stay that way:

```
int a = 5;
int b = a;    // a and b are now equal
a = 3;        // a and b are no longer equal
```

The third line changes the value of `a`, **but** it **does not** change the value of `b`, so they are **no** longer equal.

Taken together, the variables in a program and their current **values** make up the program's **state**. Figure 2.1 shows the state of the program after these **assignment** statements run.



Memory diagram of the variables `a` and `b`.

Diagrams like this one that show the state of the program are called **memory diagrams**. Each variable is represented with a box showing the name of the variable on the outside and its current value inside.

As the program runs, the state of memory changes, so memory diagrams show **only** a particular point in **time**. For example, if we added the line `int c = 0`; to the previous example, the memory diagram would look like Figure 2.2.

a 3

b 5

c 0

Memory diagram of the variables `a`, `b`, and `c`.

2.4 Printing Variables

You can display the current value of a variable by using `print` or `println`. The following statements declare a variable named `firstLine`, assign it the value `"Hello, again!"`, and display that value:

```
String firstLine = "Hello, again!";  
System.out.println(firstLine);
```

When we talk about displaying a variable, we generally mean the *value* of the variable. To display the *name* of a variable, you have to put it in quotes:

```
System.out.print("The value of firstLine is ");  
System.out.println(firstLine);
```

For this example, the output is as follows:

```
The value of firstLine is Hello, again!
```

Conveniently, the code for displaying a variable is the same regardless of its type. For example:

```
int hour = 11;  
int 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 shown here:

```
The current time is 11:59.
```

To output multiple *values* on the same line, it's common to use several `print` statements followed by `println` at the end. **But** don't forget the `println`! On many computers, the output from `print` is stored without being displayed until `println` is run; then the entire line is displayed at once. **If** you omit the `println`, the program might display the stored output at unexpected times or even terminate without displaying anything.

2.5 Arithmetic Operators

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

The following program converts a **time** of day to minutes:

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

The output is as follows:

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

In this program, `hour * 60 + minute` is an **expression**, which represents a single value to be computed (`719`). **When** the program runs, each variable is replaced by its current value, and then the operators are applied. The **values** that operators work with are called **operands**.

Expressions are generally a combination of numbers, variables, and operators. **When** compiled and executed, they become a single value. For example, the expression `1 + 1` has the value `2`. In the expression `hour - 1`, **Java** replaces the variable with its value, yielding `11 - 1`, which has the value `10`.

In the expression `hour * 60 + minute`, both variables get replaced, yielding `11 * 60 + 59`. The multiplication happens first, yielding `660 + 59`. Then the addition yields `719`.

Addition, subtraction, and multiplication all do what you expect, **but** you might be surprised by division. For example, the following fragment tries 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);
```

The output is as follows:

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

This result often confuses people. The value of `minute` is `59`, and 59 divided by 60 **should** be 0.98333, **not** 0. The **problem** is that **Java** performs *integer division* **when** the operands are integers. By design, integer division **always** rounds toward zero, even in cases like this one where the next integer is close.

As an **alternative**, we can 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 as follows:

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

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

2.6 Floating-Point Numbers

A more general solution is to use **floating-point** numbers, which represent **values** with decimal places. 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 the same way we did for the other types:

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

Java performs *floating-point division* **when** one or more operands are **double values**. So we can solve the **problem** from the previous section:

```
double minute = 59.0;  
System.out.print("Fraction of the hour that has passed: ");  
System.out.println(minute / 60.0);
```

The output is shown here:

```
Fraction of the hour that has passed: 0.9833333333333333
```

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 = 1.1; // compiler error
```

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

```
double y = 1; // legal, but bad style
```

The preceding 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 = 1 / 3; // common mistake
```

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 expression on the right divides two integers, so **Java** **does** integer division, which yields the `int` value `0`. Converted to `double`, the value assigned to `y` is `0.0`.

