HW 0: A Java Crash Course

HW Goals

In this assignment, we will go through basic Java syntax concepts. While this assignment is optional and you will not submit your answers, it is highly recommended for those with no prior Java experience. The lectures will NOT cover this material explicitly, though you are expected to understand it.

This tutorial assumes that you have significant (one semester) experience with at least some programming language, and is intended only to highlight the Java way of doing some previously familiar things.

While I hope this document should stand alone for the curious and self-motivated student, you may find it helpful to read the suggested supplementary reading when provided.

Feel free to skim and read at whatever pace you feel comfortable with. It is OK to skip parts of this HW. Use your best judgment. The directions are a bit more verbose than is probably necessary.

A Basic Program

Supplementary Reading: N/A

In Lab 1, we'll learn how to run Java code on your computer. Since lab 1 hasn't happened yet, we'll instead use an in-browser Java compiler for this HW only. Head to this link, or try this link if the link if that first one doesn't work.

You'll find yourself looking at what is perhaps your first Java program. There sure is a lot of weird stuff here, like `public class` and `public static void main(String[] args)`. We'll discuss these in more detail later, but for this HW, you should ignore all of this mysterious garbage.

Click on the 'Forward >' link twice. You'll see an `x` appear in a blue box the right with the value `5` once the line `int x = 5 is executed`. Perhaps unsurprisingly, this statement assigns the value 5 to the variable x.

Unlike other programming languages (like Python, Scheme, and MATLAB), Java variables have a static type. By this, I mean that `x` will only ever be able to store an integer. If you tried to put a number like `5.3` into it, the code would fail.

Also unlike these other languages, every statement in Java must be followed by a semicolon. The semicolon is very important, as it tells Java where one statement ends and another begins.

Click forward again, and you'll see that `x` has changed to `6`. **Click forward one more time**, and you'll see that `x` is printed in the Program output box below using the rather verbose command name `System.out.println`. Yes, this is really how you print in Java. And, it'll get more verbose, trust me.

Ordinarily, when you write Java programs, you won't be able

to see into your program's brain (i.e. there will be no blue box listing all the variables). However, this visualizer is a pedagogical tool that makes such brain scanning possible.

Click forward until the program completes execution. Everything should behave more or less as you'd expect. If anything surprises you, post to the HW0 thread on the course Piazza page.

Optional: Try editing the code and running it again. Experiment and see what happens as you tweak the program. If you have interesting observations or any questions arise, post them on Piazza. Maybe try assigning a real number (e.g. 3.3) and see what occurs (I promise your computer won't explode).

Conditionals

Optional Supplementary Reading: Shewchuk

Basic Conditionals

Now open up this program (or this), shown below:

```
public class ClassNameHere {
    public static void main(String[] args) {
        int x = 5;

        if (x < 10)
            x = x + 10;

        if (x < 10)
            x = x + 10;

        System.out.println(x);
    }
}
```

Step forward until the program completes and observe the flow of the program. The `if` statement in java checks the condition that you put inside parentheses, and if the result is true, it executes the next statement below.

Curly Braces (and Conditionals)

It is also possible to execute multiple statements in response to a single condition. We do this by wrapping the statements in curly braces, as in this program (or this), shown below:

```
public class ConditionalsWithBlocks {
   public static void main(String[] args) {
      int x = 5;

      if (x < 10) {
         System.out.println("I shall increment x by
         x = x + 10;
      }

      if (x < 10) {
         System.out.println("I shall increment x by
         x = x + 10;
      }

      System.out.println(x);
   }
}
```

Curly braces are very important in Java! Unlike Python, statements are grouped by braces, and NOT by indentation. For an example of how this can go terribly wrong, try running the following program (or use this link), which is supposed to print the absolute value of X. **Then try changing the value of x to a positive number. Run it and make sure you understand why things go wrong.**

```
public class PrintAbsoluteValue {
```

```
    public static void main(String[] args) {
        int x = -5;

        if (x < 0)
            System.out.println("I should negate X")
            x = -x;

        System.out.println(x);
    }
}
```

Unlike Python, most whitespace including indentation does not matter with regards to the functionality of your program. In fact, you can get away with replacing every whitespace in your entire program with a single space (given that semicolons are the separators between statements), though this is a horrible idea and we will be very sad if you write programs like the following valid Java program:

```
public class ClassNameHere { public static void mai
```

Curly Brace Standards

There are two common styles for curly braces:

```
if (x > 5) {
    x = x + 5;
}
```

```
if (x > 5)
{
   x = x + 5;
}
```

Which of these two styles is a bit of a holy war. Both are fine for use in 61B. Note that in this example, we've wrapped curly braces around a single statement, which isn't required in Java. In 61B, we'll ALWAYS use curly braces, even if we have only one statement to execute. This is to avoid bugs. Don't fret too much about these little details, the automated

style checker will yell at you if you do something uncouth.

