Lesson 15 – Testing, Debugging, & Analysis

This lesson will be a little bit different from prior lessons because you aren't going to be learning new *coding* principles per se, but some general stuff related to testing, debugging, and analyzing your programs. There will be a lot of reading, and not that much coding. This is motivated in large part by the APCS curriculum, as they expect you to know a few terms and concepts. But we'll add in our own stuff as well. While you won't learn as much syntax or coding in this lesson, I think the concepts will help you reason about programs. Debugging practice and tips are always very helpful as well.

Error Types

Ok, let's jump in. We'll start with the easy stuff. There are three "categories" of error (that's not a technical term.) Syntax, runtime, and logic.

Syntax errors, also known as **compile-time** are the kind of error that result in red squiggly lines and you being unable to run your program. More literally, they are problems with your code that make Java unable to compile your program. For example, you could forget a semicolon or a bracket, attempt to call a method that does not exist, have a typo in a variable name, etc. There are a ton of different kinds of syntax errors, but basically anything that will prevent your program from compiling is a syntax error. Syntax errors are maybe the most annoying because they stop you from running your program at all, but they are also usually the lowest risk and easiest to fix. Since you can't run your program, you *know* you have a bug, and your IDE will typically point out the exact location for you. This makes it easy to fix, and you don't ship your program with a syntax error unless you never even attempt to run it. So, despite being annoying, syntax errors are usually the cheapest and least-scary kind of error.

Runtime errors are similar to syntax errors in that they crash your program so you know you have a bug, however, they do not *prevent* you from running your program, because they are caused by attempting some illegal operation in your code that isn't necessarily identifiable beforehand. An example of this would be if you took an integer as user input, and then divided 10 by that number. If the user enters zero, you will run ten divided by zero, which is an impossible operation and your program will crash. However, the IDE can't detect that this will happen ahead of time. You've probably seen other runtime errors as well, such as "IndexOutOfBoundsException" or "NullPointerException". Maybe if you made a mistake in the recursion assignment you ran into "StackOverflowException". There a ton of different runtime errors. Although an IDE can't definitely detect them ahead of time, it can tell you what kind crashed the program, and what line the crash occurred on, which are big clues in terms of fixing them. Since they can't be detected ahead of time, they are scarier than syntax errors. After all, you could test your program a hundred times before happening to run across the particular condition that crashes it. However, when you do crash it, you get a lot of good info on how to fix it, so runtime errors are usually not too bad to fix once they identified. Of the three error categories, these are in the middle in terms of risk and scariness.

Logic errors are the scariest kind of error because they don't crash your program at all. Simply put, a logic error is when your program *runs*, but it doesn't do what it is supposed to do. Sometimes this is super obvious. For example, if your goal is to output a number, and no number is output, clearly something is wrong. However, it can be less obvious. If your goal is to output a specific number (e.g. you did some calculation), and your program outputs a number, that number may or may not be the *correct* number. So you have to do some work to verify that it is. And if it *isn't*, your IDE will not give you any hints as to *why* it isn't – you'll have to debug it yourself. Often logic errors are reasonably easy to detect and solve. For example, if you're writing a calculator program and your add method returns nine when you run three plus three, you'll probably notice it quickly, see that you did multiplication instead of addition, and fix it. However, the larger the program gets, the harder logic errors become to identify and fix. In your Steamworks or Power Up projects, you may have had an issue at some point where scores were calculated incorrectly, or did not reset when a new game started, or some other issue where your program *ran* but didn't *work*. These are logic errors. For companies, these can be exceedingly costly. For example, you could write code that saves data to a database. It might save data *slightly* incorrectly. No one notices that it's wrong, and it goes about its business for six months. Then someone notices there's a problem, but there are now six months of irreparable bad data saved. This could cost millions of dollars. As such, logic errors are the scariest kind of error.

Detecting Errors

Detecting errors is the first step in fixing them. The most basic thing for detecting errors is simply to **test your code**. Simply attempting to run your program will immediately detect any syntax errors. Running your code and doing stuff in your program will detect a lot of runtime errors. Paying attention to the results and checking that they are correct will unveil some logic errors.

But, simple testing is not enough. You have to test really thoroughly and cover edge cases. What happens if the user types in garbage input – will the program tell them to try again, or will it crash in a ball of fire? And what if a really strange but valid series of events happens? Think about a robot – what happens if you run twenty different commands in quick succession, then spin around three times. Does this make your robot code crash, and your robot disables itself for the rest of the match? It's important to test all the paths your code can take. (When you have conditionals in your code (e.g. if statements or loops), there are different "paths" through the code.)

