CS101 Lecture Notes, Mock
Introduction to Java Programming

Hello, World

The first program that many people write is one that outputs a line of text. In keeping with this vein, we will start with a program that prints, "Hello, world". First, here is the program in its entirety.

File: HelloWorld.java

```
// Put your name and date here
// followed by a description of the code
// This is a Hello World object that has a single
// method to print "hello world"
/*
   We can also use a slash and a star for
   comments… end the comment with a * and a slash
*/
class HelloWorld
{
      public static void main(String[] args)
      {
            System.out.println("hello, world");
      }
}
```

Before we try to compile and run this program, let's go through a description of what is in here. First, any part of a line that begins with // is considered a comment and from the // on to the end of the line, the text is ignored. Descriptive text about what the program does should go into the comment fields. We'll use these to describe the input, output, and dependency behavior of the program. Also use comments as a place to type your name with the file!

An alternate way to enter comments is to use a /* ….(comments) … */. This is useful when you want to comment out a large number of lines and is easier than prefacing each line with //.

A Java program can be composed of many files. In this case, our program consists of a single file called HelloWorld.java. Each file corresponds to what is called a **class**. A class is Java's way of defining an object. However, a Java program should have only one file that contains the special **method** called **main**. A method is essentially a collection of code that performs a specific task.

After the comments, the first line of HelloWorld.java is "class HelloWorld". This defines the name of the class (i.e. the object). **The class name must match the name of the file!** Java is case-sensitive, so Java will complain about a file named "helloworld.java" but the contents of this file contains "HelloWorld" instead. We will place each class object into its own separate file.

The left curly brace is used to denote where the body of the class begins. The right curly brace denotes where the body of the class ends. It is common notation to put the right end brace on a line on its own in the same column as where the class begins, so one can see which curly brace matches with what statement. There is great debate regarding the placement of the curly braces. At this point it is worth mentioning that the compiler doesn't care about **whitespace** between instructions. Whitespace is spaces or carriage returns. You can add as many spaces or blank lines as you wish so that the program is easier for humans to read. You need at least one whitespace character to separate instructions, but others will be ignored. In our example, we have used carriage returns to match up the { and } curly braces, and also have used tabs and indentation to make the program a bit more readable. If you wanted to, you could write the entire program on a single line! This would not make it very human-readable, but the compiler will not care.

The next line is "public static void main(String[] args) {". This defines a method called "main" in the UseHello object. As indicated earlier, main is a special function. This is where your program begins execution! Sometimes methods are referred to as functions. I will use these terms interchangeably. You can think of a method as a collection of code that does some specific task. The main method has a few terms that will look cryptic. You'll learn more about these things later, but for now you can consider this as "boiler plate" that you will just put in all your programs to make them work. Nevertheless, here is a short description of what these terms mean:

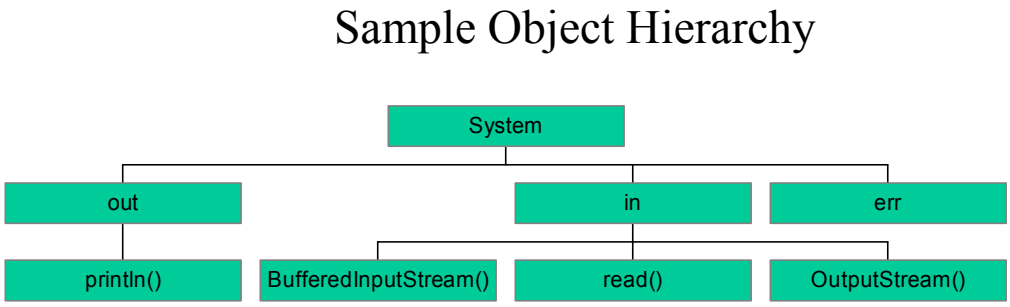
public	- The method is made available to anyone that wants to use it
static	- Only make one copy of this method
void	- This method should not return any value (e.g., functions can be designed to return values, f(x))
String[]	- A String refers to a block of ASCII characters like a word or sentence. The [] indicates that we want an array, or a collection, of many strings.
args	- This is a variable name used to refer to the strings In most of our programs we won't use this, but it refers to command-line arguments passed to our program

The next line is a left curly brace that defines where the main method begins. The right curly brace shows where the main method ends.

