Java Introduction

DS730

In this activity, you will be learning a lot of the key classes that Java has to offer you. We will be creating a simple Java program to start. Then we will follow that up with some of the more advanced features that Java has to offer. The syntax that Java uses is very similar to Python. However, Java is a bit more stringent on how the program is setup and different rules that must be followed.

In order to test the multithreaded problems at the end of this activity, you are encouraged to create a t2.2xlarge EC2 instances on AWS. You should run all of your code on that machine.

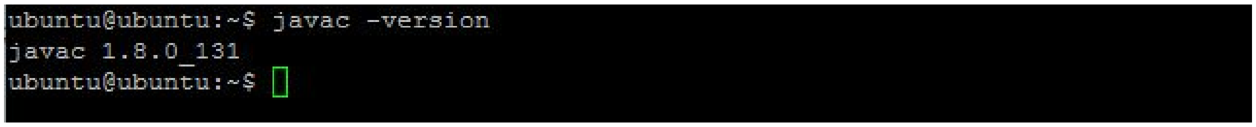
The point of learning Java is to learn how multithreading works. I do not expect you to master the language. Therefore, if you are having problems with the syntax of Java, do not spend hours trying to figure out how to do something simple. If you cannot find a solution to some compiler/syntax error after a few minutes, ask an instructor (or ask a generic question on the message board) and someone will point you in the right direction. There are many examples in this activity to point you in the right direction. You are welcome to download the multithreaded template code to use in your multithreaded solutions:

http://www.uwosh.edu/faculty_staff/krohne/ds730/javaActivity/RunnerTemplate.java
http://www.uwosh.edu/faculty_staff/krohne/ds730/javaActivity/MyTemplateThread.java

This activity is designed to show a Python programmer how to write code in Java. For those of you who are familiar with Java, you are welcome to skip the explanations and simply work on the problems. Some of this is repeating what was said in the presentations but if you are new to Java, doing it in an activity is not a bad idea.

Lastly, I would discourage you from doing a lot of searching about multithreading in Java. The threading templates that are provided in the presentations and in this activity are exactly what you should be doing. If you stray too far from what is presented, you are probably making your solution a lot harder than it needs to be. In other words, if you search for multithreading in Java, you'll come across a ton of different ways that are a lot harder than what is in the presentations. I've seen some really, really weird solutions from people. When I provided my pseudocode solution to the problems, many of them said something along the lines of "oh, that was a lot like what you showed us and a lot easier than what I did." All of this to say: search at your own risk.

1. The syntax of Java is quite similar to Python. This task is meant to show you that if you know Python, then you essentially know Java.
    a. Connect to your EC2 instance and type in **javac -version**. You should see something similar to this:



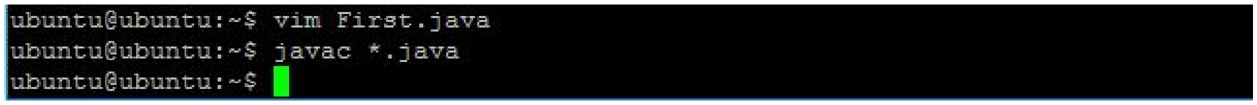
```
ubuntu@ubuntu:~$ javac -version
javac 1.8.0_131
ubuntu@ubuntu:~$
```

    b. Type in **vim First.java**. You should be mostly familiar with vim by now (or whatever your favorite editor is) so I will not talk about how to insert text, save or quit. See previous activities if you do not remember.
    c. In this course, all Java classes you create will have the same name as the file you created. Since we created a **First.java** file, we want our class to be called **First**. Insert the following text into your First.java file:

```
public class First{
    public static void main(String args[]){
        System.out.println("This is a test");
    }
}
```

The structure of this code is going to be almost identical in every other Java class you create. Start with the name of the class. Inside the class (between the red curly braces) is the name of our main function. The print statement in Java is similar to what you see in Python. The *ln* at the end of the print*ln* is a way to print a newline. If you want to just print some text without the newline, use System.out.print.

    d. Save your file and go back to the terminal window. Java is a compiled language. You cannot just run it like you do with Python. Because of this, you must *compile* your code into something the machine can understand. To do this, type in **javac *.java** to compile. Assuming everything went well, you should see this:



```
ubuntu@ubuntu:~$ vim First.java
ubuntu@ubuntu:~$ javac *.java
ubuntu@ubuntu:~$
```

    e. In order to run the program, you must type in **java First**. The Java interpreter will look for the main function that you create and will start executing.

f. Create a file called Second.java and insert the following code into that file:

```
//This import statement is needed to use the Scanner.
import java.util.*;
public class Second{
    public static void main(String args[]){
        System.out.print("Enter a number: ");
        //Scanner is used to read in values from the user
        Scanner input = new Scanner(System.in);
        //read in a value from the user and store the value into
        //a variable called number
        int number = input.nextInt();
        System.out.println("Value was: " + number);
    }
}
```

In this program, we are using the Scanner class to read in a value from the command prompt and then immediately print it back out to the user. Comments for each new statement are shown above the statement. A comment starts with two forward slashes: //.

