## **Week 4 – Test Plan Assignment**

Structural or clear-box testing techniques are based on code coverage, which measures the percentage of code executed by the tests. Some coverage criteria include:

* Statement Coverage: Every statement is executed at least once.
* Decision/Branch Coverage: Every decision point (if, else if, switch, etc) results in a true outcome and a false outcome at least once.
* Condition Coverage: Every condition in a decision point results in a true outcome and a false outcome at least once.

## Learning Objectives

* Determine if a set of tests achieves statement coverage, decision coverage, and/or condition coverage.

## Instructions

**Download and edit this document to show the test results for the following tasks:**

* Testing Task #1: Gratuity.java
* Testing Task #2: PizzaStudyAnd.java and PizzaStudyNested.java
* Testing Task #3: PlayOr.java and PlayLadder.java
* Testing Task #4: Chapter 4 Project 1.
* Testing Task #5: Chapter 4 Project 5.

The files can be downloaded at <https://github.com/linda-seiter/cmsc115_2252/tree/main/week4/src/main/java>

Refer to the Revel sample runs for the exact I/O for the Chapter 4 programming projects.

*Note: This submission is only for the test plans. The Java code for Chapter 4 Programming Projects should be submitted in the Revel environment for grading. ￼*

### Testing Task #1 – Gratuity.java

Consider the following program requirements:

*Write a program that reads the tip rate and the number of customers. A minimum 20% tip is required for large parties with 8 or more customers. The program should output the tip rate, adjusted as necessary for large parties.*

Table 1 shows a decision table that defines when a tip adjustment is necessary. The tip rate is set to 20% only when both conditions are met; otherwise, the tip remains the same.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Table 1: Tip Rate Decision Table** | | | | |
|  | **Rule 1** | **Rule 2** | **Rule 3** | **Rule 4** |
| **CONDITIONS** |  | | | |
| tip < 20 | T | T | F | F |
| customers >= 8 | T | F | T | F |
| **ACTIONS** |  | | | |
| tip | Adjust to 20% | - | - | - |

The **Gratuity** class implements the requirements, but there is an error in the code. You can run the program online at <https://onlinegdb.com/Zi6Js355C>.

Program control flow can be visually depicted using a flowchart. Decision points are drawn as diamonds and all other statements are drawn as rectangles. The line number is displayed next to each statement.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | The decision point at line 17 has two branches, creating two possible execution paths.  The sequence 13,14,15,16,17 is denoted 13-17.   |  |  | | --- | --- | | **tip < 20 || customers >= 8** | **execution path** | | true | 13-17, 18, 20 | | false | 13-17, 20 |   The code error will be detected by demonstrating structural testing techniques.  Flowchart Creator [https://app.code2fow.com/](https://app.code2flow.com/) |