The next line is "System.out.println("hello, world."); ". This entire line is called a **statement**. Every statement must end with the semicolon, also known as a statement terminator. One of the things that gets tricky for beginning programmers is where to put the semicolons; it just takes experience to learn where the semicolons go!

All this statement does is output the text "hello, world" to the screen. To do this, it uses the most basic Java operation to produce output: System.out.println("….text string here…");

The dotted notation of using periods starts at a high-level Java object, and each dot indicates a more specific Java object. Objects form a hierarchy. For example, here is a small object hierarchy for System:



System.out.println() invokes the method to print a string of data to the screen. Similarly, System.in.read() would invoke a method to read input from the keyboard. Each "dot" moves us down the hierarchy to a more specific function or object.

To print a string, note that we used double-quotes. Any double-quoted block of characters is considered a literal string, and is created by concatenating together the appropriate ASCII characters.

Finally, the file is terminated with right curly braces to close the HelloWorld class. Once again, you might not understand everything that is going on here, but it should make more sense as we go along.

Compilation

The process of entering and compiling your program is different depending upon what development environment you are using. We'll be using a most primitive development environment – using a text editor in conjunction with the java command-line compiler on mazzy. Other development environments include programs that allow more graphical views of your code. These are called IDE's (Integrated Development Environment). You'll want to use an IDE if you continue to develop larger and more complex Java programs, but the text editor method will work for the simpler programs we'll study in this class. You are also welcome to use an IDE as well, but we won't explicitly support one in class.

To compile your programs, first enter them using a text editor. For example, you could use pico under Unix or Notepad under Windows. Make sure the name of the file matches the name of the class.

After the files have been created, compile them using the javac command. The arguments are the names of the Java source code programs:

- javac HelloWorld.java

If there were any errors, the compiler will complain and tell you it encountered problems. If all goes well, you'll be returned to the prompt and you should now have a file named "HelloWorld.class" in your current directory. This is the compiled versions of the Java source code. To run it, use the Java interpreter:

- java HelloWorld

This will run the program. Note that we didn't enter the ".java" or the ".class" upon executing the program. This command invokes the Java interpreter which will convert the java byte code into native machine code and then run it.

Syntax, Program Structure, Data Types

In this section, we'll look in more detail about the format, structure, and functions involved in creating Java programs.

General Format of a Simple Java Program

In general, your programs will have the following structure:

```
import …

class ClassName {
        public variable declarations;
        private variable declarations;

        public method declarations;
        private method declarations;
}
```

The import statement at the top tells the Java compiler what other pre-compiled packages you want to use. Rather than rewrite commonly used procedures from scratch, you can use pre-built **packages**. The import statement specifies which packages you wish to use. Inside the class, you will declare variables that belong to that class. We'll focus on two categories of variables, public and private, where public is information we'll make available to the rest of the program, and private is only available within the class. Then we'll define methods that make up the class.

Here is a list of the terms that you will commonly encounter:

- **Program** : A general term describing a set of one or more Java classes that can be compiled and executed
- **Variable** : This is a name for a value we can work on. This value will be stored somewhere in memory, or perhaps in a register. Variables can refer to simple values like numbers, or to entire objects.
- **Method** : This is a function. It groups together statements of code to perform some type of task.
- **Class** : This describes the object that contains variables and methods.
- **Object** : This is used interchangeably with class. More specifically, an object is an actual instance of an object created by the new statement.
- **Identifier** : This is the name of an entity in Java, which could be a class, variable, method, etc.
- **Keyword** : This is a reserved word with special meaning to Java and can't be used as an identifier. For example, the word "class"
- **Statement** : This is a single line of code that does a particular task. A statement must be terminated by a semicolon.
- **Parameters** : These are values that are passed to a method that the method will operate on.

So far we've written our first program that could output a line of text. Obviously, we could modify our program to output different types of text if we like by replacing "hello, world" with whatever message we want to print.

Let's expand our program a little bit more to include some **identifiers**. An identifier is made up of letters, numbers, and underscores, but must begin with a letter or an underscore.

Beware: Java is case sensitive. This means that Value, VALUE, value, and vaLue are four separate identifiers. In fact, we can construct 32 distinct identifiers from these five letters by varying the capitalization. Which of these identifiers are valid?

