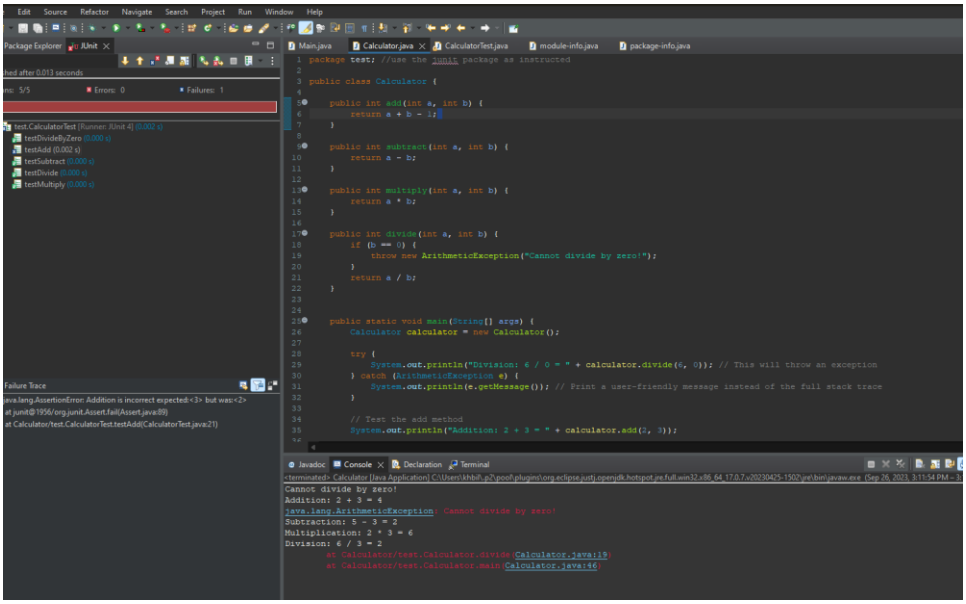
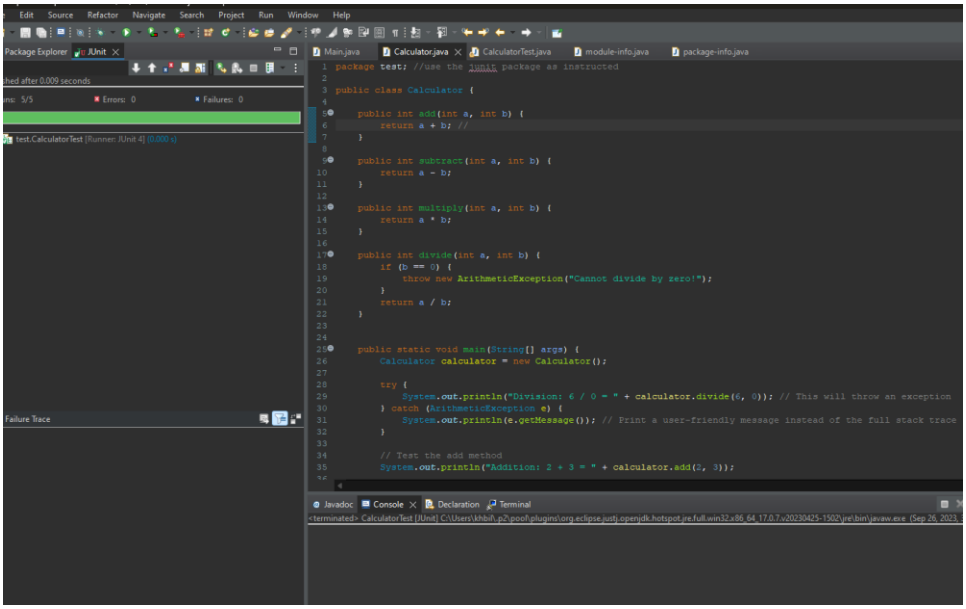
Topic 7