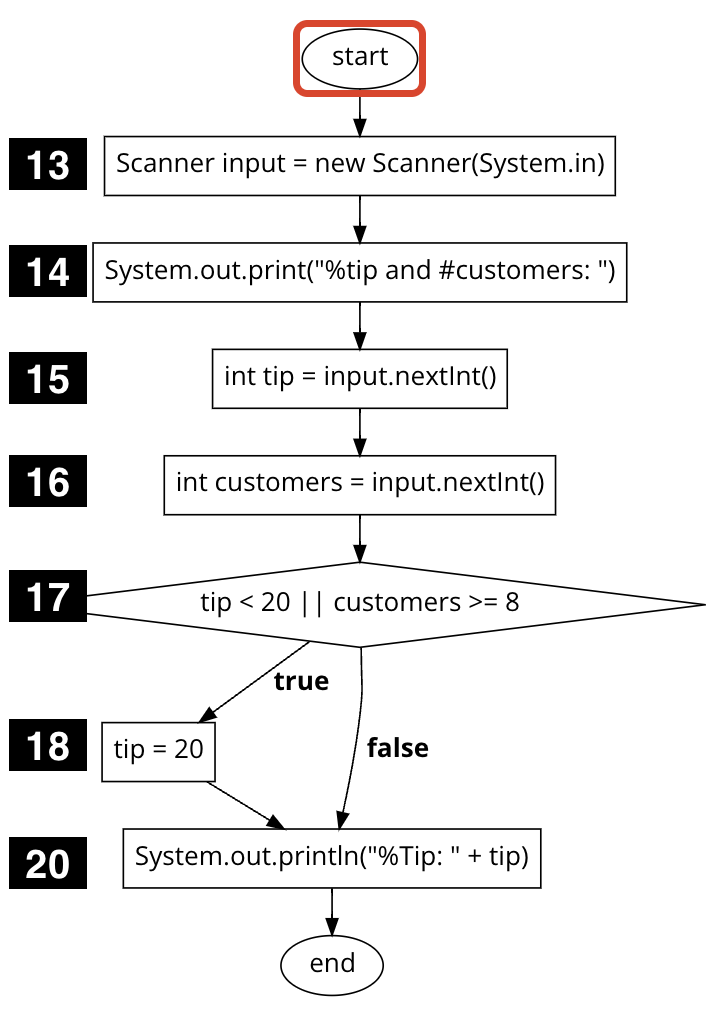
We’ll explore three basic code coverage measures for testing the **Gratuity** class.

* Statement Coverage: Every statement is executed at least once.
* Decision/Branch Coverage: Every decision point (if, else if, switch, etc) results in a true outcome and a false outcome at least once.
* Condition Coverage: Every condition results in a true outcome and a false outcome at least once.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table 2: Gratuity Test Cases**  **100% Statement Coverage** | | | | | |
|  | Expected I/O | Actual I/O | Status | Decision Point  (tip < 20 || customers >= 8) | Path |
| 1 | %Tip and #Customers: **10 12**  %Tip: 20 |  |  | true | 13-17,  18, 20 |

* Run the **Gratuity** program for the test case in Table 2 and record the results.
* Confirm test #1 passes.

Is **100% Statement Coverage** achieved by the single test in Table 2? Yes, it is. The test executes every statement along the path 13-17, 18, 20. Although the test passes, it fails to detect the error that exists in the code. Statement coverage is a weak level of code coverage that often fails to detect errors.