One way to solve this **problem** (once you figure out the bug) is to make the right-hand side a floating-point expression. The following **sets** `y` to `0.333333`, as expected:

```
double 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 simple mistake will come back and haunt you.

2.7 Rounding Errors

Most 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 as follows:

```
1.0  
0.9999999999999999
```

The **problem** is that `0.1` is a repeating fraction **when** converted into binary. So its floating-point representation stored in memory 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 lawsuits. 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 `int`, which is about 2 billion.

2.8 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 works with strings, **but** it might **not** do what you expect. For strings, the `+` operator performs **concatenation**, which means joining end-to-end. So `"Hello, " + "World!"` yields the string `"Hello, World!"`.

Likewise **if** you have a variable called `name` that has type `String`, the expression `"Hello, " + name` appends the value of `name` to the hello string, which creates 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 `3`, and `3 + "Hello"` is `"3Hello"`. **But** in the second line, `"Hello" + 1` is `"Hello1"`, and `"Hello1" + 2` is `"Hello12"`.

When more than one operator appears in an expression, they are evaluated according to the **order of operations**. Generally speaking, `Java` evaluates operators from left to right (as you saw in the previous section). **But** for numeric operators, `Java` follows mathematical conventions:

- Multiplication and division take “precedence” over addition and subtraction, which means they happen first. So `1 + 2 * 3` yields 7, **not** 9, and `2 + 4 / 2` yields 4, **not** 3.
- **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; **if** the value of `minute` is 59, we get `5900 / 60`, which yields 98. **If** these same operations had gone from right to left, the result would have been `59 * 1`, which is incorrect.
- Anytime you want to override the **order** of operations (or you are **not** sure what it is) you can use parentheses. Expressions in parentheses are evaluated first, so `(1 + 2) * 3` is 9. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even though it doesn't change the result.

See the official `Java` tutorials for a complete table of operator precedence (<https://thinkjava.org/operators>). **If** the **order** of operations is **not** obvious **when** looking at an expression, you can **always** add parentheses to make it

more clear. **But** over **time**, you **should** internalize these kinds of details about the **Java** language.

2.9 Compiler **Error** Messages

Three kinds of **errors** can occur in a program: compile-**time errors**, run-**time errors**, and logic **errors**. It is useful to distinguish among them in **order** to track them down more quickly.

Compile-time errors occur **when** you **violate** the **rules** of the **Java** language. For example, parentheses and braces have to come in matching pairs. So `(1 + 2)` is legal, **but** `8)` is **not**. In the latter case, the program **cannot** be compiled, and the compiler displays a “syntax **error**”.

Error messages from the compiler usually indicate where in the program the **error** occurred. **Sometimes** they can tell you exactly what the **error** is. As an example, let’s get back to the Hello World program from Section [1.3](#):

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

If you forget the semicolon at the end of the print statement, you might get an **error** message like this:

```
File: Hello.java [line: 5]  
Error: ';' expected
```

That’s pretty good: the location of the **error** is correct, and the **error** message tells you what’s **wrong**. **But error** messages are **not always** easy to understand. **Sometimes** the compiler reports the place in the program where the **error** was *detected*, **not** where it actually occurred. And **sometimes** the description of the **problem** is more confusing than helpful.

For example, **if** you forget the closing brace at the end of **main** (line 6), you might get a message like this:

```
File: Hello.java [line: 7]  
Error: reached end of file while parsing
```

There are two problems here. First, the **error** message is written from the compiler’s point of view, **not** yours. **Parsing** is the process of reading a program before translating; **if** the compiler gets to the end of the file **while still** parsing, that means something was omitted. **But** the compiler doesn’t **know** what. It also doesn’t **know** where. The compiler discovers the **error** at the end of the program (line 7), **but** the missing brace **should** be on the previous line.

Error messages contain useful information, so you **should** make an effort to read and understand them. **But** don’t take them too literally. During the first few weeks of your programming career, you will probably spend a lot of

time tracking down syntax and other compile-**time errors**. As you gain experience, you will make fewer mistakes and find them more quickly.

2.10 Other Types of **Errors**

The second type of **error** is a **run-time error**, so-called because it **does not** appear until after the program has started running. In **Java**, these **errors** occur **while** the interpreter is executing byte code and something goes **wrong**. These **errors** are also called “exceptions” because they usually indicate that something unexpected has happened.

Run-**time 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. **When** a run-**time error** occurs, the program “crashes” (terminates) and displays an **error** message that explains what happened and where. For example, **if** you accidentally divide by zero, you will get a message like this:

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

Error messages are very useful for debugging. The first line includes the name of the **exception**, **ArithmeticException**, and a message that indicates more specifically what happened, division by zero. The next line shows the method where the **error** occurred; **Hello.main** indicates the method **main** in the **class** **Hello**. It also reports the file where the method is **defined**, **Hello.java**, and the line number where the **error** occurred, **5**.

The third type of **error** is a **logic error**. **If** your program has a logic **error**, it will compile and run without generating **error** messages, **but** it will **not** do the right thing. Instead, it will do exactly what you told it to do. For example, here is a version of the Hello World program with a logic **error**:

