Testing

by Sophia

WHAT'S COVERED

In this lesson, you will work through testing code and identify some of the common problems and issues. Specifically, this lesson covers:

1. Syntax

It's great when programming goes well. But what happens when there are issues? To set the stage, consider a simple program shown below. The command `system.out.println( )` is a method that allows Java to output content for the user to see. Inside of the parentheses, a string is enclosed in double quotes. The quotes around the string must be double quotes and must be paired correctly. The quotes tell the program where the string starts and where it ends. There will be an error if the syntax is not written exactly in this way.

The code below shows a correct and an incorrect statement:

```
public class ErrorExample {
 public static void main(String[] args) {
   // This statement is correct
   System.out.println("Hello, world!");
   // This statement is incorrect
   System.out.println("Hello, world!);
 }
}
```

REFLECT

Can you spot the error? The second output tis missing the paired closing quotes to indicate where the string ends. The line should read as follows: **System.out.println("Hello, world!");**.

TRY IT

Directions: On the IDE, create a file called ErrorExample.java. Type in the code shown above and run the code.

The results should look like this:

```
Main.java:6: error: unclosed string literal
   System.out.println("Hello, world!);
                      ^
1 error
```

REFLECT

The IDE sometimes shows red squiggly underlining to flag **syntax errors**, but it may not always do so. In this case, the output shows that there was a syntax error on line 6. The error message “unclosed string literal” is the Java compiler’s way of telling you that the closing double quote is missing at the end of the string:

> System.out.println("Hello, world!");

Java will identify specific errors. In some cases, a single mistake in the code may produce multiple errors.

REFLECT

Writing code is an exact art. Something as simple as a period or space added in the wrong place can cause errors. Don’t let this discourage you though. As you encounter these errors you will learn from them and start to remember the correct syntax to program your ideas. And a bonus is that many of the programming constructs you learn for one language transfer to others.

HINT

To research error messages, copying and pasting the error into an internet search browser to search the error will provide insights into what issues could be causing the error. This is a good step to follow when it is not clear what the error represents. It can be used when programming in any language.

It is important to note that if there are multiple errors, which is quite common in larger programs, issues should be fixed from the top down. In many instances, an issue at the top of the error list could seemingly show that there are many more errors than there really are. Focus should be on the first error, not the total number of errors.

BIG IDEA

Remember that quotes are needed around the literal string. If quotes are not used around the literal string, Java will assume that it is a variable or another keyword. This is not critical now and will be covered in a future

challenge.

TERMS TO KNOW

System.out.println( )
A method that allows Java to output content to the Shell (screen).

Syntax Error
A syntax error means that you have violated the “grammar” rules of Java.

2. Comments

It is common to see developers add notes or comments in their code to explain what the code is doing and document the design for both themselves and others who are reviewing the code later on. If a comment is just a single line or part of a line, it should begin with `//` (double forward slash). Everything to the right of the `//` on the line is ignored by Java.

If a comment spans multiple lines, the beginning of the comment is marked by `/*` and then `*/` is added at the end of the comment. When using `/*` and `*/` to mark comments, only the material between them is ignored.

TRY IT

Directions: In the IDE, enter the following code in a file named Hello.java:

```
public class Hello {
 public static void main(String[] args) {
   // This code prints out the message 3 times.
   System.out.println("Hello, world!");
   System.out.println("Hello, world!");
   System.out.println("Hello, world!");
 }
}
```

REFLECT

As noted in the previous tutorial introducing the IDE, it is possible to select lines in a code listing like the one above, copy them, and paste them into an appropriately named .java file on the IDE, but such copying is not really a good idea when you are learning the language.

Remember that the file name is also important. As the instructions above indicate, this code must be typed into a file named Hello.java. The spelling and capitalization of the file name must match the name of the public class in the code (and the file has to have the extension .java).

Then, run the program.

The output should look like this:

```
Hello, world!
Hello, world!
Hello, world!
```

REFLECT

After running the code, you will notice the code does not produce any errors. Java simply ignores anything after the `//` comment.