For more than you ever wanted to know about indentation styles, see this wiki page.

Else

The `else` keyword allows you to specify behavior that should occur if a condition is false. For example, this program (or this), shown below:

```
int x = 9;
if (x - 3 > 8) {
    System.out.println("x - 3 is greater than 8");
} else {
    System.out.println("x - 3 is not greater than 8
}
```

We can also chain else statements, like in this program (or this):

```
int dogSize = 20;
if (dogSize >= 50) {
    System.out.println("woof!");
} else if (dogSize >= 10) {
    System.out.println("bark!");
} else {
    System.out.println("yip!");
}
```

Note that in the code above, I've used `>=`, which means greater than or equal.

While

Optional Supplementary Reading: Shewchuk

The `while` keyword lets you repeat a block of code as long as some condition is true. For example, this program (or this), shown below:

```
int bottles = 99;
while (bottles > 0) {
    System.out.println(bottles + " bottles of beer o
    bottles = bottles - 1;
}
```

Try running this program, and watch what happens. Note that as soon as the code inside curly braces is completed, we head straight back to the while condition. Optionally, experiment a bit: Try and see what happens if you start bottles off at -4. Also try and see what happens if you remove the line: `bottles = bottles - 1`;

Important note: You should think of your program as running in order, line by line. If the condition becomes false in the middle of the loop, the code does not simply stop. So for example, the code below (also here) will print "-312 bottles of beer on the wall." even though -312 is not greater than 0.

```
int bottles = 5;
while (bottles > 0) {
    bottles = -312;
    System.out.println(bottles + " bottles of beer o
}
```

Doubles and Strings

Above, all of our variables have been of type int. There are many other types that you can use in Java. Two examples of

these are double and String. double stores approximations of real numbers, and String stores strings of characters. The program below (also here) simulates a race between Achilles and a Tortoise. Achilles is twice as fast, so should overtake the Tortoise (who has a head start of 100 distance units).

```
String a = "Achilles";
String t = "Tortoise";
double aPos = 0;
double tPos = 100;
double aSpeed = 20;
double tSpeed = 10;
double totalTime = 0;
while (aPos < tPos) {
    System.out.println("At time: " + totalTime);
    System.out.println("    " + a + " is at positio
    System.out.println("    " + t + " is at positio

    double timeToReach = (tPos - aPos) / aSpeed;
    totalTime = totalTime + timeToReach;
    aPos = aPos + timeToReach * aSpeed;
    tPos = tPos + timeToReach * tSpeed;
}
```

Creative Exercise 1a: Drawing a Triangle

Finally! A chance to do something on your own.

Your goal is to create a program that prints the following figure. Your code should use loops (i.e. shouldn't just be five print statements, that's no fun).

```
*
**
***
****
*****
```

You can either write the program from scratch, or you can copy and paste lines of code from this link. You may find `System.out.print` to be a useful alternative to `System.out.println`. The difference is that `System.out.print` does not include an automatic newline.

If you go the copy and paste route, note that lines may be used once, multiple times, or not at all.

Run your code and verify that it works correctly by comparing it by eye to the program above. In next week's lab and hw, we'll discuss more sophisticated ways of verifying program correctness.

Save your code someplace (say by emailing it to yourself), as you'll need it again soon.

Defining Functions (a.k.a. Methods)

The following four pieces of code are all equivalent in Python, MATLAB, Scheme, and Java. Each defines a function that returns the maximum of two values and then prints the maximum of 5 and 15.