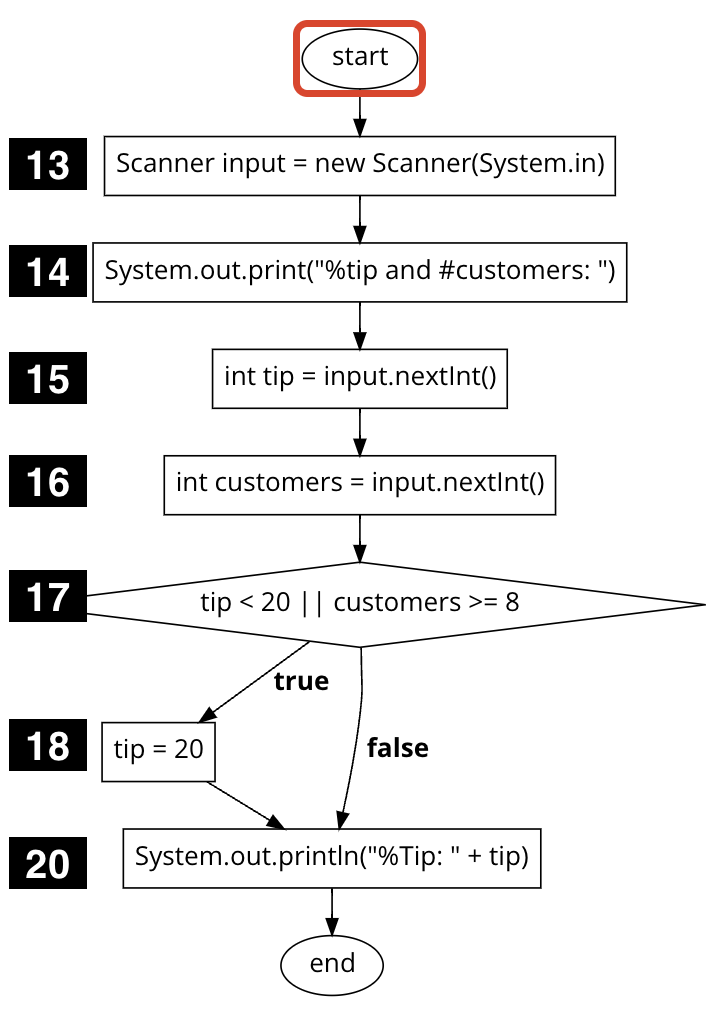


**Decision coverage** requires all branches of a decision point to be covered by at least one test. We’ll need two test cases to cover both the true and false branch of the decision point at line 17.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table 3: Gratuity Test Cases**  **100% Decision Coverage** | | | | | |
|  | Expected I/O | Actual I/O | Status | Decision Point  (tip < 20 || customers >= 8) | Path |
| 1 | %Tip and #Customers: **10 12**  %Tip: 20 |  |  | true | 13-17,  18, 20 |
| 2 | %Tip and #Customers: **22 4**  %Tip: 22 |  |  | false | 13-17,  20 |

* Run the **Gratuity** program for the test cases in Table 3 and record the results.
* Confirm both tests pass.

Do the tests in Table 3 achieve 100% decision coverage? Yes, they do. Test #1 follows the **true** branch and test #2 follows the **false** branch. But the code error remains undetected by the two passing tests.



Do the tests in Table 3 achieve 100% condition coverage? No, they don’t. It might appear as if both conditions evaluate as true in test #1 and false in test #2. However, Java short-circuits the operation and prevents the condition **customers >= 8** from evaluating as true, as shown in Table 4.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table 4: Gratuity Test Cases**  **75% Condition Coverage** | | | | | |
|  | Expected I/O | tip < 20 | customers >= 8 | Decision Point  (tip < 20 || customers >= 8) | Path |
| 1 | %Tip and #Customers: **10 12**  %Tip: 20 | true | **unevaluated** | true | 13-17,  18,20 |
| 2 | %Tip and #Customers: **22 4**  %Tip: 22 | false | false | false | 13-17,  20 |

Why does Java short-circuit logical operators **&&** and **||**? Given a compound expression such as **a || b** or **a && b**, the second condition **b** isn't evaluated if the result is clear after evaluating the first condition **a**. This is done for efficiency to avoid unnecessary computation.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 5: Java Logical Operator Short-Circuiting** | | | |
| **a** | **b** | **a || b** | **a && b** |
| true | true | true  short circuit  b not evaluated | true |
| true | false | true  short circuit  b not evaluated | false |
| false | true | true | false  short circuit  b not evaluated |
| false | false | false | false  short circuit  b not evaluated |

Given the logical expression **a || b**:

* If **a** is **true**, then Java short-circuits and returns **true** without evaluating **b**.
* If **a** is **false**, then **b** is evaluated, and its value is returned.

Similarly, given **a && b**:

* If **a** is **false**, then Java short-circuits and returns **false** without evaluating **b**.
* If **a** is **true**, then **b** is evaluated, and its value is returned.

*Notice that* ***a || b*** *and* ***a && b*** *produce the same value when the conditions are both true or both false, which is why the two tests in Table 3 failed to detect the* ***Gratuity*** *logical operator**error.*

100% condition coverage means every condition in a decision point evaluates to **true** and **false** at least once. While decision coverage requires two tests for each branch, condition coverage requires three as shown in Table 6 and 7.

