Lecture 2: Java Basics

Sam McCauley
Data Structures, Spring 2026

Basics

Let's begin with an empty Java program. It looks something like this:

```
1  public class Hello {
2       public static void main(String[] args) {
3           //fill in code here
4       }
5  }
6  /* This was a simple example of a
7     Java program */
```

Let's briefly discuss each part of this program.

- `public class Hello` means that we are creating a new class, named `Hello`. If you have seen "classes" before (say in Python or C++), this is the same kind of class. We'll be learning more about these soon.
- `public static void main(String[] args)` This is the way the "main method" is declared in Java. We'll learn more about each piece of this declaration over the course of the class.
- "fill in code here" is a comment: any text after `//` on a line is ignored by Java.
- Lines 6 and 7: "`A simple example of a`" and "`Java program`" are also comments: any text between `/*` and `*/` is considered a comment and ignored by Java. These are often called *multi-line comments* because they can span multiple lines.

Output in Java is accomplished using the methods `System.out.print()` (which prints to the screen *without* a new line at the end) and `System.out.println()` (which *does* have a new line at the end). You can enter a variable to be printed, or an actual value. In the following example, we pass in a String:

```
1  public class Hello {
2       public static void main(String[] args) {
3           System.out.println("Hello World");
4       }
5  }
```

It is likely immediately apparent that printing this way requires quite a lot of typing: it would be nice if we could say something shorter like `print()` rather than `System.out.println()`. This is a downside of Java.

Semicolons. Every command needs a semicolon at the end. Loops, conditionals, method declarations, and class declarations do not need a semicolon at the end.

Here's an example. It includes quite a bit of Java we haven't seen before, but let's go over the semicolon usage anyway.

```
1  public static int findLargestDivisor(int num) {
2      int x = num - 1;
3      while (x > 1) {
4          if(num % x == 0) {
5              return x;
6          }
7          x = x - 1;
8      }
9      return x;
10 }
```

- Line 1 does not need a semicolon because it is a method declaration
- Line 2 is a command; it has a semicolon
- Line 3 is a loop and Line 4 is an if statement; neither needs a semicolon
- Line 5 is a command; it has a semicolon
- Line 6 is a closing brace for the if statement from Line 4, so it does not need a semicolon
- Line 7 is a command; it has a semicolon
- Line 8 is a closing brace for the while loop on Line 3; no semicolon
- Line 9 is a command; it has a semicolon
- Line 10 is a closing brace for the method on Line 1; no semicolon

Basically, anything that "does something:" sets a variable, calls a method, etc., needs a semicolon. Things like conditionals or loops, that just involve what code is executed next, do not.

Semicolons are a bit of a pain! You'll get the hang of it with practice—but even experienced Java programmers forget them occasionally. The compiler will usually tell you if a semicolon is missing.

Variables

Variables in Java must be declared before they are used. A declaration tells Java that you are going to be using the variable, as well as what type it is.

```
1     int x;  //declare x
2     x = 5;
```

```
1
2     x = 5; // Error!
```

It's often useful to think of variables as values stored in a "box"—so in the above code, the variable x is like a box containing the value 5. In this analogy, the declaration is like telling Java to make the box; that way we have a place to put 5 later.

When declaring a variable, you need to state the *type* of the variable up front: what kind of data you will store in it. This cannot be changed; Java makes a "box" for the given type—if you want a different type, you'll need to make a new variable (a new "box").

In the above example, we declared an `int`, which can store integer data. Some other common types we'll see are:

- `double`: "double-precision floating point number" is a number that does not have to be an integer: `3.14` and `4.0` are both of type `double`
- `char`: a single character, like 'a' or 'Q' or '$'. Whitespace (like a single space, or a newline) can also be stored in a character. In fact, any unicode character can be stored in a `char`
- `boolean`: stores either `true` or `false`.

We've also seen another type: `String` which stores a sequence of characters, like `"Hello"`. Java handles `String`s a little differently from the other types; we'll learn about this in the next couple weeks.

There are some less common types that may come up. `long` is a version of `int` that can store larger numbers; `float` is a version of `double` that can only handle shorter, or less precise numbers.

Let's use what we've seen so far to write some simple Java code to print a variable.

```
1 int x;
2 x = 5;
3 System.out.print("The value in x is ");
4 System.out.println(x);
```

Instantiating Variables. A variable cannot be used until it is *instantiated*—until we store a value in it.[1] If we remove x=5 from the above code, Java gives an error.

[1] We will see later that some variables have default values—however, default values should not be used.

```
int x;
System.out.print("The value in x is ");
System.out.println(x); //Error!
```

This probably makes sense: we can't print the contents of x until it has some contents to print.

Variables can be declared and instantiated at the same time.

```
int x = 5;
System.out.print("The value in x is ");
System.out.println(x);
```

Arithmetic Operations

The basic arithmetic operations are similar to what you've seen in other languages: `+`, `-`, `*`, and `/` are used for addition, subtraction, multiplication, and division.

```
int yards = 300;
int feet = yards * 3;
int miles = feet/5280;
```

The percent sign `%` is for the "modulo" (meaning remainder) operation. For example:

```
if(x % 3 == 0) {
    System.out.print("x has no remainder when divided by 3, ");
    System.out.println("so x is divisible by 3");
}
```

Arithmetic and Types. One important point is that types are preserved during arithmetic: arithmetic on `int`s will always return an `int`; arithmetic on `double`s will always return a `double`.

