CS 46B

Lab 6[[1]](#footnote-0)

10 points

​​ 

***Image by Theodore Geisel, aka Dr. Seuss***

# Learning Outcomes

During this lab, you will learn how to do the following.

* Design and implement recursive functions
* Understand the behavior of the debugger when tracing recursive methods
* Set conditional breakpoints
* Use logging statements
* Turn off logging in a working program

**Timing Note:**

Please adhere to the timing schedule provided to complete each part. If you find that you cannot finish a section before the scheduled time or you are running behind, please seek extra assistance from your lab instructor or learning assistant. They will provide hints to help you complete the section on time. Additionally, during the last 45 minutes of the lab, you will complete the exit interview questions provided as the check-out quiz on Canvas. Therefore, ensure that your group completes the activity on time.

[Part 1: Recursive Factorial (20 min)](#_heading=h.3dy6vkm)

[Part 2: A Robot Navigating a Maze (30 min)](#_heading=h.lnxbz9)

[Part 3: Recursive Fibonacci Numbers (20 min)](#_heading=h.3whwml4)

Bonus activities (3pt) (20 min)

—---------

Estimated time: 1.5 hrs

**Lab Grade**

Labs are a core component of the course, and a lot of your learning happens when you have to take what you learned in the lecture and apply it in practice. Labs are a required part of the course, and missing more than two labs will result in failing the course.

# Your lab grade will be based on three components:

# **Check-in Quiz 2 points:** This is an easy quiz about the material covered during the week and takes 15 minutes at most.

# **Collaboration 3 points + Project Compile and Execution 3 points:** Your lab instructor examines and grades your project as a team on your laptop to make sure it compiles and executes. Also, make sure that you collaborate with your teammate(s). Working in groups to solve problems is an important skill that computer scientists embrace. It is important not to leave group members behind or to let each just do the work independently. Also, all team members need to submit their Jar files on Canvas. Your files will be checked randomly. If your jar file is missing or does not compile and run as expected, 50% of your grade will be deducted. So please make sure that your Jar file is complete and sound before you upload it on Canvas.

# **Exit Interview 2 points:** To receive credit for this lab, your group will complete an exit interview. To get an idea of the kinds of questions that will be asked, look at the questions highlighted in blue that you encounter as you complete the lab instructions. ***To help you prepare for the exit interview, I suggest tackling the questions when you encounter them in the lab instruction, discussing them as a group, and then writing down what you think the answer is.***

# **Important note:**

For each lab, your group will complete an exit interview. Completing this interview will give you credit for the lab. If you are absent from a lab, you can make up the interview during your lab instructor's office hours (only for those 2 allowed missing labs). Note that you can miss at most two interviews. If you miss more than two, you will fail the course.

The exit interview will be approximately 10 minutes, and you will rotate through who is the group leader (in charge of answering the questions). These interviews are not so much about getting the right answer but serve as a way for you to demonstrate how you are thinking about the problems and how your understanding of the material is evolving. They also provide an opportunity for the lab instructor or learning assistant to help me, the instructor, understand where you are struggling. You will get credit for completing the interview provided that you have made a good-faith effort to complete the lab.

One goal of the labs and homework assignments is to support you in learning how to write code. When working in small groups, it is important that all of you understand the code that is being written. If you are relying on your group mates to do all the programming, the exams and grading interviews will be challenging. **Also, note that your lab exams will be completed and graded individually.** With that said, make sure you understand each lab activity well and feel confident that, if needed, you can complete that individually, or if you find yourself understanding the concept quickly, please slow down and try to get your group mates up to speed. The best way to really understand the material is to explain it to someone else. And “explain it” doesn’t mean just showing them your code or telling them exactly what to do; it means helping them figure out how to do it on their own.

**Setup:**

Create an eclipse workspace for lab 6. Create a new Java Project called 'lab6'. In lab6->src, create the “maze” package and drag the Maze.java, Robot.java, and MazeSolver.java starter files into the package. In lab6->src, create a second package called “recursion”. You will create classes in this package from scratch.

# Let’s Program

## **Part 1: Recursive Factorial (20 min)**

For any positive integer x, the factorial of x (symbolically: x!) is defined as the product of all integers up to and including x.