_FoosBall F00sBall %FoosBall 9FoosBall% 12391 _*_ _FF99

Note: When picking identifiers try to select meaningful names!
Here is a short program that uses some variables as identifiers:

```
class OutputTest {
        public static void main(String[] args) {
                char period = '.';                          // Single quotes
                String name = "Cotty, Manny";               // Double quotes
                String foods = "cheese and pasta";
                int someNum = 0xF;                          // 0x indicates hex

                System.out.println(name + " loves to eat " + foods);
                System.out.println(someNum + " times as much as you" + period);
        }
}
```

Let's walk through the program:

Line 1 identifies the name of the class. This program should be in a file named "OutputTest.java".

Line 2 is the name of the main method, as described previously.

Lines 3 through 6 instruct the compiler to assign variables of a particular type. The format is to first indicate the data type identifier, in this case **char**, **String**, or **int**.

- char indicates that the value to be stored is to hold a single ASCII character.
- String (note the uppercase S) indicates that the value to be stored can be composed of many characters.
- int indicates that we want to store an integer value, e.g. using the 2's complement representation.

In line 3, a period is stored in the variable named *period*. In line 4, the string made up of the characters 'C', 'o', 't', 't', 'y', ',' , ' ', 'M', 'a', 'n', 'n', 'y' is stored in *name*.

In line 6, we defined one numeric value, *someNum* to 0xF. This sets *someNum* to fifteen. We have a number of ways of defining numbers:

```
Decimal: Use the normal number, e.g.  9          = decimal 9
Octal:  Use a leading 0, e.g. 011                = decimal 9
Hex: Use a leading 0x,  e.g. 0xF                 = decimal 15
```

For example, line 6 could have been defined in the following equivalent ways:

```
int someNum = 15;                    // decimal 15
int someNum = 0xF;                   // hexadecimal 15
int somenum = 017;                   // octal 15
```

When the compiler processes the variable declarations, it assigns a memory location to each one. It is intended that the data we store in each memory location is of the same

type that we defined it in the program. For example, we defined someNum to be of type int. This means we should only store integers into the memory location allocated for someNum. We shouldn't be storing floating point numbers, strings, or characters.

Also note that it is customary to define all variables used within a method at the top of the method (in this case, the method is main).

Finally, we have output statements that print the contents of the variables. Note that when we use println, we can print out variables of different types and **concatenate** or stick together the output using the + symbol. The + symbol will also serve as addition as we will see later! So be aware that a symbol may do different things in a different context.

The final output when run is:

> Cotty, Manny loves to eat cheese and pasta
> 15 times as much as you.

Words and Symbols with Special Meanings

Certain words have predefined meanings within the Java language; these are called *reserved words* or *keywords*. For example, the names of data types are reserved words. In the sample program there are reserved words: **char**, **int**, **void, main**. We aren't allowed to reserved words as names for your identifiers.

Data Types

A data type is a set of values and a set of operations on these values. In the preceding program, we used the data type identifiers **int**, **char**, and **String**.

In Java there are four integral types that can be used to refer to an integer value (whole numbers with no fractional parts). These types are **byte**, **short**, **int**, and **long** and are intended to represent integers of different sizes. These are just integers stored using the 2's complement format that by now you should know and love. The set of values for each of these integral data types is the range of numbers from the smallest value that can be represented through the largest value that can be represented. The operations on these values are the standard arithmetic operations allowed on integer values.

the sizes for the integer types are:

> **byte** - one byte, holds a number from –128 to 127
> **short** – two bytes, holds numbers up to 32767
> **int** – four bytes, holds numbers from –2,147,483,648 to 2,147,483,647 (2^{31}),
> **long** – eight bytes , holds numbers up to around 10^{18}

You might always be tempted to use type long for all numbers because it can hold the largest range. While this is possible, it would not be efficient if the values your program

is processing are all small. In this case, you'll be wasting lots of bits in allocating eight bytes of space when you might really only be using one byte.

Data type **char**, which can also be used to store bytes, has a primary use for describing one alphanumeric character. Although arithmetic operations are defined on alphanumeric characters because they are type **char** (also an integral type), such operations would not make any sense to us at this point. However, there is a collating sequence defined on each character set, so we can ask if one character comes before another character (the ASCII code). Fortunately, the uppercase letters, the lowercase letters, and the digits are in order in all character sets. The relationship between these groups varies, however. We discuss manipulating **char** data in more detail later.