This is particularly important during division. If you divide two `int`s, Java will discard the fractional part (that is to say: it will round positive numbers down, and negative numbers up).

In the following code, `halfx` will have value 4, and `halfy` will have value 4.5.

```
    int x = 9;
    double y = 9;
    int halfx = x/2;
    double halfy = y/2;
    System.out.println("Half x is " + halfx);
    System.out.println("Half y is " + halfy);
```

One may wonder what happens when operations combine different types: let's say if we evaluate `halfx + halfy`. We'll discuss this later this week.

Conditionals

We've already seen several examples with an `if` statement. The code inside the curly braces is only executed if the `if` statement's condition in parentheses is true. Note that the parentheses and braces in the following code are required.[2]

```
if(x < 0) {
    System.out.println("x is negative.");
}
```

You can also have an `else` statement, that executes only if the `if` statement is false.

```
if(x == 0) {
    System.out.println("x is zero.");
} else {
    System.out.println("x is not zero.");
}
```

Note that if we are testing for equality, we use a double equals sign `==`. This is to distinguish it with a single equals sign, which is used for variable assignment (as in `int x = 0`).

You can also have any number of `else if` statements. Each executes only if all previous conditions evaluated to false, and its condition evaluates to true.

For example, consider the following code with `x = 8`. Since `x < 0` evaluates to false, we move to the next `else if`. Again, `x == 0` evaluates to false, so we move to the next `else if`. Then `x < 10` evaluates to true, so the code prints `x has one digit.`

```
int x = 8;
if(x < 0) {
    System.out.println("x is negative.");
} else if (x == 0) {
    System.out.println("x is 0.");
} else if (x < 10) {
    System.out.println("x has one digit.");
} else {
    System.out.println("x has two or more digits.");
}
```

Java has another type of conditional, the `switch` statement; we'll learn about this later.

[2] Technically, the braces can be omitted if the contents of the if statement are a single line. In this course, however, we'll always include them.

Loops

Loops allow us to execute code repeatedly. A `while` loop continues to execute so long as the conditional is true.

```
int x = 100;
while(x > 10) {
    System.out.println(x);
    x = x/2;
}
System.out.println("all done!");
```

Let's look at what happens when the code is run:

- Since `100 > 10`, the code inside the loop will run, printing `100` and setting `x = 50`.
- Since `50 > 10`, Java will print `50` and set `x = 25`.
- Since `25 > 10`, Java will print `25` and set `x = 12`.
- Since `12 > 10`, Java will print `12` and set `x = 6`.
- Since `6 > 10` is false, Java will print: `all done!`

If the loop statement is false, the loop will not be run at all. For example, if we run the above code with `x = 8`, the only thing printed will be `all done!`

For loops. You will also frequently see `for` loops in Java. Note that these loops may not be quite like `for` loops you have seen in the past (especially in Python)—they do not necessarily iterate over a list. Instead, `for` loops in Java are shorthand for a common type of while loop.

Let's look at an example, and then discuss in more detail. The following pieces of code in Java are exactly equivalent—the for loop version is shorthand for the while loop version.

```
for(int x = 0; x < 10; x++) {
    System.out.print(x);
    System.out.print(" ");
}
System.out.println();
```

```
int x = 0;
while(x < 10) {
    System.out.print(x);
    System.out.print(" ");
    x++;
}
System.out.println();
```

A for loop consists of three parts, separated by semicolons. The first part (usually consisting of a variable declaration and instantiation) is run once, before the loop begins. The second part is a loop conditional; the loop continues to run so long as this condition is true. The third part (usually consisting of some change of variable) is done at the end of the loop, before the next loop iteration.

Other loops in Java. Java has two other kinds of loops: a *for-each* loop, and a *do-while* loop. We'll learn about these later.

Methods

When programming, it is often convenient to organize our code into smaller tasks that we can call repeatedly. These are usually called "functions," or "methods." We'll follow standard Java terminology and call them "methods" in this class.

We've already seen one method: the `main` method. (Really, I should say: the `public static void main(String[] args)` method.) This is a special method that is called when the code is run.

We can also write our own methods. Here is a simple example of how to create and use a method.

```
1 public static void sayHi() {
2     System.out.println("hello!");
3 }
4
5 public static void main(String[] args){
6     sayHi(); // prints: hello!
7     sayHi(); // (again) prints: hello!
8 }
```

Note that `public static` is (for now) required when creating a method–we'll come back to what these words mean in the next couple weeks. The `void` keyword means that the method does not return anything.

We can pass data to and from methods as well.

To pass data to a method, we may declare a method with *parameters*. When calling the method, we will pass *arguments* that give the value for those parameters. Each parameter is a variable, and therefore needs a type.

To obtain data from a method, we may have the method return a value. We will need to tell Java what this type is up front.

Here is a very simple method that takes two `int`s as parameters and returns an `int`.

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

public static void main(String[] args){
    int x = 10;
    int y = 7;
    int sum = add(x,y);
    System.out.println(sum); //17
    System.out.println(add(x,y)); //17
}
```

Methods can mix parameter and return types.

```java
public static double findAndPrintAverage(int x, int y, String message){
    double average = (x + y)/2.0;
    System.out.print(message);
    System.out.println(average);
    return average;
}

public static void main(String[] args){
    int x = 1;
    int y = 2;
    //this method call prints: Average is 1.5
    double average = findAndPrintAverage(x, y, "Average is: ");
    System.out.print("The method returned ");
    System.out.println(average); //prints 1.5
}
```