

# Chapter 1

## Getting Started

Installing Java (v1.8)  
Installing Dr. Java  
Hello, World!  
Java Basics

### Hello, Lab!

Welcome to CS 211! The early focus in this course is to get introduced to Java, and to begin getting comfortable writing code in Java, so that we can use Java intensively throughout the rest of the course when we learn about various object oriented programming concepts. So today, we will get Java installed on your machine, get an IDE installed (Dr. Java), and then begin writing some basic Java code.

This text often contains a lot of content – the lab manual chapters give you the chance to work through more problems with more step-by-step guidance. You should finish each chapter and all the associated practice problems to ensure you understand the content.

### Installing Java

In order to write and test Java code on your machine, we will need to get certain tools installed, contained in the **JDK** (not just the JRE). **Our goal is to get version 1.8 installed**, to ensure you can use all the recent additions to the Java language that we will discuss in class. Dr. Java supports version 1.8, but will require a small bit of extra work to get installed on Mac. We will use the tools directly at the command line first, instead of relying on an IDE to utilize the tools on our behalf. We want you to experience the "edit, compile, run" cycle first-hand. In later classes you take, this style of programming will be necessary/mandatory—you can't always write code on a fancy IDE on your own machine (for example, if you are writing code to run on a cluster, or are ssh'd into a remote machine to access the one licensed copy of some software), and we don't want you to think that "Eclipse==Java Programming" or "Dr. Java==Java Programming". Those tools are indeed useful, but we need to understand the basics of what is occurring to really utilize them well.

#### Quick Note

"JDK" means "Java Development Kit", and is intended for people who will create Java programs (that's you!). "JRE" means "Java Runtime Environment", and is intended only for running Java programs. If you install the JDK, you will be getting the JRE as well. So we are only trying to install the JDK, as it gets everything for us.

#### Checking for Java

First, check if you have Java installed. Open up a terminal (Mac) or command prompt (Windows: go to Start → "run...", and enter 'cmd'). Type in these two commands:

```
demo$ javac -version
demo$ java -version
```

If you don't already have it, or have a version older than 1.8 (smaller numbers), follow the instructions below.

## Downloading the latest version

- Go to [this link](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html). (<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html> )
- Click "Accept License Agreement"
- Select the version that correlates to your machine; you probably want "Windows x64", "Mac OSX x64" or some Linux version.
- Download and open the downloaded file. Follow any installer instructions.
- **Windows Users:** Update your PATH variable ([instructions here](http://docs.oracle.com/javase/8/docs/technotes/guides/install/windows_jdk_install.html#BABGDJFH) [http://docs.oracle.com/javase/8/docs/technotes/guides/install/windows\\_jdk\\_install.html#BABGDJFH](http://docs.oracle.com/javase/8/docs/technotes/guides/install/windows_jdk_install.html#BABGDJFH) ).

Next, test that you've got everything installed and working. Again, open up a *new* terminal window. Type in these two commands again:

```
demo$ javac -version
demo$ java -version
```

And make sure that they both report the same version, e.g. "1.8.0\_25" for version 1.8. We don't care what follows 1.8 (0\_25 suggested here), as long as they match. (Java might have an extended build-number after the 1.8.0\_25 part, that's okay).

- If you get error messages such as "command not found", then you didn't set up your path correctly (Windows), or you forgot to open a new command prompt after setting your PATH variable.
- Similarly, if you get an error message containing a trace-back and the phrase "unsupported major.minor version" ..., it means you have [different versions of javac and java installed](#). You would then need to update enough so they matched, meaning you should just install the latest JDK release.

At this point, you should have Java installed! If Windows users want to get the Linux feel, they could also install [Cygwin](https://cygwin.com/) (<https://cygwin.com/>). It allows you to re-compile code from source that was intended for Linux, and run it within Cygwin, on a PC.

**Still Lost?** Here's a link that gives instructions on installing Java; perhaps another perspective will help! First [WikiHow](http://www.wikihow.com/Install-the-Java-Software-Development-Kit). (<http://www.wikihow.com/Install-the-Java-Software-Development-Kit> ).

## Installing an IDE

A good coding environment makes a world of difference. Please avoid Notepad, TextEdit, Wordpad, nano, and pico. You could take the plunge and start learning something like Vim <http://www.vim.org>) or Emacs, (<http://www.gnu.org/software/emacs/>), but for this course we will install a basic IDE.