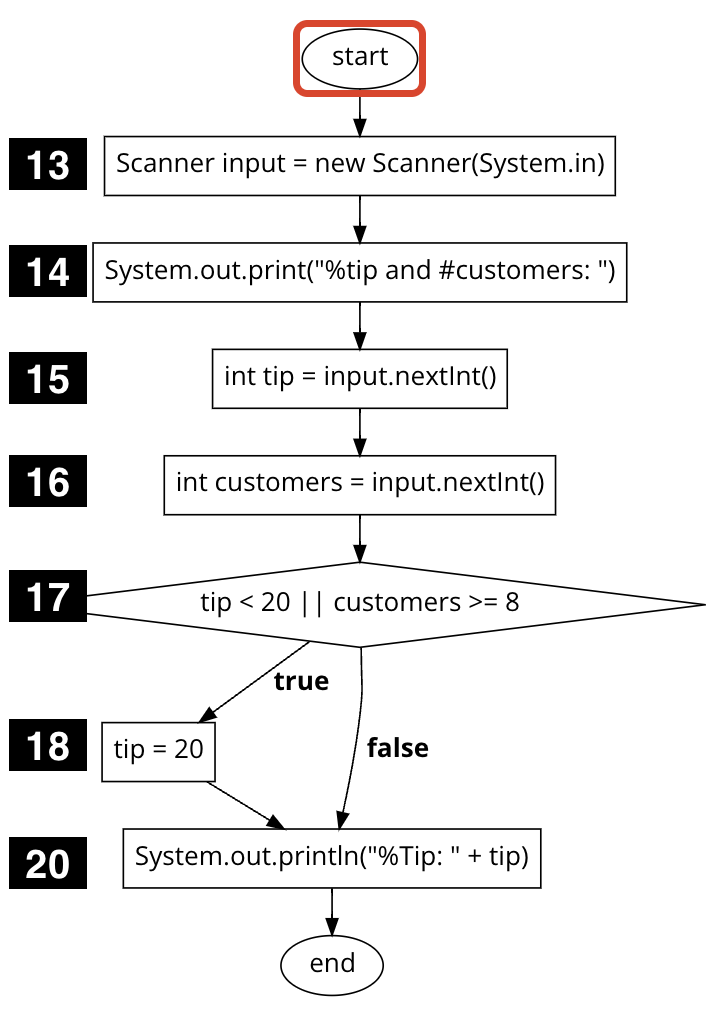
|  |  |  |
| --- | --- | --- |
| **Table 6: Condition Coverage for Logical OR** | | |
| **a** | **b** | **a || b** |
| true | - | true  short circuit |
| false | true | true |
| false | false | false |

|  |  |  |
| --- | --- | --- |
| **Table 7: Condition Coverage for Logical AND** | | |
| **a** | **b** | **a && b** |
| true | true | true |
| true | false | false |
| false | - | false  short circuit |

100% condition coverage requires an additional test case as shown in Table 8.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table 8: Gratuity Test Cases**  **100% Condition Coverage (tip < 20 || customers >= 8)** | | | | | |
|  | Expected I/O | Actual I/O | Status | tip < 20 | customers >= 8 |
| 1 | %Tip and #Customers: **10 12**  %Tip: 20 |  |  | true | unevaluated |
| 2 | %Tip and #Customers: **22 4**  %Tip: 22 |  |  | false | false |
| 3 | %Tip and #Customers: **25 10**  %Tip: 25 |  |  | false | true |

* Run the **Gratuity** program for the test cases in Table 8 and record the results.
* Confirm tests #1 and #2 pass, while test#3 fails.



Recall the rules in the Decision Table, which indicate the tip should only be altered when *both conditions* are true. The compound expression on line 17 fails to properly implement the rules.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Table 1: Tip Rate Decision Table** | | | | |
|  | **Rule 1** | **Rule 2** | **Rule 3** | **Rule 4** |
| **CONDITIONS** |  | | | |
| tip < 20 | T | T | F | F |
| customers >= 8 | T | F | T | F |
| **ACTIONS** |  | | | |
| tip | Adjust to 20% | - | - | - |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table 9: Gratuity Test Cases**  **75% Condition Coverage (tip < 20 && customers >= 8)** | | | | | |
|  | Expected I/O | Actual I/O | Status | tip < 20 | customers >= 8 |
| 1 | %Tip and #Customers: **10 12**  %Tip: 20 |  |  | true | true |
| 2 | %Tip and #Customers: **22 4**  %Tip: 22 |  |  | false | unevaluated |
| 3 | %Tip and #Customers: **25 10**  %Tip: 25 |  |  | false | unevaluated |

* Update **Gratuity** to fix the logical operator error.
* Run the program for each test in Table 9 and record the results. Verify that all tests passed.