```
public class Hello {

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

This program compiles and runs just fine, **but** the output is as follows:

```
Hello,
World!
```

Assuming that we wanted the output on one line, this is **not** correct. The **problem** is that the first line uses **println**, **when** we probably meant to use **print** (see the “Goodbye, cruel world” example of Section [\[goodbye\]](#)).

Identifying logic **errors** can be **hard** because you have to work backward, looking at the output of the program, trying to figure out why it is doing the **wrong** thing, and how to make it do the right thing. Usually, the compiler and the interpreter can't help you, since they don't **know** what the right thing is.

2.11 Vocabulary

variable:

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

value:

A number, string, or other data that can be stored in a variable. Every value belongs to a type (e.g., **int** or **String**).

type:

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

declaration:

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

keyword:

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

assignment:

A statement that gives a value to a variable.

initialize:

To assign a variable for the first **time**.

state:

The variables in a program and their current **values**.

memory diagram:

A graphical representation of the state of a program at a point in **time**.

operator:

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

operand:

One of the **values** on which an operator operates. Most operators in **Java** **require** two operands.

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 numbers with an integer part and a fractional part. In `Java`, the default floating-point type is `double`.

rounding error:

The difference between the number we want to represent and the nearest floating-point number.

concatenate:

To join two `values`, often strings, end to end.

order of operations:

The `rules` that determine in what `order` expressions are evaluated. Also known as “operator precedence”.

compile-time error:

An `error` in the source code that makes it `impossible` to compile. Also called a “syntax `error`”.

parse:

To analyze the structure of a program; what the compiler `does` first.

run-time error:

An `error` in a program that makes it `impossible` to run to completion. Also called an “`exception`”.

logic error:

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

2.12 Exercises

The code for this chapter is in the `ch02` directory of *ThinkJavaCode2*. See [page](#) for instructions on how to download the repository. Before you start the exercises, we recommend that you compile and run the examples.

If you have `not` already read Appendix [18.2](#), now might be a good `time`. It describes the DrJava Interactions pane, which is a useful way to develop and `test` short fragments of code without writing a complete `class` definition.

Exercise

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 looks 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

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 2.4, **write** a program that creates variables named **day** , **date** , **month** , and **year** . The variable **day** will contain the day of the week (like Friday), and **date** will contain the day of the month (like the 13th). Assign **values** to those variables that represent today's date.
3. Display 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. Compile and run your program before moving on.
4. Modify the program so that it displays the date in standard American format; for example: **Thursday, July 18, 2019** .
5. Modify the program so it also displays the date in European format. The final output **should** be as follows:

```
American format: Thursday, July 18, 2019
European format: Thursday 18 July 2019
```

Exercise

Exercise 2.3. The point of this exercise is to (1) use some of the arithmetic operators, and (2) 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 2.4, **create** variables named **hour** , **minute** , and **second** . Assign **values** that are roughly the current **time**. Use a 24-hour clock so that at 2:00 PM the value of **hour** is **14** .
3. Make the program calculate and display the number of seconds since midnight.
4. Calculate and display the number of seconds remaining in the day.
5. Calculate and display 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**. Then **write** code to compute the elapsed **time** since you started working on this exercise.

****Hint:** You might want to use additional variables to hold values during the computation. Variables that are used in a computation but never displayed are sometimes called “intermediate” or “temporary” variables.*