Python

```
def max(x, y):
    if (x > y):
        return x
    return y

print(max(5, 15))
```

MATLAB

```
function m = max(x, y)
    if (x > y)
        m = x
    else
        m = y
    end
end

disp(max(5, 15))
```

Scheme

```
(define max (lambda (x y) (if (> x y) x y)))
(display (max 5 15)) (newline)
```

Java

```
public static int max(int x, int y) {
    if (x > y) {
        return x;
    }
    return y;
}

public static void main(String[] args) {
    System.out.println(max(10, 15));
}
```

(program link 1, 2)

Functions in Java, like variables, have a specific return type. The `max` function has a return type of `int` (indicated by the word int right before the function name). Also functions in Java are called methods, so I'm going to start calling them from this moment on forever.

We refer to the entire string `public static int max(int x,`

`int y)` as the method's **signature**, as it lists the parameters, return type, name, and any modifiers. Here our modifiers are `public` and `static`, though we won't learn what these mean for a few days.

For this homework, all methods are going to have "public static" at the front of their signature. Just accept this for now. We'll talk more about this on Friday in lecture.

Creative Exercise 1b: DrawTriangle

Starting from the default program at our Java visualizer (also found here), create a program with one additional method (in addition to the default main method that is there when you open the visualizer).

Name this new method `drawTriangle` and give it a return type of `void` (this means that it doesn't return anything at all).

The `drawTriangle` method should take one parameter named `N`, and it should print out a triangle exactly like your triangle from exercise 1a, but `N` asterisks tall instead of 5.

After writing `DrawTriangle`, modify the main function so that it calls `DrawTriangle` with `N = 10`.

Depending on your programming background, you may find this task quite challenging. We encourage you to work with others or post to Piazza. If you're just confused about where to start, this program (also found here) starts with `DrawTriangle` already defined (but without the details

implemented).

Arrays

Optional Supplementary Reading: Shewchuk

Our final new syntax item of this HW is the array. Arrays are like vectors in Scheme, lists in Python, and arrays in MATLAB.

The following four programs in Python, MATLAB, Scheme, and Java declare a new array of the integers 4, 7, and 10, and then prints the 7.

Python

```
numbers = [4, 7, 10]
print(numbers[1])
```

MATLAB

```
numbers = [4 7 10]
disp(numbers(2))
```

Scheme

```
(define numbers #(4 7 10))
(display (vector-ref numbers 1)) (newline)
```

Java

```
int[] numbers = new int[3];
numbers[0] = 4;
numbers[1] = 7;
numbers[2] = 10;
System.out.println(numbers[1]);
```

Or in an alternate (but less general) shorthand:

Alternate Java

```
int[] numbers = new int[]{4, 7, 10};
System.out.println(numbers[1]);
```

(program link)

You can get the length of an array by using `.length`, for example, the following code would print 3:

```
int[] numbers = new int[]{4, 7, 10};
System.out.println(numbers.length);
```

Exercise 2

Using everything you've learned so far on this homework, you'll now create a function with the signature `public static int max(int[] m)` that returns the maximum value of an int array. You may assume that all of the numbers are greater than or equal to zero.

Modify the code below (also found here) so that `max` works as described. Furthermore, modify `main` so that the `max` method is called on the given array and its max printed out (in this case, it should print 22).

```
public class ClassNameHere {
    /** Returns the maximum value from m. */
    public static int max(int[] m) {
        return 0;
    }
    public static void main(String[] args) {
       int[] numbers = new int[]{9, 2, 15, 2, 22, 1
    }
}
```

For Loops

Consider the function below, which sums the elements of an array.

```
public class ClassNameHere {
    /** Uses a while loop to sum a. */
    public static int whileSum(int[] a) {
      int i = 0; //initialization
      int sum = 0;
      while (i < a.length) { //termination
            sum = sum + a[i];
            i = i + 1; //increment
      }
      return sum;
    }
}
```

Programmers in the 1950s observed that it was very common to have code that featured **initialization** of a variable, followed by a loop that begins by checking for a **termination** condition and ends with an **increment** operation. Thus was born the for loop.

The `sum` function below uses a basic for loop to do the exact same job of the `whileSum` function above.

```
public class ClassNameHere {
    /** Uses a basic for loop to sum a. */
    public static int sum(int[] a) {
```

```
        int sum = 0;
        for (int i = 0; i < a.length; i = i + 1) {
          sum = sum + a[i];
        }
        return sum;
    }
}
```

Try it out using this link.

In Java, the `for` keyword has the syntax below:

```
for (initialization; termination; increment) {
    statement(s)
}
```

The initialization, termination, and increment must be semicolon separated. Each of these three can feature multiple comma-separated statements, e.g.

```
for (int i = 0, j = 10; i < j; i += 1) {
    System.out.println(i + j);
}
```

Comma separated for loops should be used sparingly.