2a. Using Comments for Clarity

Developers may use comments after each line to add clarity as well. Consider the following example:

```
public class Hello {
 public static void main(String[] args) {
   // This code prints out the message 3 times.
   System.out.println("Hello, world!"); // 1st output
   System.out.println("Hello, world!"); // 2nd output
   System.out.println("Hello, world!"); // 3rd output
 }
}
```

These additional notes in comments are hidden from users, but they help other developers understand the code functionality and can also help when tracking down issues.

2b. Using Comments to Hide Lines of Code

Developers may also use comments to remove lines of code temporarily. At times, there may be certain parts of the code that are not functioning. These parts need to be isolated and further tested before they will function as intended.

Using comments to isolate issues can be tested by adding in an error. In this case, we have intentionally left off quotes in line 5 of the following example:

```
public class Hello {
 public static void main(String[] args) {
   // This code prints out the message 3 times.
   System.out.println("Hello, world!"); // 1st output
   System.out.println(Hello, world!);  // 2nd output
   System.out.println("Hello, world!"); // 3rd output
 }
}
```

If it is not clear how to fix the issue, and the line did not affect the rest of the code, it can be commented out using // at the start of the line as seen in line 5 below:

```
public class Hello {
 public static void main(String[] args) {
   // This code prints out the message 3 times.
   System.out.println("Hello, world!"); // 1st output
   // System.out.println(Hello, world!);  // 2nd output
   System.out.println("Hello, world!"); // 3rd output
 }
}
```

TRY IT

Directions: Run the program again with comments added before the error in the code.

The results should look like the following:

```
Hello, world!
Hello, world!
```

REFLECT

Since the code on line 5 was commented out, the code on that line wasn't run, and the second output wasn't displayed.

TRY IT

Directions: It may not be obvious which line was commented out, so try changing the output slightly by numbering the lines as demonstrated below:

```
public class Hello {
  public static void main(String[] args) {
    // This code prints out the message 3 times.
    System.out.println("1. Hello, world!");   // 1st output
    //System.out.println("2. Hello, world!"); // 2nd output
    System.out.println("3. Hello, world!");   // 3rd output
  }
}
```

The results should look like the following:

```
1. Hello, world!
3. Hello, world!
```

Numbering each line of the output makes it clear which line was commented out.

REFLECT

As you become a more experienced programmer you will find that comments are your “best friend.” You’ll use them to keep track of what you’re doing and to document your program for future reference. You’ll also find that they enable trying out different ideas quickly by using them to skip code that you have without deleting it, and then adding a new line of programming while keeping your overall programming structure. If your new idea doesn’t work you can then just delete (or comment) it, remove the comment markings for the code you skipped, and get back to your original line of thought.

---

3. Best Practices With Comments

When working with a program, it is good practice to include some details about that program. This could include details like who developed the program, when it was created, and what the program is for. Additionally, it could include details regarding how to use the program and any additional information that could be useful for someone using the program later on. It could also be beneficial to users, even if they don’t know what the code is actually doing.

Consider the following program, which may seem a bit daunting at first glance:

```
public class Max {
  public static int maxOfTwo(int firstNumber, int secondNumber) {
    if(firstNumber > secondNumber) {
      return firstNumber;
    }
    return secondNumber;
  }

  public static int maxOfThree(int firstNumber, int secondNumber, int thirdNumber) {
    return maxOfTwo(firstNumber, maxOfTwo(secondNumber, thirdNumber));
  }

  public static void main(String[] args) {
    System.out.println("Highest number is " + maxOfThree(1, 2, 3) + ".");
  }
}
```

THINK ABOUT IT

It may or may not be obvious what the program is doing by reviewing the code. And it would likely take a bit of time to figure it out. If the author would have included comments about what the program was designed for, intentions would be clear without taking extra time to decipher the code itself. Consider the following program. Comments have been added for clarification:

```
/* Author: Sophia
  Created Date: Feb 07, 2022
  Description: This program has two methods to find the maximum
  value of three numbers.
  Example of usage: maxOfThree(20, 30, -10)
  Result: method returns 30 */

public class Max {
  public static int maxOfTwo(int firstNumber, int secondNumber) {
    if(firstNumber > secondNumber) {
      return firstNumber;
    }
    return secondNumber;
  }

  public static int maxOfThree(int firstNumber, int secondNumber, int thirdNumber) {
    return maxOfTwo(firstNumber, maxOfTwo(secondNumber, thirdNumber));
  }

  public static void main(String[] args) {
    System.out.println("Highest number is " + maxOfThree(20, 30, -10) + ".");
  }
}
```