Testing

One important concept is that of an "**edge case**" or "**boundary case**". An edge case is a case at one boundary of possibilities. For example, say you have a loop that runs from 1 to 10. When i = 1 or 10, those are edge cases. Edge cases aren't just for loops or sequences of numbers, though. Any weird situation could be an edge case. When you test your programs, you should do so with the mentality of "how can I break this", and start throwing whatever weird stuff you can think of at it. "What if I run the same menu option a bunch of times in a row?" "What if I try to do something that should be impossible in real life?" "What if I try to do two things at the same time?" Etc. etc. Do whatever you can to break it. Don't *just* be random, though – when you know you have loops, sequences, or odd combinations of paths, go out of your way to test all of them.

Testing for a specific thing is called a "**test case**". For example, you could have a test case that when you start the program, choose menu option A, and press 7, the program should output the string "You did a thing!". You can run through this test case as you work on your program to make sure it's functioning more properly. This one is pretty obvious. Sometimes they are more complicated, for example if you programmed the quadratic formula and you have test cases to confirm that it's returning the proper result for various numbers. Whenever we do a "systems check" on the robot we're basically running a bunch of test cases. "When I press the joystick forward, does the robot drive forward?" Etc.

Creating a comprehensive suite of test cases is a good way to ensure your program is functioning *mostly* properly (large programs almost always have bugs in them that just haven't been discovered yet.)

The idea of creating test cases can be taken pretty far, to the point where people actually *write their code around their test cases*, in a process know as "**Test-Driven Development**", or TDD. The idea behind TDD is that if you write all possible test cases for your code, and then develop your program to the point where all your test cases are shown to be correct, you're just *done*. To make this process easier, the test cases are automated so they can run on their own and automatically notify you if any of the test cases are incorrect. This makes logic errors more similar to syntax errors, because as soon as you run your program you will see if there is a problem. These automated test cases are called "**unit tests**", and the process of writing and running them is called unit testing. Unit testing has a lot of advantages – in particular, once you're written a test case, it will work on its own indefinitely without forcing you to run through your code manually, and will easily detect problems. The downside is that it takes a fairly significant amount of extra effort to write all the unit tests. This effort is not always worth it, but for projects where it is, it can be extraordinarily valuable.

Unit tests often cover a very small chunk of code or part of your program. There is also a process called "**integration testing**" where you test that your program works together as expected. For example, say you write a bunch of classes, and write perfect unit tests for each of them. Say, you've written a Drivetrain class for your robot, and written unit tests so you're certain all the drivetrain methods are correct. Then you've done the same for Elevator and Robot classes. You know all three classes work perfectly. When you combine them together into a program, will the program as a whole work? Testing this is called integration testing.

Testing

The APCS exam uses the concepts of **preconditions** and **postconditions**. A precondition is a condition which you can assume will be true at the start of your program or method. For example, if you had a program that loops through an array of integers and divides a number by each of the integers, you might be given a precondition that none of the integers in the array are zero. This would save you from having to check if you're about to divide by zero before the division operation. If there were no precondition, you would not be able to make this assumption. A postcondition on the other hand is something that you have to make true. E.g., if the postcondition is when you are done with an array, every integer in it must be negative, you must somehow make that happen or you would get the question wrong. When you see a problem these are usually pretty self-explanatory. We don't really use these in this course or in robot code development, but maybe if I get a chance I will try to phrase some of our assumptions and requirements in terms of pre- and postconditions in the future so you're used to them.

An **assertion** is a statement that posits that some condition is true. For example, you could assert that an integer variable *myNumber* is equal to 12. The assertion could return true or false. There is an *assert* keyword in Java although it's not tested on the exam (you will see this whenever we do unit tests), but they say you need to know what an assertion is. So now you know.

Debugging

Programs of non-trivial length will never work correctly on the first try, so the ability to debug them is critical. This is a skill that takes practice, but the APCS exam will test you on a couple of things. First, you will need to be able to run through programs by hand. This means looking at code and reasoning through it in your head, or with pencil and paper. For example, if there is a for loop that does some math, can you calculate what the final result will be? Will you accurately count how many times the loop runs, and what variables get assigned to? Etc. This is called hand-tracing code.

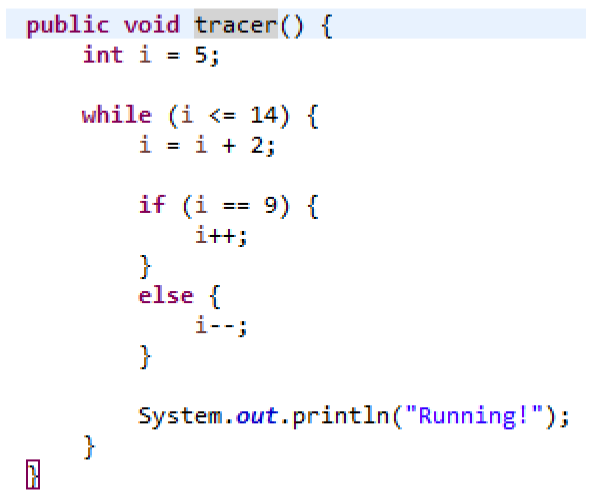
They also expect you to know about the concepts of adding output statements (SOPs) and using the debugger. Specifics of using a debugger will not be tested because there is no set IDE for the course, but there may be a multiple choice question about what debuggers are. Also, these skills are critical for actually writing code, so you'll want to be familiar with them and well-practiced.

