

**Due:**

Activity (in-lab): Monday, September 26, 2016 by the end of lab

**Goals:**

By the end of this activity you should be able to do the following:

- Gain a further understanding of *if* and *if-else* statements
- Understand the basics of *while* statements (a.k.a., *while loops*)
- Introduces the ArrayList class (java.util.ArrayList)

**Description:**

In this activity you will create a `NumberOperations` class which will hold an integer value and provide methods to perform various operations on that value. You will also download and modify a class called **NumberOpsList** which creates an ArrayList object and then reads in a value from the keyboard. While the value is not zero, the program creates an instance of the `NumberOperations` class for the value, adds the instance to the ArrayList, and then reads the next value from the keyboard.

**Directions:****Part 1: NumberOperations: Method Stubs (a.k.a. skeleton code) (40%)**

- Create a class called `NumberOperations` with an instance variable called *number* of type *int*.

```
public class NumberOperations {  
    private int number;  
}
```

- Add method stubs for the following methods. **The first two are given; do the rest on your own.**

- The constructor takes an int parameter called *numberIn*

```
public NumberOperations(int numberIn) {  
}
```

- `getValue`: takes no parameters; returns an int value

```
public int getValue() {  
    return 0; // placeholder return  
}
```

Create method stubs with placeholder returns for each method on your own.

- `oddsUnder`: takes no parameters; returns a String
- `powersTwoUnder`: takes no parameters; returns a String
- `isGreater`: takes an int parameter called *compareNumber*; returns an int
- `toString`: takes no parameters; returns a String

Compile `NumberOperations` and run the following in interactions. **Do not continue until your program compiles and the following code runs without runtime errors in interactions. Note that return values will be the placeholder values in the “stubbed” methods.**

```
▶ NumberOperations numOps = new NumberOperations(5);  
▶ String s1 = numOps.oddsUnder();  
▶ String s2 = numOps.powersTwoUnder();  
▶ int n1 = numOps.isGreater(2);  
▶ String s3 = numOps.toString();
```

**Part 2: NumberOperations: Constructor, getValue, and toString (20%)**

- In your constructor, add code that will set the value of *number* to *numberIn*.
- In your *getValue* method, delete the placeholder return and return the value of *number*.
- Replace the placeholder return in the *toString* method with the following code:

```
return number + "";
```

[Note that the result of concatenating *number*, which is an *int*, with an empty *String* is a *String*.]

Compile *NumberOperations* and run the following code in the interactions pane. **Do not continue until the following code runs without error in interactions.**

```
▶ NumberOperations numOps = new NumberOperations(5);
▶ numOps.getValue()
  5
▶ numOps // displays the toString return value
  5
```

**Part 3: NumberOperations: oddsUnder Method (10%) – Returns a String containing the positive odd integers less than the value of number.**

- Create a local variable in *oddsUnder* called *output* and initialize it to an empty string literal.

```
public String oddsUnder() {
    String output = "";
}
```

- Add a local *int* variable *i* and a while loop that will iterate through the loop as long as the value of *i* is less than the value of *number*.

```
int i = 0;
while (i < number) {

}
```

- Inside of the above loop, add code that will concatenate the value of *i* if it is an odd number. Also increment the value of *i* during each iteration of the loop.

```
if(i % 2 != 0) {
    output += i + "\t";
}
i++;
```

- After the loop, add code to **return the value of output**.

Compile *NumberOperations* and run the following code in the interactions pane. **Do not continue until the following code runs without error in interactions.**

```
▶ NumberOperations numOps = new NumberOperations(9);
▶ numOps.oddsUnder()
  1  3  5  7
```

**Part 4: NumberOperations: powersTwoUnder Method (15%) - returns a String containing the positive powers of two less than the value of number.**

- Create a local String variable in powersTwoUnder called *output* and initialize it to an empty string literal as you did in oddsUnder.
- Create another local variable of type int called *powers* and initialize its value to 1.
- Add a while loop that will iterate through each number up until the value of *number*.

```
while (powers < number) {
```

- Inside of the while loop, add code that will concatenate the value of powers to output if it is a power of 2 and then calculate the next power of two (the comments below are optional).

```
    output += powers + "\t"; // concatenate to output
    powers = powers * 2; // get next power of 2
```

- After the loop, add code to **return the value of output**.