That was a good starting point to begin clarifying specific details. It is clear who created the program. If there are questions, the author can be contacted now. Knowing when the program was created can also be helpful. As code is updated during the maintenance phase, this will allow for the author to be contacted if any issues arise. Notice as well, there is a description that explains what the code does and an example of how to use it.

HINT

This example program uses two separate methods (`maxOfTwo()` and `maxOfThree()`). For now, consider methods to be small reusable pieces of code. Rather than redeveloping everything from scratch each time, they can be made and used later on. We will explore methods in detail later.

The author could add comments to help users understand the purpose of these methods without understanding exactly what the code does. Consider the following revision:

```
/* Author: Sophia
  Created Date: Feb 07, 2022
  Description: This program has 2 methods to find the maximum value of 3 numbers.
  Example of usage: maxOfThree(20, 30, -10)
  Result: method returns 30 */

public class Max {
  /* Method: maxOfTwo
        Purpose: This method accepts two numbers, compares them
        and returns the value that is larger. */
  public static int maxOfTwo(int firstNumber, int secondNumber) {
    if(firstNumber > secondNumber) {
      return firstNumber;
    }
    return secondNumber;
  }

 /* Method: maxOfThree
   Purpose: This method accepts three numbers, sends second and third
        number to maxOfTwo. It then takes the result of that comparison to
        send the first number and the result to get the larger value
        of all three. */
 public static int maxOfThree(int firstNumber, int secondNumber, int thirdNumber) {
    return maxOfTwo(firstNumber, maxOfTwo(secondNumber, thirdNumber));
 }

 // Test of maxOfThree() method
 public static void main(String[] args) {
   System.out.println("Highest number is " + maxOfThree(20, 30, -10) + ".");
 }
}
```

Even though the logic of the code may not be understood, the intent of what the code is programmed to do should be clear with the additional comments.

TRY IT

Directions: Copy in the following code into a file name Max,java on the IDE and test to see if it functions as expected:

```
/* Author: Sophia
  Created Date: Feb 07, 2022
  Description: This program has 2 methods to find the maximum value of 3 numbers.
```

```
  Example of usage: maxOfThree(20, 30, -10)
  Result: method returns 30 */

public class Max {
  /* Method: maxOfTwo
     Purpose: This method accepts two numbers, compares them
              and returns the value that is larger. */
  public static int maxOfTwo(int firstNumber, int secondNumber) {
    if(firstNumber > secondNumber) {
      return firstNumber;
    }
    return secondNumber;
  }
  /* Method: maxOfThree
     Purpose: This method accepts three numbers, sends second and third
          number to maxOfTwo. It then takes the result of that comparison to
          send the first number and the result to get the larger value
          of all three. */

  public static int maxOfThree(int firstNumber, int secondNumber, int thirdNumber) {
    return maxOfTwo(firstNumber, maxOfTwo(secondNumber, thirdNumber));
  }
  // Test of maxOfThree() method
  public static void main(String[] args) {
    System.out.println("Highest number is " + maxOfThree(20, 30, -10) + ".");
  }
}
```

The result should look like this:

```
Highest number is 30.
```

REFLECT

Knowing who created a program, when it was created and what version of the program you are looking at is essential to understanding it and how to deploy it; and comments are the first tool to use to capture this information. Configuration management and software version control is a full discipline to itself with larger organizations having dedicated teams of employees to perform these tasks. The programs developed throughout these lessons won't get big enough to be an issue, but as you write larger and more complex programs, and integrate them into an overall programming architecture, keeping up with "who's on first" will become ever more important.

BRAINSTORM

Try to change around the values that are being passed in as well and see if you can accurately predict the results.

