

Modern Computer Programming -- An Expressions First Approach with Java (working title)

**Note that this is an incomplete
preprint. Do not distribute. (c)
2018 Joseph Kendall-Morwick**

**Modern Computer Programming -- An Expressions First Approach
with Java (working title): Note that this is an incomplete preprint.
Do not distribute. (c) 2018 Joseph Kendall-Morwick**

Table of Contents

1. Evaluating Simple Expressions	1
Read Evaluate Print Loops (REPLs)	1
Expressions	2
Literal Expressions and Data Types	2
Floating Point Values	3
Casting	3
Arithmetic Expressions	3
Variables	5
Method Calls	6
Static Imports	6
Method Call Expressions	6
2. Forming Simple Programs from Statements	8
Algorithms	8
Statements	9
Declaration Statements	9
Assignment Statements	10
Scripts	12
JShell Scripts	12
Inline Comments	13
3. Object Data Types and Strings	16
Object Types	16
Classes, Packages, and Import Statements	16
Constructor Calls	17
Object Type Variables	17
Instance Method Calls	18
Object Fields	18
The Character Primitive Data Type	18
Methods for Characters	19
Strings	19
String Literals	19
String Concatenation	20
String Instance Methods	20
Standard Input / Output	21
Reading from Standard Input	21
Reading in Other Datatypes	22
4. Decision Making in Imperative Programs	24
Method Definitions	24
Booleans	26
Comparison Operators	26
Boolean Operators	27
5. Conditional Statements	28
Conditional Statements	28
If Statements	28
Else Statements	29
Nested If Statements	30
if/else Chains	30
Switch Statements	31
6. Loops	33
While Loops	33
For Loops	33

List of Figures

1.1. Using the JShell REPL	2
1.2. Anatomy of a Method Call	7
2.1. Anatomy of a Declaration Statement	9
2.2. Anatomy of an Assignment Statement	10
2.3. Anatomy of an Initialization Statement	11
2.4. Anatomy of an Initialization Statement with Type Inference	11
3.1. Anatomy of a Constructor Method Call	17
3.2. Anatomy of an Instance Method Call	18
4.1. Anatomy of a Method Definition	24
5.1. Anatomy of a Simple if Statement	28
5.2. Anatomy of an if Statement with a Block Body	28
5.3. Anatomy of an if / else Statement	29
6.1. Anatomy of a While Loop	33
6.2. Anatomy of a For Loop	34

List of Examples

1.1. Averaging Several Integers	4
1.2. Calling the Square Root Method	6
1.3. Revisiting Averaging Several Integers	7
2.1. Averaging Algorithm	8
2.2. Declaring a Variable and Using it in an Expression	9
2.3. Assigning a Value to a Variable	10
2.4. Declaring and Initializing a Variable with a Single Statement	11
2.5. Averaging Script	12
2.6. Adding End-of-Line Comments to the Averaging Example	14
3.1. Basic Concatenation	20
3.2. Concatenating non-Strings	20
3.3. Finding the Length of a String	20
3.4. <code>char String.charAt(int)</code>	21
3.5. <code>String String.substring(int,int)</code>	21
4.1. Square a Number	25
4.2. Add Two Numbers	25
4.3. Ultimate Answer	26
5.1. Implementing Absolute Value with <code>if/else</code>	29
5.2. Nested <code>if</code> Statements to Determine Best Temperature	30
5.3. Ice Cream Choices with <code>if/else</code> Chains	30
5.4. Switching Over Letter-Grade Characters	32
6.1. Counting with a While Loop	33
6.2. Totaling Numbers with a For Loop	34
6.3. Iterate Through a String	34

Chapter 1. Evaluating Simple Expressions

Developing computer programs in a programming language such as Java has its parallels in writing a book in a natural language like English. Both natural languages and programming languages have rules of *syntax* that govern how different elements of the language can be joined together. In the case of English, these are the rules for putting words together in a sentence that you may have studied in an English grammar class. For instance, the prior sentence in this paragraph is well-formed (it follows the rules of proper English grammar), but sentence this one not because rules syntax properly follow not.

That last sentence may have taken you a moment to understand, but I expect that you did. This is the principal difference between a natural language and a programming language: the latter doesn't make any sense at all to the computer unless it is well-formed (strictly follows all rules of syntax). Thus, it's important to have a good foundation of understanding for what these rules are in Java before diving in to writing complex programs. The good news is that some of the more simple syntax in Java, namely expressions, are also considerably powerful, especially when used with a REPL (such as the recently release Java REPL JShell). We'll dive in to what a REPL is, how to use a REPL, and several types of expressions in this first chapter.

Read Evaluate Print Loops (REPLs)

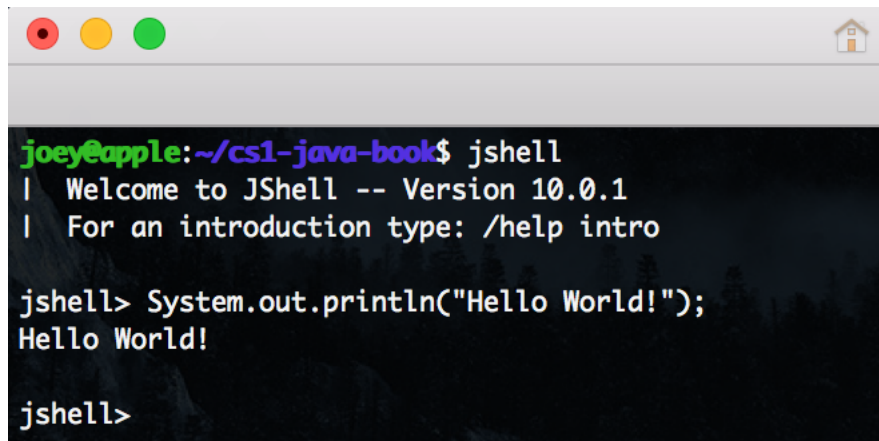
There are many ways to categorize programming languages. For instance, some programming languages are called *object-oriented*, *functional*, or *statically typed*, to name a few, and many of these categories overlap. An important category, as it relates to teaching Java to a beginning programmer, is that of *scripting languages*. This is important because Java has never in the past been considered one¹, and this has led to problems when introducing it to someone who has never programmed before.

A scripting language is one that is intended for quickly writing short, ad-hoc programs that perform a simple task, possibly only once. An important tool for working with a scripting language is a REPL (Read Evaluate Print Loop). A REPL allows you to run small snippets of a program, one at a time, and observe what the computer does in real time. This way the programmer can quickly react, correct errors, and experiment. This functionality is of obvious importance to a new programmer learning how to use the language.

Java, for a very long time, was infamous for its *wordiness*. That is, it took a lot of typing to say anything meaningful. For example, the first program most introductory programming books show the reader is *Hello World!* -- a program that simply prints these words to the screen. In Java, this required the programmer to write 5 lines of code. This level of wordiness is essentially the opposite of the goal of a scripting language.

This changed somewhat in September 2017 when an official REPL was added to Java, called JShell. In JShell, *Hello World!* is just one line:

¹Do not confuse Java with Javascript, which is an entirely different language and is indeed a scripting language.

Figure 1.1. Using the JShell REPLA screenshot of a terminal window with a dark background. The window title bar shows three colored circles (red, yellow, green) and a home icon. The terminal text is as follows:

```
joe@apple:~/cs1-java-book$ jshell
| Welcome to JShell -- Version 10.0.1
| For an introduction type: /help intro

jshell> System.out.println("Hello World!");
Hello World!

jshell>
```

JShell is an example of a command-line program run from the system terminal. You can see in the figure that the first thing JShell did after I started it up was to welcome me and indicate what version is running. After that you see the *command prompt*.

```
jshell>
```

At that prompt I typed `System.out.println("Hello World!");` which is a snippet of Java that instructs the computer to print the same message back to me. JShell read this snippet in from me (the R for read in REPL), then did as it was instructed (the E for evaluate and P for Print steps). You can see the JShell REPL complying with that command below and then returning me to the command prompt for another command (L for loop -- after each command, it waits for another, looping indefinitely).

There are many commands you can enter at the command prompt that are not Java code snippets but are unique to using JShell. A very important one is the `/help` command. Typing in this command causes JShell to report on all the other special commands available to you. Another important command is `/exit`, which terminates JShell. We will see other commands as they become relevant in later sections.

Expressions

Expressions are the most basic syntactic building blocks of a computer program. Think of an expression as a simple question that you're asking of the computer. The answer you get back from the computer is called a *value*, and the process of determining the value for an expression is called *evaluation*.

You've worked with expressions before if you've ever used a calculator. For instance, an expression you might feed in to a calculator is `2 + 3`. The answer the calculator would give you back is 5. In this case, the calculator read the expression `2 + 3`, evaluated the expression by performing addition, and returned the resulting value 5 back to you. This is similar to how expressions are used in programming languages, with the difference being that there are many more types of expressions in Java than are likely supported by your calculator!