**We will officially support you in this class with Dr. Java**, but won't promise to solve all your installation woes if you have installed other editors and IDEs. Some other popular more lightweight options are JEdit, JGrasp, or Notepad++ but of course you're on your own for installation if you use any of these, and you will be missing out on some features that are directly available in Dr. Java later on.

We will also use JUnit, which also works directly with both Dr. Java and Eclipse (as well as other IDEs, and of course on the command line as well).

### Installing Dr. Java

Go here and get our specially-patched version of Dr Java: <http://cs.gmu.edu/~kauffman/drjava/>

## Windows/Linux Users

- An alternate (less cool) version is here: <http://drjava.sourceforge.net/> (download the app version; after a short countdown the download begins). It directly gives you the application. You might want to move it to some more permanent location and just have a handy link or pin it to your dock/task bar.

## Mac Users

- Mac machines need convincing to open a file that doesn't go through their fancy App Store. The first time you open this file (it's double-clickable), you might need to agree to open it despite having downloaded it from the internet. If you have trouble opening it, try right-clicking and selecting Open. This is actually supposed to make a difference.
- **Missing the Debugger?** This should be solved in our patched version above. When you've got Dr. Java open, look for a menu named "Debugger". If it's not present, then you need to tell Dr. Java where to find tools.jar.
  - Copy this file: <http://cs.gmu.edu/~marks/211/share/tools.jar> to a place that doesn't have any spaces in the full path. Perhaps /Users/youraccountname/tools.jar would be satisfactory.
  - Go to Preferences, Resource Locations, edit the tools.jar location to be your chosen location, e.g. /Users/youraccountname/tools.jar
  - Try re-opening Dr. Java and ensure you see the Debugger menu. We definitely want to use this later in the semester.

You type code into the largest pane, and then can save it where you like. There are two buttons of current interest across the top: **compile**, and **run**. Compile will cause Dr. Java to try compiling the code on your behalf. You must successfully compile your code with no errors before you are able to then run the file (with the run button).

## Installing Eclipse (unsupported alternative option)

This section is here as a courtesy for anyone who wants to install Eclipse, but you should install Dr. Java for this class. Be sure to focus on getting started on writing Java code, and maybe come back here if you want to install Eclipse later on.

First, go to this link <http://www.eclipse.org/downloads/> and download "Eclipse for Java Developers". Follow the installation instructions.

Eclipse is written in Java, and has support for writing code in many languages. It has many extra features (common to most IDEs), such as code completion, type checking in the background to immediately bring up errors or warnings, and management of your packages and files.

One reason not to use a big IDE right away is that it also creates extra files and folders, requires setting many properties for a "project", and can make it confusing what files and folder structures are actually required to be writing Java code. It also blurs the distinction between editing, compiling, and executing, because you get a nice, shiny "play" button that will compile and run your code all at once, if it can. Some future programming situations will explicitly require you to write code command-line style, so we wanted to cement comfortability with that mode of programming first.

When you open Eclipse, you'll need to specify your workspace. This is a directory where Eclipse can create "projects". It's important that you manually "close project" whenever you go between projects; the play button sometimes is trying to run the other project. To close a project, in the "package explorer" panel, right-click the project and "close project", so that just the one you want to create is open.

We think of each assignment or program as a "project". We can create a new project via `File → New → Java Project`. We similarly add classes with `File → New → Class`. Eclipse tries to be helpful with all sorts of checkboxes and things, but you can always just type things in manually later on, in case you forget to check "create method stubs for inherited abstract methods", or something similar.

Eclipse will create a folder for the project in its workspace directory, and inside of that, all your code will go by default into a "src" directory (source). Finding this workspace directory is sometimes a pain. You will also have to be careful when turning in your work, as this doesn't match the expected folder/file structure at all.

## Navigating through the DOS Prompt / Terminal

This is a skill you will need to get comfortable with at some early point during your computer science career (or any engineering-related career). You might as well get comfortable with it now, or else it will make you very unhappy during some project in this class and later classes, when it will be assumed you know how to navigate in a terminal. However, if you get stuck too much on this, you can rely on Dr. Java's `compile` and `run` buttons instead. Just be sure to get comfortable using the tools at the command line at some point.