Assignment

Some of these should be answered in prose, others are code assignments. For any written questions that ask you to provide examples, do not use examples that were used in this document itself.

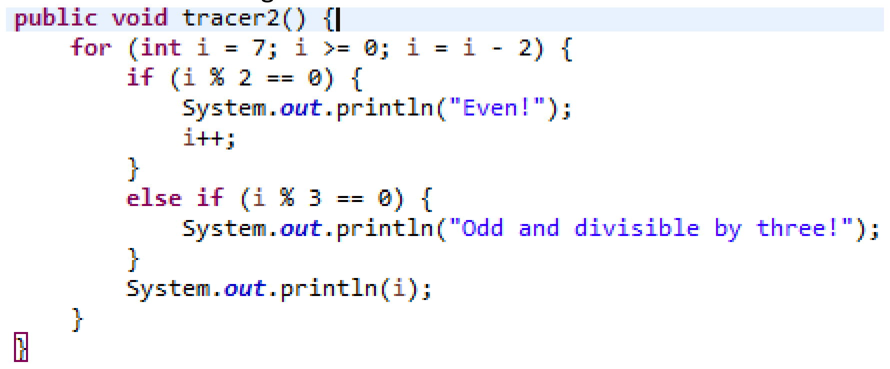
1. Describe in your own words the three categories of errors. Discuss the differences between them and how "dangerous" each kind is. Describe a hypothetical example of each category.
2. What is an edge case? Describe a hypothetical example.
3. What is a test case? Describe a hypothetical example.
4. Explain what unit tests are and what TDD is. Discuss some advantages and disadvantages of these approaches.
5. Describe integration testing and include a hypothetical example.
6. Describe pre- and postconditions and include hypothetical examples.
7. What is an assertion? Provide an example.
8. Describe each of the different debugging methods mentioned in this document.
9. Create a program called error tester that has a syntax error, a runtime error, and logic error in it. Add comments to the code that clearly indicate where these errors are. Then create the same program but with all the errors fixed. Submit both programs, clearly labeling which is which.
10. Consider the following method. I cannot enforce this, but please answer 11 and 12 without implementing this code for yourself (e.g., figure out the answers by hand by tracing the code

in your head.)



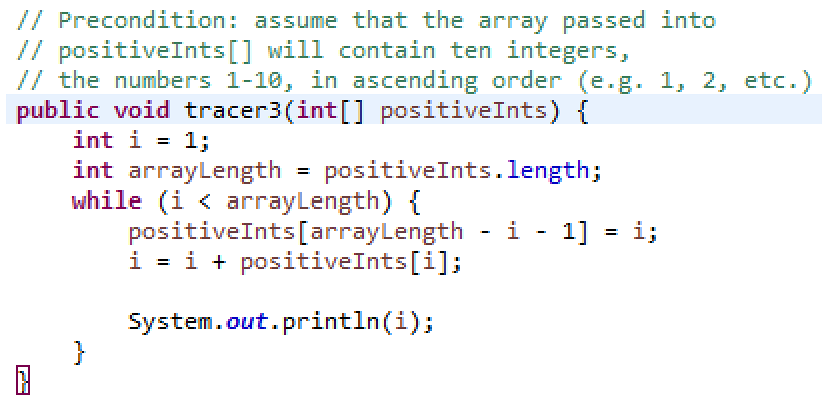
```
public void tracer() {
    int i = 5;

    while (i <= 14) {
        i = i + 2;

        if (i == 9) {
            i++;
        }
        else {
            i--;
        }

        System.out.println("Running!");
    }
}
```

11. When called, how many times will this method print the line “Running!”?
12. What will the value of i be when the loop stops running?
13. Consider the following method:



```
public void tracer2() {
    for (int i = 7; i >= 0; i = i - 2) {
        if (i % 2 == 0) {
            System.out.println("Even!");
            i++;
        }
        else if (i % 3 == 0) {
            System.out.println("Odd and divisible by three!");
        }
        System.out.println(i);
    }
}
```

14. How many times will the loop output “Even!”?
15. How many times will the for loop run?
16. What will be the last value of i for which the loop runs?
17. Consider the following method. **Look closely at the precondition.** Also note that i does not start at zero. This one is a little bit tricky.



```
// Precondition: assume that the array passed into
// positiveInts[] will contain ten integers,
// the numbers 1-10, in ascending order (e.g. 1, 2, etc.)
public void tracer3(int[] positiveInts) {
    int i = 1;
    int arrayLength = positiveInts.length;
    while (i < arrayLength) {
        positiveInts[arrayLength - i - 1] = i;
        i = i + positiveInts[i];

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

18. How many lines of output will this program have? What will the output be?
19. What is the final value of the positiveInts array?