

Homework 2
CS 361

Jen McCall, Gianna Sorrentino

Homework 2

Programming Languages Principles and Implementation

Instructions:

- Due date: 10/9 (No late homework will be accepted. The solution of the homework will be posted on 10/9 after class. The midterm is on 10/11.)
- This homework assignment is to be done alone or in a group of 2 students.
- Problems must be done in order.
- You need to fill out this document with your answers. Homeworks with answers only will not be accepted.
- All Java code must be written and tested in the Eclipse IDE (<http://www.eclipse.org>) (or similar).
- Code must be provided in annex and printed directly from Eclipse.
- Code that does not compile will be graded as 0.
- All your code must be available on GitHub under the CS361 and Homework2 directories.
 - Your homework must be well presented and have a cover page. 10 points will be reduced from your grade if you do not do have a cover page.
 - The presentation of the hard copy of your homework assignment must contain your name(s).
 - In case of problems with this homework, contact me by email cscharff@pace.edu.
 - Grade: 100 points

Question 1: History of programming languages

Put the following programming languages on a chronological timeline. The year must be provided. **In addition**, indicate the name of the designer of the programming language, where it was created (company, national lab, higher education institution etc.), and the country.

- Fortran – 1957, John Backus & IBM, USA
- Lisp -1958, John McCarthy & MIT, USA
- Cobol – 1959, Jean E Sammet & CODASYL & DoD, USA
- PASCAL - 1970, Niklaus Wirth & UCSD, USA
- Prolog - 1972, Alain Colmerauer & Aix-Marseille University, France
- C - 1972, Dennis Ritchie & Bell Labs, USA
- ADA - 1980, Jean Ichbiah & Tucker Taft & HOWLG & DoD, USA & UK
- C++ - 1983, Bjarne Stroustrup & Bell Labs, USA
- ISETL - 1986, Gary Levin & Berkeley & Clarkson, USA
- EIFFEL - 1986, Bertrand Meyer & Eiffel Software, France
- Perl - 1987, Larry Wall & Unisys, USA
- SML - 1990, Robin Milner & Mads Tofte & Robert Harper & University of Edinburgh, UK
- Python - 1991, Guido van Rossum & Python Software Foundation & CWI, Netherlands
- Java - 1995, James Gosling & Sun Microsystems, USA
- Ruby - 1995, Yukihiro Matsumoto & Open Standards Promotion Center of the IT Promotion Agency, Japan
- Kotlin - 2011, Dmitry Jemerov (lead) & JetBrains, Russia

Question 2:

Consider the following code. Each *draw* method has a number.

```
public class Circle{
    public double center_x, center_y;
    public double radius;

    public void draw() {
        // (1) method to draw circle on the screen
    }

    public void draw(Color color) {
        // (2) method to draw circle on the screen with a
        // given color
    }
}

public class ColoredCircle extends Circle{
    public int color;

    public void draw() {
        // (3) method to draw the colored circle
    }
}
```

a) Explain polymorphism on the code above.

The code is polymorphic because the draw method is defined in more than one way, but the method that will be called depends on which object the code is called with. The code knows what to do but it makes the decision on which draw method to call based on which object it is called with.

b) c is of type Circle and d is of type ColoredCircle. Can we write `d = c;`? Why?

Yes because it passes the “IS-A” test. ColoredCircle extends Circle and “IS-A” type of circle. Therefore it is okay to say `d = c;`.

c) c is of type Circle and d is of type ColoredCircle. Can we write `c = d;`? Why? What happens if we execute the code below? What method called *draw* is called? Why?

```
c = d;
c.draw();
```

Although I initially believed we could not write `c = d`, after some research online I discovered we indeed can. I learned a bit more about polymorphism from tutorialspoint.com. We can write `c = d` because since ColoredCircle is a form of Circle, Circle can inherit

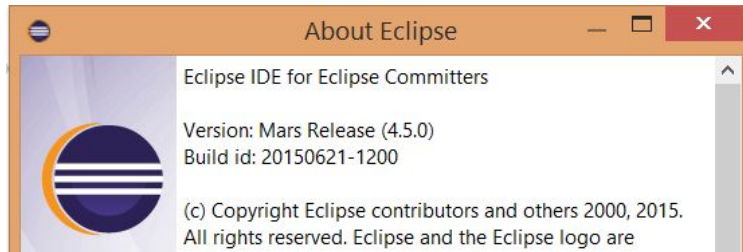
all the properties of Circle. Therefore, draw method 1 would be called.

Question 3:

Install the following Eclipse Bytecode Outline plugin from: <http://asm.objectweb.org/eclipse/index.html> or from the Eclipse MarketPlace.

[Dr. Scharff tested with the Neon version of Eclipse and with Eclipse Marketplace Byte Outline 2.4.3 plugin and it works!]

a) What Eclipse version are you using?



b) What Java version are you using?

```
C:\Users\jenspacely>java -version
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) Client VM (build 25.131-b11, mixed mode, sharing)
```

c) What is the Bytecode generated by the following statements?