* Example: 6! = 1x2x3x4x5x6 = 720.
* **NOTE**: 0! Is defined to be 1.

Before doing any programming, come up with a plan for how you will implement the following class using recursion.

Create a class called FactorialGenerator in the recursion package with 3 methods:

1. public long nthFactorial(int n). This method should call computeFactorialRecurse(n).
2. private long computeFactorialRecurse(int n), which should recurse (that is, call itself) unless n is 0 (if n is 0, the method should return 1).
3. A main method that constructs a FactorialGenerator instance and calls nthFactorial(), passing in interesting values.

So far the class should be very simple. Try computing x! for x = 6, 10, and 20.

Now put a loop in main() that prints x!, where x goes from 1 to 32. For what values of x does x! look wrong? Describe how they look wrong. Jot down an example of one of the wrong numbers on some scratch paper.

**To find out why, explore the optional exercises on the next page.**

Now change main() so that it calls nthFactorial(-1). What happened?

Your computeFactorialRecurse() method needs a precondition check. Before it does anything else, it should make sure n is >= 0. If n is negative, you can either throw IllegalArgumentException or use assertions. Let’s use an assertion here. Write an assertion that asserts n>=0. If it isn’t, write a meaningful message. (HINT: remember assert boolean: object. The assertion error is thrown if the boolean value is false. In this case, we want to pass a String object with a message).

Run the app again. What happened? If the assertion didn’t have any effect, you probably don’t have assertions enabled. To enable assertions, select “Run Configurations…” in the Run menu. In the wizard that pops up, select the “Arguments” tab. In the lower text area (“VM Arguments”, not “Program Arguments”), type –ea and then click “Apply”. Run the app again. You should now see the result of triggering the assertion statement.

#### An Aside on Assertions

In large development projects, development usually happens with assertions enabled, so that problems can be detected early. When the product is delivered, customers run it with assertions disabled so that they don’t see embarrassing error messages.

**STOP**

*Make sure everyone in your group has completed Part 1 before moving on.*

### Factorial Gets Big FAST

20! is a large positive number, and it roughly looks like it’s about 20 times 19! So 20! is probably ok. But how can 21! be negative? Let’s write some code to figure this out. Add a line to main() that prints the maximum value for the long data type. This is available as a sta up in the API.tic constant in the java.lang.Long class; look it ***What is the name and value of the constant?*** Make sure everyone in your group is on the same page.

#### Arithmetic Overflow

When Java multiplies 20! by 21, the result is greater than the maximum long value, so it cannot be represented by a long. This situation, where doing math on numbers so that the result is bigger than your data type can accommodate, is called “arithmetic overflow”. **In Java, arithmetic overflow is silent. You might expect some kind of runtime exception to be thrown, but no.** *The long type is 64 bits; 21! requires 69 bits. Java just retains the low-order 64 bits and throws away the rest.* n our result, the leftmost bit of the result (also called the high-order bit or bit 63 or the most significant bit) is a 1. In computer arithmetic, any integral type with its high-order bit = 1 is interpreted as negative. That’s why your method says that 21! is negative.

#### Let’s try using Float instead

Maybe long isn’t the best return type for nthFactorial() and computeFactorialRecurse(). Try changing it to float, and again print out 1! through 32! ***Are there any negative values now?***

Obviously, float is better than long. But compare the two values of 18! (using longs and using floats). ***How are they different?***

The float type is better than long in this application because it can cope with large numbers, but it doesn’t always have great precision.

#### Let’s try using Double instead

For more precision, use double. Change the return type for nthFactorial() and computeFactorialRecurse() to double. Again, ***Are there any negative values now? Is 18! computed this way exactly equal to what you computed when the return type was long?***

## 

## **Part 2: A Robot Navigating a Maze (30 min)**

Your job is to help the robot navigate through the maze.