Literal Expressions and Data Types

Literal expressions are the most simple expressions in Java. A *literal expression* is one that directly represents the value it evaluates to. For instance, the expression 5 evaluates to, wait for it..., 5. Other examples of literal expressions are 21, 90000 (note, no commas), or 0.

One catch is that there are more than one type of numeric value in Java! The type of a value is called a *data type* (frequently abbreviated to *type*). Types are simply a collection of possible values that a given value is restricted to. For example, 5 is an example of what is called an integer (shortened to `int` in Java). There are some constraints on the size of an integer (the minimum value is -2^{31} and the maximum value is $2^{31}-1$ or 2147483647). Considering this rule, the expression

2147483648 would be a syntax error, since it is too large to be an int. Note that integer values also have no fractional component. For instance, 0.5 is not a legal integer value either.

There are also other integer types with different ranges of values. For instance, the type long has a range of -2^{63} to 2^{63} . This type of value is often used when a number is too large to be represented as an int. Note that you can specify you want to create a long value using a special syntax for the literal expression. For instance, 46L is a legal expression that evaluates to the long integer value 46 and 2147483648L is also legal (where 2147483648 was not).

Floating Point Values

9.2 is an example of a floating point value. These are values with a decimal component. Like integer values, there is more than one floating point type in Java (single and double precision). The difference between these types is the precision (essentially the number of significant digits) to which a value will approximate some real number. By default, the literal expressions you type in that contain a decimal point will be double precision (shortened to double).

Internally, these values are stored in the same way you would use scientific notation. Each floating point value consists of two components: an exponent and a mantissa. The value is determined by multiplying the mantissa by 10 to the power of the exponent. In addition to writing out floating point literals with a decimal point, you can also format them by identifying these two components explicitly. Literal values that use the scientific notation format place a letter E between the mantissa and the exponent. For instance, 5E3 is the same as 5000.0.

Casting

Values can also be converted to new values with a different data type by using a *casting* expression. This can be important in situations when it is important to use the advantages or avoid the disadvantages of a particular data type. To cast a value to a new data type, type the name of the data type, surrounded by parentheses, immediately before the expression you want to convert. For example,

```
(double)8
```

evaluates to the value 8.0

Casting a value produces a value that is different from the one you started with, and it may be impossible to cast it back to the original value, depending on the types you're casting between. For instance, if you cast an int to a double, such as we did in the last example, you're expanding the values that can be represented by introducing the fractional component. But if you cast a double to an int, such as in the following expression:

```
(int)5.8
```

, you're narrowing the possible values that can be used. In this case, when casting a double to an int, the fractional component is *truncated*, yielding the value 5. Note that it is not rounded up to the nearest integer.

Arithmetic Expressions

Expressions can be combined together to form a larger, more complex, compound expression. To combine two expressions, a *binary operator* is placed between them. The operator performs some function over the values that the two expressions evaluate to (these values are called *operands*). For instance, + is a binary operator representing the addition operation. 5 is a numeric literal expression, and 5 + 5 is a more complex, compound expression using the addition operator that evaluates to the value 10.

There are several binary arithmetic operators which work over numeric expressions in Java. Each is described below:

- + (the addition operator)
- - (the subtraction operator)
- * (the multiplication operator)
- / (the division operator)
- % (the modulus operator)

Note that, when using the / operator with integers, you get back the integer quotient (rather than a floating point value). Modulus gives you the remainder when performing integer division.

There is also a unary operator in Java. This is an operator that works over only one operand rather than two. This is the negation operator, which, like subtraction, is represented with -. For instance, -5 is a compound expression that is the negation of 5.

Also similar to algebra, note that these operators follow the same order of precedence. Multiplication and division are evaluated before addition and subtraction. For example, $2 * 3 + 1$ is equal to 7, not 8.

You can also use parentheses to force a certain order of evaluation for your expressions, just as you did in algebra. For example, the expression $2 * (3 + 1)$ does, in fact, evaluate to 8.

Example 1.1. Averaging Several Integers

Consider the following arithmetic expression that averages three integer values:

```
(5 + 7 + 8) / 3
```

This expression evaluates to the value 6, since it simplifies to:

```
20 / 3
```

If we wanted the average including the fractional component, we could cast the original sum to a double value, or we could just use a double formatted literal such as 3.0 as the divisor. For instance, evaluating

```
(double)(5 + 7 + 8) / (double)3
```

yields the value 6.666666666666667. Perhaps it's surprising then that our original expression yielded 6 instead of 7. The average should round up to 7, but when performing division with integers, the remainder is simply thrown away. If we wanted to round up to the nearest integer, we could take advantage of casting to do this. Consider the following expression:

```
(int)((double)(5 + 7 + 8) / (double)3 + 0.5)
```

This first simplifies to

```
(int)((double)20 / (double)3 + 0.5)
```

as before, and then

```
(int)(20.0 / 3.0 + 0.5)
```

and next

```
(int)(6.666666666666667 + 0.5)
```

but now we'll add 0.5 to the result of the division, in effect rounding it up and simplifying to

```
(int)(7.166666666666667)
```

All that's remaining now is the cast expression, which yields the value 7. Note that if we changed the last 8 to a 7 in the original expression, making the result of the division `6.333333333333333`, adding `0.5` would only bring it up to `6.833333333333333` and truncating it by converting to an `int` would yield 6, essentially rounding down to the nearest integer.

Variables

Notice that when you type an expression in to the JShell REPL, more than just the resulting value is printed to the screen. For example, when you type the literal expression `5` in to the REPL, this is what you see:

```
jshell> 5
$1 ==> 5
```

The `$1` that you see indicates the name of an automatically generated variable. [Definition: A *variable* is a named portion of computer memory that holds a single value.] In this case, the value 5 is stored in computer memory at a location accessible with the variable name `$1`.

Variables in programming are similar to the idea of a variable you've seen in algebra. In algebra, the variable represents an unknown quantity. Likewise, when you write a program, you may not know what value a variable you're using will eventually hold. However, at the time when your program is running (known as *run time*), the variable's value will be known to the computer, just as the value of a variable in algebra is known when you solve an equation.

When you type an expression in to the REPL, another variable will be automatically generated to hold the value. For instance:

```
jshell> 5
$1 ==> 5
```

```
jshell> 9
$2 ==> 9
```

Variables aren't just used for storing values but can also be used to retrieve values. A *variable expression* is simply the name of a variable and evaluates to the value held by the variable. For example:

```
jshell> 5
$1 ==> 5
```

```
jshell> 9
$2 ==> 9
```

```
jshell> $1
$1 ==> 5
```

```
jshell> $2
$2 ==> 9
```

Note that when a variable expression is evaluated by itself, another new variable is not automatically generated by the REPL to hold the value (that would be redundant).

Variable expressions can also be part of more complex expressions by combining them with operators. For instance:

```
jshell> 5
$1 ==> 5

jshell> 9
$2 ==> 9

jshell> $1 + $2
$3 ==> 14
```

Method Calls

The arithmetic operators we've used thus far are simple. We can combine them together to create more complex operations, but that also makes our code more difficult to read. More complicated operations are often handled by *methods* which put a simple name to a potentially complex operation. This is an example of *abstraction* -- a very important concept in computer programming in which the complexities of operations are hidden to make programs easier to read, modify, and combine with other programs. Methods also allow Java to be *extensible*, in that you can create your own methods to perform custom tasks (we will see how to do this in future chapters).

We formed arithmetic expressions with arithmetic operators. Similarly, we can form *method call* expressions by providing arguments to a method (arguments, in this case, are values derived from other expressions -- similar to the operands we supplied to operators in section 1.3). When the method call expression is evaluated, the method will calculate a value based on the arguments it was given. In this context, the term *function* is also applicable.

Static Imports

In future sections you'll learn how to define your own methods, but initially you'll need to rely on *static methods* that are provided for you. Think of a static method as a *normal* method -- they're easier to define in contrast to special kinds of methods we'll see in later chapters.

To use these static methods, you need to import them from *classes* that store them. To import them, you'll need to use an import statement. Import statements simply begin with the keyword *import*. When we import static methods, we also need to follow that with the keyword *static*. What follows is a package name, class name, and the name of the method we want to import, all separated by periods. Classes and packages will be covered in more depth in the objects section -- for now just understand them as containers of the static methods that Java makes available to you. For example, below is an import statement that allows us to use the `pow` method:

```
import static java.lang.Math.pow;
```

In this case, the package name is `java.lang` and the class name is `Math`. We could import all of the static methods in the `Math` class at the same time if we wished by replacing the method name with a wildcard:

```
import static java.lang.Math.*;
```

This would allow us to use the complete collection of static methods organized in to the `Math` class.