g. Edit the Second.java to be the following:

```
import java.util.*;
public class Second{
    public static void main(String args[]){
        System.out.print("Enter a number: ");
        Scanner input = new Scanner(System.in);
        int number = input.nextInt();
        //if statement checks if the number was less than 0
        //else if checks if the number was 0
        //else is everything else, i.e. it was positive

        if(number < 0){
            System.out.println("Number was negative.");
        } else if(number == 0){
            System.out.println("Number was 0.");
        } else{
            System.out.println("Number was positive.");
        }

    }
}
```

In this program, we are using a simple if, else if, else clause to do 1 of 3 things. The *else if* in Java is identical to the *elif* in Python. You are able to add more than one check in the condition. For example, if you want to check if *number* < *0* and *value* > *0*, you would use the following if statement:

```
if(number < 0 && value > 0)
```

The double & is the way to check if both of them are true. If you only care about one of them being true, you can use the *or* operator:

```
if(number < 0 || value > 0)
```

Lastly, if you want to check the opposite of something, you use the exclamation point:

```
if(!(number < 0))
```

These are similar in Python. Feel free to compile and run the above code. You are also encouraged to make tweaks to the code to ensure you understand how it works. For example, what happens if you only change the else if part to say *number* == *-5* instead of *number* == *0*. What happens when the user types in -5? If you are not 100% sure you know what happens, test it out and see if you were right.

h. Edit the Second.java to be the following:

```
import java.util.*;
public class Second{
    public static void main(String args[]){
        System.out.print("Enter a number: ");
        Scanner input = new Scanner(System.in);
        int number = input.nextInt();
        //a repetitive structure that prints out the integers
        //from number down to 0 without printing out 0.
        while(number > 0){
            System.out.println(number);
            number--;
        }
    }
}
```

In this program, we are using a simple while loop. The while loop syntax in Java is almost identical to the while loop in Python. As with the if step, feel free to compile and run the above code. You are also encouraged to make tweaks to the code to ensure you understand how it works.

i. Edit the Second.java to be the following:

```
import java.util.*;
```

```
public class Second{
    public static void main(String args[]){
        System.out.print("Enter a number: ");
        Scanner input = new Scanner(System.in);
        int number = input.nextInt();
        //a repetitive structure that prints out the integers
        //from 0 up to number without printing out number.
        for(int count = 0; count < number; count++){
            System.out.println(count);
        }

    }
}
```

In this program, we are using a simple for loop. The for loop syntax in Java is a bit different from the for loop in Python. The first statement is an initial statement. It's the first thing that happens and it only happens once:
int count = 0;
Once the initial statement happens, the condition is checked:
count < number;
If count is less than number, then the body of the for loop is executed. In this case, we are just printing out the value of count. Once the body has executed, the increment step is executed:
count++;
After the incrementing step is executed, the condition is checked again. If count is less than number, the body is executed (i.e. count is printed). Then the incrementing step is done (i.e. count++) and the condition is checked… This continues until the condition is no longer true. In this case, when count >= number. As with the previous steps, feel free to compile and run the above code. You are also encouraged to make tweaks to the code to ensure you understand how it works.

j. Edit the Second.java to be the following:

```
import java.util.*;
public class Second{
    public static void main(String args[]){
        System.out.print("How many numbers: ");
        Scanner input = new Scanner(System.in);
        int size = input.nextInt();
        ArrayList<Integer> values = new ArrayList<>();
        for(int count = 0; count < size; count++){
            System.out.print("Enter value "+(count+1)+": ");
            int temp = input.nextInt();
            values.add(temp);
        }
        for(int value : values){
            System.out.println(value);
        }
    }
}
```

In this program, we've added an ArrayList. An ArrayList is similar to a list in Python. The following:

```
ArrayList<Integer> values = new ArrayList<>();
```

is similar to saying the following in Python:

```
values = []
```

Adding values to an ArrayList is similar to Python as well. This statement:

```
values.add(temp);
```

is similar to saying this in Python:

```
values.append(temp)
```

An added feature at the end is called a *for-each* loop and it is very similar to what you have seen in Python. The for-each loop in Python looks something like this:

```
values = []
values.append(4)
values.append(5)
for x in values:
    print(x)
```

The for-each loop in Java is essentially the same thing. As with the previous steps, feel free to compile and run the above code. You are also encouraged to make tweaks to the code to ensure you understand how it works.

k. There are many great classes already built into Java that you can use. We will look at two of them in this example. The first one is a HashSet. A HashSet is a great structure for storing data if you don't care about the order. A HashSet should be used if you want to know if some value exists or doesn't exist. If you need your items to be in some kind of order (numerical, alphabetical, etc), then you should use a TreeSet. A Set and a List are different from each other and the difference is subtle. In a Set, a specific item can only exist once. In other words, you cannot add the

number 4 to a set more than once. If you do, nothing will happen.
Consider the following code to see the difference between a List and a Set (note I changed the class to Third):

```
import java.util.*;
public class Third{
    public static void main(String args[]){
        ArrayList<Integer> myList = new ArrayList<>();
        myList.add(4);
        myList.add(4);
        myList.add(4);
        HashSet<Integer> mySet = new HashSet<>();
        mySet.add(4);
        mySet.add(4);
        mySet.add(4);
        System.out.println("In list:");
        for(int value : myList){
            System.out.println(value);
        }
        System.out.println("In set:");
        for(int value : mySet){
            System.out.println(value);
        }
    }
}
```

