JUnit

Table of Contents

1. Introduction to JUnit Testing

1.1. Overview of JUnit

JUnit is a popular unit testing framework in the Java programming environment. Developed by Kent Beck and Erich Gamma, it has become a standard tool for implementing unit tests in Java projects. JUnit provides annotations to identify test methods, assertions to test expected results, and test runners for running tests.

1.2. Introduction to JUnit 5 (Jupiter)

JUnit 5, also known as Jupiter, is the next generation of the JUnit framework, introducing many new features and improvements over JUnit 4. It is designed to be more flexible and modular, making it easier to write and maintain tests. JUnit 5 is composed of three main subprojects:

JUnit Platform: Launches testing frameworks on the JVM. **JUnit Jupiter:** Provides new programming and extension models for writing tests and extensions. **JUnit Vintage:** Provides a test engine for running JUnit 3 and JUnit 4 tests on the JUnit 5 platform.

Key Features of JUnit 5

- **Annotations:** JUnit 5 introduces several new annotations, such as `@Nested`, `@DisplayName`, `@Tag`, and `@ExtendWith`, allowing for more descriptive tests and custom extensions.

- **Dynamic Tests:** Tests can be dynamically generated at runtime, offering more flexibility in test design.
- **Parameterized Tests:** Support for parameterized tests has been significantly improved, allowing for more robust data-driven testing.
- **Improved Assertions:** JUnit 5 includes a new Assertions class with better support for asserting conditions in tests.
- **Modularity:** The modular architecture of JUnit 5 makes it easier to integrate with other tools and frameworks.

Getting Started with JUnit 5

To start using JUnit 5 in your Java projects, you need to include the JUnit Jupiter API and the JUnit Platform Runner in your project's dependencies. For Maven projects, add the following dependencies to your `pom.xml`:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
</dependency>
```

For Gradle projects, include:

```
testImplementation 'org.junit.jupiter:junit-jupiter-api:5.7.0'
testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.7.0'
```

Ensure you have the JUnit 5 setup in your IDE to start writing and running JUnit 5 tests.

---

2. Setting Up JUnit with JavaFX

2.1 Overview of JavaFX

JavaFX is a rich client platform for building cross-platform desktop applications in Java. It offers a wide range of functionalities, including 2D and 3D graphics, UI controls, multimedia, and web views. JavaFX applications are known for their high performance and customizable interfaces.

2.2 Setting Up a JavaFX Project

1. **Create a New JavaFX Project:** Use your IDE to create a new JavaFX project. Specify the project SDK (Java Development Kit).

2. **Add JavaFX Libraries:** Ensure JavaFX libraries are included in your project's dependencies. For Maven and Gradle projects, add JavaFX dependencies in your `pom.xml` or `build.gradle` file, respectively.

For Maven, add:

```
<dependency>
  <groupId>org.openjfx</groupId>
  <artifactId>javafx-controls</artifactId>
  <version>11</version>
</dependency>
```

For Gradle, add:

```
implementation 'org.openjfx:javafx-controls:11'
```

3. **Configure the Module-Info.java:** If you're using modules, ensure your module-info.java file correctly requires the necessary JavaFX modules. For example:

```
module your.module.name {
  requires javafx.controls;
  exports your.main.package;
}
```

2.3 Integrating JUnit 5 into a JavaFX Project

With your JavaFX project set up, the next step is to integrate JUnit 5, enabling you to write and run tests.

1. **Add JUnit 5 Dependencies:** Just as in setting up a regular Java project with JUnit 5, add the JUnit Jupiter API and Engine dependencies to your project. Use the dependency snippets provided in Part 1, adjusting the version numbers as necessary to match the latest releases.

2. **Configure Your IDE:** Most modern IDEs support JUnit 5 out of the box. Ensure that your project is configured to use JUnit 5 for testing. This usually involves setting the test framework in your project's testing settings to JUnit 5.

3. **Write a Simple Test:** To verify that JUnit 5 is correctly set up with your JavaFX application, write a simple test case. For instance, you might create a test class for a simple JavaFX controller.

Example test class:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertTrue;

public class SimpleTest {
    @Test
    void simpleAssertion() {
        assertTrue(true, "The assertion should pass");
    }
}
```

4. **Run Your Test:** Use your IDE's test runner to run the test. Ensure the test passes, indicating that JUnit 5 is correctly set up and ready for more complex test scenarios involving your JavaFX application components.