`int i = 5;`

`i = i+5;`

Explain the syntax of the Bytecode. Provide a screenshot to support your work.

[Explanations here are supporting using Java Documentation found here](#)
[And using this presentation](#)

```
1 public class bytecode {
2     public static void main(String[] args){
3         int i = 5;
4         i = i+5;
5     }
6 }
7
8
```

```
bytecode
// class version 52.0 (52)
// access flags 0x21
public class bytecode {

    // compiled from: bytecode.java

    // access flags 0x1
    public <init>(): void
    L0
    LINENUMBER 2 L0
    ALOAD 0: this
    INVOKESPECIAL Object.<init>(): void
    RETURN
    L1
    LOCALVARIABLE this bytecode L0 L1 0
    MAXSTACK = 1
    MAXLOCALS = 1

    // access flags 0x9
    public static main(String[]): void
    L0
    LINENUMBER 4 L0
    ICONST_5
    ISTORE 1
    L1
    LINENUMBER 5 L1
    IINC 1: i 5
    L2
    LINENUMBER 7 L2
    RETURN
    L3
    LOCALVARIABLE args String[] L0 L3 0
    LOCALVARIABLE i int L1 L3 1
    MAXSTACK = 1
    MAXLOCALS = 2
}
```

As we see in the screenshot above, the bytecode is displayed on the right hand side of the screen. The syntax of the bytecode is extremely different looking than the Java code. The Java code defines an integer “i” and sets it equal to five. After that in the next line it redefines i as the value of i + 5. In the bytecode version, we will look under the public static... method to see the relevant code and furthermore the bytecode for lines 4 and 5 since that’s what we’re interested in.

First, we see L0 our first action. Everything indented below this shows what on the first line in our method. In our Java code, this is int i = 5; but in bytecode we see three lines. In this case, each of these mean:

1. LINENUMBER 4 L0 - our first action in this method, as seen by L0 (the first action), but it’s on line 4 of our total code
2. ICONST_5 - we’re loading the value 5 onto the stack
3. ISTORE 1 - we’re storing the value of the int (5) into the index

Second, we see L1, our second line of this method:

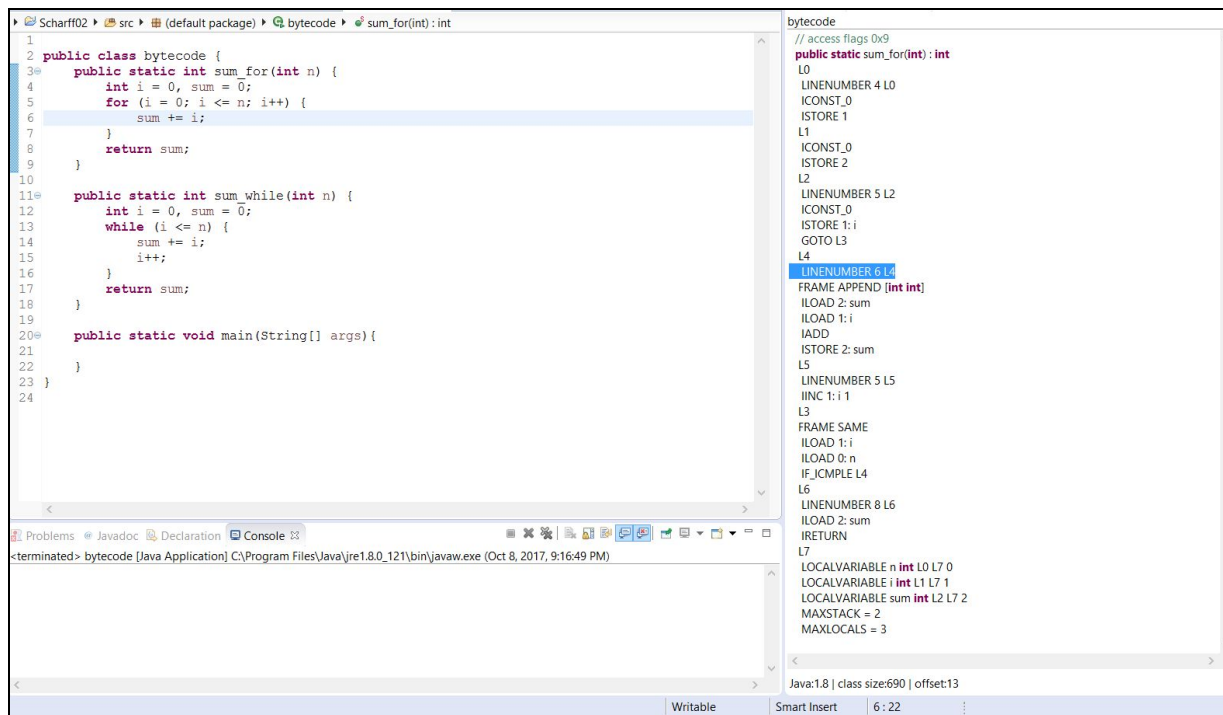
1. LINENUMBER 5 L1 - our second action is on line 5 of this code
2. IINC 1: i 5 - We’re going to increment the value of the index 1 (which we defined before) by the defined 5

After this we see L2, LINENUMBER 6, RETURN. This means that the method is closing out and everything past that we don’t care about (at least for this particular example).

d) Compare the Bytecode generated by the 2 functions below and write down your conclusions.