When running this program, there are three 4’s in the list. There is only one 4 in the set.
When considering the difference between a Tree and a Hash, a Tree will keep things in order, a Hash will not. Consider the following:

```
import java.util.*;
public class Third{
    public static void main(String args[]){
        TreeSet<Integer> myTree = new TreeSet<>();
        for(int i=0; i<100; i+=10){
            myTree.add(i);
        }
        HashSet<Integer> mySet = new HashSet<>();
        for(int i=0; i<100; i+=10){
            mySet.add(i);
        }
        System.out.println("In tree:");
        for(int value : myTree){
            System.out.println(value);
        }
        System.out.println("In hash:");
        for(int value : mySet){
            System.out.println(value);
        }
    }
}
```

When running the above code, you'll notice the TreeSet prints out everything in order. The HashSet prints things out in a seemingly random way. A HashSet runs faster than a TreeSet so if speed is vital, choose a HashSet. However, if ordering is important, then your best choice is going to be a TreeSet.

l. A very useful collection in Java is called a Map. A Map is essentially a dictionary that you should know from Python. In a Map, you have unique keys that point to some value. Keys must be unique but values need not be unique. Consider the following example that maps an area code to state:

```
import java.util.*;
public class Third{
    public static void main(String args[]){
        HashMap<Integer, String> codes = new HashMap<>();
        codes.put(715, "WI");
        codes.put(319, "IA");
        codes.put(920, "WI");
        Set<Integer> keys = codes.keySet();
        for(Integer key : keys){
            if(codes.containsKey(key)){
                System.out.println(key+" : "+codes.get(key));
            }
        }
    }
}
```

In this example, a HashMap takes a (key,value) pair and adds (puts) it into the map. This statement gets all of the keys out of the map and stores them into a set (in no particular order because it's a *Hash*Map):

```
Set<Integer> keys = codes.keySet();
```

Once we have all of the keys, the next for-each loop simply iterates through all of the keys. The if statement is not necessary as we already know the key exists in the HashMap. It is only there for demonstration. The *get* method used here:
`codes.get(key)`
returns the value of the (key,value) pair that was inserted earlier. If the key were 920, codes.get(key) would return “WI” as the answer.

m. The above Map example shows what happens when you add distinct (key,value) pairs to the map. However, just like a dictionary in Python, if you add a new (key,value) pair where the key is identical to one already in the map, the old (key,value) pair is replaced with the new (key,value) pair. Consider this example that will print out 3 (key,value) pairs. The value for 319 will be TX.

```
import java.util.*;
public class Third{
    public static void main(String args[]){
        HashMap<Integer, String> codes = new HashMap<>();
        codes.put(715, "WI");
        codes.put(319, "IA");
        codes.put(920, "WI");
        codes.put(319, "TX");
        Set<Integer> keys = codes.keySet();
        for(Integer key : keys){
            if(codes.containsKey(key)){
                System.out.println(key+" : "+codes.get(key));
            }
        }
    }
}
```

n. The previous example showed us how to map an area code to a state. What if you wanted to return all of the area codes for a given state? In the previous example, one would have to iterate over the entire map to get all of the area codes for a particular state. Instead of mapping an area code to a state, we can map a state to a list of area codes. The following code shows us how to do that:

```
import java.util.*;
public class Third{
    public static void main(String args[]){
        TreeMap<String, ArrayList<Integer>> codes = new
TreeMap<>();
        ArrayList<Integer> forWI = new ArrayList<>();
        forWI.add(715);
        forWI.add(920);
        codes.put("WI", forWI);
        ArrayList<Integer> forIA = new ArrayList<>();
        forIA.add(319);
        forIA.add(651);
        codes.put("IA", forIA);
        for(Map.Entry<String, ArrayList<Integer>> values :
codes.entrySet()){
            for(Integer code : values.getValue()){
               System.out.println(values.getKey()+" "+code);
            }
        }
    }
}
```

In the above example, the TreeMap maps a String to a list of Integers instead of just 1 integer. The for loops at the bottom show another way to iterate through the collections.

o. Often, it is convenient to create functions to split up your code and reduce the amount of code you need to write. Consider the following example that reads in 3 numbers and prints out the factorial of each number:

```
import java.util.*;
public class Second{
    public static void main(String args[]){
        Scanner input = new Scanner(System.in);
        System.out.print("Enter number: ");
        int value = input.nextInt();
        //ensure the value read in is non-negative
        while(value < 0){
            System.out.print("Enter in a positive number: ");
            value = input.nextInt();
        }
        int answer = 1;
        for(int i=1; i<=value; i++){
            answer = answer * i;
        }
        System.out.println("Factorial of "+value+" is "+answer);

        //second number
        System.out.print("Enter number: ");
        value = input.nextInt();
        //ensure the value read in is non-negative
        while(value < 0){
            System.out.print("Enter in a positive number: ");
            value = input.nextInt();
        }
        answer = 1;
        for(int i=1; i<=value; i++){
            answer = answer * i;
        }
        System.out.println("Factorial of "+value+" is "+answer);

        //third number
        System.out.print("Enter number: ");
        value = input.nextInt();
        //ensure the value read in is non-negative
        while(value < 0){
            System.out.print("Enter in a positive number: ");
            value = input.nextInt();
        }
        answer = 1;
        for(int i=1; i<=value; i++){
            answer = answer * i;
        }
        System.out.println("Factorial of "+value+" is "+answer);
    }
}
```