3. Writing Your First JUnit Test

Understanding the structure and components of a JUnit test is crucial for any developer. This section outlines the basics of writing unit tests using JUnit 5, focusing on simple yet essential test cases related to a hypothetical login page in a JavaFX application.

3.1 Anatomy of a JUnit Test

JUnit tests are methods annotated with @Test and are contained within test classes. Here's a breakdown of the typical components in a JUnit test class:

- **Test Methods:** Methods annotated with `@Test` that contain the actual test code.
- **Assertions:** Statements that check if the test conditions are met.
- **Setup and Teardown Methods:** Methods annotated with `@BeforeEach`and `@AfterEach` (or `@BeforeAll` and `@AfterAll`) for setting up test preconditions and cleaning up after tests.
- **Test Class:** Contains test methods and possibly `setup/teardown` methods. Usually named after the class it tests (e.g., `LoginControllerTest` for testing `LoginController`).

3.2 Implementation

JavaFX Application Structure Your JavaFX application will consist of three **primary scenes**:

> - `Login Page`
> - `Registration Page`
> - `Todo Page`

Each page is represented by an FXML file for the layout and a Controller class to handle the user interactions. This separation adheres to the Model-View-Controller (MVC) pattern, enhancing maintainability and testability.

Focusing on a detailed implementation of the **Registration Page** in a JavaFX application, we'll step through creating the **UI** with **FXML**, building the controller, and setting up a comprehensive testing strategy for the registration logic.

Step 1: Designing the Registration Page UI with FXML

`Registration.fxml`

Create an FXML file named `Registration.fxml`. This file will define the user interface for the registration page, including input fields for the `user's name`, `email`, `password`, and a `registration button`.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.GridPane?>
```

```
<?import javafx.scene.text.Text?>

<GridPane fx:controller="your.package.RegistrationController"
          xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10"
vgap="10">
    <padding><Insets top="25" right="25" bottom="10" left="25"/></padding>

    <Text text="Register" GridPane.columnIndex="0" GridPane.rowIndex="0"
GridPane.columnSpan="2"/>
    <Label text="Name:" GridPane.columnIndex="0" GridPane.rowIndex="1"/>
    <TextField fx:id="nameField" GridPane.columnIndex="1" GridPane.rowIndex="1"/>
    <Label text="Email:" GridPane.columnIndex="0" GridPane.rowIndex="2"/>
    <TextField fx:id="emailField" GridPane.columnIndex="1" GridPane.rowIndex="2"/>
    <Label text="Password:" GridPane.columnIndex="0" GridPane.rowIndex="3"/>
    <PasswordField fx:id="passwordField" GridPane.columnIndex="1"
GridPane.rowIndex="3"/>
    <Button text="Register" onAction="#handleRegistrationAction"
GridPane.columnIndex="1" GridPane.rowIndex="4"/>
</GridPane>
```

Step 2: Implementing the RegistrationController The RegistrationController class will handle user input and registration logic.

`RegistrationController.java`

```java
package your.package;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.PasswordField;
import javafx.scene.control.TextField;

public class RegistrationController {

    @FXML
    private TextField nameField;

    @FXML
    private TextField emailField;

    @FXML
    private PasswordField passwordField;

    @FXML
    protected void handleRegistrationAction(ActionEvent event) {
        String name = nameField.getText();
        String email = emailField.getText();
        String password = passwordField.getText();

        if (registerUser(name, email, password)) {
            // Registration successful
            // Navigate to login page or show success message
        } else {
            // Registration failed
            // Show error message
        }
    }

    private boolean registerUser(String name, String email, String password) {
        // Placeholder for registration logic
        // In a real application, this would involve validating the input and
storing the user data
        return !name.isEmpty() && !email.isEmpty() && password.length() >= 8;
    }
}
```

Step 3: Writing Test Cases for the Registration Logic Now, let's focus on how to test the registration logic using JUnit 5. The registerUser method checks that none of the fields are empty and that the password is at least 8 characters long.

3.2.1 Unit Testing registerUser Method

`RegistrationControllerTest.java`

We'll create a test class named RegistrationControllerTest. This class will contain test methods to verify the registration logic under various conditions.