Provide screenshots to support your work.

```
public static int sum_for(int n) {  
    int i = 0, sum = 0;  
    for (i = 0; i <= n; i++) {  
        sum += i;  
    }  
    return sum;  
}
```



```
public static int sum_while(int n) {  
    int i = 0, sum = 0;  
    while (i <= n) {  
        sum += i;  
        i++;  
    }  
    return sum;  
}
```

```

1 public class bytecode {
2     public static int sum_for(int n) {
3         int i = 0, sum = 0;
4         for (i = 0; i <= n; i++) {
5             sum += i;
6         }
7         return sum;
8     }
9 }
10
11 public static int sum_while(int n) {
12     int i = 0, sum = 0;
13     while (i <= n) {
14         sum += i;
15         i++;
16     }
17     return sum;
18 }
19
20 public static void main(String[] args) {
21 }
22 }
23
24

```

```

bytecode
// access flags 0x9
public static sum_while(int): int
L0
LINENUMBER 12 L0
ICONST_0
ISTORE 1
L1
ICONST_0
ISTORE 2
L2
LINENUMBER 13 L2
GOTO L3
L4
LINENUMBER 14 L4
FRAME APPEND (int int)
ILOAD 2: sum
ILOAD 1: i
IADD
ISTORE 2: sum
L5
LINENUMBER 15 L5
INCR 1: i
L3
LINENUMBER 13 L3
FRAME SAME
ILOAD 1: i
ILOAD 0: n
IF_JCMPL L4
L6
LINENUMBER 17 L6
ILOAD 2: sum
IRETURN
L7
LOCALVARIABLE n int L0 L7 0
LOCALVARIABLE i int L1 L7 1
LOCALVARIABLE sum int L2 L7 2
MAXSTACK = 2
MAXLOCALS = 3

```

We can see that these bytecodes are different. Each method is executed in 8 actions (L0-L7), but the code is different in each one. We see something a little strange in each code, or at least something that took us by surprise. Instead of a numerical progression (0, 1, 2, ..., 7), we see the bytecode jump around in its action numbering. In both bytecodes, we see L3 come after L4 in one case and L5 in another. We conclude that the inside of the loop, be it the FOR loop or the WHILE loop. This makes sense because our FOR loop handles the “i++” interaction inside its initialization whereas the WHILE loop does this step inside the actual loop. From this as well as number of identical commands, we can see that the two loops, while executed differently, do the exact same thing. This is really no surprise since we see this same concept in our Java code in many different applications. There are many ways to do one particular problem.

e) Write the factorial function (with the profile: `public static fact(int n)`) and describe the bytecode generated by this function.

The screenshot shows an IDE with the following content:

```
1 public class factorial {
2     public static int fact(int n) {
3         if (n == 0)
4             return 0;
5         else if (n == 1)
6             return 1;
7         else
8             return n*fact(n-1);
9     }
10 }
11
12 public static void main(String[] args) {
13     int i = 3;
14     System.out.println(fact(i));
15 }
16 }
17 }
18 }
```

The bytecode for the `fact` method is shown on the right:

```
factorial
// access flags 0x9
public static fact(int) : int
L0
LINENUMBER 4 L0
ILOAD 0: n
IFNE L1
L2
LINENUMBER 5 L2
ICONST_0
IRETURN
L1
LINENUMBER 6 L1
FRAME SAME
ILOAD 0: n
ICONST_1
IF_ICMPNE L3
L4
LINENUMBER 7 L4
ICONST_1
IRETURN
L3
LINENUMBER 9 L3
FRAME SAME
ILOAD 0: n
ILOAD 0: n
ICONST_1
ISUB
INVOKESTATIC factorial.fact (int) : int
IMUL
IRETURN
L5
LOCALVARIABLE n int L0 L5 0
MAXSTACK = 3
MAXLOCALS = 1
```

The console at the bottom shows the output of the program:

```
<terminated> factorial (1) [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (Oct 9, 2017, 12:52:05 AM)
6
```