### DOS>

In order to navigate through folders, create folders, and so on in the DOS prompt, you will need to get used to some commands. Here are some basics to get you started: Basic DOS Commands (<https://www.sophos.com/en-us/support/knowledgebase/13195.aspx>). For now, just focus on `cd`, `del`, `dir`, `md` & `rd`.

### Terminal\$

UNIX tutorial (<http://people.ischool.berkeley.edu/~kevin/unix-tutorial/toc.html>): useful for navigating around in the terminal (for macs & linux). For now, focus mainly on "looking around", "managing files and folders", and "viewing and editing files".

## Writing Our First Java Program

Now that Java is installed, we can finally get to the simpler world of writing a file, compiling it, and running it. This "edit-compile-run" cycle is at the heart of what we do when we develop a program.

### Writing a Java Program

To write a Java program, we could use any old text editor to create a document with a `.java` extension, containing valid Java code. Any old editor will do, but you will quickly discover that writing code in editors designed for writing code is a far, far superior experience. We've already installed Dr. Java so we will use that.

Once you've decided on an editor (or perhaps only used old Notepad or TextEdit just today!), copy the following into a file named `HelloLab.java` :

```
public class HelloLab {  
  
    public static void main (String[] args) {  
        System.out.println("Hello, Lab!");  
        //more code to go here!  
    }  
}
```

## Compiling a Java Program

Now that you have `HelloLab.java` stored onto your computer, you need to compile it. Open up a prompt/terminal, and use the `cd` command to navigate to the folder containing the file `HelloLab.java`, and execute the following instruction to compile the `.java` file:

```
javac HelloLab.java
```

Unless some odd character encodings occurred during copy-pasting, there should be no errors, and the file `HelloLab.class` has been created. This file is the compiled version of our `HelloLab` class definition. Every Java program contains a `main` method that is run to execute the program. Running a Java program means to call its `main` method. In our case, the only method in `HelloLab` is `main`, and we will cause this method to be run with the next terminal command. Our entire program is the contents of `main`.

In general, when we write code that doesn't represent valid Java, we will get compilation errors here.

## Executing a Java Program

Now that we have a compiled version of our `HelloLab` program, we want to run it. In the prompt/terminal, run:

```
java HelloLab
```

Notice that we did not have any extension after `HelloLab`; `java` looks for `HelloLab.class`, and looks for the `main` method in it and then runs it. It prints out "Hello, Lab!".

## Edit-Compile-Run!

Now that we have a source file that we can edit, a way to compile it, and a way to run it, we are ready to start writing code! Contrast this cycle to an interpreted language with an interactive prompt; if we want to try a new expression, we pretty much have to edit the source file so that it calculates and uses/prints the new value, and then compile and run it again. Later on we'll let an IDE like Dr. Java or Eclipse help speed up this process, but the three phases—edit, compile, run—will still be present, whenever we develop Java code.

## Running Code Through Fancy IDEs

Okay, okay, of course you want to use that fancy editor that also lets you run your code. With Dr. Java installed, you can just click the `compile` and `run` buttons instead of manually invoking `javac` and `java` at the command prompt. But if you get comfortable now using the command-line version, you will thank yourself in some later class where all tools are command-line-only.

## Practice: Java Basics

Now it's time to try out some basic Java language features.

**Literals.** Literals are small, atomic 'building blocks' of our data in Java, such as `5`, `true`, or `'c'`. Java understands each literal to be of a particular **primitive type**. The eight primitive types in Java are:

<b>char:</b>	16-bit unsigned integer representation; used to represent Unicode characters.
<b>byte:</b>	8-bit signed integer representation.
<b>short:</b>	16-bit signed integer representation.

<b>int:</b>	32-bit signed integer representation.
<b>long:</b>	64-bit signed integer representation.
<b>boolean:</b>	truth values. <code>true</code> and <code>false</code> are the only two values. They are lower-case.
<b>float:</b>	32-bit floating-point representation (approximates real numbers). Place an <code>f</code> at the end of the number to distinguish it from double values.
<b>double:</b>	64-bit floating-point representation (approximates real numbers).

Signed means that we have positive and negative numbers available; unsigned means that only non-negative numbers are available (zero and up, to a maximum of  $2^n-1$  for  $n$ -bit numbers).

→ We often just use **int** and **double** for numbers, ignoring `byte`, `short`, and `long`.

**Variables.** Variables are our names for storage in memory. We can declare new variables, and assign values to them; we can later look up these values. The two things we must do in order to use a variable are **declare** it and then **instantiate** (initialize) it.