```
// package your.package;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class RegistrationControllerTest {

    private RegistrationController controller;

    @BeforeEach
    void setUp() {
        controller = new RegistrationController();
    }

    @Test
    void testRegisterUserWithValidData() {
        assertTrue(controller.registerUser("John Doe", "john@example.com",
"password123"),
                   "Registration should succeed with valid data.");
    }

    @Test
    void testRegisterUserWithEmptyName() {
        assertFalse(controller.registerUser("", "john@example.com",
"password123"),
                    "Registration should fail with an empty name.");
    }

    @Test
    void testRegisterUserWithShortPassword() {
        assertFalse(controller.registerUser("John Doe", "john@example.com",
"pass"),
                    "Registration should fail with a password shorter than 8
characters.");
    }

    // Additional tests can be added to cover more cases, such as invalid email
formats.
}
```

Parameterized Testing For more comprehensive testing, especially to cover various input combinations efficiently, you can use JUnit 5's parameterized tests. Here's how you might extend the testing to cover multiple scenarios using @ParameterizedTest.

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

class RegistrationControllerTest {

    private RegistrationController controller = new RegistrationController();

    @ParameterizedTest
    @CsvSource({
        "John Doe, john@example.com, password123, true",
        ", john@example.com, password123, false",
        "John Doe, , password123, false",
        "John Doe, john@example.com, pass, false"
    })
    void testRegisterUser(String name, String email, String password, boolean
expectedOutcome) {
        assertEquals(expectedOutcome, controller.registerUser(name, email,
password),
                     "Registration validation failed.");
    }
}
```

Step 4: Designing the LoginPage UI with FXML

`Login.fxml`

Create an FXML file named `Login.fxml` to define the user interface for the login page, including input fields for the username and password, and a login button.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.text.Text?>

<GridPane fx:controller="your.package.LoginController"
          xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10"
vgap="10">
    <padding><Insets top="25" right="25" bottom="10" left="25"/></padding>

    <Text text="Login" GridPane.columnIndex="0" GridPane.rowIndex="0"
GridPane.columnSpan="2"/>
    <Label text="Username:" GridPane.columnIndex="0" GridPane.rowIndex="1"/>
    <TextField fx:id="usernameField" GridPane.columnIndex="1"
GridPane.rowIndex="1"/>
    <Label text="Password:" GridPane.columnIndex="0" GridPane.rowIndex="2"/>
    <PasswordField fx:id="passwordField" GridPane.columnIndex="1"
GridPane.rowIndex="2"/>
    <Button text="Login" onAction="#handleLoginAction" GridPane.columnIndex="1"
GridPane.rowIndex="3"/>
</GridPane>
```

Step 5: Implementing the LoginController The LoginController class will handle user input for login actions.

`LoginController.java`

```
// package your.package;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.PasswordField;
import javafx.scene.control.TextField;

public class LoginController {

    @FXML
    private TextField usernameField;

    @FXML
    private PasswordField passwordField;

    @FXML
    protected void handleLoginAction(ActionEvent event) {
        String username = usernameField.getText();
        String password = passwordField.getText();

        if (authenticateUser(username, password)) {
            // Login successful
            // Navigate to the Todo Page or show success message
        } else {
            // Login failed
            // Show error message
        }
    }

    private boolean authenticateUser(String username, String password) {
        // Placeholder for authentication logic
        // In a real application, this would check the credentials against a user
store (e.g., database)
        return "admin".equals(username) && "password".equals(password);
    }
}
```

Step 3: Writing Test Cases for the Login Logic Testing the authenticateUser method is crucial to ensure that only valid users can access the application.

Unit Testing authenticateUser Method LoginControllerTest.java

Create a test class named LoginControllerTest for verifying the login functionality.

```
package your.package;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class LoginControllerTest {

    private LoginController controller;

    @BeforeEach
    void setUp() {
        controller = new LoginController();
    }

    @Test
    void testAuthenticateUserWithValidCredentials() {
        assertTrue(controller.authenticateUser("admin", "password"),
                   "Authentication should succeed with valid credentials.");
    }

    @Test
    void testAuthenticateUserWithInvalidUsername() {
        assertFalse(controller.authenticateUser("wrongUser", "password"),
                    "Authentication should fail with an invalid username.");
    }

    @Test
    void testAuthenticateUserWithInvalidPassword() {
        assertFalse(controller.authenticateUser("admin", "wrongPassword"),
                    "Authentication should fail with an invalid password.");
    }
}
```

Parameterized Testing for Login Leverage parameterized tests to cover a broader range of input scenarios for the login logic efficiently.

```java
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

class LoginControllerTest {

    private LoginController controller = new LoginController();

    @ParameterizedTest
    @CsvSource({
        "admin, password, true",
        "admin, wrongPassword, false",
        "wrongUser, password, false",
        "admin, , false",
        ", password, false"
    })
    void testAuthenticateUser(String username, String password, boolean
expectedOutcome) {
        assertEquals(expectedOutcome, controller.authenticateUser(username,
password),
                     "Authentication validation failed.");
    }
}
```

Moving forward to the Todo Page in our JavaFX application, we'll illustrate how to design the UI with FXML, implement the functionality with a controller, and discuss testing strategies for the ***CRUD*** operations of todo items using JUnit 5. The Todo Page allows users to add, view, update, and delete todo items.

Designing the Todo Page UI with FXML `Todo.fxml`

Create an FXML file named Todo.fxml to define the user interface for the Todo Page. This interface includes a ListView to display todo items, a TextField to enter new todos, and buttons for adding and deleting todos.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.collections.FXCollections?>
<?import javafx.collections.ObservableList?>
<?import javafx.geometry.Insets?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.text.Text?>

<VBox fx:controller="your.package.TodoController"
      xmlns:fx="http://javafx.com/fxml" spacing="10">
```

```
    <padding><Insets top="20" right="20" bottom="10" left="20"/></padding>

    <Text text="Your Todos"/>
    <ListView fx:id="todoListView" prefHeight="200"/>
    <HBox spacing="10">
        <TextField fx:id="todoInputField" HBox.hgrow="ALWAYS"/>
        <Button text="Add" onAction="#handleAddTodoAction"/>
        <Button text="Delete" onAction="#handleDeleteTodoAction"/>
    </HBox>
</VBox>
```

Implementing the TodoController The TodoController class manages the interaction logic for the Todo Page, including adding and deleting todo items.

`TodoController.java`

```
// package your.package;

import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXML;
import javafx.scene.control.ListView;
import javafx.scene.control.TextField;

public class TodoController {

    @FXML
    private ListView<String> todoListView;

    @FXML
    private TextField todoInputField;

    private ObservableList<String> todoItems;

    @FXML
    public void initialize() {
        todoItems = FXCollections.observableArrayList();
        todoListView.setItems(todoItems);
    }

    @FXML
    protected void handleAddTodoAction() {
        String newTodo = todoInputField.getText().trim();
        if (!newTodo.isEmpty()) {
            todoItems.add(newTodo);
            todoInputField.clear();
        }
    }

    @FXML
    protected void handleDeleteTodoAction() {
        String selectedTodo = todoListView.getSelectionModel().getSelectedItem();
        if (selectedTodo != null) {
            todoItems.remove(selectedTodo);
        }
    }
}
```

Writing Test Cases for Todo Operations Testing the Todo Page functionality involves verifying that todo items can be added and deleted as expected. However, testing UI controllers directly in JUnit can be complex due to the need for initializing JavaFX components. Here, we focus on testing the logic behind adding and deleting todos, assuming these methods are made accessible (e.g., package-private or public for testing) or refactored into a separate testable class.

Unit Testing Todo Operations

`TodoControllerTest.java`

Create a test class named TodoControllerTest. This class will contain methods to test adding and deleting todo items.

```
package your.package;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class TodoControllerTest {

    private TodoController controller;

    @BeforeEach
    void setUp() {
        controller = new TodoController();
        controller.initialize(); // Manually initialize to setup the todoItems
list
    }

    @Test
    void testAddTodoItem() {
        controller.todoInputField.setText("New Todo");
        controller.handleAddTodoAction();
        assertFalse(controller.todoItems.isEmpty(), "Todo list should not be empty
after adding an item.");
    }

    @Test
    void testDeleteTodoItem() {
        // Setup - Add an item first
        controller.todoInputField.setText("Todo to Delete");
        controller.handleAddTodoAction();
        // Select the item to delete
        controller.todoListView.getSelectionModel().select(0);
        controller.handleDeleteTodoAction();
        assertTrue(controller.todoItems.isEmpty(), "Todo list should be empty
after deleting the item.");
    }
}
```

4. Advanced JUnit Testing Techniques

4.1 Testing with Assertions

Assertions are fundamental in JUnit tests; they validate the conditions that the test expects to be true. JUnit 5 introduces more powerful assertion methods compared to its predecessors, offering a wide range of options to test with precision.

> - **Basic Assertions:** Test simple conditions. Use `assertEquals`, `assertTrue`, `assertFalse`, and `assertNull`.

```
assertEquals(4, calculator.add(2, 2), "Optional failure message");
assertTrue('a' < 'b', () -> "Assertion messages can be lazily evaluated
-- to avoid constructing complex messages unnecessarily.");
```

> - **Grouped Assertions:** Execute a group of assertions together, reporting any failures collectively after all assertions are executed.

```
import static org.junit.jupiter.api.Assertions.assertAll;

assertAll("Multiple assertions",
    () -> assertEquals(4, calculator.multiply(2, 2)),
    () -> assertEquals(0, calculator.divide(1, 0), "Division by zero should result
in zero")
);
```

> - **Exception Assertions:** Test that your code throws an expected exception.

```
import static org.junit.jupiter.api.Assertions.assertThrows;

assertThrows(ArithmeticException.class, () -> calculator.divide(1, 0));
```

> - **Timeout Assertions:** Ensure that your code completes within a specified time.

```
import static org.junit.jupiter.api.Assertions.assertTimeout;
```

```
assertTimeout(Duration.ofMillis(100), () -> {
    // Perform task that should not take more than 100 ms
});
```

4.2 Grouping and Tagging Tests

Grouping and tagging tests in JUnit 5 allow you to categorize your tests logically, making it easier to manage and execute subsets of tests.

- **Using Tags:** You can tag your test methods with `@Tag` annotation. Tags are useful for filtering tests during execution.

```
import org.junit.jupiter.api.Tag;

@Tag("fast")
@Test
void aFastTest() {
    // This test is tagged as "fast"
}

@Tag("slow")
@Test
void aSlowTest() {
    // This test is tagged as "slow"
}
```

Filtering Tests: When running your tests, you can specify which tags to include or exclude. This capability is particularly useful in build tools like Maven and Gradle, or within IDEs, to run only a specific subset of tests.

4.3 Testing Exceptions

JUnit 5 provides the assertThrows method to assert that execution of a particular code snippet throws a specific exception.

> - **Basic Exception Testing:**

```
import static org.junit.jupiter.api.Assertions.assertThrows;

@Test
void whenDivideByZero_thenThrowArithmeticException() {
    Calculator calculator = new Calculator();
    assertThrows(ArithmeticException.class, () -> calculator.divide(1, 0));
}
```