4. Edge and Corner Cases

It is critical to ensure that a program works correctly and logically. Although every potential input cannot be assumed, it is generally a good idea to also test edge and corner cases. The **edge cases** are values that are at the end of a testing range.

Let's take the following program that should take in a score for an exam that was given to us as a request. If the score is between 90 and 100, it should state that the student received an A. If the score is between 80 and 89, it should indicate that the student received a B. If the score is between 70 and 79, it should report a C. If the score is less than 70, it should indicate that the student did not pass the exam.

Given this description, this was the (flawed) program that was created:

```
import java.util.Scanner;

public class Grade {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("What is your exam score (0 - 100): ");
    int score = input.nextInt();
    // && is logical/Boolean AND
    if(score > 90 && score < 100) {
      System.out.println("You got an A. Congratulations!");
    }
    else if(score > 80 && score < 90) {
      System.out.println("You got a B. Well done!");
    }
    else if(score > 70 && score < 80) {
      System.out.println("You got a C.");
    }
    else {
      System.out.println("You did not pass the exam.");
    }
  }
}
```

TRY IT

Directions: Let's just try this program as is to see if it works. Copy the code into a file name Grade.java on the IDE for testing.

EXAMPLE First, let's try entering in 91:

```
What is your exam score (0 - 100): 91
You got an A. Congratulations!
```

REFLECT

Look, it seems to work, right?

EXAMPLE Let's try entering 92:

```
What is your exam score (0 - 100): 92
You got an A. Congratulations!
```

EXAMPLE Let's try entering 93:

```
What is your exam score (0 - 100): 93
You got an A. Congratulations!
```

EXAMPLE Let's try entering 94:

```
What is your exam score (0 - 100): 94
You got an A. Congratulations!
```

REFLECT

Our first instinct when testing is to try out what we think will or should work. These interior tests are an example of this. You may run into errors with these tests too. Errors here will often be embedded in the fundamental logic of your program since it is not doing the most basic tasks for which it was designed.

TRY IT

Directions: Try to enter in 100, as that is at the end of the first range and see what happens:

```
What is your exam score (0 - 100): 100
You did not pass the exam.
```

REFLECT

Oops! You would generally expect a student that had a perfect grade to have the same congratulatory message as the one that had a 94, right? You will dig into the code a bit further in later tutorials, but the issue here is that in the check, the score doesn't include 100; it only checks if the score is less than 100.

TRY IT

Directions: As a quick fix, try changing that code as follows:

```
import java.util.Scanner;
```

```
public class Grade {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("What is your exam score (0 - 100): ");
    int score = input.nextInt();
    // && is logical/Boolean AND
    if(score > 90 && score <= 100) {
      System.out.println("You got an A. Congratulations!");
    }
    else if(score > 80 && score < 90) {
      System.out.println("You got a B. Well done!");
    }
    else if(score > 70 && score < 80) {
      System.out.println("You got a C.");
    }
    else {
      System.out.println("You did not pass the exam.");
    }
  }
}
```

EXAMPLE Let's test the code again with an entry of 100:

```
What is your exam score (0 - 100): 100
You got an A. Congratulations!
```

REFLECT

Success! Well, for now. It is important to continue testing all of the other edge and corner cases as well. A **corner case** is when the value being tested is in between two edge cases. In this scenario, a value of 90 and 80 would be considered as a corner case since it is in between those two edge cases.

EXAMPLE Let's see what happens if we enter in 90:

```
What is your exam score (0 - 100): 90
You did not pass the exam.
```

REFLECT

Oops again, similar to the check on the 100, it is important to ensure that the value 90 is part of one of the ranges. At this point, 90 is not in either range, so let's just go ahead and do the same thing as we did on the 100.