Compile NumberOperations and run the following code in the interactions pane. **Do not continue until the following code runs without error in interactions.**

```
▶ NumberOperations numOps = new NumberOperations(20);
▶ numOps.powersTwoUnder()
1    2    4    8    16
```

**Part 4: NumberOperations: isGreater Method (15%)**

- Delete the placeholder return from isGreater and add code that will return 1 if number is greater than compareNumber, -1 if number is less than compareNumber, or 0 if the numbers are equal.

```
if (number ____ compareNumber) {
    return 1;
}
else if (number ____ compareNumber) {
    return -1;
}
else {
    return 0;
}
```

Compile NumberOperations and run the following code in the interactions pane. **Do not continue until the following code runs without error in interactions.**

```
▶ NumberOperations numOps = new NumberOperations(10);
▶ numOps.isGreater(2)
1
▶ numOps.isGreater(15)
-1
▶ numOps.isGreater(10)
0
```

**Part 5: NumberOpsDriver Class**

- Download the driver program called *NumberOpsDriver.java* and then add the indicated code (described in the `//` comments) so that it does the following: prompts the users for a list of numbers (ints) separated by a space followed by the number 0; reads the first number in the list; enters a loop and while the number is not 0, creates a *NumberOperations* object, adds the *NumberOperations* object to an *ArrayList* called *numOpsList*, then read the next number in the list (i.e., iterates through the loop). Once the value of 0 is read from the list, the loop terminates. Now using a second while loop, print out each *NumberOperations* object in the *ArrayList* along with its “odds under” and its “powers of 2 under”. Example output:



```

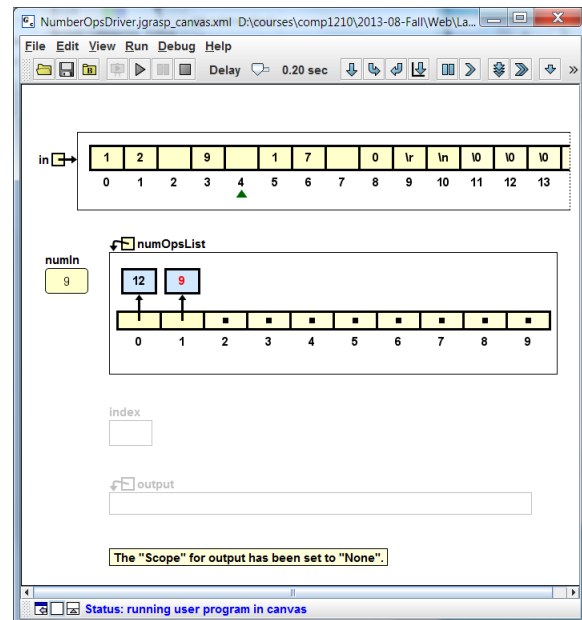
----jGRASP exec: java -ea NumberOpsDriver








Enter a list of positive integers separated with a space followed by 0:
12 9 17 0
For: 12
  Odds under:   1       3       5       7       9       11
  Powers of 2 under:  1       2       4       8
For: 9
  Odds under:   1       3       5       7
  Powers of 2 under:  1       2       4       8
For: 17
  Odds under:   1       3       5       7       9       11       13       15
  Powers of 2 under:  1       2       4       8       16
----jGRASP: operation complete.

```

**Part 5: Running NumberOpsDriver in the Canvas**

- Download the jGRASP canvas file *NumberOpsDriver.jgrasp\_canvas.xml* and save in the folder with your source files for this activity.
- Now click the Run in Canvas button  on the desktop toolbar. After the canvas file opens, click the Play button  and you should see the viewers for Scanner *in* and *ArrayList numOpsList* become active although empty. After entering the input as shown in the example above, you should see the values in the Scanner *in* viewer. As the each value is read in by your program you should see the Scanner viewer move through the values in its buffer. And as your program creates each *NumberOperations* object for the value and adds the *NumberOperations* object to *ArrayList numOpsList*, you should see *numOpsList* viewer updated in the canvas. The figure at right shows the **canvas after two values have been read in and two *NumberOperations* objects have been created and added to the *ArrayList numOpsList*.**



- As the program plays you should see the viewer for *output* which is a local variable in the *oddsUnder()* and *powersTwoUnder()* methods. Note that the “Scope” for *output* has been set to “None” by using the viewer pull-down menu ▼ and selecting **Scope** and then **None**. This allows the viewer to display the value of *scope* regardless of which local variable it is. The figure at right shows the canvas while the program is executing the *powersUnder()* method on the third element of *numOpsList* (i.e., the element at index 2). The viewer for *output* shows the five powers of two numbers below 17.
- While running on the canvas, you can pause  and then step  or step-in  as appropriate to view the behavior of interest. You can also stop  the canvas and then run in canvas  and play  again. To regulate the speed of the program in the canvas, decrease or increase the delay between steps using the Delay slider **Delay**  **0.30 sec**.
- You should experiment with running this program in the canvas until you are sure that you understand both the *NumberOperations* and *NumberOpsDriver* classes. Since this activity introduces the *while* statement, you should pause the canvas and then single step through each of the *while* statements to ensure that you understand them. In the main method, the first *while* statement reads in the values and the second one populates the *ArrayList* of *NumberOperations* objects. You should also single step through each of the *while* loops in the *oddsUnder()* and *powersTwoUnder()* methods.