Method Call Expressions

Now that we have some methods available for use, we can make calls to them. Here is an example of a simple function call:

Example 1.2. Calling the Square Root Method

We will see many types of methods in future chapters, but the methods we're calling in this chapter are called *static* methods. In order to use some of the methods that come standard with Java, we should

first import them. Typing in the following line will make several mathematical methods available. The method we'll use in this example is named `sqrt`).

```
import static java.lang.Math.*;
```

Note that `Math` must be capitalized and the line should end in a semicolon. Once you've typed this in, you're allowed to make calls to the square root function and several others at will. For example:

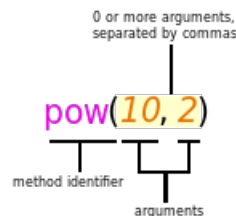
```
sqrt(4)
```

This expression should yield the value `2.0`. Note that the method returns a double value, even though `4` is a perfect square. Try calling this method with some other, messier values.

A list of all the other methods you can use after the import statement above can be found here: <https://docs.oracle.com/javase/10/docs/api/java/lang/Math.html>

To call a method, first give the name of the method you wish to call. Following the name should be a pair of parenthesis, within which is a comma-separated list of expressions for each argument. In the last example, there was only one argument, but a method could take more than one argument, or could even take zero arguments.

Figure 1.2. Anatomy of a Method Call



In the diagram above, we're calling a method named `pow` which takes two arguments: the base and the power. The method returns the result of raising the specified base to the specified power (that is to say, this expression evaluates to the value 100).

Example 1.3. Revisiting Averaging Several Integers

Another method available from the import in Example 1.2 is the `round` method that rounds a floating point value to the nearest integer value. This method abstracts away some of the details of averaging we implemented in Example 1.1. For example, evaluating the expression

```
average(5.9)
```

yields the value 6 whereas evaluating the expression

```
average(5.2)
```

yields the value 5. Note that this is much easier to understand for a programmer than deciphering

```
(int)(5.9 + 0.5)
```

or

```
(int)(5.2 + 0.5)
```

which both perform the same rounding operation.

Chapter 2. Forming Simple Programs from Statements

So far you've only seen small snippets of code (expressions, mostly) that accomplish something very simple. Clearly we use computers and computer programs to accomplish much more complex tasks than those we've accomplished thus far in JShell. Of course, we could build up very large and complex expressions that accomplish more complex tasks. This style of programming is called *functional* programming. While Java does have some support for this programming paradigm, more so Java (like most popular programming languages) is an *imperative* programming language. This means that Java programs, for the most part, follow a series of discrete steps, each of which updates the *state of the program* (perhaps the values of variables, the contents of the screen, etc).

For a beginning programmer, the imperative programming paradigm will be more familiar. You've followed what amounts to an imperative program whenever you've been given a list of instructions. In fact, such an abstract list of instructions has a name in math and computer science: an algorithm. Algorithms are important to programmers because they reveal, more abstractly, how part of a program can or should work. Using algorithms doesn't depend on knowing Java or Python or any other specific language; they are written in English (or another natural language) in a way that any programmer should understand. The programmer can then use the algorithm to guide them through the process of developing the code for a portion of their program.

This chapter will start with a closer look at algorithms. Then you will learn how to use statements (essentially a single *step* in an imperative program). Finally, you'll learn how to develop your own simple but complete computer programs as JShell scripts. Note that JShell scripts are not the only way to develop a complete program in Java, and in fact they are relatively new to Java, but they are the most simple way for a beginner to write their first program. We'll also take a look at Java applications in later chapters.

Algorithms

[Definition: An *Algorithm* is a list of abstract instructions for solving a problem.] These instructions need not be computer codes -- an algorithm, as a concept, transcends any one particular programming language.

Algorithms are often used to describe how to perform a complex mathematical operation by breaking it down in to a number of smaller steps. Though algorithms are not limited to manipulating numbers, some simple numerical examples work well to get the idea across.

Example 2.1. Averaging Algorithm

Let's consider a very simple algorithm for computing the average of n different numbers (labeled x_1 through x_n):

1. add each number x_i together (for every i between 1 and n)
2. divide this sum by n
3. the result is the average value

Note that the first step involves quite a bit of work! This is what we would call a relatively high-level instruction, in that it makes a rather heavy assumption about the reader's problem solving knowledge (specifically, that they understand they need to add x_1 to x_2 , add that sum to x_3 , etc...).

Let's see what it would look like to apply this algorithm to some numbers. Let's say we wanted to average the numbers 4, 5 and 9. In this case, x_1 would be 4, x_2 would be 5, and x_3 would be 9. First, step one would involve finding the sum of all three of these numbers, which is 18. Next, in step 2, you would divide this sum by n , which in this case is 3. The resulting value, which is 6, is the average.


```
int x;
```

Evaluating the expression `x` yields the value 0, since 0 is the default value given to a variable of type `int`. You can see both the declaration statement and the variable expression being evaluated below:

```
jshell> int x;  
x ==> 0
```

```
jshell> x  
x ==> 0
```

Note that if you use a variable in an expression before its defined, an error will result.

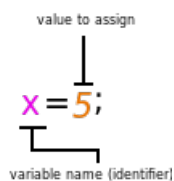
```
jshell> y  
| Error:  
| cannot find symbol  
|   symbol:   variable y  
|   y  
|   ^
```

Assignment Statements

The main difference between variables in programming and variables you've used in math classes is that, rather than trying to determine what value the variable represents, you get to dictate what that value is yourself. In Java, this is done using an assignment statement. The automatically generated variables from JShell (`$1`, `$2`, etc) aren't intended to be changed, but the variables we declare ourselves are!

Below is a syntax diagram for a typical assignment statement:

Figure 2.2. Anatomy of an Assignment Statement



There are two distinct parts to an assignment statement: on the left is an identifier for the variable you want to make an assignment to, and on the right is an expression which yields the value you want to assign to the variable.

This syntax may be a little confusing since it uses the equals operator despite the fact that this is not an equation! This is one situation where the semantics (meaning) of Java syntax diverges from what you might be accustomed to from algebra. Rather than trying to determine what the `x`'s value should be, this statement makes `x` equal to 5.

Example 2.3. Assigning a Value to a Variable

The following code example includes two statements. The first is a variable declaration that creates a variable named `x` of type `int`. Second is an assignment statement that will assign the value 5 to the variable `x`.

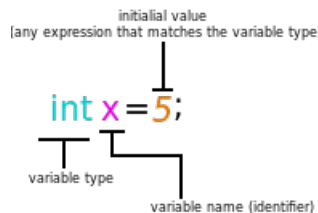
```
int x;  
x = 5;
```

Now evaluating the variable expression `x` at the shell prompt will yield the value 5:

```
jshell> x  
x ==> 5
```

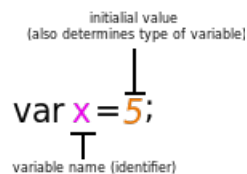
[Definition: When a variable is assigned its first (initial) value, we say that the variable has been *initialized*.] Variables must be initialized before they can be used in another part of a program, and variables must be declared before they can be initialized. In the previous example, two statements were used for these two tasks, respectively. However variables can be declared and initialized in a single statement, which saves the programmer some typing. These combination statements take the following form:

Figure 2.3. Anatomy of an Initialization Statement



It's also not always necessary to provide a type in an initialization statement. Since Java 10 was released, Java will infer the proper type of the variable from the type of the expression used to initialize it. In this case, you can simply use the keyword `var` rather than a type when declaring and initializing a variable:

Figure 2.4. Anatomy of an Initialization Statement with Type Inference



Note that the only difference between this diagram and the previous one is the `var` keyword in place of the variable type. This is one of several ways that the Java maintainers have recently integrated *type inference*. When the term type inference is used, it means that the programmer won't need to explicitly name a type of a variable when the proper type is obvious from context. Note that the type is still static (that is, you can't later change the type of the variable). Though either form of these initialization statements are acceptable, type inference has made Java a little less wordy and a little easier on programmers, so as long as you're using a more recent version of Java (in this case, Java 10 or later), the latter syntax is preferred in most cases.

Example 2.4. Declaring and Initializing a Variable with a Single Statement

The following declares a new variable named `y` of type `int` and assigns the value 34 to it:

```
int y = 34;
```


Instead of giving a type (int), we can instead use the **var** keyword in its place. Since the variable is being initialized on the same line, Java can infer what the type of the variable should be from the type of value being assigned to it. The following code produces the exact same result as the previous example: a variable named `y` of type `int` initialized to the value 34:

```
var y = 34;
```

Throughout this book we will use the **var** keyword wherever possible in our variable definitions. However, there are cases where a type must be explicitly provided in a variable definition, and we will make special note of those cases.

Scripts

Together, a collection of statements comprises an imperative computer *program*. Even one statement by itself can be a program (such as the hello world example). However, typing each of these statements in to the REPL is tedious. Any useful programming language has a means to easily store and run programs.