### Maze.java

The class Maze initializes mazes such as this 13 X 29 maze.  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* \*\*\* \*

\*\* \*\*\* \* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* \*\*\* \* \* \*

\*\* \*\*\* \* \*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*

\*\* \* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\* \*\*\*\*\*\*\* \*\*\*\*\*\*\* \*\*\*\*\*\*

\*\*\*\*\*\* \*\*\*\*\*\*\* \*\*\*\*\*\*

\* \*\*\*\*\*\*\* \*\*\*\*\*\*\* \*\*\*\*\*\*

\* \*\*\*\* \*\*\*\*\*\*\* \*\* \*

\* \* \*\*\*\*\*\*\* \*\*\*\*\*\*\* \*\*\*\*\*\*

\* \*\*\*\* \*\*\*\*\*\*\* \*\*\* \*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*

The \* are walls, and the blank spaces are corridors. Each corridor is one space wide (horizontally or vertically). Given a row and column, what method in the Maze classes lets you know if you are an exit to the maze?

### Robot.java

To help the robot that is trapped in the maze escape, you need to complete the recursive escape method in the Robot class so that the robot escapes the maze and prints the path it took. The MazeSolver class tests your implementation. Before you start programming read through the following steps to help you formulate your plan. The goal is to have your robot use the “right hand rule”: Always have a wall to the right to navigate the maze. You may assume the ro*bot has been placed with a wall to the right when it was constructed.*

1. Start with the base case:
   * If the robot is already at an exit, return. It’s escaped!
2. Now for the Recursive Case
   * If you can't move forward, turn left.
     1. There is no point turning right? Why?
   * If you can move forward, do it.
   * You may have lost the wall to the right. Make a sketch showing how this can happen.
3. * If it has lost the wall (the block on the right is not a wall block ( \* ) ), then have the robot turn right and move.
   * It must then again have a wall to the right. Why?
   * If it hasn't lost the wall (there is a wall to the right), turn back to the original position. (You had to turn to see the wall.)
4. Now the robot is at a new position. Then call escape so that it can escape from that position. That's your recursion.

Now that you understand the basics of how the robot should escape the maze, implement the escape method. **Before you begin implementation, make sure your entire group understands the plan.**

### Running the MazeSolver

What happens when you run the MazeSolver program? It would be a good idea to scroll through all the output and be sure you understand why the robot moves as it does.

Could you have written this program without recursion? Don't try to rewrite it, just think about it.

**STOP**

*Make sure everyone in your group has completed Part 2 before moving on.*

### 

### **Bonus (1pt)**

### l: What if the maze has cycles.

The right hand rule doesn't work if the maze has cycles—paths that lead back to the same spot. Sketch an example.

We'll use a more complex algorithm for this purpose. Make a second method called escape2 that does the following:

* Make the robot turn around in all four directions.
* In each direction with a path emanating from it (i.e canMove is true),
  + The robot *clones* itself.
  + Move the clone into the path (one step), and
  + Let it escape (call escape2()).

Complete the loop:  
public boolean escape2() {

if (atExit()) return true;

for (int i = ...){//this loop makes the robot check all possible directions

turnRight();

if (...) {

Robot cloned = clone();

cloned....();//what should the clone do?

if (cloned.escape2()) {

visited... //how does visited need to be updated?

return true;

}

}

}

return false;

}

1. What happens when you run the program? Remember to change the call in the MazeSolver to escape2
2. This code will work for any maze.
3. Can you easily rewrite escape2 without recursion? Why or why not?

## **Part 3: Recursive Fibonacci Numbers (20 min)**

fib(1) = fib(2) = 1.

For x > 2, fib(x) = fib(x-2) + fib(x-1).

Create a class called FibGenerator in the recursion package with 3 methods:

1. public int nthFib(int n). This method should call computeFibRecurse(n).
2. private int computeFibRecurse(int n), which should recurse (that is, call itself) unless n is 1 or 2. If n is 1 or 2, the method should return 1.
3. A main method that prints “STARTING”, then constructs a FibGenerator generator and then calls nthFib(), passing in interesting values.

Note that the return types are ints. Compared to factorials, the Fibonacci series grows relatively slowly, and ints will be sufficient for today’s exercise.

Put a loop in the main() method that computes and prints fib(x) for x = 1 through 20. Check to make sure that your method is correct by looking up the fibonacci on Google.