**Declaration:** give a type, followed by an identifier (name). Multiple variables may be created with a single type followed by a comma-separated listing of identifiers for the variables.

Add these examples to your `HelloLab.java` file (at the `//more code goes here` location):

```
int x;
double f,g;
```

**Instantiation:** use an assignment statement (`varname = expr;`) to give a value to the variable.

Add these examples to your `HelloLab.java` file:

```
x = 5;
f = 2.3;
g = 3.14159;
```

**All together:** We can declare and instantiate all in one go: give a type, a new identifier, `=`, and an initial expression.

Add this example to your `HelloLab.java` file:

```
int y = 7;
```

**Printing things—just the basics.** Later on we'll learn about `System` and everything we need to know about printing, but for now, we just need to feed an expression as the argument to this method call, as shown. We'll also pretend we know all about `Strings` (characters in double-quotes):

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

It will be printed to the terminal when we run our program. So let's print some of our variables out:

```
System.out.println(x);
System.out.println("y equals " + y);
System.out.println("f="+f);
System.out.print("g: ");
System.out.println(g);
```

**Your Turn!** Create a couple variables of each primitive type (including `boolean`). Print them out. It might help to just print them each individually at the end, and change the code ahead of it; you might also prefer to make sequential changes interleaved with print statements. It's all for your benefit, so however you want to do this is up to you.

### Getting input.

Programming is much more interesting when our code can request information from the user. We will take a very brief look at the [Scanner](#) class. Right now we view it more as boilerplate that lets us make the right incantation to get the job done, and later on we will understand what is happening behind the abstraction that a Scanner provides us.

First, to use the Scanner class, we have to include its code with an import statement. Add the following import statement on a line before `public class HelloLab`:

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

We could have also stated `"import java.util.*; "`. This version allows access to any class in the `java.util` package. We will take a much better look at packages later.

Now that we are allowed to use the Scanner class, we create a Scanner object (also stuff we will explain in better detail later on). Place the following line inside the main method before the first time you want to get input from the user—you might as well put it at the very beginning of the method.

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

We are now allowed to use the Scanner object, which we named `sc`, by calling methods on it. Of particular interest to us are the `nextInt()` method and `nextDouble()` methods. There are other similarly-named ones.

It's pretty polite to ask for input before we actually demand it, so we tend to see printing on the line before:

```
System.out.println("please give me a number: ");  
int userNum = sc.nextInt();
```

That's all: each time you want a value of a particular type `Foo`, use the corresponding `nextFoo` method of our `sc` Scanner object, and you'll get the next thing they type (after hitting enter).

Note: the `nextFoo` methods can grab parts of a line; it can be convenient to read many numbers from one line, or it can be annoying to need to manually get to the next line by calling `sc.nextLine()` and discarding the result.

**Your Turn!** Create variables of three different types; ask the user for values that will fit in those types, and then use them (print them, change them, whatever you want to try). This will be useful in the rest of the lab, semester, and your major.

### Expressions versus Statements.

Remember, an expression is just a representation of a calculation that would result in some value (it might even already be a value, like a literal). A statement is a command to the programming language to



do something or perform some action, often involving expressions to describe what should be done. Let's add many assignment statements, using more and more sorts of expressions as our righthand-sides.

### Basic mathematical operations.

We have the usual mathematical operations that operate over all of our numerical types: `+`, `-`, `*`, `/`, `%` (modulo, or remainder). As you may have encountered in other languages, division (`/`) is *closed*, meaning that if we divide an `int` by an `int`, we get back an `int` (by truncating the result—throwing away the remainder). Similarly, if we encounter different types in a division, the "wider" type is chosen. So an `int` divided by a `float` will return a `float`. The order of operations is the same (PEMDAS) as in mathematics, so no surprises there.

**Your Turn!** Try to use all the different mathematical operators between your numerical variables.

### Converting between numerical types.

The action of coercing or converting a number from one type to another is known as 'casting' the value from one type to another. There are default assumptions in Java as to when casting is automatically applied (as in `int/float` division, where the `int` is cast to a `float`), and we can also manually apply a cast.

Here is an idealized notion of the widening of types. Each type on the left is 'smaller', or a 'subset' of the types to its right. (It is idealized because of the imprecision of floating-point representations).