One thing you should quickly notice is that there is a lot of duplicated code. The code in green, blue and red are essentially the exact same code. When this happens, you should consider using a function. The same code is shown below using a function instead:

```
import java.util.*;
public class Second{
    public static void factorial(){
        Scanner input = new Scanner(System.in);
        System.out.print("Enter number: ");
        int value = input.nextInt();
        //ensure the value read in is non-negative
        while(value < 0){
            System.out.print("Enter in a positive number: ");
            value = input.nextInt();
        }
        int answer = 1;
        for(int i=1; i<=value; i++){
            answer = answer * i;
        }
        System.out.println("Factorial of "+value+" is "+answer);
    }

    public static void main(String args[]){
        factorial();
        factorial();
        factorial();
    }
}
```

The code above should be fairly self-explanatory and is no different from Python other than syntax. The factorial() function code in the main function is used to call the function. The function executes and execution is returned back to the main function. As with Python, it is also possible to pass information to the function and also return information from the function. Consider the following example:

```
import java.util.*;
public class Second{

    public static int factorial(int value){
        int answer = 1;
        for(int i=1; i<=value; i++){
            answer = answer * i;
        }
        return answer;
    }

    public static void main(String args[]){
        int returnedValue = factorial(6);
        System.out.println(returnedValue);
    }
}
```

In this example, the value 6 is passed to the function for it to use. When the factorial function starts, value is equal to 6. The factorial function runs as normal and when it is finished, it returns its answer. We know it returns an int because the function was defined that way:
`public static int factorial…`
The answer that was returned is then stored into the returnedValue variable in the main function.
All of the concepts of creating functions, calling them, passing values to them and returning values are the same in Python as they are in Java. The syntax is a little different but the concepts are the same.

p. In this example, we are reading in from a file and outputting some information about the contents of that file. Create a file called *temp.txt* and store a bunch of numbers in that file. The numbers can be separated by spaces, tabs or newlines. One can use the Scanner class to read in from a file similar to what was done to read in from the command prompt. The code should be fairly self-explanatory. Non-obvious statements have a comment near them:

```
import java.util.*;
//needed import to use File class
import java.io.*;
public class Second{
    //throws Exception is added because of our usage of File.
    //basically, if something goes wrong, the program quits.
    public static void main(String args[]) throws Exception{
        //instead of Scanner.in, there is: new File("temp.txt")
        Scanner input = new Scanner(new File("temp.txt"));
        ArrayList<Integer> values = new ArrayList<>();
        //while there are more values to be read in
        while(input.hasNext()){
            int temp = input.nextInt();
            values.add(temp);
        }
        int total = 0;
        for(int value : values){
            total += value;
        }
        double average = total / (double)values.size();
        System.out.println(average);
    }
}
```

A quick aside. The `(double)values.size()` is there to ensure we are doing the correct type of division. In Java, 5 / 3 is 1. This is called integer division. The Python equivalent is this:

```
print(5/3)  #this prints out 1.66666666667
print(5//3)  #this prints out 1
```

Integer division is when you divide the two numbers but ignore the remainder. In Java, when you divide two ints, you get integer division. Therefore, to get around this, I casted values.size() to be a double. Essentially, I told Java to consider the second argument a double. When the division was being done, it was now dividing an int by a double which gives you “normal” double division. This gives me an accurate average.

An Exception is being handled in the previous class by simply stating that the main method may have an Exception. If this happens (e.g. the file isn’t found or is inaccessible), the method simply quits. However, when we are dealing with multithreaded applications, we cannot have the *run* method throw an Exception like you see in the above code. In order to handle code that may throw an Exception, we need to use a try/catch block as shown in the following example:

```
import java.util.*;
import java.io.*;
public class MyThread extends Thread{
    public void run(){
        try{
            Scanner input = new Scanner(new File("temp.txt"));
            //code that does stuff with the input
        }catch(Exception e){
            System.out.println(e);
        }
    }
}
```

q. In this example, we are seeing how to use the Scanner to read in and parse a csv file.

```
import java.util.*;
import java.io.*;
public class Third{
    public static void main(String args[]) throws Exception{
        Scanner input = new Scanner(new File("temp.csv"));
        while(input.hasNext()){
            String curLine = input.nextLine();
            String[] values = curLine.split(",");
            //values is an array where the first column of the
            //curLine is stored in values[0], the second column
            //of the curLine is stored in values[1] and so on.
        }
    }
}
```