JShell Scripts

JShell also introduces a facility for this purpose that wasn't previously available to Java programmers. All of the statements you type in to a JShell session can be saved to a file by typing the following command in to JShell:

```
/save myscript.jsh
```

where `myscript.jsh` can be any file name of your choosing. A series of statements saved to a file in any programming language referred to as a *script*. The `.jsh` extension distinguishes in the filename distinguishes this file as a JShell script that can be loaded and executed. In fact, you can load and execute a script you've previously saved in JShell by typing in the following command:

```
/open myscript.jsh
```

Example 2.5. Averaging Script

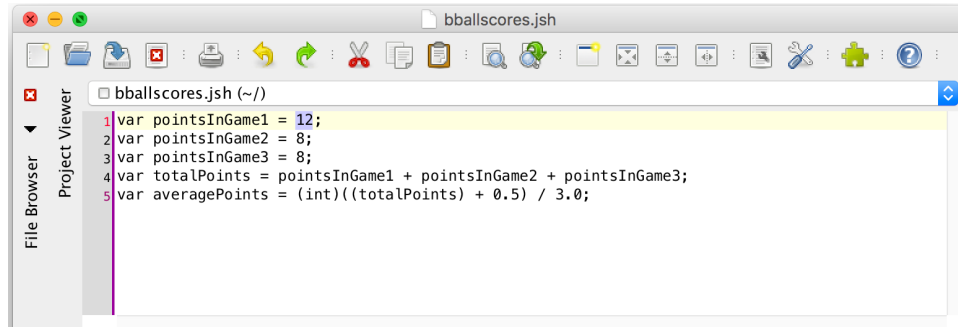
We've seen an algorithm for averaging numbers and we've averaged some numbers interactively in the REPL, now let's develop a script that averages several numbers. Perhaps, in this case, each number will represent the points a player scored in a basketball game. Consider typing the following statements in to the REPL:

```
var pointsInGame1 = 6;
var pointsInGame2 = 8;
var pointsInGame3 = 8;
var totalPoints = pointsInGame1 + pointsInGame2 + pointsInGame3;
var averagePoints = (int)(totalPoints / 3.0);
/save bballscores.jsh
```

This will save a script that creates three different variables, each storing the number of points scored in 3 different games. Following that, the fourth line calculates the total of these scores, and the last line calculates the average number of points the player scored, rounded to the nearest integer. Now let's say we exit the REPL:

```
jshell> /exit
| Goodbye
```

If we wanted to, we could open up this script with a text editor and make changes to the initialization statements. Perhaps there was an error in recording the scores and in game #1 the player actually scored 12 points. In the figure below, I'm using a text editor to update that score to 12:



Now we can open JShell again and load the altered script.

```
| Welcome to JShell -- Version 10.0.1
| For an introduction type: /help intro

jshell> /open bballscores.jsh
```

It appears nothing happens right after we load the script, but if we look for the variables defined in that script, we can see that they will have the expected, updated, values. Note that the `/vars` command will list all defined variables and their values.

```
jshell> pointsInGame1
pointsInGame1 ==> 12

jshell> averagePoints
averagePoints ==> 9.333333333333334

jshell> /vars
|   int pointsInGame1 = 12
|   int pointsInGame2 = 8
|   int pointsInGame3 = 8
|   int totalPoints = 28
|   double averagePoints = 9.333333333333334

jshell>
```

You may need to download a text editor if you wish to edit these scripts that you create. You can also use a text editor to create a new script from scratch. Some text editors have nice features for programmers like syntax highlighting. If you're using Windows, my favorite editor is Notepad++: <https://notepad-plus-plus.org/>. For any system, jedit (which was developed in Java) is also a good option: <http://www.jedit.org/>.

Inline Comments

Programs that you write will quickly become so complex that it may not be easy to look at your code and understand what it will do (or what it is intended to do). Programmers deal with this problem with

a liberal use of comments -- portions of source code that are intended to be read by other programmers and not interpreted as an instruction in the program.

The most common type of comments you'll find in any programming language are called inline comments (also called end-of-line comments, since they appear at the end of lines). In Java, at any point in a line of code, you can type two slashes. Everything following those slashes is ignored. Thus, its only purpose is to be read by you or other programmers to help understand your program.

Inline comments can also be used as placeholders for future instructions while you're writing a program. In this way, they can be seen as a go-between when converting the steps of an algorithm in to an actual computer program.

More often inline comments are used to explain what a significant or particularly complex snippet of code is intended to do (in plain English). It is often helpful to write these comments *first* and then write the code the comments are describing, but they can also be added to code later after a programmer realizes their code might not be so easy to follow.

There is a bit of an art to writing comments well, just as there is an art to writing programs well. Too few comments and it may not be clear what a program is doing. This can also make errors in the program harder to detect. Too many comments can make a program laborious to read and also wastes the programmer's time since writing comments takes careful thought. There are many rules of thumb, but some important ones are:

- Use descriptive variable names that do a lot of the job a comment might be intended for. If the variable's role is very simple, you may not need a comment describing its purpose.
- Do not use comments that merely repeat what a statement is doing, but rather explain the purpose behind one or more statements if that purpose is not immediately obvious.
- Otherwise, if it's not immediately clear what a statement is doing, it may deserve a comment, but put some thought in to whether your comment should cover just that one line of code, or perhaps several that are working closely together.

Example 2.6. Adding End-of-Line Comments to the Averaging Example

Our previous averaging example relied on a trick that may not be obvious to a programmer reading the code. In fact, it might even look like a mistake if they don't realize the intention of the code! Part of the problem is that one statement is doing a lot of work. We could break the complex statement down in to multiple statements:

```
var pointsInGame1 = 6;
var pointsInGame2 = 8;
var pointsInGame3 = 8;
var totalPoints = (pointsInGame1 + pointsInGame2 + pointsInGame3);
var rawAverage = (double)totalPoints / 3.0;
var roundedAverage = (int)(rawAverage + 0.5);    // round to the nearest point
```

An inline comment on the last line helps explain what the line is doing. The variable names on the other lines do a fairly sufficient job of explaining what each of their lines are responsible for, but perhaps the reader is not seeing the forest from the trees. A comment before the three scores indicating what that section of the script (not just one of the lines) is responsible for could help.

Another critique of this approach is that it uses more lines (and more variables) than are necessary and that this may lead to long and convoluted code. Since this is a very simple example, it can't get all that convoluted, but taking this approach with a more complex algorithm would likely be overkill. If you end up writing some long lines that you don't want to split in to multiple statements, feel free to break the statement across multiple lines. There is no rule in Java that each statement must fit on a single line. You can also include in-line comments after several expressions in a complex statement split across

several lines. If you're going to break a statement in to multiple lines, make sure to indent consistently on each line every time an expression is nested (such as whenever you open a pair of parentheses). This makes your code easier to read and helps defend against errors involving too many open or too many close parentheses.

A final critique is that the averaging trick, even with a comment, is perhaps less ideal than calling a method that accomplishes the same task. It's harder to make a mistake when calling a method, but when using the averaging trick, perhaps you accidentally add 5 instead of 0.5, or perhaps you add this value *before* you divide. These errors can be difficult to detect and comments won't necessarily save you from making them.

Below is an example incorporating some of the ideas from the previous paragraphs:

```
import static java.util.Math.*;

// record points scored in each game
var pointsInGame1 = 6;
var pointsInGame2 = 8;
var pointsInGame3 = 8;

var averagePoints = round(
    (pointsInGame1 + // find total points
     pointsInGame2 +
     pointsInGame3)
    / 3.0           // average the total points
); // round the average up to nearest whole number
```

The last line is still a little clunky. There will be times that this technique of using multi-line statements will do a lot to make your code easier to understand and there will be others when a multi-line statement will make it more confusing. This situation is in somewhat of a gray area.

Chapter 3. Object Data Types and Strings

So far you've seen several examples of what are called *primitive* data types that represent very simple data. This chapter will introduce a whole new category of data types: object types. One particularly important object type we'll explore is the `String` type.

Object types are more flexible and useful in complex tasks. We'll also explore such a task in this chapter: communicating with the user of your program through input and output streams. Being able to communicate with the user is necessary to write a useful program, so this chapter will get you off the ground to creating your own programs that do something useful!

Object Types

Thus far we've worked with very simple data in Java (integers, floating point numbers, etc). These data types represent single values with no structure to them and can represent such real-world data as test scores, temperatures, etc. However, how do we represent more complex real-world data? How would we be able to send data about a book, or a completed exam, or an immunization record?

Some data doesn't consist of one simple value. More complex data may have multiple named *fields*. For example, an employee record may have a name, a hire date, a department, etc. Furthermore, operations on this data are also going to be more complicated. We can add and subtract integers, but we hire or fire employees, or put them to work on tasks. We also need new operations for more complex data.

Both of these problems are solved by a whole new category of data types: objects. Object types are fundamentally different from primitive types. Object data types can have multiple fields associated with them and also have *instance methods* that allow you to perform operations over the objects.

Objects are a very important part of the Java programming language (and are the reason Java is referred to as an *Object Oriented* programming language). In Java, you are also able to create new object types if some of the standard ones that ship with the JDK are not what you need or what you want -- however creating new object types is a more advanced topic that will be covered in a future course. This section will show you how to work with some of the object types available to you with a standard JDK.