`byte` → `short` → `int` → `long` → `float` → `double`

We perform a cast manually by adding a type in parentheses to the left of the expression:

```
int z = (int) 2.46f;    //convert 2.46 from a float to an int.
z = ((int) 3.14) * 10;  //use parentheses to cast within expr.
```

### Casting Implications.

We say that a cast to a 'wider' type (a type with all the other type's values, and more) preserves the value, but that a cast to a 'narrower' type will lose precision in some sense. Casting from a floating-point number to an integer number will lose the fractional part; casting from a `double` to a `float` may lose some precision. Casting from a larger integer type to a smaller integer type may mean that the number wasn't even representable, and an 'overflow' (modular division) will occur to fit the value into the new, smaller, type.

For more information, read the primitives section of the section on casting <http://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html> in the JLS (especially 5.1.2, 5.5).

**Your Turn!** Try casting between various primitive numerical types. See what happens (via printing) when you cast between wider and smaller types, or from smaller to wider types. See what happens to fractional parts, negative fractional numbers, and so on. Try to create each of the situations:

- A cast completely preserves the old value in the new type.
- A cast causes a truncation (loss of information after the decimal place).
- A cast causes an overflow.
- A cast causes division between two integers to be performed as a float.

**Operators.** There are a variety of operators available to us in Java that we are currently interested in. Try out each of the following:



**Relational operators.**      <      <=      >      >=      ==      !=

- These operators take two numbers, and result in a boolean matching whether the relation holds. E.g., `3<4` results in `true`; `(5%2)<=0` results in `false`.
- Keep in mind that `==` and `!=` are also defined for all primitive types (so, `boolean` and `char`, and the various number types). Just have compatible types on both sides. Java will complain if you attempt something like `false!=4`, because there's no type that contains both `booleans` and `ints`. But it's safe to express `3==3.2` (and the result is `false`). This one works, because Java has implicit casting between the numerical types. The casting always 'upgrades' values to make the types match, rather than 'degrading' values: 3 can easily become 3.0 and also be a double; but for 3.2 to become 3, there's a loss of information. This basic idea should help you figure out which direction the conversion is occurring.

**Mathematical operators.**    +      -      \*      /      %

We mentioned these above (they're listed here for completeness' sake).

**Boolean operators.**      &      |      &&      ||      !      !=

These operators only accept boolean expressions, and result in a boolean value.

- `&` and `&&` both mean "and"; then return `true` only when the expressions on both sides are true, `false` otherwise.
- `|` and `||` both mean "or"; they return `false` only when the expressions on both side end up being false, and `true` otherwise.
- What's the difference here? `&` and `|` will always evaluate the expressions on both side, no matter what. `&&` and `||` are called "short-circuiting operators": when possible, they skip evaluating the righthand expression. `false && whatever` will always be `false`, no matter what value `whatever` has. Similarly, `true || whatever` will be `true`, regardless of `whatever`'s value.
- `!` means "not"; it is a unary operator, meaning we only supply one boolean expression after it. It turns a true value into false, and a false value into true. It gives us alternate ways to phrase things: `!(x>4)` is the same as `(x<=4)`. For instance, we might say `(x<4) || (x>4)` more simply by stating that `!(x==4)` (or even shorter, `x!=4`).

**Ternary operator:**      `boolexpr ? expr : expr`

- The 'conditional expression' is a ternary operator: it requires three expressions, sandwiched around the `?` and the `:`. This is essentially an if-statement occurring at the expression-level. Used at the right time, it is extremely useful! Examples:

```
x = (x<0) ? 0 : x;
String status = (x>0) ? "good" : "bad";
```

- Compare this with an if-statement version:

```
if (x>0){
    status = "good";
}
else {
    status = "bad";
}
```

- The first expression must be a boolean expression; this serves as the decision of whether this entire expression results in the middle-expression (when true), or the last expression (when false).

## **Keep Trying Things!**

You know best what features at this basic level are the most confusing to you; explore things you were able to do in your previous language, and try them out in Java. Lab is a great place to ask questions one-on-one, through the TAs. Next we discuss control structures and methods, but you can try to figure those out now too, if you're already comfortable with basic operators, creating variables, and user input/command-line output.

Keep in mind that the first couple of weeks of this course is heavily focused on getting proficient and comfortable in Java. If you aren't writing code and running it, you aren't getting prepared for the rest of the semester. It's really that simple, so go ahead and get prepared!