We’ve identified and fixed the error by replacing || with &&, but Table 9 shows tests #2 and #3 now short-circuiting, meaning the second condition never evaluates as false.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table 10: Gratuity Test Cases**  **100% Condition Coverage (tip < 20 && customers >= 8)** | | | | | |
|  | Expected I/O | Actual I/O | Status | tip < 20 | customers >= 8 |
| 1 | %Tip and #Customers: **10 12**  %Tip: 20 |  |  | true | true |
| 2 | %Tip and #Customers: **22 4**  %Tip: 22 |  |  | false | unevaluated |
| 3 | %Tip and #Customers: **25 10**  %Tip: 25 |  |  | false | unevaluated |
|  |  |  |  |  |  |

* Add a new test in Table 10 to attain 100% condition coverage. Use Table 7 as a guideline for logical &&. Note that only one of test #2 or #3 is required for 100% condition coverage for &&.
* Run the program for each test in Table 10 and record the results. Verify that all tests passed.
* Insert a screen print of your code solution that shows the console result from the new test.

### Testing Task #2 – Equivalence of && and nesting

Consider the following program requirements:

*Write a program that reads in how much cash you have and whether you’re hungry. The program recommends ordering a pizza if you have at least $20 and are hungry, otherwise just keep studying.*

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 11: Pizza/Study Decision Table** | | | |
|  | **Rule 1** | **Rule 2** | **Rule 3** |
| **CONDITIONS** |  | | |
| cash >= 20 | T | F | - |
| isHungry | T | - | F |
| **ACTIONS** |  | | |
| Suggest | Order Pizza | Study | Study |

We’ll explore two ways to implement the requirements to gain a clearer understanding of short-circuit Boolean expression evaluation.

The logical && (AND) operator can be considered equivalent to a nested conditional in terms of control flow. Both evaluate the second condition only if the first condition is true. The two conditional statements in Table 11 are equivalent.

* PizzaStudyAnd.java uses && to test the two conditions
* PizzaStudyNested uses nesting to test the two conditions

|  |  |
| --- | --- |
| **Table 11: Equivalence between logical && and nested if/else** | |
| [**https://onlinegdb.com/uYJysqQV7**](https://onlinegdb.com/uYJysqQV7)  **PizzaStudyAnd.java** | [**https://onlinegdb.com/Q-Wi6hYcM**](https://onlinegdb.com/Q-Wi6hYcM)  **PizzaStudyNested.java** |
| if (cash >= 20 && isHungry) {  System.out.println("Order Pizza");  } else {  System.out.println("Study");  } | if (cash >= 20) {  if (isHungry) {  System.out.println("Order Pizza");  } else {  System.out.println("Study");  }  } else {  System.out.println("Study");  } |
|  |  |
| **isHungry** is evaluated only if **cash >= 20** is true | **isHungry** is evaluated only if **cash >= 20** is true |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table 12: Test Cases**  **PizzaStudyAnd.java Condition Coverage** | | | | | |
|  | Expected I/O | cash >= 20 | isHungry | path |  |
| 1 | Cash: **30**  Hungry: **true**  Order Pizza |  |  |  |
| 2 | Cash: **25**  Hungry: **false**  Study |  |  |  |
| 3 | Cash: **18**  Hungry: **true**  Study |  |  |  |
| 4 | Cash: **10**  Hungry: **false**  Study |  |  |  |