r. Writing output to a file is a very common thing to do. If you are comfortable with the System.out syntax, then a PrintWriter is a simple thing to use for output. In the following example, a file is read into an ArrayList of Strings. The code then filters out all words that are less than 5 characters. The code then prints out all unique words to a file in alphabetical order.

```
import java.util.*;
import java.io.*;
public class FilterWords{
    public static void main(String args[]) throws Exception{
        Scanner input = new Scanner(new File("temp.txt"));
        ArrayList<String> words = new ArrayList<>();
        while(input.hasNext()){
            words.add(input.next());
        }
        TreeSet<String> outputMe = new TreeSet<>();
        for(int index = 0; index < words.size(); index++){
            if(words.get(index).length()>=5){
               outputMe.add(words.get(index));
            }
        }
        //An iterator goes through all elements of a Set, it is
a
        //common operation with lists/sets of elements.
        Iterator<String> iter = outputMe.iterator();
        //Open up a file for writing output.
        PrintWriter output = new PrintWriter(new
          FileWriter("output.txt"));
        while(iter.hasNext()){
            output.print(iter.next()+" ");
        }
        //Close the file to ensure all values are printed.
        output.close();
    }
}
```

s. Comparing Strings is a common thing to do in Java and it's important to know how to compare them. Everything in Java is an Object (except for the primitive types of int, double, boolean, etc). Since a String is an Object, when you are comparing two Objects, you are comparing whether or not they are the exact same Object. Therefore, the following code will output false:

```
public class StringTest{
    public static void main(String args[]) throws Exception{
          String first = new String("hello");
          String second = new String("hello");
          if(first==second){
               System.out.println("true");
          } else{
               System.out.println("false");
          }
    }
}
```

This is because first and second are two separate Objects and therefore are not equal to each other. In a more concrete example, think about going to the store and buying two copies of an Algorithms book. You now have two separate books. Let's say you put one book in your left hand and one in your right hand and you ask yourself, are these two books equal to each other? If you use the == notation in Java, you are asking, are these two books the exact same book. The answer to that is no. Even though the contents of the book are identical, they are not the exact same book. The question that you probably want to answer is this: are the contents of the book identical? In order to do that, you use the following notation:

```
public class StringTest{
    public static void main(String args[]) throws Exception{
          String first = new String("hello");
          String second = new String("hello");
          if(first.equals(second)){
               System.out.println("true");
          } else{
               System.out.println("false");
          }
    }
}
```

The print out above is true because the contents of the String are the same.

t. It is common to read in values from a file where the values you are reading in are actually integers or floats. However, when you read in from a file, you are reading them in as Strings. You cannot do arithmetic on integers if they are stored as Strings. The following example shows you how to convert a String into an integer. It also shows you how to convert part of your String into an integer using the substring method.

```
public class StringToIntTest{
    public static void main(String args[]) throws Exception{
          String first = new String("22");
          String second = new String("45");
          //doesn't work: int total = first+second;
          //rather you must convert first and second to ints
          int firstInt = Integer.parseInt(first);
          int secondInt = Integer.parseInt(second);
          int total = firstInt + secondInt;

          //another common issue to converting part of a
          //string to an int, for example, a date
          String date = new String("05/11/2004");
          //in order to pull out the 11, we use substring
          //similar to the str[3:5] syntax of Python
          String day = date.substring(3, 5);
    }
}
```

u. Another common stumbling block is the concept of **declaring** Objects and **creating** Objects. An Object is declared whenever you specify a type:

```
ArrayList<Integer> myList;
```

That Object has not been created yet. You need to create the Object before you use it:

```
myList = new ArrayList<>();
```

An issue arises when you declare an Object globally but that repeat that ***declaration*** in a method. If you have code that looks like this:

```
public class ObjTest{
     private ArrayList<Integer> myList;
     public void someMethod(){
          ArrayList<Integer> myList = new ArrayList<>();
          myList.add(10);
          myList.add(20);
     }
     public ArrayList<Integer> getList(){
          //myList is null and has nothing stored in it
          return myList;
     }
}
```

The above someMethod method redeclared the Object inside of the method. In the above code, you have 2 separate myList Objects. When you redeclare it in someMethod, it is only accessible inside of someMethod. When someMethod finishes, you are left with a null global ArrayList called myList. If you want to use the global myList Object in someMethod, your code would be updated to look like this:

```
public class ObjTest{
     private ArrayList<Integer> myList;
     public void someMethod(){
          myList = new ArrayList<>();
          myList.add(10);
          myList.add(20);
     }
     public ArrayList<Integer> getList(){
          //myList is not null here assuming
          //someMethod was called first
          return myList;
     }
}
```

v. The issue of scope is something that is quite important in Java. You can only use a variable in the correct scope. For example, consider the following example:

```
public class ScopeTest{
     public void someMethod(){
          int y = 10;
          if(y > 5){
               int x = 20;
               x = y + x;
               System.out.println(x);
          }
          System.out.println(x); //error!
     }
}
```

In the previous example, you do not have access to x in the second print statement because it was created in a different scope. The easiest way to figure out the scope of a variable is this. Look where you declare the variable (in the example below, the declaration is in red) and go up the code until you find the first left curly brace (highlighted in blue below). Find the matching right curly brace (highlighted in yellow below). From the point you declare your code until you reach the matching right curly brace is the scope of that variable (the highlighted area below).

```
public class ScopeTest{
     public void someMethod(){
          int y = 10;
          if(y > 5){
               int j = 6;
               int x = 20;
               x = y + x;
               System.out.println(x+j);
          }
          System.out.println(x); //error!
     }
}
```

w. A few miscellaneous things that you need to know for future projects. In order to time your code, you want to use the following command:

```
long now = System.currentTimeMillis();
```