Try raising the maximum value of x from 20 to 30. If that works ok, change 30 to 40, and keep going until the app begins to run slowly. If it runs really slowly and you lose patience, you can kill the execution by clicking on the red square in the banner of your console window. (**Note**: *That red square only appears if you have a console, and you don’t always have a console. But if your app prints anything, you get a console. That’s why at the beginning of most apps I print “STARTING” … it ensures I always have a console in case I need to kill the app.*)

Approximately what value of x causes your app to slow down noticeably? This is hard to answer if your main() just prints out fib(x). It’s easy if your main prints output like this:

fib(3) = 6

fib(4) = 24

…

Why is the recursive Fibonacci algorithm slow? Work with your group to come up with a hypothesis and write it down.

### Conditional Breakpoints.

Let’s use the debugger to help us figure this out. First let’s modify our code just a bit. Use the following code for your recursive call. This will let us more clearly see the values of the recursive calls in the debugger.

else {

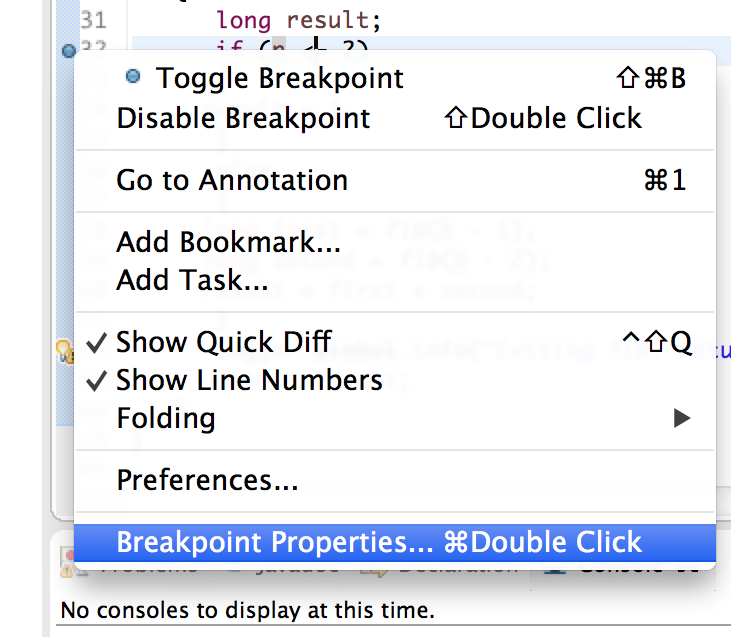
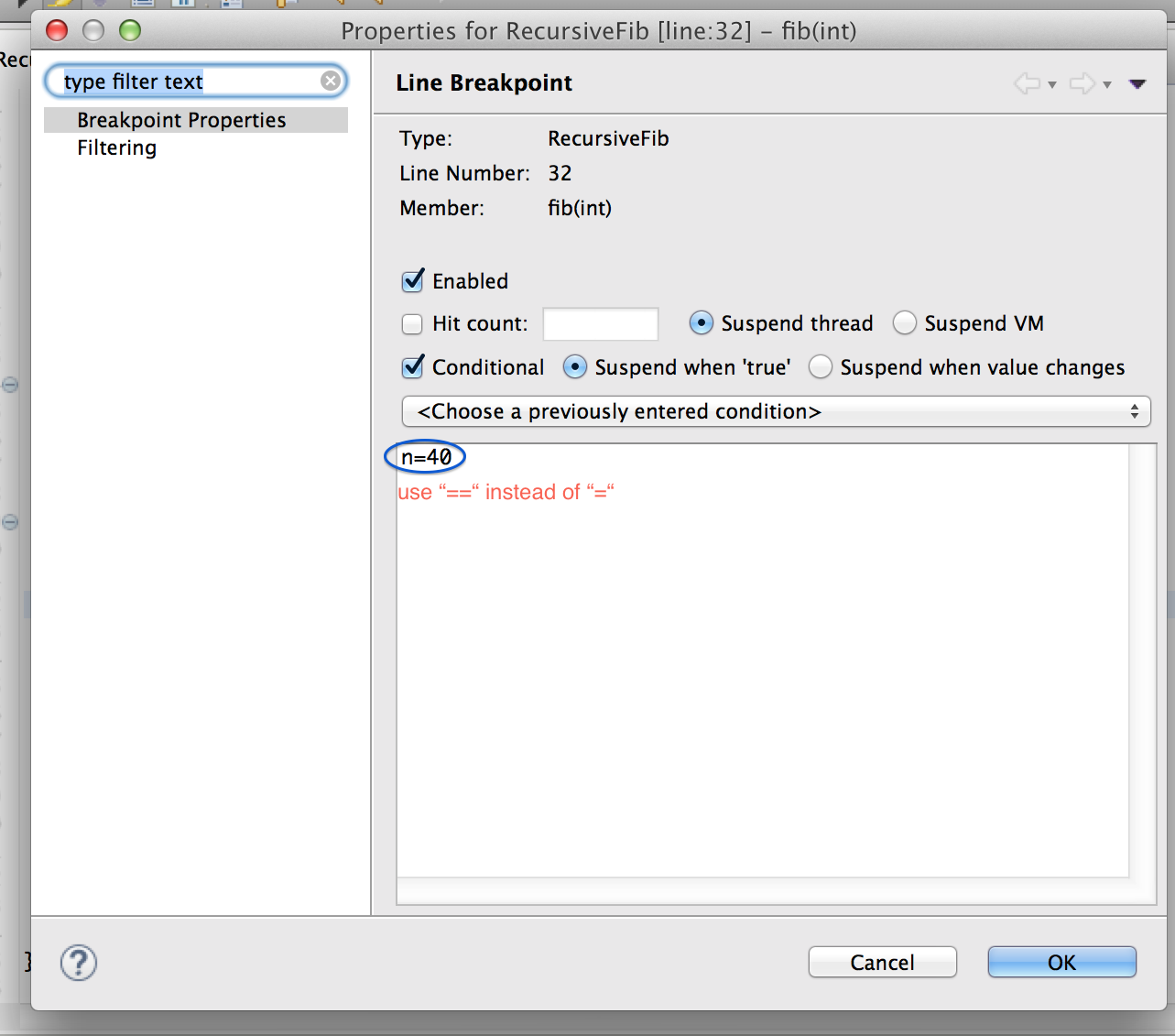
int first = computeFibRecurse(n-1);

int second = computeFibRecurse(n-2);

return first + second;

}

Now we want to trace into the fib method. Let's say we want to step through nthFib(40). But it is called in a loop. We don't want to click the "Step Over" button 40 times. That's where *conditional breakpoints* come in.

Make a breakpoint on line if (n <= 2). (or whatever your condition for your base case is) and Right-click on it and select Breakpoint -> Properties. Click on “Break when hit count” and enter 40 after “equals to”. **Make sure to use ==**.  
  
  
Now select **Run → Debug** from the menu. What happens?  
If the program does not stop, you may not be in debug perspective. Look at the upper right. That shows the current perspective. If Debug is not selected, select it. Now try Run → Debug. ***If this is moving very slowly (it was for me), try decreasing the conditional on the breakpoint to 20 and debug again.***

1. Step into the fib method. Keep stepping until you reach the line
   1. int first = fib(n - 1);
2. Step into that call again. And keep clicking on the “Step Into” button. Watch the call stack (Window -> Debugging -> Call stack) and the variable display (Window -> Debugging -> Variables). Describe what you see. What is n? What is on the call stack?
3. Not surprisingly, when you debug a recursive method, you can observe that the method calls itself multiple times.
4. It would be pretty tedious to keep stepping in all the way to the base case, so set a breakpoint inside the base case (e.g. on the line that says return 1;) click the Continue/Resume button
5. How many invocations of fib are on the run time stack now?
6. Remove the breakpoint on the return 1 line and set a breakpoint at the line  
   int result = first + second;
7. Click the Continue/Resume button (or hit the F5 key). What is the value of n? What is the value that is about to be returned? (You may need to click on the Variables tab to see the values of n, first, and second.)
8. Click Continue a few more times. Write down the values of n, first, and second, that you encounter. Do this for at least ten values.
9. Now look at the values that you recorded. What do you notice? Why does that explain that the method runs so slowly?