* Fill in the condition columns with either **unevaluated**, **true**, or **false**. Fill in the execution path.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table 13: Test Cases**  **PizzaStudyNested.java Condition Coverage** | | | | | |
|  | Expected I/O | cash >= 20 | isHungry | path |  |
| 1 | Cash: **30**  Hungry: **true**  Order Pizza |  |  |  |
| 2 | Cash: **25**  Hungry: **false**  Study |  |  |  |
| 3 | Cash: **18**  Hungry: **true**  Study |  |  |  |
| 4 | Cash: **10**  Hungry: **false**  Study |  |  |  |

* Fill in the condition columns with either **unevaluated**, **true**, or **false**. Fill in the execution path.

### Testing Task #3 – Equivalence of || and chained conditional ladder

Consider the following program requirements:

*Write a program that reads in weather conditions and determines where to play. Play inside if it is raining or below freezing (<32 Fahrenheit), otherwise play outside.*

Table 13 shows a decision table that defines where to play based on outdoor weather conditions.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 14: Play Location Decision Table** | | | |
|  | **Rule 1** | **Rule 2** | **Rule 3** |
| **CONDITIONS** |  | | |
| isRaining | T | - | F |
| belowFreezing | - | T | F |
| **ACTIONS** |  | | |
| Play | Inside | Inside | Outside |

Once again, we’ll look at two ways to implement the requirements. A common programming structure that is based on nested if is the if/else if/.../else **chained conditional ladder**. It looks like:

if (condition1) {

statements

} else if (condition2) {

statements

} else if (condition3) {

statements

}

...

else {

statements

}

The conditional expressions are evaluated starting at the top with **condition1**. As soon as a true condition is found, the statements associated with it are executed, and the rest of the ladder is bypassed.

The logical || (OR) operator can be considered equivalent to a chained conditional ladder in terms of control flow. Both evaluate the second condition only if the first condition is false. The two conditional statements in Table 15 are equivalent.

* PlayOr.java uses || to test the two conditions
* PlayLadder.java uses a chained ladder to test the two conditions

|  |  |
| --- | --- |
| **Table 15: Equivalence between logical || and chained conditional ladder** | |
| <https://onlinegdb.com/QcxjdF8qN>  **PlayOr.java** | [**https://onlinegdb.com/7WudfWDuF**](https://onlinegdb.com/7WudfWDuF)  **PlayLadder.java** |
| if (isRaining || belowFreezing) {  System.out.println("Play Inside");  } else {  System.out.println("Play Outside");  } | if (isRaining) {  System.out.println("Play Inside");  } else if (belowFreezing) {  System.out.println("Play Inside");  } else {  System.out.println("Play Outside");  } |
|  |  |
| belowFreezing evaluated only if isRaining is false | belowFreezing evaluated only if isRaining is false |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table 16: Test Cases**  **PlayOr.java Condition Coverage** | | | | | |
|  | Expected I/O | isRaining | belowFreezing | path |  |
| 1 | Raining: **true**  Temperature: **30**  Play Inside |  |  |  |
| 2 | Raining: **true**  Temperature: **50**  Play Inside |  |  |  |
| 3 | Raining: **false**  Temperature: **24**  Play Inside |  |  |  |
| 4 | Raining: **false**  Temperature: **48**  Play Outside |  |  |  |

* Fill in the condition columns with either **unevaluated**, **true**, or **false**. Fill in the execution path.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table 17: Test Cases**  **PlayLadder.java Condition Coverage** | | | | | |
|  | Expected I/O | isRaining | belowFreezing | path |  |
| 1 | Raining: **true**  Temperature: **30**  Play Inside |  |  |  |
| 2 | Raining: **true**  Temperature: **50**  Play Inside |  |  |  |
| 3 | Raining: **false**  Temperature: **24**  Play Inside |  |  |  |
| 4 | Raining: **false**  Temperature: **48**  Play Outside |  |  |  |

* Fill in the condition columns with either **unevaluated**, **true**, or **false**. Fill in the execution path.

