Modern Computer Programming

Modern Computer Programming

Table of Contents

List of Examples

Chapter 1. Algorithms

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.

In this course you will write many computer programs in Java that implement algorithms. However, before we make that leap, we'll first learn about the basic building blocks of a Java program and weave them together to implement the steps in abstract algorithms.

Chapter 2. Basic Elements of Imperative Programming

Expressions

Throughout the first unit we will be mostly looking at Java expressions. Think of expressions as the most basic building blocks of a Java program. An expression is something that is evaluated in your Java program (that is, it's interpreted to represent some value). A common exercise / exam question you will have this semester is ``what value does this expression evaluate to?''.

Throughout the first unit, you will be using a program called a REPL (Read, Evaluate, Print Loop) to evaluate Java expressions. If you type a Java expression in to the REPL, the REPL will interpret it and print out the value that it evaluates to.

We'll start by working with numeric expressions. These are expressions that always evaluate to some sort of numeric value (such as 5, or 9.2).

Numeric Literal Expressions

Literal expressions are the most simple expressions in Java. The expression you type in is literally the value you get back. For instance, the expression `5` evaluates to, wait for it..., `5`.

One catch is that there is more than one type of numeric value in Java! `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 `-2147483648` or -2^{31} and the maximum value is 2^{31}-1 or `2147483648`). There are different integer types with different sizes. For instance, the type long has a range of -2^{63} to -2^{63} There are also the lesser used short and byte integer types, but we won't work with these specifically (this semester).

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.

`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). By default, the literal expressions you type in that contain a decimal point will be of type double (float is the other legal type name).

Internally, these values are stored in the same way you've used 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.

There is a format for literal values that uses scientific notation using the letter E between these two components. For instance, `5E3` is the same as `5000.0`

Arithmetic Expressions

A binary operator is something that operates over two expressions in Java, forming a larger, more complex, compound expression. 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. For example, 5 + 5 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 reminder when performing integer division.

There is also a unary operator in Java. This is an operator that works over only one operand (expression), as opposed to 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. That is, 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.

Statements

[Definition: A *statement* is essentially a single instruction.] This is a fairly abstract definition, but statements serve many purposes in imperative programming languages like Java and can embody simple or complex instructions. Generally, statements will end with a semicolon (think of this like the period at the end of a sentence in English).

There are several types of statements in Java that we will examine in the early portions of this book, but we'll start with two of the most simple statements: declaration statements and assignment statements.

Declaration Statements

Before we can explore declaration statements, we must first understand variables:

One of the essential components of any program is the ability to remember and recall values. In terms of computer hardware, this is what we refer to as memory. The most basic way to utilize memory in just about any programming language is by using variables. [Definition: A *variable* is a named portion of computer memory that holds a single value.] 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.

You give variables a name in Java by using a Java identifier. Thus `x` is a legal variable name, but so is `x5` or `reallylongvariablename`. Note that the last variable name is somewhat difficult to read. For this reason, there are *rmconventions*> in Java which determine the proper way to name variables. `reallylongvariablename` is not an error, but it does break the proper naming convention in Java, which is called *camel case*. In camel case, you capitalize each of the sequential words. We must do this for identifiers consisting of multiple words since spaces are not allowed in identifiers. Thus, a proper convention-following identifier we could use as a variable name would be `reallyLongVariableName`. Notice that this is a bit easier to read?

In order to use a variable in Java, you must declare it exists using a *declaration statement*. Java is known as a statically typed language, which means that when we declare variables, we must also declare what type of value they hold. The first element of a declaration statement is the type of data the variable will hold, and the second is the identifier (variable name).

Example 2.1. Declaring a Variable and Using it in an Expression

We can create a variable named `x` which holds integer values with the following statement:

```
int x;
```

Once a variable is declared, you can use it as an expression (called a *variable expression*). The expression will evaluate to whatever the variable holds. 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 evaluted 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 you can change the values variables hold. In Java, this is done using an assignment statement. Assignment statements may be a little confusing since they use the equals operator, even though this is not an equation! 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.

Example 2.2. 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, however variables can be declared and initialized in a single

statement. It's also not always necessary to provide a type in a declaration. The example below walks you through the relevant Java syntax:

Example 2.3. 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;
```

Since Java 10, it is also unnecessary to provide the variable type in an assignment statement if you are initializing it with the same statement. Instead of giving a type, you 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.

Chapter 3. Decision Making in Imperative Programs

Booleans

This section will introduce a significant new primitive type: \begin{textbf}booleans\end{textbf}. The boolean type is actually the most simple type because they have only two possible values: \begin{textit}true\end{textit} and \begin{textit}false\end{textit}. Both \begin{textit}true\end{textit} and \begin{textit}false\end{textit} are also literal boolean values, and \begin{textit}boolean\end{textit} is the name of the type. For instance, you can declare and initialize a boolean variable as follows:

```
boolean mybool = true;
```

Comparison Operators

The boolean type is useful for answering ``yes'' or ``no'' quesions, 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 you learned last week. However, they operate over values of any type, and the evaluate to a boolean value. For instance, == 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 evalute 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 instance:

```
true && false
```

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

```
true && true
```

...evalutes 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 subexpressions. For instance, the following expression evaluates to true:

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

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

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 and if statements allow us to make conditional actions.

An if statement is a new kind of statement that will only execute when a certain boolean expression evaluates to true. These statements start with the keyword \begin{textbf}if\end{textbf}, 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 conditional statement can be included in an if statement. In fact, you can include as many as you like by opening up a \begin{textbf}block\end{textbf} of code using curly braces surrounding each of the conditional statements:

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

Conditional Expressions

You can also create expressions that evaluate conditionally. These expressions use two operators in conjunction: a question mark (?) and a colon (:). Conditional expressions consist of 3 parts: a boolean

expression, followed by a question mark, followed by an expression that wll be evaluated if the initial boolean expression is true, followed by a colon, followed by an expression that will be evaluated if the initial boolean expression is false. For example, consider the following expression:

```
5 > 10 ? 30 : 40
```

This expression evaluates to the value 40 since the initial boolean expression evaluates to false, the expression following the colon is evaluated. The middle expression is ignored in this case. If we changed the comparison operator to < the conditional expression would have evaluated to 30. Here's another example that would evaluate to the maximum of x and y:

```
x > y ? x : y
```