We used two variables of data type **String**, *name* and *foods*. String is actually not an integral data type; it is an object. We'll say more about the difference later. For now, you can use String to hold a sequence of characters. Note that string data is denoted with double quotes, not a single quote as for char. "I" refers to the string with the letter I in it, while 'I' refers to the character with the letter I. The two are different, they are different types and are stored in a dramatically different way!

Finally, there are separate data types for fractional or floating point numbers. These are numbers that are stored using the IEEE 754 format we discussed previously in class. There are two types of floating point storage:

float - 4 bytes, using IEEE 754. Values up to 10^{38} possible
double – 8 bytes, using IEEE 754. Values up to 10^{308} possible

Arithmetic Expressions

Variables and constants of integral and floating point types can be combined into expressions using arithmetic operators. The operations between constants or variables of these types are addition (+), subtraction (-), multiplication (*), and division (/). If the operands of the division operation are integral, the result is the integral quotient. If the operands are floating point types, the result is a floating point type with the division carried out to as many decimal places as the type allows. There is an additional operator for integral types, the modulus operator (**%**). This operator returns the remainder from integer division.

In addition to the standard arithmetic operators, Java provides an *increment* operator and a *decrement* operator. The increment operator ++ adds one to its operand; the decrement operator -- subtracts one from its operand.

Examples:

```
class OutputTest {
      public static void main(String[] args) {
            int x=1;

            x = x + 55;
            System.out.println(x);
      }
}
```

This produces the output of: 56

We have set x to 1, then we set x to 1+55 or 56 which is then output.

If we change the line "x = x + 55" to:

x = 5 * 10 * 2;	the output is: 100
x = 14 % 5;	the output is: 4
x = 10 /2 ;	the output is: 5
x++;	the output is: 2 (this adds 1 to x)
x--;	the output is 0 (this subtracts 1)
x = 11 / 2;	the output is: 5 (remainder discarded)
x = 1 / 2;	the output is 0 (remainder discarded)
x = 100000000 * 100000000;	the output is some weird value (may get overflow warning)

Note : Truncation, not rounded to nearest integer

The last three examples may require some explanation.

For the line x = 11 / 2;

We divide 11 by 2 to get 5.5 However, x is an integer. It cannot store floating point values like 5.5. Java truncates the results, i.e. it throws away any value after the decimal point. So we simply get 5.

For the line x = 1 / 2, we get 0.5 As in the above example, we throw away anything after the decimal point to get 0.

The last example results in a strange value because of an overflow. Java's representation for an integer does not contain enough bits to accurately store an integer this large, so we instead get a weird number.

Precedence Rules

The precedence rules of arithmetic apply to arithmetic expressions in a program. That is, the order of execution of an expression that contains more than one operation is determined by the precedence rules of arithmetic. These rules state that:

1. parentheses have the highest precedence
2. multiplication, division, and modulus have the next highest precedence
3. addition and subtraction have the lowest precedence.

Because parentheses have the highest precedence, they can be used to change the order in which operations are executed. When operators have the same precedence, order is left to right.

Examples:

x = 1 + 2 + 3 / 6;	x ← 1 + 2 + 0 = 3
x = (1 + 2 + 3) / 6;	x ← 6 /6 = 1
x = 2*3 + 4 * 5;	x ← 6 + 20 = 26
x = 2 / 4 * 4 / 2;	x ← 0 * 4 / 2 = 0 / 2 = 0
x = 4 / 2 * 2 / 4;	x ← 2 * 2 / 4 = 4 / 4 = 1
x = 10 % 2 + 1;	x ← 0 + 1 = 1

Converting Numeric Types

If an integral and a floating point variable or constant are mixed in an operation, the integral value is changed temporarily to its equivalent floating point representation before the operation is executed. This automatic conversion of an integral value to a floating point value is called *type coercion*. Type coercion also occurs when a floating point value is assigned to an integral variable. Coercion from an integer to a floating point is exact. Although the two values are represented differently in memory, both representations are exact. However, when a floating point value is coerced into an integral value, loss of information occurs unless the floating point value is a whole number. That is, 1.0 can be coerced into 1, but what about 1.5? Is it coerced into 1 or 2? In Java when a floating point value is coerced into an integral value, the floating point value is **truncated**. Thus, the floating point value 1.5 is coerced into 1.