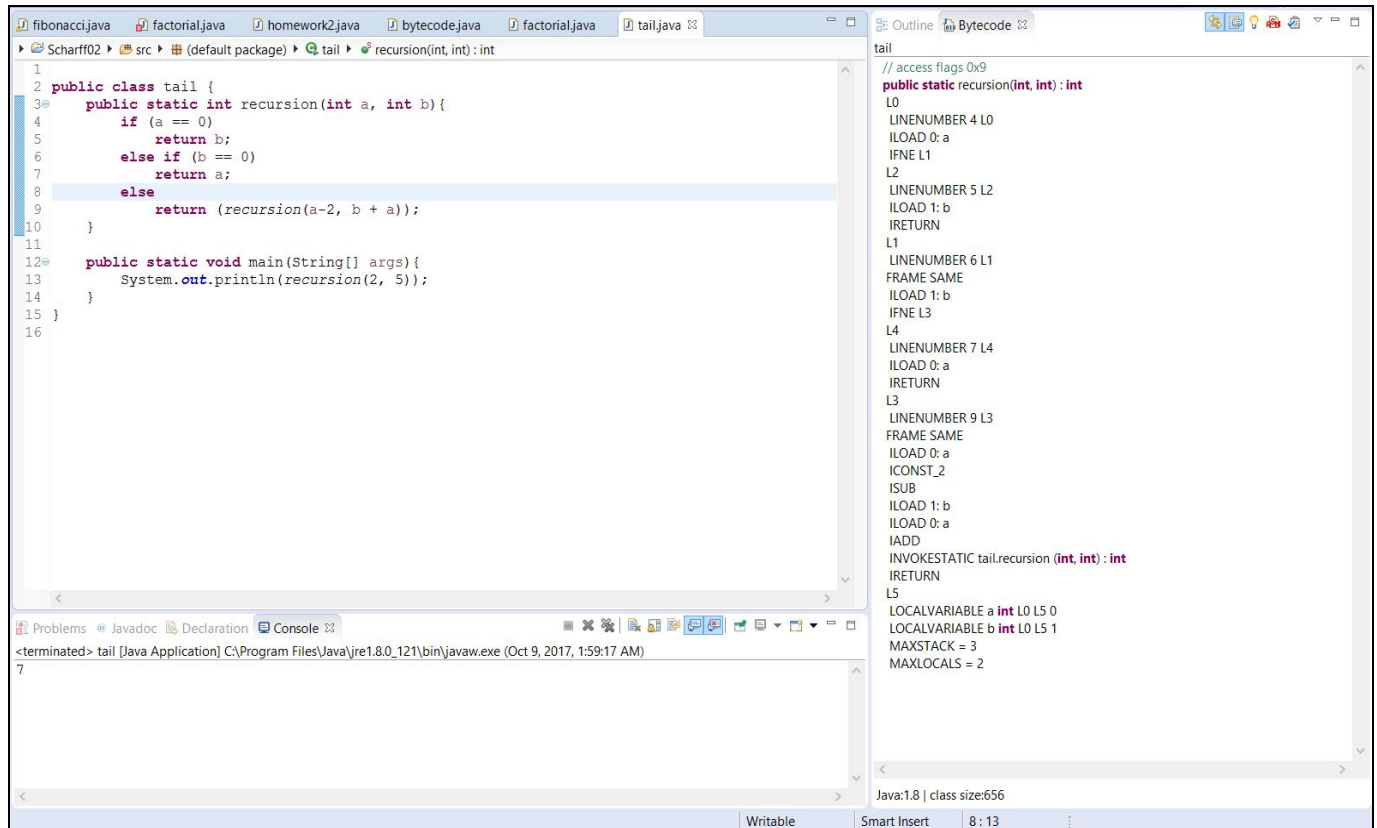
Here we see the actions we are doing represented by our LX numbers. We see the same alternating as we did in the previous question. L0, action 1, is done on line 4, loading the value of our int given and starting our IF statement. Now bytecode follows by line, so the next line (the bulk of our first IF catch, is not necessarily the next action. In this function, L2 is on the next line but the third action potentially done. This sets 0 as our value and returns the value. Next is L1 (action 2) where we are checking if `n == 1` or our value set is 1. Like before, this might not be true so the action on the next line is not the next numerical action. This brings us to the next line, action 5 (L4), which would set and return our value as 1. Next line is action 4 (L3) which aims to store our value by first doing some variable math. In the previous steps, we were just setting our value to a specific number, but in this one we need to handle some variables. This is why there's more body to the bit of bytecode, as it has to store and calculate $n*(n-1)$ as well as returning that. Lastly, in L5 we tell our system how many of these stacks we can fit which is the number of possible answers (3 statement options) and also define that only 1 variable can be held in each one at a time.

This is a tad confusing, so here's a bit of an easier way to follow it



f) Choose a tail recursive function and describe the bytecode generated by this function. Compare with the code generated for a recursive function obtained in c).