As you have written the main function, you may have noticed the String args[]. These are arguments that you can pass into your program at runtime. Add the following to your main function:

```
System.out.println(args[0] + " " + args[1]);
```

...and then run that code using the following command:

```
java FileName FirstArgument SecondArgument
```

Your code will print out, as expected: FirstArgument SecondArgument. If you are given a folder name as a String, the way to get all of the files in that folder is to use the following statement:

```
File[] theFiles = new File(folderName).listFiles();
```

2. You will be creating a program that prints off all of the prime numbers that are strictly between two numbers.
    a. Create a file called **First.java**. Inside of that file, create a Java class called First.

    b. Inside the First class, create an isPrime function that accepts a number as an argument and returns true if the number is prime, false otherwise. Your function definition must look like this:

```
public static boolean isPrime(int value)
```

    c. Inside the First class, create a function called **printPrime** that accepts two numbers as arguments and prints out all of the prime numbers strictly between those two numbers (i.e. not including those two numbers). Each prime number in your output is separated by a space. Your function definition must look like:

```
public static void printPrime(int first, int second)
```

You should not assume first is less than or greater than second inside of this function (i.e. it is possible that first > second). Your code should still work if the first variable is greater than the second variable. If there are no prime numbers between first and second, your program should print: No primes.

    d. Inside the First class, create a main function and prompt the user to enter two integers. You can assume that the value being entered is an integer but you should not assume that it is greater than 0. If the number entered is less than 0, prompt the user again until the number entered is greater than or equal to 0. Once you have a number that is greater than or equal to 0, prompt the user for a second number. Ensure the second number is greater than or equal to zero. Once you have two numbers, call the printPrime function with those values.

3. Make sure to follow the directions exactly as described above. If you do not, my tester code will not work. For example, in steps 2a and 2b, if you do not call your file First.java or create a function called isprime instead of isPrime, my tests will fail.

What to Submit

When you are finished, zip up your First.java file from step 2 and upload your `a6.zip` file to the Activity 6 dropbox.

Optional Problems

The following problems are optional as the solutions are provided for you. However, if you are struggling with Java or struggling with threads, these problems will help. You are encouraged to solve the problems first before looking at the hints. If you need to look at the hints, then you are encouraged to write the solutions before looking at my code.

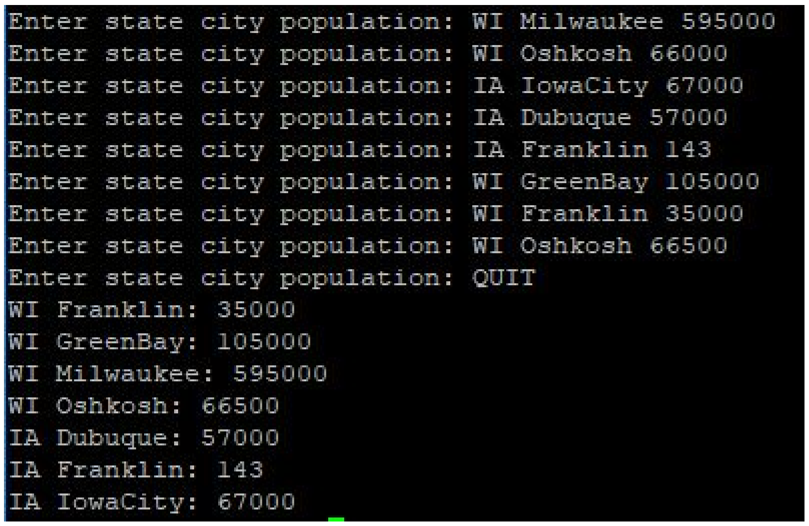
1. Write code that asks the user to enter in 3 pieces of information:
    a. A state name
    b. A city name
    c. The city's population

   Each piece will be separated by a space and there will be no spaces in the state, city or population. If the user enters in a state/city combination again, it will be assumed that the subsequent time is the correct population (i.e. maybe the population was updated). When the user is finished, the user will enter in QUIT for the state name and the program will print out all of the data. The following information should print out (see sample input/output below):
    a. The state name followed by the city and it's population for each city. It does not matter what order the states print out as long as the cities are

grouped by state (i.e. it is possible to print out all of the Wisconsin cities before printing out the Iowa cities).

b. The cities should be a alphabetical order for each state no matter how the data was input.

Your goal is to make this as efficient as possible. If there are millions of states and billions of cities, the choice of data structure will be very important. A sample running of the program is below:



```
Enter state city population: WI Milwaukee 595000
Enter state city population: WI Oshkosh 66000
Enter state city population: IA IowaCity 67000
Enter state city population: IA Dubuque 57000
Enter state city population: IA Franklin 143
Enter state city population: WI GreenBay 105000
Enter state city population: WI Franklin 35000
Enter state city population: WI Oshkosh 66500
Enter state city population: QUIT
WI Franklin: 35000
WI GreenBay: 105000
WI Milwaukee: 595000
WI Oshkosh: 66500
IA Dubuque: 57000
IA Franklin: 143
IA IowaCity: 67000
```

The next 2 problems assume you have gone through all of the parallel programming presentations in the course. If you haven't seen them all yet, then go watch them and then come back to solve these.