### Logging

### In the previous section, you saw how to use the debugger for analyzing, and that is often the best strategy. It is, however, not the strategy that students use most commonly. Often students sprinkle their code with print statements like these:

System.out.println("Here");

System.out.println(n);

Print statements can waste a lot of your time. You put them in, you take them out, you put them back in, you comment them out, you remove the comments, and when it all works you take them out for good. Until the next bug appears and you wish you had left them in. In this section, you will practice the use of *logging* statements. Logging is easy. Instead of System.out.println, simply use Logger.getGlobal().info.

Logging is better than System.out.println for two reasons:

* You can leave the logging statements in your code when you turn it in, demonstrating good software engineering practice.
* It is easy to turn all logging statements on or off.

Logging can be better than using the debugger.

* The price of entry is much lower. It is very easy to add a logging statement, but it takes some effort to start a debugger session.
* With logging, you have a complete record of what happens (provided, of course, you inserted enough statements to have a complete record). With a debugger, you only have that if you diligently wrote your observations into your notebook.

Let's try this with the fib method. At appropriate places, add the following logging statements. You may need to modify your code slightly to add the returned value to the logging statement.

Logger.getGlobal().info("Entering fib. n=" + n);  
Logger.getGlobal().info("Exiting fib. return=" + ...);

1. In the main method, add Logger.getGlobal().setLevel(Level.ALL);  
   Run your program (changing the upper limit of the loop in the main method to a small enough value). What is the output?
2. How does the output demonstrate the inefficiency of the method?
3. Now change the setLevel call of the main method to  
   Logger.getGlobal().setLevel(Level.OFF);  
   Run your program again. What happens?
4. That is the power of logging. You can deactivate the messages without having to remove them. Suppose you find another bug and want to turn logging back on. How do you do that?

In a small program, using Logger.getGlobal().info works fine. As your programs grow larger, you can control your logging in more sophisticated ways. You can use different logging levels. Instead of info, call methods severe for important messages or fine for "fine-grained" messages. Then call the setLevel method to set the level of the messages that you want to see in a particular program run. You can also define your own logger objects in addition to Logger.global. Keep this in the back of your mind; it will come in handy in the future.

**STOP**

*Make sure everyone in your group has completed Part 3 before moving on to the exit interview.*

### Optional

These are two more optional exercises to help you improve your programming skills and better understand recursion.

### **Bonus (2pt)**

#### Optimizing recursive Fibonacci **(1pt)**

One way of making the method more efficient is to use a technique called *memoization*. Remember previously computed results and don't recompute them.

1. Add a static array knownFibonacciValues to the class.
2. In the computeFibRecurse method, first look up knownFibonacciValues[n].
   1. If it is non-zero, return it.
   2. Otherwise, compute it in the normal way. Just before returning, call knownFibonacciValues[n] = *return value* so that it is known the next time.
3. Turn the logging back on. What is the logging output now?
4. What happens when you try again with n = 50?

### 

#### Count the number of recursive calls **(1pt)**

To look into this problem, you’re going to use software to analyze software. Add an instance variable: private int[] callCounter. In nthFib(n), initialize call counter so that its length is n+1. In computeFibRecurse(n), increment callCounter[n]. Add a printCallCounter() method that prints like this:

0 calls to fib(0)

2584 calls to fib(1)

4181 calls to fib(2)

2584 calls to fib(3)

1597 calls to fib(4)

...

Run your app to compute fib(20) and then call printCallCounter(). You should notice a pattern in the call counts. You have seen these numbers very recently. What is the pattern?

# Saving your work

# It can be a good idea to back up your work in case you ever accidentally delete your eclipse-workspace. You can export your work as a jar file like you do for the homework as a form of back. You can import jar files directly into Eclipse. Later this semester, we will learn better ways to back up your work using version control and GitHub.

**Please submit your lab6.jar file on Canvas.**

1. Modified from material provided by Dr. Philip Heller and Dr. Cay Horstmann and Dr. Chakarov [↑](#footnote-ref-0)