### Testing Task #4 – Chapter 4 Project 1

Submit your Java project solution to Revel for auto-grading. The CodeGrade tests seem to evaluate only the first two digits following the decimal point. Your Actual I/O may differ slightly from the Sample Run, but your code should not attempt to round or truncate the area. CodeGrade *does no*t check invalid input (i.e. <3 sides, negative lengths) so you can ignore those cases and assume valid input.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Table 18: Chapter 4 Project 1 Test Cases** | | | | |
|  | Expected I/O | Actual I/O | Status | Comment |
| 1 | Enter the number of sides: **5**  Enter the side: **6.5**  The area of the polygon is 72.6903 |  |  | Sample Run |
| 2 |  |  |  | Square with side length 4 |
| 3 |  |  |  | Square with side length 6 |
| 4 |  |  |  | Triangle with side length 4 |
| 5 |  |  |  | Triangle with side length 6 |

* Edit Table 18 to fill in the Expected I/O for tests 2-5. You can search “area of equilateral triangle” to find an [online calculator](https://www.google.com/search?q=area+of+equilateral+triangle&oq=area+of+equi&gs_lcrp=EgZjaHJvbWUqDQgAEAAYkQIYgAQYigUyDQgAEAAYkQIYgAQYigUyBggBEEUYOTIHCAIQABiABDIHCAMQABiABDIHCAQQABiABDIHCAUQABiABDIHCAYQABiABDIHCAcQABiABDIHCAgQABiABDIHCAkQABiABNIBCDIyNTlqMGo3qAIIsAIB&sourceid=chrome&ie=UTF-8) and compute the expected area for test #4 and #5.
* Run your Project 1 solution for each test case in Table 18 and record the results.
* Try to fix errors identified by the tests. If you are unable to get a test case to pass, mention it in the lessons learned.
* Describe lessons learned while implementing Chapter 4 Project 1.

LESSONS LEARNED FOR PROJECT 1:

### Testing Task #5– Chapter 4 Project 5 Test

Submit your Java project solution to Revel for auto-grading.

The 11 characters in a Social Security Number have the format ddd-dd-dddd. Each character involves a binary condition (i.e. digit true/false, dash true/false). It is not feasible to test 211 combinations so we will test a subset of conditions including a string length check, a character digit check, and a character dash check.

Writing the correct code for even the first test in Table 19 can be challenging. It is best to use an incremental approach where you slowly build up the Boolean expression that checks for a valid SSN. The first program version checks just one condition, the second version checks two conditions, etc. Test each program version with input that results in valid/invalid based on that version’s notion of valid SSN.

PROPOSED PROGRAM VERSIONS:

1. A valid SSN has length 11.
2. A valid SSN has length 11 and the first character is a digit.
3. A valid SSN has length 11, the first character is a digit, and the fourth character is a dash.
4. Etc.

Once you think the entire valid SSN condition is working, test your complete program with the test cases in Table 19. The tests do not check every character position so it is possible that an error may go undetected, but the tests do exercise length, digit, and dash conditions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Table 19: Chapter 4 Project 5 Test Cases** | | | | |
|  | Expected I/O | Actual I/O | Status | Comment |
| 1 | Enter a SSN: **232-23-5435** 232-23-5435 is a valid social security number |  |  | Valid:  ddd-dd-dddd |
| 2 |  |  |  | Invalid length:  ddd |
| 3 |  |  |  | Invalid length:  ddd-dd-ddddd |
| 4 |  |  |  | 1st not digit:  ?dd-dd-dddd |
| 5 |  |  |  | 4th not dash:  ddd#dd-dddd |

* Edit Table 19 to fill in the Expected I/O for tests 3-6 based on the Comment column. You can choose any input that conforms to the pattern specified in the comment.
* Run your Project 5 solution for each test case in Table 19 and record the results.
* Try to fix errors identified by the tests. If you are unable to get a test case to pass, mention it in the lessons learned.
* Describe lessons learned while implementing Project 5.

LESSONS LEARNED FOR PROJECT 5: