**Guided LAB - 306.1.1 - JUNIT: Test Driven Development**

## **Learning Objective:**

In this lab, we will write a Java JUnit5 test program for a requirement to validate if the password entered matches the criteria needed.

**Project Example**: Passwords are used for every web application to validate a user. A strong password is desired to safeguard the authenticated user and prevent against hacks and easy guesses by unwanted users. Below are the minimum criteria to authenticate a password entered. Password entered must be:

* Criteria#1 - Length is greater or equal to 8.
* Criteria#2 - Length is less than or equal to 12.
* Criteria#3 - Has Alphanumeric characters.
* Criteria#4 - Has at least one special character.
* Criteria#5 - Does not already exist.
* Criteria#6 - Not already in use.
* Criteria#7 - Does not have “NOT ALLOWED” characters.

We will be writing a JUnit5 test case for some of the above criteria before actually validating the password.

We will write Java methods for each of the above seven criteria that will verify if the password is valid. For each of the methods, we will write JUnit test cases. The seven methods will be iteratively modified until the test cases pass, which is Test Driven Development.

## **Prerequisite:**

### **Add following SQL queries in MySQL workbench**

| CREATE SCHEMA `PERSCHOLAS` ; |
| --- |

### 

## **Prerequisite: JUnit5 libraries.**

For this lab, we will be adding Junit5 libraries to the pom.xml file as below:

### **Step 1: Setup New project**

### Create a new Maven project by using the Eclipse IDE or STS (Spring Tool Suite4) and add JUnit5 dependencies.

| <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.perscholas.javacasestudy</groupId>  <artifactId>junit5</artifactId>  <version>0.0.1-SNAPSHOT</version>  <name>junit5</name>  <properties>  <maven.compiler.target>11</maven.compiler.target>  <maven.compiler.source>11</maven.compiler.source>  <junit.jupiter.version>5.4.0</junit.jupiter.version>  </properties>  <dependencies> <dependency>  <groupId>org.junit.jupiter</groupId>  <artifactId>junit-jupiter-engine</artifactId>  <version>${junit.jupiter.version}</version>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.junit.jupiter</groupId>  <artifactId>junit-jupiter-api</artifactId>  <version>${junit.jupiter.version}</version>  <scope>test</scope>  </dependency> </dependencies> <build> <plugins>  <plugin>  <artifactId>maven-surefire-plugin</artifactId>  <version>2.22.1</version>  </plugin>  </plugins>  </build>  </project> |
| --- |

### **Step 2: Create a package named "junit.test"under the *src* folder.**

### **Step 3: Under junit.test package, create a class named PasswordComply and write a code in the class as shown below:**

| package model;  package junit.test;  import java.sql.CallableStatement;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.SQLException;  public class PasswordComply {    private String passwordString;  private final int minPasswordLength = 8;  private final int maxPasswordLength = 12;    static final String DB\_URL = "jdbc:mysql://localhost/PERSCHOLAS";  static final String USER = "root";  static final String PASS = "password";  static final String QUERY = "{call getEmpName (?, ?)}";    PasswordComply (String verifyPassword) {  passwordString = verifyPassword;  }    private boolean verifyPasswordLength() {    if(!passwordString.isEmpty()) {  if(passwordString.length() >= minPasswordLength && passwordString.length() <= maxPasswordLength) {  return true;  }  }  return false;  }    private boolean verifyAplhaNumeric() {    return true;  }    private boolean hasAllowedspecailCharacters() {    return true;  }    // This is a dummy method and needs to implement the real code to validate // password against database entries.  public boolean doesNotAlreadyExist() throws SQLException {    Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);  CallableStatement stmt = conn.prepareCall(QUERY);  stmt.setString(1, passwordString);  stmt.registerOutParameter(2, java.sql.Types.VARCHAR);  System.out.println("Executing stored procedure..." );  stmt.execute();  //Retrieve password  String existingPassword = stmt.getString(2);  System.out.println("Password already exist" + existingPassword);  return true;  }    private boolean hasNoSpecailCharacters() {    return true;  }    public void setPassword(String givenPassword) {  passwordString = givenPassword;  }    public boolean doesPasswordComply() {    return verifyPasswordLength() && verifyAplhaNumeric() && hasAllowedspecailCharacters() && hasNoSpecailCharacters();  }  } |
| --- |

* The above class is our main Password class that will validate the seven criterias. It has two variables/properties to keep things simple.