For this example, we decided to go with a tail recursive sum function as pictured below. This code and our understanding of tail recursion partially [comes from this source](#), as well as our recursive GCD classwork:



The screenshot shows an IDE with two main panes. The left pane displays the Java source code for a class named `tail`. The right pane displays the corresponding bytecode for the `recursion` method.

```
1 public class tail {
2     public static int recursion(int a, int b){
3         if (a == 0)
4             return b;
5         else if (b == 0)
6             return a;
7         else
8             return (recursion(a-2, b + a));
9     }
10 }
11
12 public static void main(String[] args){
13     System.out.println(recursion(2, 5));
14 }
15 }
16
```

The bytecode pane shows the following instructions for the `recursion` method:

```
tail
// access flags 0x9
public static recursion(int, int) : int
L0
LINENUMBER 4 L0
ILOAD 0: a
IFNE L1
L2
LINENUMBER 5 L2
ILOAD 1: b
IRETURN
L1
LINENUMBER 6 L1
FRAME SAME
ILOAD 1: b
IFNE L3
L4
LINENUMBER 7 L4
ILOAD 0: a
IRETURN
L3
LINENUMBER 9 L3
FRAME SAME
ILOAD 0: a
ICONST_2
ISUB
ILOAD 1: b
ILOAD 0: a
IADD
INVOKESTATIC tail.recursion (int, int) : int
IRETURN
L5
LOCALVARIABLE a int L0 L5 0
LOCALVARIABLE b int L0 L5 1
MAXSTACK = 3
MAXLOCALS = 2
```

So as we can see here, our bytecode looks pretty much the same as the last examples we did. We're calling a stack, putting different variables `i` to compare them and loading them. In L3, however, we see a difference because now we have to do some math to get our comparison. This includes invoking a static method to store our local variables, loading them up, subtracting, adding, comparing, and then returning. While this sounds more complicated for this example, it does the exact same thing as head recursion. There are certain circumstances, however, such as a recursive factorial function, that tail recursion doesn't necessarily make sense and head recursion becomes the better option.

References

- The Java Virtual Machine Specification
<https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf> (Java 8 SE)
- Java Bytecode Basics <http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.html> (1996)
- <http://www.beyondjava.net/blog/java-programmers-guide-java-byte-code/> (2015)

Question 4:

a. Write a PROLOG program that describes the British family until nowadays. Kate, William and their children should be cited in the facts. Your program will start with the facts available in the slides (slide 31) and ends with Kate, William and their children. (M = male, P = parent)

```

M(Phillip).
M(Edward VII).
M(George V).
M(George VI).
M(Charles).
M(William).
M(Harry).
M(George VIII).

P(Edward VII, George V).
P(Victoria, Edward VII).
P(Alexandra, George V).
P(George VI, Elizabeth II).
P(George V, George VI).
P(Elizabeth II, Charles).
P(Phillip, Charles).
P(Charles, William).
P(Charles, Harry).
P(Diana, William).
P(Diana, Harry).
P(William, George VIII).
P(Kate, George VIII).
P(William, Charlotte).
P(Kate, Charlotte).
G(x,y): -P(x,z), P(z,y)

```

b. Write a **rule** that describes the father predicate. *Father*(X, Y) means that X is the father of Y.
 $P(X,Y) \wedge M(X) \rightarrow \text{Father}(X,Y)$

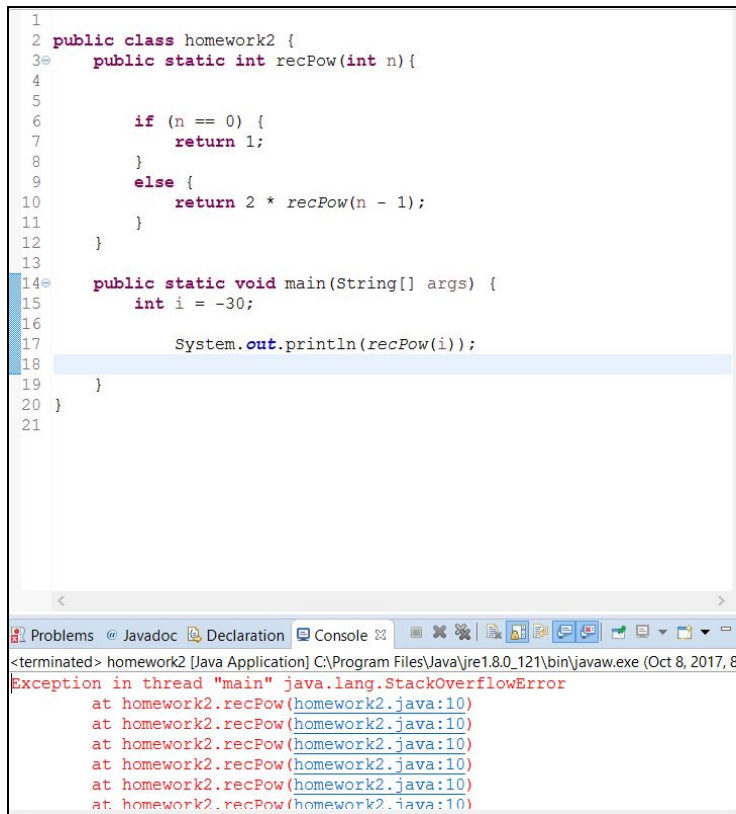
Question 5:

Write a **recursive** function *recPow* that computes 2^n for $n \geq 0$ in Java. The function will have the following profile:

```
public static int recPow(int n)
```

The function must consider all cases and be tested exhaustively. Show your testing!

For this problem, we need to write an integer function that calls upon itself to find the 2 to the nth power. There are a couple of restrictions right off the bat that we need to be concerned with. First of all, the problem defines that we'll be looking for an answer that is a non-negative integer. An integer in Java is defined at its maximum as 2,147,483,647 or, conveniently for our current case, 1 less than 2^{31} . This means that anything above this cannot be accepted in our function. So we know now that our function can handle integers anywhere from 0 to 30 without running into errors. That being said, we need to make sure our program doesn't run into anything above or below these conditions and give us an error. As you can see, without accounting for these conditions, we run into some errors below:



```
1
2 public class homework2 {
3     public static int recPow(int n){
4
5
6         if (n == 0) {
7             return 1;
8         }
9         else {
10            return 2 * recPow(n - 1);
11        }
12    }
13
14    public static void main(String[] args) {
15        int i = -30;
16
17        System.out.println(recPow(i));
18    }
19 }
20
21
```

Problems @ Javadoc Declaration Console

<terminated> homework2 [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (Oct 8, 2017, 8

Exception in thread "main" java.lang.StackOverflowError

at homework2.recPow(homework2.java:10)

at homework2.recPow(homework2.java:10)

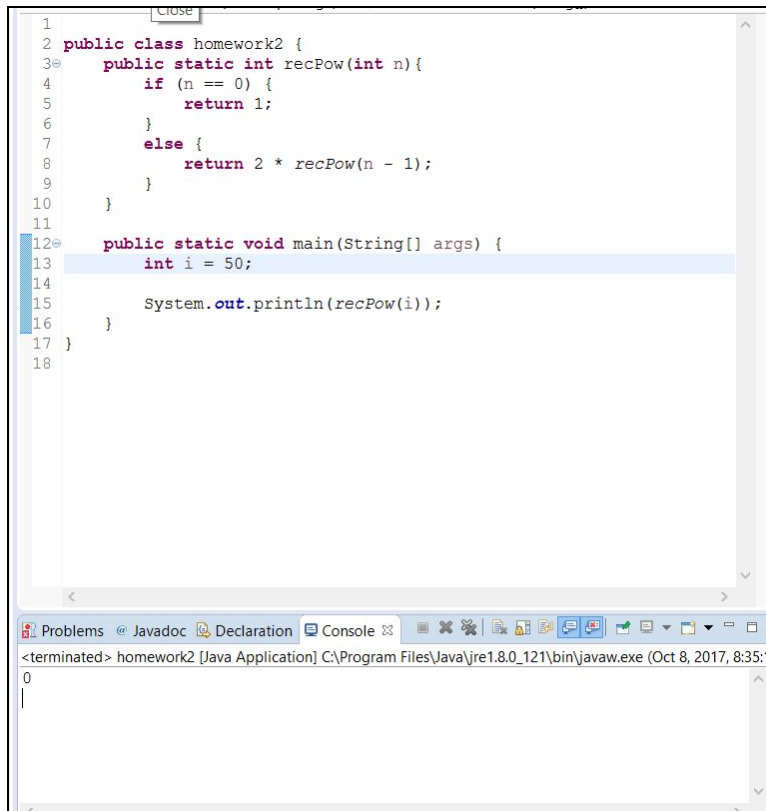
at homework2.recPow(homework2.java:10)

at homework2.recPow(homework2.java:10)

at homework2.recPow(homework2.java:10)

at homework2.recPow(homework2.java:10)

Ex 1. a negative number will throw the program into an overflow error



```
1 public class homework2 {
2     public static int recPow(int n){
3         if (n == 0) {
4             return 1;
5         }
6         else {
7             return 2 * recPow(n - 1);
8         }
9     }
10 }
11
12 public static void main(String[] args) {
13     int i = 50;
14
15     System.out.println(recPow(i));
16 }
17 }
18
```

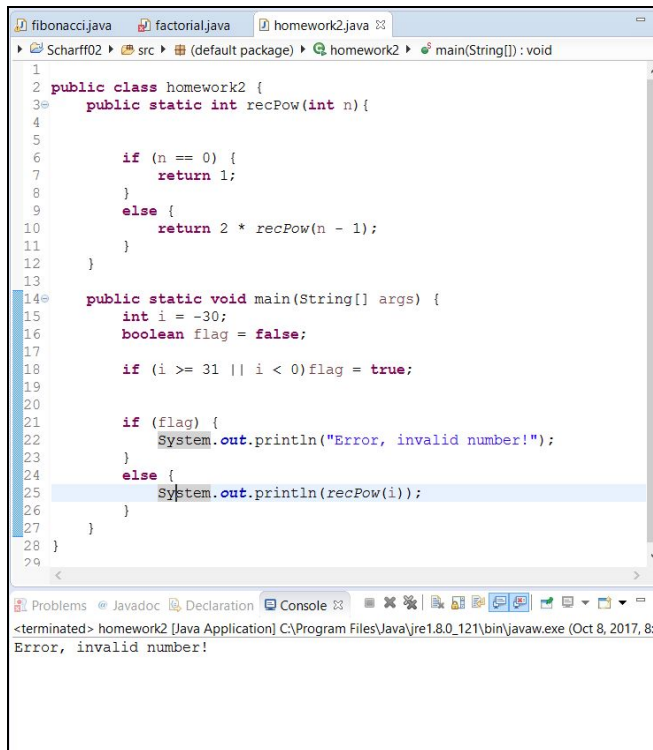
Problems @ Javadoc Declaration Console

<terminated> homework2 [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (Oct 8, 2017, 8:35:1

0

Ex 2. Any $n > 30$ will break the code and result in a wrong answer like 0

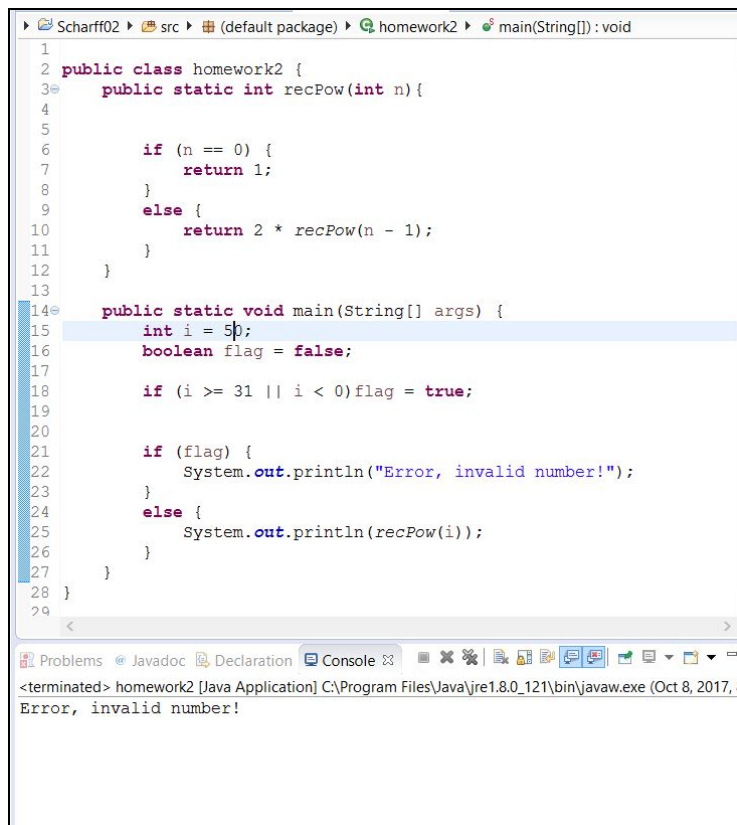
So to fix this and catch any integers above or below our defined accepted range, we need to make sure that our test integer i isn't going to break our code. To do this, in our main method (see code examples below), we initialize a boolean to check to see if the integer we're testing fits our criteria. So if our integer doesn't match our defined range, the boolean flag will be true as if to throw up a red flag and stop our code from running and inevitably breaking. Since we have 2 integer ranges that won't work in our function (that is the range below our designated accepted and the range above our designated accepted), we need two different triggers to set off our flag. To do this simply, we added an if statement checking both the "below" range OR (designated by the `||` in our code) the "above" range. As you can see in the screenshots below, both of the values that broke our function before now have an error message attached to them, keeping the function from running and breaking itself.



```
1 public class homework2 {
2     public static int recPow(int n) {
3
4         if (n == 0) {
5             return 1;
6         }
7         else {
8             return 2 * recPow(n - 1);
9         }
10    }
11
12    public static void main(String[] args) {
13        int i = -30;
14        boolean flag = false;
15
16        if (i >= 31 || i < 0) flag = true;
17
18        if (flag) {
19            System.out.println("Error, invalid number!");
20        }
21        else {
22            System.out.println(recPow(i));
23        }
24    }
25 }
26
27
28
29
```

<terminated> homework2 [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (Oct 8, 2017, 8:00:00 AM) Error, invalid number!