Classes, Packages, and Import Statements

Objects are also called *instances* of a *class* in Java. A class is essentially a definition of an object type that determines what fields an object will have and what methods you can use to interact with the object. For example, there might be an `Employee` class, but you could potentially have many instances of that class (objects) that represent individual employees with their own names, hire dates, etc. The `Employee` class is merely a data type that ensures each of the objects that are instances of it have these features in common.

Java provides literally thousands of classes with the JDK that you may use to create programs. Because there are so many classes, there was a need to keep them organized (it's possible two classes might even have the same name, such as a `Deal` class that might refer to a business deal or the cards laid down in a poker game). Java organizes classes by placing them in named *packages*. For example, Java has a class called `Point` that is used to represent a point in a 2 dimensional Cartesian plane. However, to avoid possible name conflicts and to help keep things organized, this class is placed in a package named `java.awt`, along with several other classes that assist with computer graphics. Remember that each class represents a data type, and data types have names (such as `int`). In this case, the name of this data type, including its package name, is `java.awt.Point`.

It can be cumbersome to have to type in all of that every time you want to refer to a data type, so Java provides *import statements* to allow you to shorten it to just the class name (in this case, `Point`). In

order to use `Point` as the name of the data type rather than `java.awt.Point`, enter this import statement in the REPL:

```
import java.awt.Point;
```

You can also import all of the classes in the `java.awt` package at the same time if you want to by using a *wildcard* in your import statement in place of the class name:

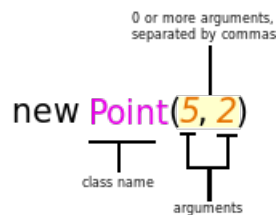
```
import java.awt.*;
```

This would allow you to use the short name for `Point`, but also `Polygon` (instead of `java.awt.Polygon` and many other classes in the `java.awt` package). Generally the use of wildcards is discouraged since you often will only need a handful of classes from a single package, and having each of them listed explicitly in an import statement makes it clear what classes you're using in your code. It also helps avoid the possibility of name conflicts.

Constructor Calls

Before you can use an object, you need to be able to create one. To create an object, you must call a special method called a *constructor*. There is a special syntax for calling constructors outlined in the diagram below:

Figure 3.1. Anatomy of a Constructor Method Call



Assuming you've already imported the `Point` class, this is a method call expression that creates a new `Point` object with its `x` field set to 5 and its `y` set to 2. Some constructors will require more or fewer arguments, and some will require none at all. In fact, if you provide no arguments to the constructor for the `Point` class, it will create a `Point` object with its `x` field set to 0 and its `y` field set to 0:

```
new Point()
```

Object Type Variables

If we want to store and manipulate the object values we create with constructor calls, we need to be able to put them in to variables. Object type variables work exactly the same as primitive type variables. You can simply use the `var` keyword and the proper data type will be assigned to the variable you create and initialize with a constructor call. The following example stores a point at `x` coordinate 4 and `y` coordinate 1 in a variable called `homeLocation`:

```
var homeLocation = new Point(4, 1);
```

Also, if you want to explicitly name the type rather than using the `var` keyword, you can simply use the class name as the name of the data type. The following example does the exact same thing as the previous example:

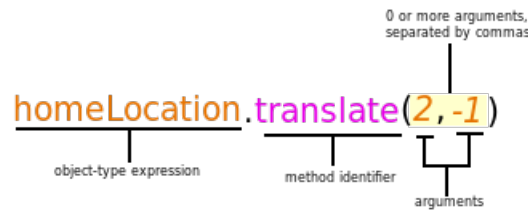
```
Point homeLocation = new Point(4, 1);
```

Again, the `var` syntax is preferred wherever possible, but there will be some cases where you will need to name the type explicitly.

Instance Method Calls

Once you create an object and store it in to a variable, you can interact with that object by calling its instance methods. Calling an instance method looks a lot like calling a normal method with the difference that you start with an expression of an object type, follow it with a dot, and then follow that with the method call. Typically, you will just use a variable reference as the object type expression. Below is a diagram of an instance method call on the `Point` object stored in the `homeLocation` variable we declared in the last section:

Figure 3.2. Anatomy of an Instance Method Call



This method call moves the `Point` object forward 2 in the *x* direction and backward 1 in the *y* direction, meaning that the `Point` object will now be at location 6, 0 on the coordinate plane. Note that *translation* is an operation frequently applied to points in computer graphics, so it was developed as an instance method for the `Point` class. This is an example of how objects provide useful abstractions in object-oriented programming, just as static methods do.

The `Point` class has several other useful instance methods that can be called. Some of them have return values and some of them don't. You can find them all summarized here: <https://docs.oracle.com/javase/10/docs/api/java/awt/Point.html>

Object Fields

Some classes will allow you to access the object's fields directly and `Point` is one of them. You can access a field just as you would an instance method. For example, to directly access the *x* coordinate of the `Point` object stored in `homeLocation`, you could use the following expression:

```
homeLocation.x
```

Most classes will prevent you from accessing the object's fields in this way as it's generally considered bad style to write classes that expose their fields (though it's debatable whether it was a bad choice for the `Point` class, specifically). There are two primary reasons for this: is that it is expected that the designer of the class has already provided sufficiently powerful instance methods that serve as the proper means for manipulating the object, and more importantly also because it helps prevent errors that could occur when another programmer manipulates a field in a way that the designer of the class didn't anticipate (such as setting the balance of a bank account to a negative number). By only allowing objects to be manipulated through carefully designed instance methods, objects serve as more accurate abstractions of some real-world concept you're attempting to model in your program.

The Character Primitive Data Type

A character is essentially another type of numeric primitive, but with a special purpose. In order to work with textual data on a computer (such as the letters you're reading right now, which were printed off by a computer), each letter (and also other symbols such as punctuation, numbers, etc) are each given a special code number.

Characters are a primitive type in Java, and the name of the type is `char`. Thus, just like `int` and `double`, you can create a variable of type `char`:

```
char letterGrade;
```

There is also a special format for character literals. You can surround a character with single-quotes and Java will interpret that as a literal character value. For instance:

```
letterGrade = 'A';
```

Since Java was designed to work on any available computing platform, naturally it should also support any language as well. Thus, the standard used for the characters in Java include codes not only for the letters used in English, but also all other natural languages (including those with many characters, such as Chinese)! This system is called Unicode: <https://unicode.org/>

The actual codes used for each character in Java can be looked up on the UTF-8 chart here: <https://www.unicode.org/charts/>

Methods for Characters

Just as with mathematical functions, there are also some *built in* methods for working with characters. You can make them available in the REPL by typing in the following statement:

```
import static java.lang.Character.*;
```

A couple important ones are summarized below:

- `char toUpperCase(char)` - Takes a character as input and returns that character in upper-case (or leaves it alone if it was already upper-case)
- `char toLowerCase(char)` - Takes a character as input and returns that character in lower-case (or leaves it alone if it was already lower-case)

Strings

A single character by itself doesn't convey much. The kind of data we more often want to work with is a sequence of multiple characters, like a name or address field, or even something longer and more complex like a news article or even a computer program. Java has a data type that fits this role: the `String`. While a character literal only holds a single value, such as `'h'`, a string can hold 0 or more character values. Thus, strings can represent more complex data like names, sentences, or even this entire book! You can create a variable of type `String` as follows:

```
String name;
```

Note that the type in that variable declaration is capitalized. That's because, unlike `char`'s and `int`'s, the string type is an object type (not a primitive type). Thus, when we use the string type in Java code (such as the example above), it must be capitalized. However, unlike `Points` and other objects we've played with, strings are a special object type in Java.

String Literals

Unlike `Points` and other objects we've played with, strings are a special object type in Java in that they have their own special format for literal values. Character literals are a single character surrounded by single quotes, such as `'a'`. String literals are similar, but as noted, there may be more than one character in the string, and we use double quotes to surround them. For instance, the following code initializes the `name` variable from the previous example with my name as a value:

```
name = "Joesph Kendall-Morwick";
```

Note that string literals can contain all sorts of characters, including spaces and dashes. Strings also distinguish between upper and lower case characters. Character literals work just the same way, but character literals hold exactly one character.

String literals may also hold zero characters. For instance:

```
name = "";
```

is also legal. Note that the empty string (above) is still a string value! It doesn't mean there isn't a string held by the variable `name`. If `name` held no value, we would say it holds the `null` value, just as any other object-type variable that doesn't hold a value. Recall that `null` is a keyword representing this value:

```
name = null;
```

Note that there are no quotes around `null`, since it is a keyword.

String Concatenation

Two string values may be joined together with the concatenation operator. Unfortunately, you've technically already seen the concatenation operator because it uses the same symbol as the addition operator (+). However, they are considered distinct operators since they take distinct actions on their operands. You will end up with addition or concatenation depending on the data types of the operands. Allowing an operator to take on multiple meanings depending on the types of the operands is more generally referred to as *operator overloading*.

Example 3.1. Basic Concatenation

The following code:

```
String fullName = "Joseph" + " " + "Kendall-Morwick";
```

will store a single string value representing my full name in the variable `fullName`.