**IMPORTANT CONSIDERATIONS:**

* In this class (PasswordComply.java), we will be **hard-coding** the password, and we will change each test. This password can also be sent from another calling program.
* In Test Driven Development, we write methods for each criteria and write a JUnit test case, which will first fail. Once the criteria code is complete, iteratively the Junit test will pass.
* For the purpose of this lab, each of the methods is returning “true” to pass the JUnit test.
* **Change the return statement of each method from “false” to “fail” a JUnit test, and from “true” to “pass” a JUnit test.**
  + **Example :**

private boolean hasNoSpecailCharacters() {

return false ;

}

OR

private boolean hasNoSpecailCharacters() {

return true;

}

* PasswordComply class has the following methods:
  + verifyPasswordLength() method checks if the password is greater than 8 and less than 12. It returns ***true*** if the Criteria#1 and Criteria#2 is met.
  + verifyAplhaNumeric() method checks Criteria#3 if the password has Alphanumeric characters. For Test Driven Development, this method returns ***true*** for ***passing*** a Junit test, and ***changes*** the value to ***false*** for ***failing*** a Junit test. Iteratively a programmer is required to add the **real** code to this method.
  + hasAllowedspecailCharacters() method checks Criteria#4 if the password has at least one allowed special character. For Test Driven Development, this method returns ***true*** for ***passing*** a Junit test and ***changes*** the value to ***false*** for ***failing*** a Junit test. Iteratively, a programmer is required to add the **real** code to this method.
  + doesNotAlreadyExist() method checks Criteria#5 and Criteria#6 if the password already exists in a database or is already in use. For the purpose of JUnit TDD, we have some dummy code that programmers are required to add to the **real** code to this method, iteratively. We are returning ***true*** for ***passing*** a Junit test and ***change*** the value to ***false*** for ***failing*** a Junit test. Iteratively, a programmer is required to add the **real** code to this method.
  + hasNoSpecailCharacters method checks Criteria#7 if no special characters exist in the password. For Test Driven Development, this method returns ***true*** for ***passing*** a Junit test and ***changes*** the value to ***false*** for ***failing*** a Junit test. Iteratively, a programmer is required to add the **real** code to this method.
  + doesPasswordComply method return true if all the 7 criteria and met otherwise false.

### 

### **Step 4: Create a class named PasswordComplyTest under the “src/test/java” folder (used for JUnit test cases).**

In this class, we would write java code (JUnit test cases) to **test** the criteria methods of **PasswordComply** class.

Add the code below in the **"PasswordComplyTest"** class:

| package junit.test;  import static org.junit.jupiter.api.Assertions.\*;  import org.junit.jupiter.api.Test;  class PasswordComplyTest {  @Test  void testDoesPasswordComply() {  boolean expectedResult = true;  PasswordComply password = new PasswordComply("abcd1234");  boolean actualResult = password.doesPasswordComply();  assertEquals(expectedResult,actualResult, "Password conditions failed!");  System.out.println("Password conditions success!");  }    @Test  void testDoesPasswordExist() {  PasswordComply password = new PasswordComply("abcd1234");  assertThrows(java.sql.SQLException.class,() -> password.doesNotAlreadyExist(), "SQL Exception was thrown.");  }    @Test  void testEmptyPassword() {  PasswordComply password = new PasswordComply("Abcd1234!");  assertThrows(NullPointerException.class,() -> password.doesNotAlreadyExist(), "Null Exception was thrown but received SQL Exception.");  }  } |
| --- |

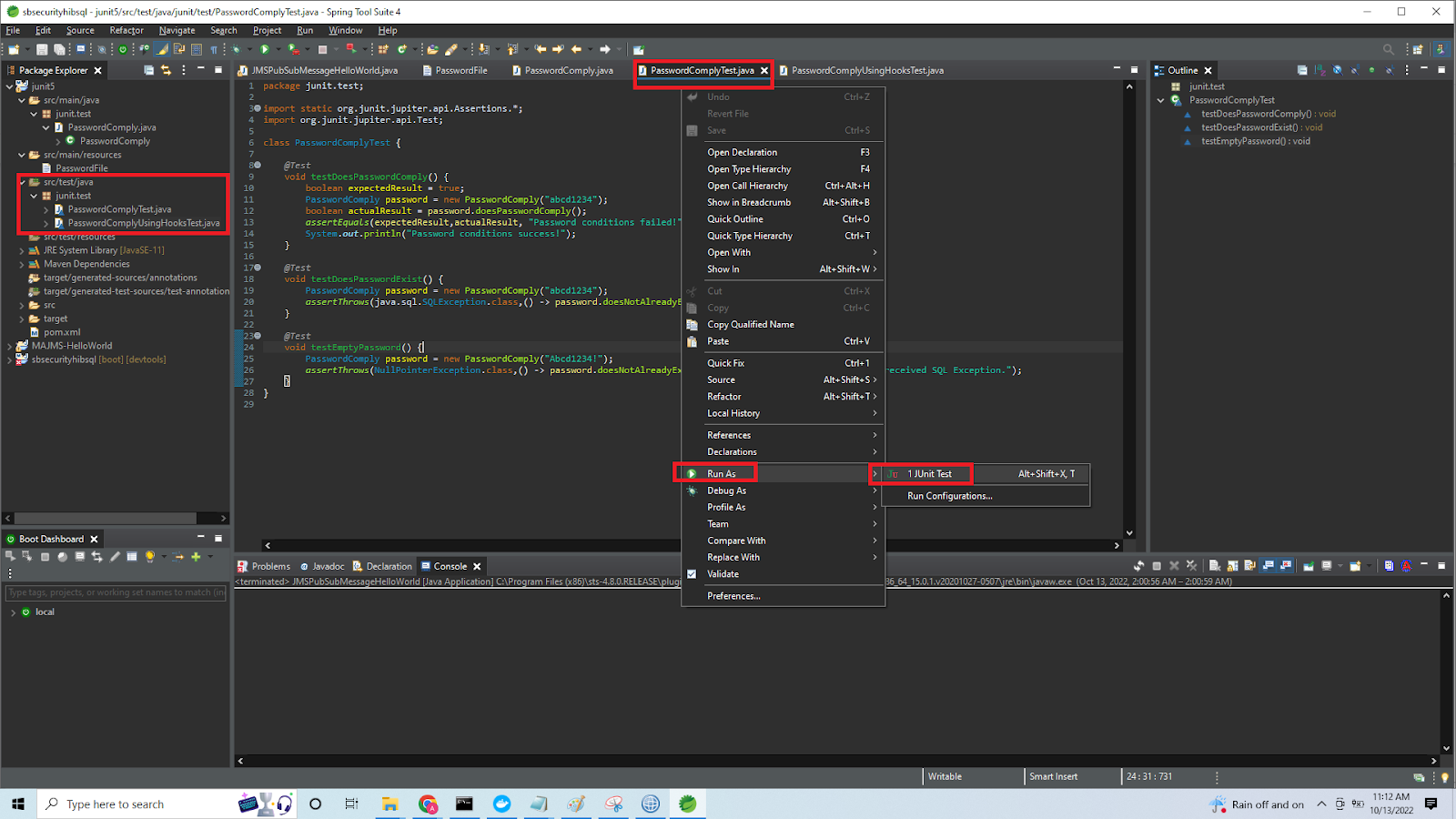
* In the above PasswordComplyTest, we create an instance of the PasswordComply and pass the password.
  + **Example :**

PasswordComply password = new PasswordComply("abcd1234");

PasswordComply password = new PasswordComply("Abcd1234!");

**Running Test Cases**:

* + **TEST CASE#1** - testDoesPasswordComply() method is a test case. It calls the doesPasswordComply() method of the PasswordComply class and asserts actual versus expected results. **Run (Right-Click on** PasswordComplyTest **file->Select “Run As”->Select “JUnit Test”, screenshot shown below)** this test case and change the return value of hasAllowedspecailCharacters method in PasswordComply class to “**true**” to pass a test and “**false**” to fail a test.



* + **TEST CASE#2** - testDoesPasswordExist() method is another test case. It calls the doesNotAlreadyExist() method of the PasswordComply class and throws a SQL exception for Database errors or invalid data. **Run (Right-Click on** PasswordComplyTest **file->Select “Run As”->Select “JUnit Test”)** this test case and change the return value of doesNotAlreadyExist method in PasswordComply class to “**true**” to pass a test and “**false**” to fail a test.
  + **TEST CASE#3** - testEmptyPassword() method is another test case. It calls the doesNotAlreadyExist() method of the PasswordComply class and throws a **Null** exception for Database errors or invalid data. **Run (Right-Click on** PasswordComplyTest **file->Select “Run As”->Select “JUnit Test”)** this test case and change the return value of doesNotAlreadyExist method in PasswordComply class to “**true**” to pass a test and “**false**” to fail a test.
* **Testing Different Use Cases -** 
  + Change the value passed in the PasswordComply constructor to test different use cases.
  + **Example:**

PasswordComply password = new PasswordComply("abcd12");

PasswordComply password = new PasswordComply("Abcd!");

### **Step 5: Create a class named PasswordComplyUsingHooksTest under “src/test/java” folder (which is used for JUnit test cases)**

In this class, we would write java code (JUnit test cases) to **test** the criteria methods of **PasswordComply** class.

Add the code below in the **"PasswordComplyUsingHooksTest"** class:

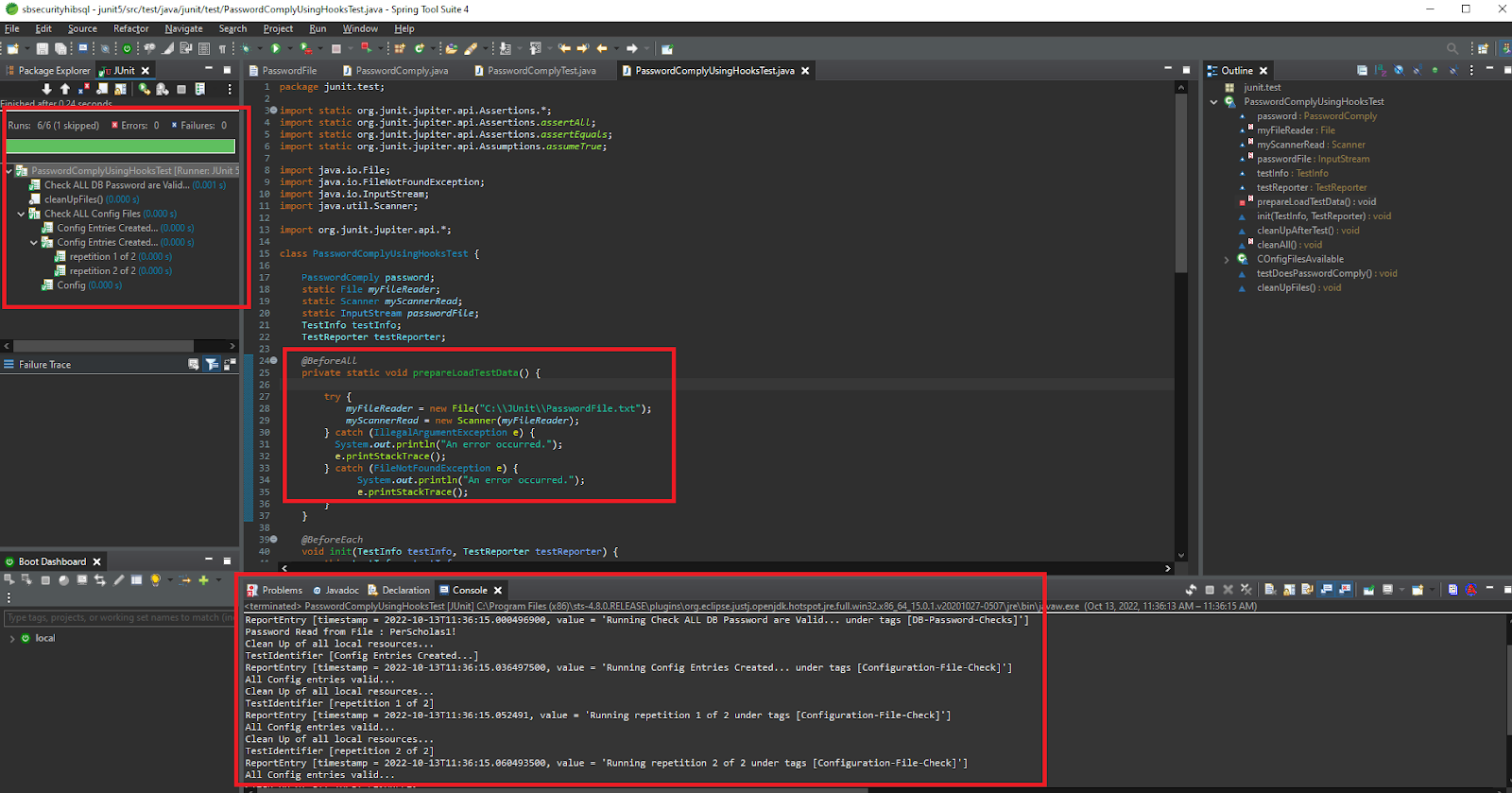
| import static org.junit.jupiter.api.Assertions.\*;  import static org.junit.jupiter.api.Assertions.assertAll;  import static org.junit.jupiter.api.Assertions.assertEquals;  import static org.junit.jupiter.api.Assumptions.assumeTrue;  import java.io.File;  import java.io.FileNotFoundException;  import java.io.InputStream;  import java.util.Scanner;  import org.junit.jupiter.api.\*;  class PasswordComplyUsingHooksTest {  PasswordComply password;  static File myFileReader;  static Scanner myScannerRead;  static InputStream passwordFile;  TestInfo testInfo;  TestReporter testReporter;    @BeforeAll  private static void prepareLoadTestData() {  try {  // Do not forget to change the location of file  myFileReader = new File("C:\\JUnit\\PasswordFile.txt");  myScannerRead = new Scanner(myFileReader);  } catch (IllegalArgumentException e) {  System.out.println("An error occurred.");  e.printStackTrace();  } catch (FileNotFoundException e) {  System.out.println("An error occurred.");  e.printStackTrace();  }  }    @BeforeEach  void init(TestInfo testInfo, TestReporter testReporter) {  this.testInfo = testInfo;  this.testReporter = testReporter;  testReporter.publishEntry("Running " + testInfo.getDisplayName() + " under tags " + testInfo.getTags());  password = new PasswordComply("");  }    @AfterEach  void cleanUpAfterTest() {  myScannerRead.close();  System.out.println("Clean Up of all local resources...");  }  @AfterAll  static void cleanAll() {  // File Pointers, DB Logs, Application Logs, Metrics Data Etc..  Runtime. getRuntime(). gc();  System.out.println("Clean up of Application Level DB data, Logs, resources etc.. done.");  }    @Nested  @Tag("Configuration-File-Check")  @DisplayName("Check ALL Config Files")  class COnfigFilesAvailable {    @Test  @DisplayName("Config")  void checkALLConfigFiles() {  System.out.println("All Config files created...");  }    @Test  @DisplayName("Config Entries Created...")  @RepeatedTest(2)  void checkALLConfigEntries() {  System.out.println("All Config entries valid...");  }  }  @Test  @Tag("DB-Password-Checks")  @DisplayName("Check ALL DB Password are Valid...")  void testDoesPasswordComply() {  assumeTrue((myFileReader != null));  boolean expectedResult = true;  while (myScannerRead.hasNextLine()) {  String passwordRead = myScannerRead.nextLine();  System.out.println("Password Read from File : " + passwordRead);  password.setPassword(passwordRead);  boolean actualResult = password.doesPasswordComply();  assertAll(  () -> assertEquals(expectedResult,actualResult, "Password conditions failed!")  );  }  }  @Disabled  @Test  void cleanUpFiles() {  myFileReader.delete();  System.out.println("Deleting all config, password file created.");  }  } |
| --- |

* The above PasswordComplyUsingHooksTest class uses ***different annotations and hooks*** provided by JUnit to run test cases.
* The above PasswordComplyUsingHooksTest class is an example of how enterprise applications test passwords of different components of the applications. The above code is used to write sample test cases to test these passwords, which are stored in the PasswordFile.txt

**IMPORTANT CONSIDERATIONS:**

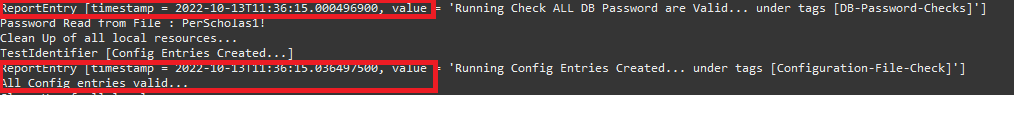
* Create a text file under the “**C:\\JUnit\\PasswordFile.txt**” folder to read the passwords.
  + myFileReader = new File("C:\\JUnit\\PasswordFile.txt");
* Add a password into the file ( “**C:\\JUnit\\PasswordFile.txt**”). Contents of the file should like:
  + PerScholas1!
* Adding Test Cases -
  + **TEST CASE#1** - prepareLoadTestData() method runs before any test case because of the annotation “**@BeforeAll.**” This method is a test case to ensure applications have valid passwords before running a load test. Method reads the password(s) from the C:\\JUnit\\PasswordFile.txt file. If there are no passwords present, it throws an exception and “**fails**” the test. **Run (Right-Click on** PasswordComplyTest **file->Select “Run As”->Select “JUnit Test,” screenshot shown below)** this test case, and the output should look like the screenshot below.

**Output:**



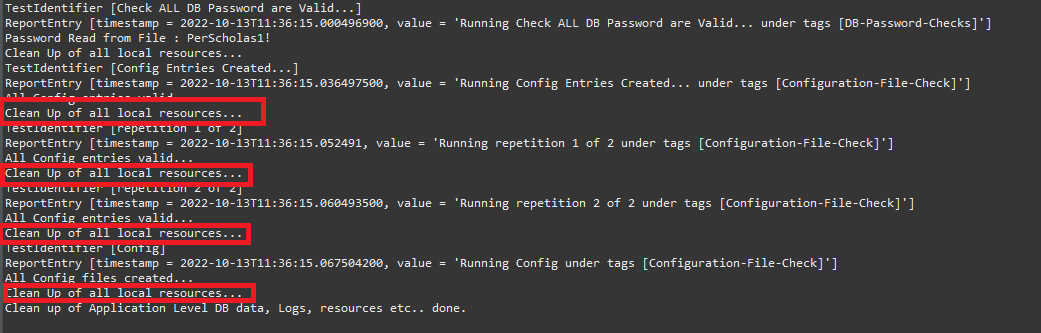
* + **TEST CASE#2** - init() method runs before any test case and initializes, and makes data preparation for running the test case by using the annotation **“@BeforeEach.**” This method is used to print important test run information for later review in “Test Runners” by calling method “testReporter.publishEntry().” **Run (Right-Click on** **PasswordComplyUsingHooksTest** **file →Select “Run As” → Select “JUnit Test”)** this test case.

**Output:**



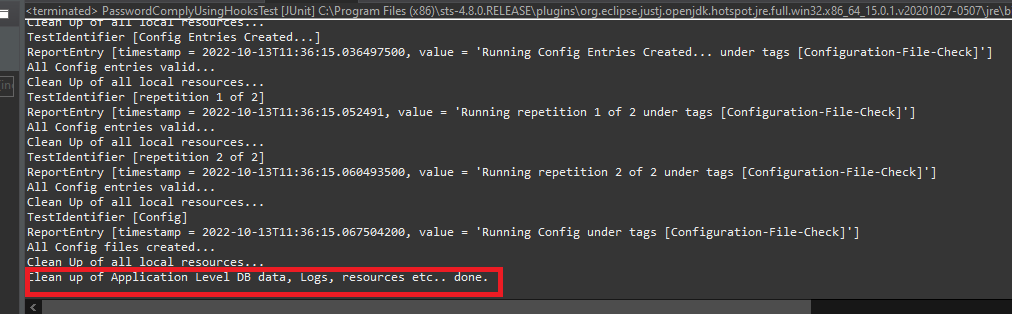
* + **TEST CASE#3** - cleanUpAfterTest() method runs after every test case and is used for clean-ups like closing resources, files, connections, sessions, etc. It is called by using the annotation **“@AfterEach.**” **Run (Right-Click on** PasswordComplyUsingHooksTest **file→ Select “Run As” →Select “JUnit Test”)** this test case.

**Output:**



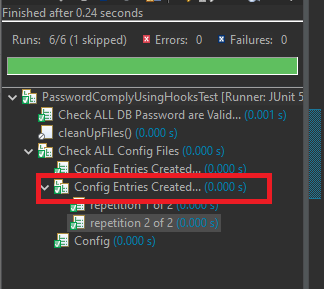
* + **TEST CASE#4** - cleanAll() method runs after all test cases are completed and the last method has been run. It is called for gc (garbage collector) and other clean-up purposes like clearing cache and clearing metrics data, etc. in enterprise applications. It is called by using the annotation **“@AfterAll.**” **Run (Right-Click on** PasswordComplyUsingHooksTest **file->Select “Run As”->Select “JUnit Test”)** this test case.

**Output:**



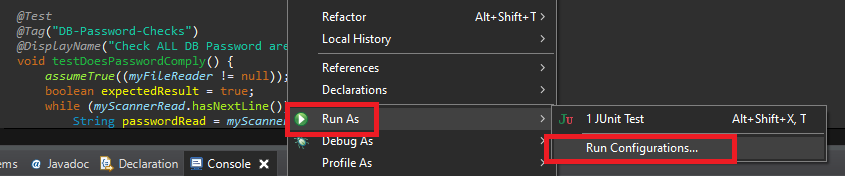
* + **TEST CASE#5** - ConfigFilesAvailable() method runs a test case to check if config files of application components are available. It calls **“@Nested**” annotation to indicate its call within a test. **Run (Right-Click on** PasswordComplyUsingHooksTest **file->Select “Run As”->Select “JUnit Test”)** this test case.

Output:



* + **TEST CASE#6** - COnfigFilesAvailable() method runs a test case based on a tag. It calls the **“@Tag**” annotation to run the specific “**tagged”** method @Tag("Configuration-File-Check"), and not all. **Run (Right-Click on** PasswordComplyUsingHooksTest **file->Select “Run As”->Select “Run Configurations”->Click “Configure”-> Check “Include Tags” -> type “Configuration-File-Check” in test box-> Click “OK”)** this test case.

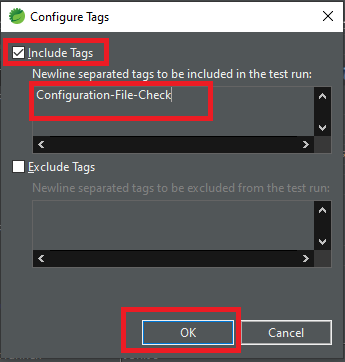
Step#1 :



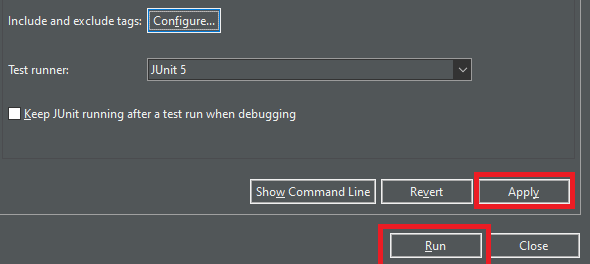
Step#2 : Click Configure:



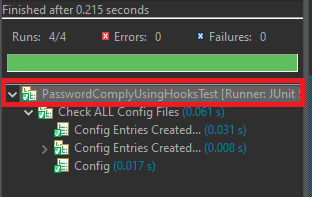
Step#3 : Check “Include Tags.” and type **“Configuration-File-Check.”** Click “OK.”



Step#4 : Click “Apply,” and then Click “Run.”

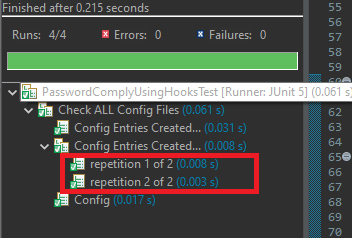


Output: It runs those methods that have annotation “@Tag("Configuration-File-Check").”



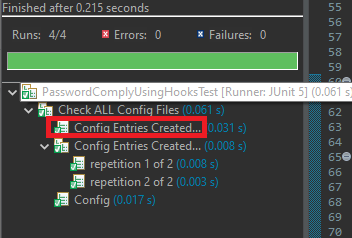
* + **TEST CASE#7** - checkALLConfigEntries() method in nested class COnfigFilesAvailable runs a test repeatedly based on the number of times passed (Dependency Injection Pattern). It calls **“@RepeatedTest(2)**” annotation to run specific times. **Run (Right-Click on** PasswordComplyUsingHooksTest **file->Select “Run As”->Select “Run Configurations”->Click “Configure”-> Check “Include Tags” -> type “Configuration-File-Check” in test box-> Click “OK”)** this test case.

Output:



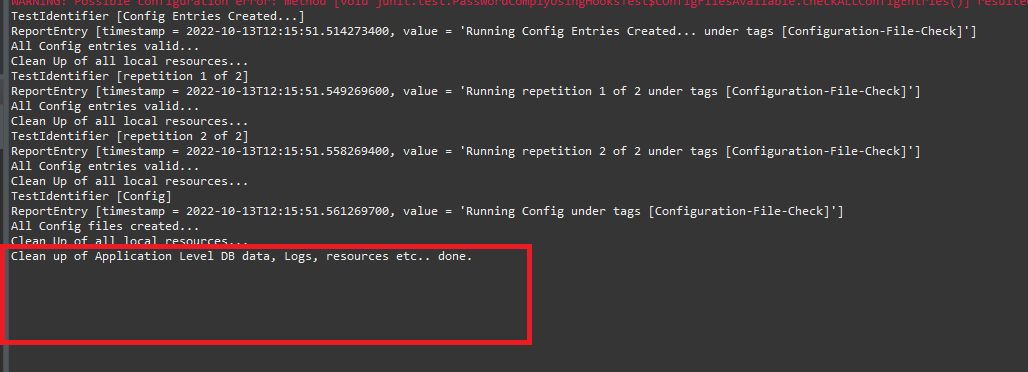
* + **TEST CASE#6** - testDoesPasswordComply() method reads the file for a password and checks if they comply. It uses the **“@DisplayName”** annotation to name and identify a specific test run ex: @DisplayName("Config Entries Created..."). **Run (Right-Click on** PasswordComplyUsingHooksTest **file->Select “Run As”->Select “Run Configurations”->Click “Configure”-> Check “Include Tags” -> type “Configuration-File-Check” in test box-> Click “OK”)** this test case.

Output:



* + **TEST CASE#7** - cleanUpFiles() method is never called. Use cases include calling or not calling methods based on test/staging/production environments. It uses the **“@Disabled”** annotation to *not* run a test case. **Run (Right-Click on** PasswordComplyUsingHooksTest **file->Select “Run As”->Select “Run Configurations”->Click “Configure”-> Check “Include Tags” -> type “Configuration-File-Check” in test box-> Click “OK”)** this test case.

Output:



Note: The System.out.println ("Deleting all config, password file created."); statement is never executed and there is no output to the console.

[**Click here to download the Full Project**](https://drive.google.com/file/d/1YVJO_qOB3xfGQ-lZqE2UzATu1OpQC8qT/view?usp=share_link)

### 

### **Conclusion:**

In this lab, we discussed Test Driven Development and how to create JUnit5 test cases.

**Submission Instructions:**

Include the following deliverables in your submission:

* + Submit your source code or screenshots using the Start Assignment button in the top-right corner of the assignment page in Canvas.