Ex 3. Fix 1 for Ex 1: i is less than 0, so our flag is true and the function does not run



```
1 public class homework2 {
2     public static int recPow(int n) {
3
4         if (n == 0) {
5             return 1;
6         }
7         else {
8             return 2 * recPow(n - 1);
9         }
10    }
11
12    public static void main(String[] args) {
13        int i = 50;
14        boolean flag = false;
15
16        if (i >= 31 || i < 0) flag = true;
17
18        if (flag) {
19            System.out.println("Error, invalid number!");
20        }
21        else {
22            System.out.println(recPow(i));
23        }
24    }
25 }
26
27
28
29
```

<terminated> homework2 [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (Oct 8, 2017, 8:00:00 AM) Error, invalid number!

Ex 4. Fix 2 for Ex 2: i is more than 31 , so our flag is true and our function does not run

Question 6:

Write a **recursive** function `merge` that merges 2 arrays in Java. . The function will have the following profile:

```
public static int[] mergeSort(int[] a, int[] b)
```

You will use the `split` function of slide 18 (odd and even positions).

The function must be tested exhaustively. Show your testing!

If you use code online, you will need to cite your sources.

Citations: <http://www.java2novice.com/java-sorting-algorithms/merge-sort/>

<https://www.buildingjavaprograms.com/code-files/4ed/ch13/MergeSort.java>

MergeSort is an algorithm that uses the divide and conquer method to sort numbers. It divides them in half into groups, sorts those groups, and then calls on itself recursively to do that again and again until it has sorted everything. After everything is sorted, it merges them back together to now form a complete sorted array.

```
*recursiveMerge.java
1 // citations:
2 // http://www.java2novice.com/java-sorting-algorithms/merge-sort/
3 // https://www.buildingjavaprograms.com/code-files/4ed/ch13/MergeSort.java
4
5 import java.util.*;
6
7 public class recursiveMerge {
8
9     // This program implements the merge sort algorithm for
10    // arrays of integers.
11
12    public static void main(String[] args) {
13        int[] a = {40, 94, 7, 11, 23, 0, 38};
14        System.out.println("Initial array: ");
15        for(int i = 0; i < a.length; i++)
16            System.out.print(a[i] + " ");
17
18        mergeSort(a);
19        System.out.println("\nArray after sorting: ");
20        for(int i = 0; i < a.length; i++)
21            System.out.print(a[i] + " ");
22    }
23
24    // Puts the integers in order using mergesort
25    public static void mergeSort(int[] array) {
26        if (array.length > 1) {
27            // split array into two halves
28            int[] left = left(array);
29            int[] right = right(array);
30
31            // recursively sort the two halves
```

The main method provides our test for the code. This picture contains the input for our first output, we also did a second output with negative numbers to show it worked for them as well. It then prints a before and after of the unsorted/sorted array.

The `mergeSort` creates the “left” and “right” arrays we will be using for when we split the array and then recursively calls on “left” and “right”. After that is done, it calls on `merge` to make them one array again.


```

32         mergeSort(left);
33         mergeSort(right);
34
35         // merge the sorted halves into a whole
36         merge(array, left, right);
37     }
38 }
39
40 // Returns the first half of the array
41 public static int[] left(int[] array) {
42     int size1 = array.length / 2;
43     int[] left = new int[size1];
44     for (int i = 0; i < size1; i++) {
45         left[i] = array[i];
46     }
47     return left;
48 }
49
50 // Returns the second half of the array
51 public static int[] right(int[] array) {
52     int size1 = array.length / 2;
53     int size2 = array.length - size1;
54     int[] right = new int[size2];
55     for (int i = 0; i < size2; i++) {
56         right[i] = array[i + size1];
57     }
58     return right;
59 }
60

```

The left and right methods help us access the two halves of the array we created.

```

60
61 // Merges the left and right arrays
62 public static void merge(int[] result,
63     int[] left, int[] right) {
64     int k = 0; // index into left array
65     int j = 0; // index into right array
66
67     for (int i = 0; i < result.length; i++) {
68         if (j >= right.length || (k < left.length && left[k] <= right[j])) {
69             result[i] = left[k]; // take from left
70             k++;
71         } else {
72             result[i] = right[j]; // take from right
73             j++;
74         }
75     }
76 }
77 }
78

```

Merge is the method we called on earlier. It has quite a detailed for loop that must satisfy some requirements. Our index for the right array must be greater than the length of the right array OR the index for the left array must be less than the length of the left array. In addition to satisfying that requirement for the left, the index of the left array must be less than that or equal to that of the index on the right. If it is, it takes a value from the left and adds it back to the array, sorted. If not, it takes a value from the right and adds it back to the array, sorted.

Initial array:

40 94 7 11 23 0 38

Array after sorting:

0 7 11 23 38 40 94

Initial array:

-5 94 -21 11 23 0 -38

Array after sorting:

-38 -21 -5 0 11 23 94

We did not figure this code out entirely by ourselves. We did need to cite a bit of information from the two links above, however we did learn quite a bit about the coding for mergeSort in the process.