In fact, concatenation is performed when even just one of the operands is a string value. The other value will be coerced (transformed) in to a string value.

Example 3.2. Concatenating non-Strings

The following code:

```
String message = "my favorite number is " + 5;
```

will store a single string value holding the message "my favorite number is 5" in the variable `message`.

String Instance Methods

The string class has many useful instance methods. We'll take a look at four of the most important ones.

`int String.length()`

The `length` instance method has no parameters but returns the length of the string (as an integer).

Example 3.3. Finding the Length of a String

```
"hello".length()
```

evaluates to the value 5 whereas

```
"".length()
```

evaluates to the value 0

char String.charAt(int)

The `charAt` instance method takes the location (called the index or offset) of a character in the string and returns just that character (as a character-type value). The tricky part of working with string indexes is that the first index is zero (not one).

Example 3.4. char String.charAt(int)

```
"hello".charAt(0)
```

evaluates to the value 'h' whereas

```
"hello".charAt(4)
```

evaluates to the value 'o'

int String.indexOf(char)

The `indexOf` instance method takes a character you're searching for as input and returns the first index of where it's found in the string. If the character is not present in the string, -1 is returned instead.

String String.substring(int,int)

The `substring` instance method takes two integer values (indexes) as input and returns a substring of the original string starting at the first index and ending just before the last index. Be mindful of the fact that

Example 3.5. String String.substring(int,int)

```
"hello".substring(1, 3)
```

evaluates to the value "el"

Standard Input / Output

The programs you write will need to be able to interact with a user and thus require some sort of *user interface*. A text-based user interface (or command-line interface) is a very simple and straight forward way to start interacting with the users of your programs by printing information to the screen for the user in text format and taking in data when the user types something at the keyboard. JShell is an excellent example of a program that uses a text-based user interface. This was a very popular style of application on older computer systems in the 90's and earlier but has remained relevant and has perhaps even gained popularity more recently.

Now that you've had experience with characters and strings, you can use these data types to send messages back and forth with the user.

Reading from Standard Input

A text-based application writes to *standard output* to communicate something to the user. You've already seen how to do this in the "Hello World!" example by using `System.out.println`. What you haven't seen yet is how to receive a response from the user while your program is running. This chapter will introduce `Scanner` objects which allow you to read messages from *standard input*.

To create a scanner, you need to provide it with an `InputStream` object to read data in from. Java prepares an input stream that will read data from the keyboard and stores it in `System.in`, so simply

provide that as the argument to the constructor for the `Scanner` class. The example below creates a scanner object stored in a variable named `in`:

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

Now we need to do something with the scanner! A line of input can be read in from the user by calling the `nextLine` instance method. Note that this is a method that *blocks*, meaning that execution of your program will halt at the point that this method is called and will not resume until the user hits the enter key. This method has a return value of type `String`. The string that it returns will be whatever the user typed in. For example, running the following program will ask the user to type something in, and then will simply repeat it back to them:

```
var in = new Scanner(System.in);
System.out.print("Please enter a message: ");
var message = in.nextLine();
System.out.println("You typed: " + message);
in.close();
```

Note that `System.out.print` is called first instead of `System.out.println`. These methods work the same way with the exception that the latter will jump to the next line of output when it's finished. By calling `System.out.print` instead, the user will type something on the same line that they're being prompted on (similar to how `JSHELL` itself works).

Also note that a call is made to the `close` instance method for the scanner on the last line of the program. This essentially lets the system know that you're done using the standard input stream. You won't always run in to trouble if you neglect to close your scanner, but there are many times where this sort of cleanup code is important, so it's good form to always close scanners when you're finished with them.

Reading in Other Datatypes

If you ask a user for a number and read it in with `nextLine`, you will have a string, not an `int` or `double`! That means that if you try to add something to it, you'll be concatenating, not adding.

Scanners have many instance methods you can use to read in data other than strings. For example, you can call the `nextInt` or `nextDouble` methods to read in numeric data as an `int` or `double`, respectively. However, there is a problem with using these methods: They don't necessarily read an entire line of input, they only read the next value stored in an input buffer for the scanner. What this means is that if the user types in two different numbers separated by a space in one line, such as `23 54`, calling `nextInt` once will return the value `23` and calling it a second time will return the value `54` without blocking and waiting for the user to enter another line of input. Sometimes this is exactly the behaviour you want from your scanner, but more often when writing text-based applications, this will create a confusing and error-prone experience for your users.

In a text-based application, a better approach is to always use `nextLine` to read in an entire line of input. Once you've read this line in, if you want to perform arithmetic with it, you can convert it from a string in to an integer with the static method `parseInt`. It's available for use after entering the following import statement:

```
import static java.lang.Integer.parseInt;
```

This method takes a `String` as input and returns an `int`. The following program uses the `parseInt` method to read two numbers in from the user and print out their sum:

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

```
import static java.lang.Integer.parseInt;

var in = new Scanner(System.in);
System.out.print("Enter a number: ");
var firstNumber = parseInt(in.readLine());
System.out.print("Enter another number: ");
var secondNumber = parseInt(in.readLine());
System.out.println("Their sum is: " + (firstNumber + secondNumber));
```

Note that if the string contains data that is not properly formatted as an integer (such as "23.43" or "ham sandwich"), an error will occur. That means that the user can cause this program to misbehave by typing in bad data! This will also happen with the `nextInt` instance method. We'll see some ways to defend against foolishness on the user's part in future chapters.

Chapter 4. Decision Making in Imperative Programs

In the last unit you explored several different kinds of expressions and statements that could be combined to form programs. You also saw how methods can hide a lot of the details of a complicated operation and make your programs cleaner. Although Java makes many methods available to you, you might not have always had a method that did exactly what you wanted or even worked exactly the way you wanted it to.

Starting this unit, you'll begin developing your own methods. Writing methods make for good programming exercises for the same reason that they are good abstractions: you get to write a very complete, free standing piece of a program that can be re-used over and over in many programs you write! This chapter will begin with developing some methods similar to the ones you used last week.

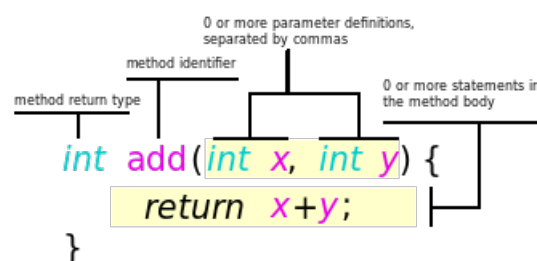
In the GCD algorithm exercise, you had to manually work through the algorithm in jshell because, this far, you have not seen a way to handle the portions of the algorithm where a decision had to be made. In particular, you had to check if a value was greater than zero or not and repeat some of the code if it was. In this chapter, we'll begin looking at conditional statements which allow you to take one of several actions depending on the state of your program. This is only half of the issue for the GCD algorithm, but later in this unit you'll also learn how to have a snippet of code repeat multiple times as well.

Method Definitions

In prior sections we used method calls to do some of the computing for us. For example, a call to the `max` method in the `Math` class was a simple way to achieve what would have otherwise required more detailed code. In this section you will learn how to define your own custom methods.

Method definitions are a bit more involved than our previous topics. Methods take data (arguments) as input (through *parameters*) and return data as output (called the *return value*). The data types for both a method's input and output must be specified (similar to how you could specify the data type when you create a variable, though in this case it's required). Below is a diagram of the syntax of a method definition.

Figure 4.1. Anatomy of a Method Definition



A method definition consists of a header (or signature) defining the input and output data types, and a method body. Let's first take a look at the contents of the signature. First is a data type for the data returned by the method. For instance, the `abs` method in the `Math` class returns a double, so its return type would be double. Next is the name of the method.

So far, the definition looks a lot like a variable definition, but we're not quite finished. We also need to define parameters for the method. Parameters are variables that will hold the input values for the method (the input values are what we referred to as arguments in prior sections). Following the name of the method is a pair of parentheses with definitions of the parameters in between the parentheses. Each parameter definition consists of a data type followed by the name of the parameter (just like the variable definitions we saw previously but without the semicolon at the end). There may be zero or more parameter definitions, separated by commas.

The method body consists of an open curly brace, 0 or more statements, and a closing curly brace. Unlike if statements, the method body must be contained within two curly braces (there is not option for a single-statement body with no curly braces). These statements can do anything you want with the restriction that your method must reach a special kind of statement called return statement when it is finished (with some exceptions -- but for now assume this is always true). The return statement must be the last statement executed in the method and it returns the value that a call to the method would evaluate to.

Example 4.1. Square a Number

Below is the definition of a method that will simply square the number passed in to it.

```
int square(int x) {  
    return x * x;  
}
```

Note that it has only one parameter, `x`, and one statement in the method body, the return statement. Defining and then calling this method in jshell will look like this:

```
jshell> int square(int x) {  
    ...>     return x * x;  
    ...> }  
| created method square(int)  
  
jshell> square(4)  
$2 ==> 16  
  
jshell> square(10)  
$3 ==> 100  
  
jshell>
```

Notice that jshell doesn't include the name of the parameter `x` in the message reporting that you successfully defined the method. That's because the name of the parameter is ultimately not important. The parameter's name is meant to help make the code defining your method easier to read and understand, and also to describe the purpose of the parameter to the programmer writing code that calls it. What's most important from a functional perspective is that you've defined a method that can take an int type argument and that it will return another int type value to you.

Example 4.2. Add Two Numbers

Below is an example of a method definition with more than one parameter. It might not be particularly useful, but it demonstrates how to work with more than one argument in your methods:

```
int add(int x, int y) {  
    return x + y;  
}
```

Calling this method in jshell would look like this:

```
jshell> add(4, 5)  
$2 ==> 9  
  
jshell> add (10, -2)  
$3 ==> 8
```

Example 4.3. Ultimate Answer

Some methods don't have any parameters at all. For example, below is the definition of a method that simply returns the value 42:

```
int ultimateAnswer() {  
    return 42;  
}
```

Calling this method in jshell would look like this:

```
jshell> ultimateAnswer()  
$2 ==> 42
```

This method clearly doesn't do much and there are better alternatives to storing a single value, but there will be many opportunities for writing a meaningful method with no parameters as long as it performs some useful side effect, like printing to the screen.

Booleans

This section will introduce a significant new primitive type: *booleans*. The boolean type is actually the most simple type because it has only two possible values: `true` and `false`. Both `true` and `false` are also literal boolean values, and `boolean` is the name of the type. For example, you can declare and initialize a boolean variable as follows:

```
boolean mybool = true;
```

or simply

```
var mybool = true;
```

Comparison Operators

The boolean type is useful for answering *yes* or *no* questions, such as *is 2 equal to 1 + 1*? We can ask these types of questions with comparison operators.

The comparison operators are binary operators, just like the arithmetic operators from prior sections. However, they operate over values of any type (instead of numeric values), and they evaluate to a boolean value (instead of a numeric value). For example, `==` is the comparison operator. It determines whether two values are the same or not. So, to answer the question we posed in the last paragraph, we could use an expression like this:

```
(1 + 1) == 2
```

If you type this in to the REPL, it will evaluate to `true`. Note that the parentheses are not necessary, either. `+` will be evaluated before any comparison operators will be. There are several other comparison operators that will be useful to us, summarized below:

- `!=` Not equal to
- `>` greater than
- `<` less than
- `>=` greater than or equal to

- `<=` less than or equal to

For example, the expression

```
4 < 3 + 3
```

would evaluate to `true`.

Boolean Operators

There are also operators that work strictly over boolean values, just as the arithmetic operators we used last week worked strictly over numeric values. These are called *boolean operators*. They operate over one or two boolean values and evaluate to another boolean value. These operations are quite simple but form the basis for what is known as *boolean algebra*.

The first operator we'll consider is the and operator: `&&`. This is a boolean operator that evaluates to `true` only when both the left and right operands also evaluate to `true`. For example:

```
true && false
```

evaluates to `false`, since the right-hand operand is `false`. However:

```
true && true
```

evaluates to `true`, since both operands are `true`.

The next operator we'll consider is the or operator: `||`. It is similar to the and operator, but it evaluates to `true` when either of the operands are `true`. Put another way, this operator will only evaluate to `false` when both of its operands evaluate to `false`. For example, if you replaced `&&` with `||` in both of the last two expressions, they would both evaluate to `true`.

The last boolean operator we'll consider is unique in that it's a unary operator. Recall that this means it has only one operand. It is called the not operator: `!`. This operator simply reverses the boolean value of its operand. For example:

```
!false
```

evaluates to `true`. Note that all of these operators can be combined and nested in sub expressions. For example, the following expression evaluates to `true`:

```
!((false && false) || false)
```

These operators also have an order of precedence: `!`, `||`, `&&`

Chapter 5. Conditional Statements

This chapter has just a single section since it is a more complex topic than we have seen in prior chapters. Throughout this chapter we'll be looking at conditional statements and the critical use they have in writing methods and programs. In addition to variables (and some of the statements that manipulate variables), conditional statements make the second of three essential ingredients of any imperative programming language. We'll see the third and final necessary ingredient in the next chapter.

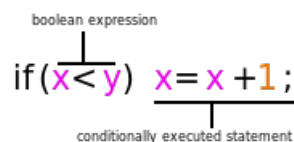
Conditional Statements

Booleans are important because you can use them to make decisions in your programs. Note that several of the algorithms we've worked with asked us to do one of two things depending on some condition. Boolean expressions allow us to specify the condition and conditional expressions allow us to make conditional actions.

If Statements

An if statement is a new kind of statement that will only execute when a certain boolean expression evaluates to true.

Figure 5.1. Anatomy of a Simple if Statement



If statements start with the keyword `if`, then a boolean expression (surrounded in parentheses -- the parentheses are necessary), then the statement we want to conditionally execute. For instance, consider the following statements:

```
int x = 10;
int y = 0;

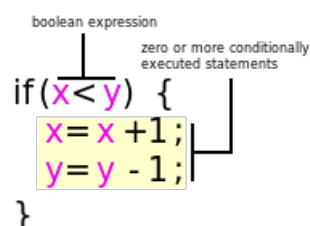
if (y > x) x = 30;

if (x > y) y = 20;
```

The first if statement does nothing. `y` is not greater than `x` at this point, so the assignment statement that follows it (`x = 30;`) does not execute. The second if statement will set the variable `y` to the value 20. Since `x` is greater than `y`, the statement (`y = 20;`) will be executed.

More than one statement can be conditionally executed in an if statement. In fact, you can include as many as you like by opening up a *block* of code using curly braces surrounding each of the conditionally executed statements.

Figure 5.2. Anatomy of an if Statement with a Block Body



```
if (x > y) {  
    x = 30;  
    y = 40;  
    z = 50;  
}
```

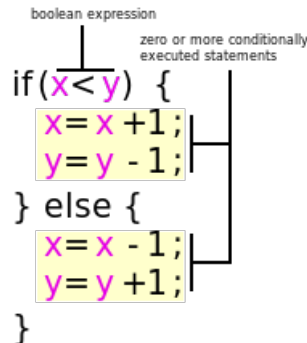
There are a couple important points to keep in mind about code blocks. First, notice that there is no semicolon at the end of the block of code following the closing curly brace. Even though we say there is an *if statement* here, the *if* portion doesn't actually have a semicolon on the end. Including a semicolon following the *if* portion of this code snippet is actually a common error that can be difficult to debug (the body of the *if* would always execute in this case and no error message would be given).

Another important consideration is style. Note that all of the statements inside of the block are indented. This is an important convention in Java that makes the code much easier to read. Soon you will be including blocks of code inside of blocks of code, and each should be further indented, helping the programmer to see how deeply *nested* the blocks of code are. Typically each nested statement is indented two spaces. Sometimes a tab or four spaces is used instead, but this can become a problem with larger programs. Whatever choice you make, the most important consideration is that you use consistent indentation throughout your program.

Else Statements

Conditional expressions provided a value whether our boolean expression evaluated to *true* or *false*. We can similarly provide an alternative block of statements we want executed if the boolean expression used in our *if* statement evaluates to *false* by using an *else* statement.

Figure 5.3. Anatomy of an if / else Statement



Else statements look exactly like *if* statements but must immediately follow the end of the *if* statement.

Example 5.1. Implementing Absolute Value with if/else

Recall the *abs* method from the *Math* class that determined the absolute value of a number. In this example, we'll write our own absolute valued method that does the exact same thing using an *if* and an *else* statement. We'll name this method *myAbs* to make it distinct:

```
double myAbs(double x) {  
    if (x > 0) {  
        return x;  
    } else {  
        return -x;  
    }  
}
```

If `x` is greater than 0, the first `if` statement will execute, meaning that the value stored in `x` will be returned. If `x` is not greater than zero, the `return` statement inside the `else` block will execute instead, meaning that the negation of the value stored in `x` will be returned. Note that `-0` is returned when `x` stores the value 0, but since these two values are equivalent, it doesn't make a difference. Also note that the `return` statement is still always the last statement executed, no matter what the value of `x` is.

Nested If Statements

Any statement can be included in the body of an `if` statement, which means you can include an `if` statement inside an `if` statement! This is called a nested `if` statement.

Example 5.2. Nested if Statements to Determine Best Temperature

The following method will return `true` *if* the temperature (an integer type variable) is between 68 and 72:

```
boolean justRight(int temperature) {
    if(temperature >= 68) {
        if(temperature <= 72) {
            return true;
        }
    }
    return false;
}
```

One thing to note about this example is that there are no `else` statements. There could be, but they aren't necessary since in all cases they would return `false`, which is also what the last statement of the method does.