Type changes can be made explicit by placing the new type in parentheses in front of the value:

```
intValue = (int) 10.66;
```

produce the value 10 in intValue

Here are some same typecasts:

(double) intValue;	(int) doubleValue;
(long) intValue;	(int) longValue;
(long) floatValue;	

Examples with float:

```
class OutputTest {
      public static void main(String[] args) {
            float x=1;

            x = 11 / 2;
            System.out.println(x);
      }
}
```

You might think this would produce 5.5. But instead it produces 5.0. Why?
The answer is because the expression 11/2 is computed as an integer expression. This throws away anything after the decimal point. So then we get x = 5 even though x is capable of holding 5.5

How about the following:

```
x = (float) 11 / 2;          // Since 11 is a float, 2 is also turned into a float
                             // and we also get x ← 5
x = 11 / (float) 2;          // Similar to above

x =  2 / 4 * 4 / 2;          // This one computes 0 * 4 / 2 or x ← 0

x = (float) 2 / 4 * 4 / 2;   // This one works!  By turning 2 into a float
                             // the rest of the computation is done as a float

x = 2 / 4.0 * 4 / 2;         // 4.0 treated as a double, turns the entire
                             // computation into a double, and we get
                             // x ← 0.5 * 4 / 2 or x = 1
                             // Compiler may complain about double to float
```

The bottom line here is to be careful if you are mixing integers with floating point values in arithmetic expressions. Especially if performing division, you might end up with zero when you really want a floating point fractional answer. The solution is to coerce one of the integers into a float or double so the entire calculation is made using floating point.

Let's put together what we know so far with an example program. Here is the problem:

You are running a marathon (26.2 miles) and would like to know what your finishing time will be if you run a particular pace. Most runners calculate pace in terms of minutes per mile. So for example, let's say you can run at 7 minutes and 30 seconds per mile. Write a program that calculates the finishing time and outputs the answer in hours, minutes, and seconds.

Input:

Distance : 26.2
PaceMinutes: 7
PaceSeconds: 30

Output:

3 hours, 16 minutes, 30 seconds

Here is one algorithm to solve this problem:

1. Express pace in terms of seconds per mile, call this SecsPerMile
2. Multiply SecsPerMile * 26.2 to get the total number of seconds to finish. Call this result TotalSeconds.
3. There are 60 seconds per minute and 60 minutes per hour, for a total of 60*60 = 3600 seconds per hour. If we divide TotalSeconds by 3600 and throw away the remainder, this is how many hours it takes to finish.
4. TotalSeconds mod 3600 gives us the number of seconds leftover after the hours have been accounted for. If we divide this value by 60, it gives us the number of minutes.
5. TotalSeconds mod 3600 gives us the number of seconds leftover after the hours have been accounted for. If we mod this value by 60, it gives us the number of seconds leftover. (We could also divide by 60, but that doesn't change the result).
6. Output the values we calculated!

Code:

```
class RacePace {
      public static void main(String[] args) {
        double distance = 26.2;
        int paceMinutes = 7;
        int paceSeconds = 30;

        long secsPerMile, totalSeconds;

        secsPerMile = (paceMinutes * 60) + paceSeconds;
        totalSeconds = (long) (distance * secsPerMile);
        System.out.print("You will finish in: ");
        System.out.print(totalSeconds / 3600 + " hours, ");
        System.out.print((totalSeconds % 3600) / 60 + " minutes, ");
        System.out.println((totalSeconds % 3600) % 60 + " seconds.");
      }
}
```

A few things to note about this program:

```
totalSeconds = (long) (distance * secsPerMile);
```

Since distance is a double and secsPerMile is an int, the whole thing is typecast to a long, which is the type of totalSeconds.

System.out.print is the same as System.out.println, except System.out.print does not add a newline to the end of the output. This means that everything gets printed onto the same line until the very final statement, which prints a newline with the println statement.

The output is;

```
You will finish in: 3 hours, 16 minutes, 30 seconds.
```

If we wanted to calculate the finish time for different distances and different paces, we'll need to change the values in the program and recompile it. This can be inconvenient – it would be nice to have the user input any values he or she desires. In the next few lectures we'll see how to input values from the user using the book's Console class.