2. The Monty Hall door problem is a fun statistical problem that we can simulate. For those unfamiliar, the game is this. A player chooses between door 1, door 2 and door 3. Behind one of the doors is a car. Behind two of the doors is a goat. The player is allowed to choose any door initially. Let's see a sample running of the game.
Let's say the player chooses door 1. Because there are 2 goats, either door 2 or door 3 must have a goat behind it. The host, knowing what the right answer is, opens up either door 2 or door 3 depending on which one had the goat behind it. Let's say there was a goat behind door 2. The host opens up door 2. Now there are 2 doors remaining, door 1 and door 3. The player initially chose door 1. At this point in the game, the player is given the option to switch doors. The player could choose to switch to door 3 or the player could stick with door 1.

   The question is this, does the player benefit from switching doors? Or does the player have a better chance of winning by not switching doors? Or does it not matter at all? These are the questions we are going to answer with a simulation. Create code that simulates this problem[1]. We will run this scenario a few billion times and that should give us an idea of which method is best[2].

You should create a program that accepts 1 parameter (call it X) at runtime, the number of simulations you should run for each decision (i.e. run it X times for choosing to switch and run it X times for choosing not to switch). You should time your program to see how long it takes to simulate that many runnings. Once you are finished with that version, you should create a multithreaded program that splits up the work appropriately[3].

3. Assume you work for a marketing company. You have a huge database of people and a long list of interests for each person. Maybe person A likes golf, tennis, cooking and traveling. Maybe person B likes tennis, traveling, biking, avocados and cats. And so on.

   Your goal is this: you want to assemble a subset of people such that for each interest, at least 1 person in your subset has that interest.

   After talking with your computer science friends, you realize that this problem is actually really difficult. A natural solution that comes to mind quickly is this: pick the person with the most interests we haven't yet covered and repeat. Such a solution is actually quite bad, in general[4]. Realizing your initial thought is

[1] Since this problem deals with random choices, to get a random number between 0 and 1 (not including 1), use the following statement: Math.random(). If you want a number between 0 and 5, use this code: (int)(Math.random()*6). The last statement saying: get a random number between 0 and 1, multiply it by 6… so now the number is between 0 and 6 (not including 6). Convert the number to an int which means you drop the decimal points and now you have a number that is either 0, 1, 2, 3, 4 or 5.
[2] Or we could solve this using math but eh, this is a programming course so let's write code!
[3] Because this problem is so simple to model, it is possible that your parallelized version might run slower than your non-threaded version. If the amount of work that you have to do in each step is so small, the benefit of parallelization is minimal and sometimes, it's worse. The compiler may improve your serialized code so dramatically that there is no benefit to multithreading. To ensure that my parallelized version ran faster, I added a small "slowdown" to the code such that each scenario test takes an extra microsecond to execute. When this added slowdown is used, the problem takes 28 seconds to run 100000 simulations in parallel and 227 seconds to run in serial. This makes sense, 28*8 == 224.
[4] If you are not convinced, consider this grouping of people. A is interested in 0, 2, 4, 6, 8, 10, and 12. B is interested in 1, 3, 5, 7, 9, 11, 13. Together, A and B are interested in everything. However, C is interested in 6, 7, 8, 9, 10, 11, 12, 13. D is interested in 2, 3, 4, 5. E is interested in 0, 1. The greedy solution of picking the person with the most uncovered interests would pick C because C has 8 interests we haven't covered yet. A and B only have 7 we haven't covered. So we pick C for the solution. This takes care of 6-13. Of the interests that remain, A and B are interested in 3 of them separately but D is interested in 4 things that remain. So we pick D. Finally we pick E and our solution is {C, D, E} whereas the optimal

incorrect, you decide to do a brute force solution and simply check all of the possibilities. Does any 1 person have all of the interests? If no, do any 2 people have all of the interests? If no, do any 3 people have all of the interests… and so on until you find an answer.

The problem is this: write a program to read in an arbitrary list of people/interests and output a minimal subset of people such that all interests are covered by the people in your subset.

Optional Problems Hints

1. For this problem, there are a few clues as to what to do. The first is that all cities are associated with a state. Therefore, having a list of cities that are mapped to a state seems like a good idea. You want to have some sort of Map where the key is a String (the state) and the value is a list of things. One false start would be to do something like this:

   ```
   Map<String, String>
   ```

   The problem is that you need to store more than 1 city for each state so having 1 String as the value doesn't make sense. Rather, you want to have a list of cities. Something like this is closer:

   ```
   Map<String, List<String>>
   ```

   The issue with this code is that you also need to store a population along with the city. Therefore, storing just the city name doesn't do you any good. Rather, you want to store a population to go along with the city name. In other words, you want your city to also map to something, i.e. the population:

   ```
   Map<String, Map<String, Integer>>
   ```

   Now the question becomes what kind of map do we use. Since we don't care if the states are sorted, using a Hash is the fastest thing. Since we do care if the cities are sorted, using a Tree is the smartest thing because it keeps the cities sorted. This is the best data structure to use for our problem:

solution is {A, B}. This example can be extended so that you pick an arbitrary number of people whereas the optimal solution is picking just 2 people.