Also notice that the nested `if` block is indented further than the block it is nested in. This is an important style convention that you should follow whenever opening nested blocks of code, not just with nested `if` statements.

if/else Chains

When a situation calls for one of two kinds of action, we could use a `if` statement with an `else` statement following it. However, if there are more than two actions we might take, we can nest a new `if` statement inside of each `else` statement, forming a chain of `if / else` statements (ending in an `else` statement with no nested `if` statement). This way, rather than just two choices in the conditional statement, this `if/else` structure allows us to have 3 or more choices.

One issue with this level of nesting is that the code can become very complex if we're following the indentation conventions. For instance, if there were 10 different choices in an `if/else` chain (not unheard of), the last `else` statement would be indented at least 20 spaces!

To avoid this complexity in the code, there are coding conventions that dictate when you should use curly braces and how much you should indent to avoid very wide lines in your code, summarized in the figure above. Rather than opening a brace for each `else` statement, instead, when an `else` statement is followed by another `if`, no brace is opened and the `if` immediately follows the `else` on the same line. Then a brace is opened for the `if` statement and also the final trailing `else` statement.

Example 5.3. Ice Cream Choices with if/else Chains

An application is being developed that can order an appropriate snack for you automatically. If the temperature outside is high, it will order an ice cream, and if the temperature is low, it will order a hot chocolate. Assume that there are two methods available that can order either of these foods. The code might look like this:

```
if ( temperature > 65 ) {  
    orderIceCream();  
} else {  
    orderHotChocolate();  
}
```

Now consider that the application might also order popcorn if the temperature is more moderate. We would have to use a nested if statement to accomplish this:

```
if ( temperature > 80 ) {  
    orderIceCream();  
} else {  
    if ( temperature > 50 ) {  
        orderPopcorn();  
    } else {  
        orderHotChocolate();  
    }  
}
```

If we wanted to add, for example, 5 more snack options, this would become quite a mess, with many layers of nested if statements inside of else statements! Your code would be so far indented to the right, that it may begin to run over the length of the screen.

This is avoided by not opening a new curly brace for each if statement that immediately follows an else, and instead putting both on the same line. The body of the else then becomes the if statement that follows the else, and the following if statement is not indented further than the else that it follows. For example, we could re-write the code above this way:

```
if ( temperature > 80 ) {  
    orderIceCream();  
} else if ( temperature > 50 ) {  
    orderPopcorn();  
} else {  
    orderHotChocolate();  
}
```

Switch Statements

If each of the `if` statements in an if/else chain are simply comparing some expression to one of several literal values, there is a special syntax in Java called a *switch* statement that can be used instead. In some of these cases, this will be a little easier to read and more efficient.

The syntax for switch statements is a little trickier than the other conditional statements we've seen. Following the `switch` keyword is an expression in parentheses, but it won't be a boolean expression. Instead, it can be any other type of primitive or String value. This value will be compared against a number of literal values inside of the switch statement body, which follows.

The body is contained within a pair of curly braces and consists of a series of *case statements*. Each case statement begins with the `case` keyword, followed by the literal value that the resulting value from the original expression at the head of the switch statement will be compared to, and then a colon. Following that will be a series of statements. Curly braces are not necessary.

Note that after a case statement is complete, the next case statement will be executed! Unless this is intentional, you must explicitly end the switch statement using a *break statement*. This is simply the keyword `break` followed by a semicolon. Break statements are not needed if a return statement is used, since execution of the enclosing method will end at that point anyway.

One last important feature of the switch statement syntax is the *default case* which can handle the case where the value we get back from the initial expression doesn't match any of the literal values in the cases. A switch statement without a default case will simply do nothing.

There is a special syntax for defining the default case. Default cases look like the other case statements, but they use the keyword `default` instead of `case` and are immediately followed by a colon instead of some literal value to compare with.

Example 5.4. Switching Over Letter-Grade Characters

A grade book application needs to determine the minimum grade needed for each letter grade. Letter grades will be represented with capital letter characters. This functionality could be developed using an if/else chain:

```
int getMinimumGrade(char letterGrade) {
    if ( letterGrade == 'A' ) {
        return 90;
    } else if ( letterGrade == 'B' ) {
        return 80;
    } else if ( letterGrade == 'C' ) {
        return 70;
    } else if ( letterGrade == 'D' ) {
        return 60;
    } else {
        return 0;
    }
}
```

However, this is a good opportunity to use a switch statement. The switch statement below takes equivalent action, but will run a little more efficiently and may be a little easier to read and modify:

```
int getMinimumGrade(char letterGrade) {
    switch(letterGrade) {
        case 'A':
            return 90;
        case 'B':
            return 80;
        case 'C':
            return 70;
        case 'D':
            return 60;
        default:
            return 0;
    }
}
```

Note that no `break` statements are needed in this example since `return` statements are used in each case.

Chapter 6. Loops

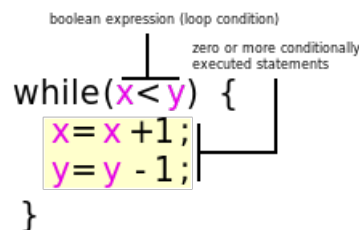
Loops are the last piece of the puzzle for a *real* programming language (or what computer scientists call a *Turing Complete* language). What this means is that, technically, you can now write a program that is capable of doing whatever any other program written in any other language can do.

You won't see as much new syntax in this chapter as you have in prior chapters. Loops look a lot like conditional statements. However, they add a level of complexity to programs that takes a lot of practice to master. Take your time with this material and read through it multiple times if necessary after going through examples, demos, and attempting practice problems.

While Loops

While loops are a lot like if statements. In fact, they have a nearly identical syntax.

Figure 6.1. Anatomy of a While Loop



While loops start with the `while` keyword, and then a boolean expression in parentheses, and then a block of code (a series of statements surrounded by curly braces) or a single statement. The only difference is that when this block of code or single statement completes, program control returns to the beginning of the while loop. Think of it as an if statement that repeats itself indefinitely.

Now you may be concerned by the word *indefinitely*, and you should be -- it's a distinct possibility that a while loop can run *forever*. However, after each *iteration* of the loop (an iteration is complete after execution reaches the end of the code block), the boolean expression at the front of the while loop is evaluated again. If you wrote the loop correctly, that boolean expression should evaluate to `false` when you don't want the loop executing anymore.

Example 6.1. Counting with a While Loop

The following code will add up the numbers 1 through 10:

```
int count = 0;
int total = 0;
while(count <= 10) {
    count = count + 1;
    total = total + count;
}
```

After the first iteration of the loop, `count` will increase by 1 and `total` will increase by 1. Thus, when the boolean expression is evaluated the second time, `count` will be 1 (and not 0). However, 1 is still less than 10, so the loop will run a second time. After the second iteration, `count` will increase to 2 and `total` will increase to 3. Now you might see where this is going. The loop will run 8 more times until `count` is equal to 11. At that point, `count` will be greater than 10, so the boolean expression will be false, and the statements inside the block will be skipped.

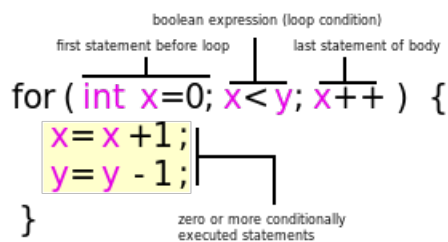
For Loops

When writing loops, three components are very commonly included:

- Some local variable is declared and initialized, for instance a variable named `counter` could be initialized to 1.
- That variable is tested as part of the while condition, for instance looping while `counter` is less than 10.
- At the end of each iteration of the loop, the variable is altered in some way, for instance `counter` could be incremented at the end of each iteration of the loop.

Since nearly every while loop that we write contains these three components, and also because they play an important role in determining how the loop executes, Java provides a special syntax called a `for loop` that simply re-organizes these components to all reside at the front of the loop.

Figure 6.2. Anatomy of a For Loop



for loops start with the `for` keyword and a pair of parentheses, just like a while loop, but inside the parentheses is first an expression (ending in a semicolon) that is executed before the loop starts, secondly a boolean expression (the same as the looping condition for the while loop), also ending in a semicolon, and finally a statement that is executed at the end of each loop (typically incrementing a variable). The last statement doesn't have a semicolon following it, but all three should be within the parentheses. The rest is identical to a while loop.

Example 6.2. Totaling Numbers with a For Loop

Consider the following while loop which adds the numbers between 1 and 10:

```

int total = 0;
int count = 1;
while(count <= 10) {
    total = total + count;
    count++;
}

```

Here is identical code written as a for loop:

```

int total = 0;
for(int count = 1; count <= 10; count++) {
    total = total + count;
}

```

Because code written with while loops and for loops perform the exact same task, we call for loops *syntactic sugar*, in that it is meant only to help with the look of your code, not the function of your code. Essentially, it is used to improve your coding `style`.

Example 6.3. Iterate Through a String

The following code will print each character of the string message, one character per line:

```
String message = "hello";  
for(int i=0; i < message.length(); i++) {  
    System.out.println(message.charAt(i));  
}
```