Exercise 3

Rewrite your solution to Exercise 2 so that it uses a for loop. Use your original solution as starter code, or if you've lost it, you can use this or this).

```
public class ClassNameHere {
    /** Returns the maximum value from m using a fo
    public static int forMax(int[] m) {
        return 0;
    }
    public static void main(String[] args) {
```

```
        int[] numbers = new int[]{9, 2, 15, 2, 22, 1
    }
}
```

Break and Continue

Occasionally, you may find it useful to use the `break` or `continue` keywords. The `continue` statement skips the current iteration of the loop, effectively jumping straight to the increment condition.

For example the code below prints each String from an array three times, but skips any strings that contain “horse”. You can try it out at this link.

```
public class ContinueDemo {
    public static void main(String[] args) {
        String[] a = {"cat", "dog", "laser horse",

        for (int i = 0; i < a.length; i += 1) {
            if (a[i].contains("horse")) {
                continue;
            }
            for (int j = 0; j < 3; j += 1) {
                System.out.println(a[i]);
            }
        }
    }
}
```

By contrast, the `break` keyword completely terminates the innermost loop when it is called. For example the code below prints each String from an array three times, except for strings that contain horse, which are only printed once. You can try it out at this link.

```
public class BreakDemo {
    public static void main(String[] args) {
```

```
        String[] a = {"cat", "dog", "laser horse",

        for (int i = 0; i < a.length; i += 1) {
            for (int j = 0; j < 3; j += 1) {
                System.out.println(a[i]);
                if (a[i].contains("horse")) {
                    break;
                }
            }
        }
    }
}
```

`break` and `continue` also work for `while` loops and `do-while` loops. If you're curious about `do-while` loops, see the official Java looping tutorial.

Optional: Exercise 4

This is a particularly challenging exercise, but strongly recommended.

Write a function `windowPosSum(int[] a, int n)` that replaces each element a[i] with the sum of a[i] through a[i + n], but only if a[i] is positive valued. If there are not enough values because we reach the end of the array, we sum only as many values as we have.

For example, suppose we call `windowPosSum` with the array `a = {1, 2, -3, 4, 5, 4}`, and `n = 3`. In this case, we'd:

- Replace a[0] with a[0] + a[1] + a[2] + a[3].
- Replace a[1] with a[1] + a[2] + a[3] + a[4].
- Not do anything to a[2] because it's negative.
- Replace a[3] with a[3] + a[4] + a[5].
- Replace a[4] with a[4] + a[5].

- Not do anything with a[5] because there are no values after a[5].

Thus, the result after calling `windowPosSum` would be `{4, 8, -3, 13, 9, 4}`.

As another example, if we called `windowPosSum` with the array `a = {1, -1, -1, 10, 5, -1}`, and `n = 2`, we'd get `{-1, -1, -1, 14, 4, -1}`.

```
public class BreakContinue {
  public static void windowPosSum(int[] a, int n) {
    /** your code here */
  }

  public static void main(String[] args) {
    int[] a = {1, 2, -3, 4, 5, 4};
    int n = 3;
    windowPosSum(a, n);

    // Should print 4, 8, -3, 13, 9, 4
    System.out.println(java.util.Arrays.toString(a)
  }
}
```

Starter code is available at this link.

Hint 1: Use two for loops.

Hint 2: Use `continue` to skip negative values.

Hint 3: Use `break` to avoid going over the end of the array.

The Enhanced For Loop

Java also supports iteration through an array using an "enhanced for loop". The basic idea is that there are many circumstances where we don't actually care about the index at all. In this case, we avoid creating an index variable using

a special syntax involving a colon.

For example, in the code below, we do the exact thing as in `BreakDemo` above. However, in this case, we do not create an index `i`. Instead, the String `s` takes on the identity of each String in `a` exactly once, starting from `a[0]`, all the way up to `a[a.length - 1]`. You can try out this code at this link.

```
public class EnhancedForBreakDemo {
    public static void main(String[] args) {
        String[] a = {"cat", "dog", "laser horse",

        for (String s : a) {
            for (int j = 0; j < 3; j += 1) {
                System.out.println(s);
                if (s.contains("horse")) {
                    break;
                }
            }
        }
    }
}
```