TRY IT

Directions: Change the code to change the second condition to include 90:

```
import java.util.Scanner;

public class Grade {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("What is your exam score (0 - 100): ");
    int score = input.nextInt();
    // && is logical/Boolean AND
    if(score > 90 && score <= 100) {
      System.out.println("You got an A. Congratulations!");
    }
    else if(score > 80 && score <= 90) {
      System.out.println("You got a B. Well done!");
    }
    else if(score > 70 && score < 80) {
      System.out.println("You got a C.");
    }
    else {
      System.out.println("You did not pass the exam.");
    }
  }
}
```

EXAMPLE Let's run the program again and try entering in 90:

```
What is your exam score (0 - 100): 90
You got a B. Well done!
```

REFLECT

That's not quite right, as a score of 90 should be an A. Instead, we should have changed the line with the first condition to include 90 as part of the range as demonstrated below:

```
import java.util.Scanner;

public class Grade {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("What is your exam score (0 - 100): ");
    int score = input.nextInt();
    // && is logical/Boolean AND
    if(score >= 90 && score <= 100) {
```

```
      System.out.println("You got an A. Congratulations!");
    }
    else if(score > 80 && score < 90) {
      System.out.println("You got a B. Well done!");
    }
    else if(score > 70 && score < 80) {
      System.out.println("You got a C.");
    }
    else {
      System.out.println("You did not pass the exam.");
    }
  }
}
```

EXAMPLE Let's test the program again and enter a score of 90:

```
What is your exam score (0 - 100): 90
You got an A. Congratulations!
```

We'll also need to do the same thing for the next two conditions as well, so let's fix those issues as demonstrated in the following code:

```
import java.util.Scanner;

public class Grade {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("What is your exam score (0 - 100): ");
    int score = input.nextInt();
    // && is logical/Boolean AND
    if(score >= 90 && score <= 100) {
      System.out.println("You got an A. Congratulations!");
    }
    else if(score >= 80 && score < 90) {
      System.out.println("You got a B. Well done!");
    }
    else if(score >= 70 && score < 80) {
      System.out.println("You got a C.");
    }
    else {
      System.out.println("You did not pass the exam.");
    }
```

```
    }
}
```

REFLECT

Edge and corner case tests move you out of the “I know this works” realm into the gray area of testing where “I think this might work.” You’ll need to do these types of tests to make your programs robust. So, check above, below and in-between to keep your programs clean.

Now let’s enter the fixed code into the IDE to see what happens.

TRY IT

Directions: Enter the fixed code above into the IDE in a file named Grade.java. Go ahead to test those corner and edge cases within that range and see what happens. Once you’ve done so, try to consider what happens if the user enters in 101 or -1, which is on the edge cases of the entire program. With a negative number, having a note that the user didn’t pass the example is acceptable. However, if the number is set to a value larger than 100, it’s most likely a larger error that we have to handle.

EXAMPLE Running the code should produce a result like this with a score of 101:

```
What is your exam score (0 - 100): 101
You did not pass the exam.
```

REFLECT

The only way to really know what your program is doing is to test it, and to do so thoroughly. It will always be best for you to take the first cut at it, since you wrote it. Don’t skimp on this process. There will almost always be a case where your original code won’t do what you expected it to do.

BRAINSTORM

Try to think about alternatives to handle this scenario. What should the result be? Should the user be informed that they entered an incorrect value?

TERMS TO KNOW

Interior Test
Test values that are within a specific range where the precise values that we entered within that range did not matter.

Edge Case
Values that are at the end of a testing range.

Corner Case
The value you are testing is in between two edge cases.

SUMMARY

In this lesson, you learned what a **syntax** error is and how Java helps to identify the right line and character where it notices a violation of the "grammar" rules. You then looked at how to **create comments** within code, both **for clarity** and **to hide lines of code**, to help with testing and documenting. You identified some of the **best practices for commenting**. Finally, you learned how to test a program using interior testing and testing values at **edge and corner cases**.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source www.cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from: "PYTHON FOR EVERYBODY" BY DR. CHARLES R. SEVERANCE. Source www.py4e.com/html3/

TERMS TO KNOW

Corner case
Value you are testing is in between two edge cases.

Edge case
Values that are at the end of a testing range.

Interior test
Test values that are within a specific range where the precise values that we entered within that range did not matter.

Syntax error
A syntax error means that you have violated the "grammar" rules of Java.

print( ) and println()
Methods that allow Java to output content to the Shell (screen). The println() method adds a new line at the end of the output.