HashMap<String, TreeMap<String, Integer>>

The rest of the problem simply lies in how to insert data into such a structure. See step n for clues on how to do this. If you are still not sure, a solution to this problem can be found at http://www.uwosh.edu/faculty_staff/krohne/ds730/javaActivity/ProblemOne.java.

2. The problem itself should be easy to solve. Since there are many ways to actually write code, I won't elaborate on a solution here. The more interesting part comes with splitting up the work. If one needs to run 10,000,000 scenarios and you have 8 cores, then having each thread simulate 1,250,000 scenarios seems like a reasonable solution. Since there are 2 separate ways to run a scenario, the question is, do you create 8 cores to solve the scenario of not switching doors and follow it up with 8 cores of solving the scenario of switching doors. Or do you use 4 cores for the switching scenario and 4 cores for the non-switching scenario. In this particular problem, it won't matter. However, these are the kinds of things to think about when thinking about how to split up your problem. There is no magic formula for how to split up your computation. You have to analyze your problem and determine the best way to split it up. A solution for this problem can be found at:

   http://www.uwosh.edu/faculty_staff/krohne/ds730/javaActivity/ProblemTwoRunner.java
   http://www.uwosh.edu/faculty_staff/krohne/ds730/javaActivity/MyStayThread.java
   http://www.uwosh.edu/faculty_staff/krohne/ds730/javaActivity/MySwitchingThread.java

   My solution split up the computation into 2 separate thread types where thread A solved it one way and thread B solved it another way. I created 8 cores to thread A, solved it, then created 8 cores of thread B and solved it. However, there would have been nothing wrong with creating 4 cores of thread A and 4 cores of thread B.

3. I will start with an OK solution here and show you why such a solution probably isn't the best. One natural way to split up the computation would be to check all subsets of size 1 in a thread, check all subsets of size 2 in another thread, check all subsets of size 3 in another thread and so on. The problem with this is that the threads that are checking smaller subsets will finish very quickly leaving your threads that are checking bigger subsets to work a lot harder. Such a split of

computation is better than nothing but the gain will not be as large as it could be[5].

A potentially better solution would be to think about what a potential solution might look like. Let’s say you have 8 cores and you have 32 people. A better solution would be to split up those 32 people into 8 groups of 4 each. Then you will send those 4 people to each thread. In your thread, you will then assume that each person you receive is a part of the solution and check subsets from there. For example, let’s say thread A receives people numbered 0, 1, 2 and 3. Thread A will check all subsets of size 1 with the people in its group. It will check if 0 is a solution, then 1, then 2 and finally 3. Since it is unlikely any of those are solutions, the thread will now check if any of those people belong in a solution of size 2. Thread A will check (0, 1), then (0, 2), then (0, 3), then (0, 4), then (0, 5) ... , then (0, 30), then (0, 31), then (1, 0), then (1, 2), then (1, 3), …, then (3, 29), then (3, 30), then (3,31). If none of those are a solution, then we move on to subsets of size 3 where the solution contains at least one of (0, 1, 2, 3).

You might be looking at that solution and saying to yourself, wait a minute, that’s less than brilliant. There is no difference between (0, 1) and (1, 0) so why check both of them! If you noticed this, then well done. It is not ideal to do that. Therefore, a better solution would be to only check subsets where the values are increasing. In other words, (0, 1) would be checked but (1, 0) would never be considered because the 0 comes after the 1 in the list. This ensures that subsets will never be checked more than once.

A final very subtle issue with this solution is the fact that the threads with the smaller numbered people will do more work. Think about solutions of size 2. In the thread that contains 0, it must check (0, 1) and (0, 2) and (0, 3) all the way up to (0, 31). But think about the thread that contains person 29 that is checking subsets of size 2. It only has to check (29, 30) and (29, 31). Therefore, you want to be sure that your people are split up appropriately. Sending (0, 10, 21, 31) to a thread seems reasonable. Sending (1, 11, 20, 30) to another thread seems reasonable as well. Splitting up your people like this is a reasonable way to split up the computation.

[5] As an aside, if this problem were assigned and this is the solution you came up with and submitted, it would be more than acceptable for this course. The goal of this course is to show you how to multithread. This is not an upper level CS course where you have to create great algorithms to solve problems. As long as you are creating a reasonable split of your computation, that is good enough for this course. The rest of the hint is more of a thought experiment on how you might be able to split up your computation to get better results.

General Threading Tips

As was said at the end of the second hint above, there is no algorithm or formula I can give you to tell you how to split up your computation. Many problems are unique and have their own solution space to search through. The best advice on how to split up computation is this:

1. If you have a ton of input files and can split up the input in a certain way, that is probably going to be a good solution (e.g. see MapReduce).
2. If you have a lot of computation and there a natural way to split up the computation. For example, if there are K simulations to run, then sending K/C simulations (where C is the number of CPUs available) to each thread is a good idea.
3. The question of splitting up the computation can also lie in the solution. What does the solution look like? What are all of the potential solutions? Is there any way to split up the potential solution space such that you are checking S/C possible solutions in 1 thread, S/C possible solutions in another thread and so on (where S is the number of potential solutions and C is the number of CPUs).