KariAnn Harjo

10/01/2023

Part 1: Basic Unit Tests





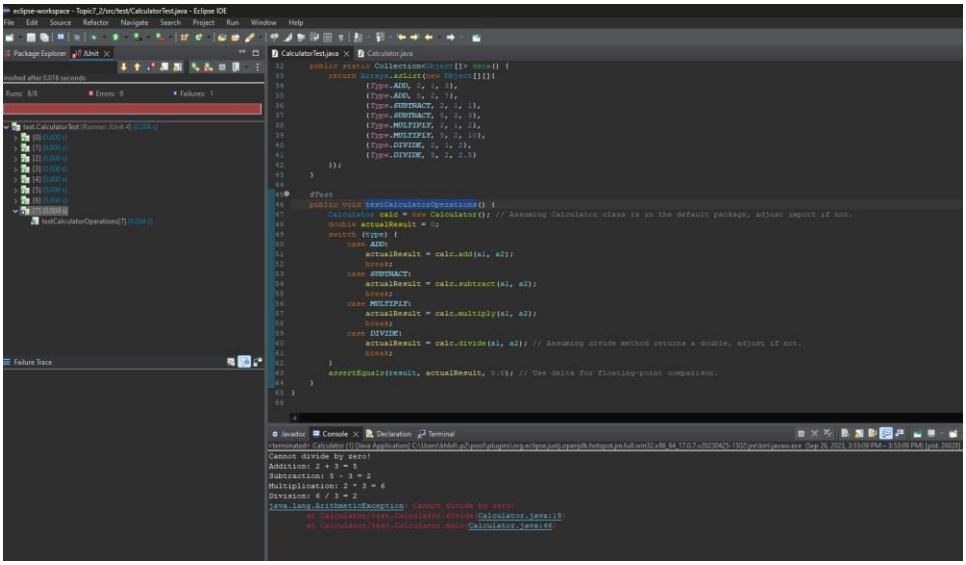
a. When developing applications like a Calculator, meticulous test case creation for each method is paramount. A minimum of one test case per method is essential, but thorough testing necessitates multiple cases per method to account for various inputs and error conditions. For the Calculator application, particular attention must be given to the divide method, ensuring graceful handling of division by zero through critical test cases to prevent application crash and ensure the return of appropriate messages or values.

b. White and Black Box Testing are foundational testing paradigms used to validate software functionality and structure. White Box Testing is concerned with testing the internal structures of an

application, necessitating knowledge of the application's internal workings, design, and implementation, typically performed at the unit testing level. Conversely, Black Box Testing is performed without knowledge of the internal structures, focusing solely on verifying that the application functions correctly according to requirements, often utilized at the system and acceptance testing levels.

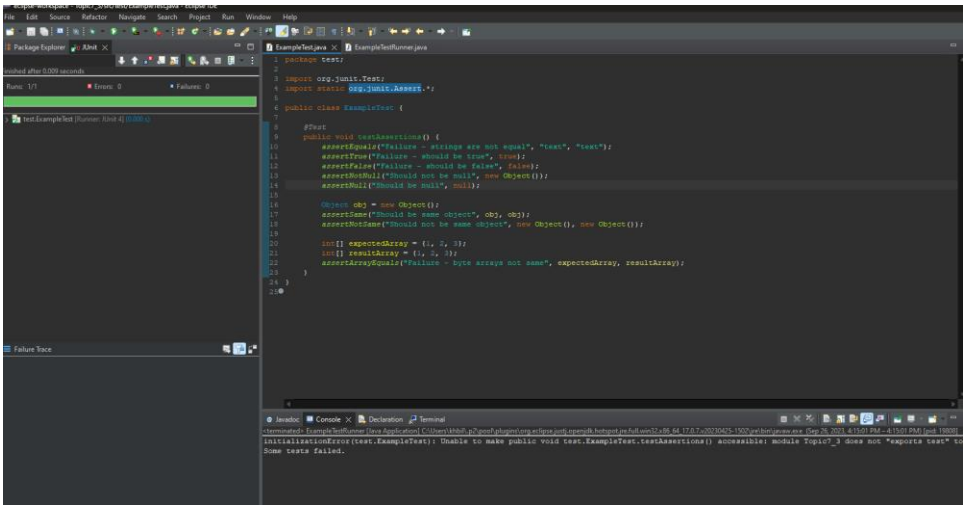
c. In software testing, a Test Suite is a collection of test cases intended to validate different aspects of software functionality. Utilizing a Test Suite is crucial when multiple test classes need to be grouped and automated, enabling efficient regression testing for each modification made to the software. Particularly in Continuous Integration/Continuous Deployment (CI/CD) environments, Test Suites ascertain the stability and reliability of the code before deployment or merging. They also play an essential role in organizing and managing tests when dealing with a large number of test cases, facilitating the delivery of reliable and high-quality software products.

Part 2: Parameterized Unit Tests



Utilizing Parameterized Tests in JUnit profoundly optimizes the software testing process by consolidating multiple test cases into a more concise and manageable form, significantly reducing redundancy and enhancing readability. This approach allows comprehensive and thorough testing of various input combinations and edge cases, improving the detection of potential issues and ensuring robust software behavior. The structured format of parameterized tests enables swift identification of errors, allowing developers to efficiently rectify specific input-related issues, ultimately saving time and enhancing the scalability and reliability of the test suite. This comprehensive approach bolsters confidence in the software's resilience and functionality, fostering the development of high-quality software products.

Part 3: Advanced Unit Tests



a.

In JUnit, testing whether a piece of code throws an expected exception can be achieved using the @Test(expected = ExceptionClass.class) annotation, which will make the test pass if the specified exception is thrown. In JUnit 5, a more versatile approach is the Assertions.assertThrows method, allowing not only the assertion that an exception of a specified type is thrown but also enabling further inspection of the thrown exception, facilitating more detailed and comprehensive exception testing and validation.

b.

Testing for all error conditions and exceptions is challenging due to the complexity and diversity of possible inputs, states, and execution paths, especially in intricate systems. The unpredictability and variability of external dependencies, services, and libraries further complicate exhaustive testing, making it intricate to foresee and test every conceivable exception or error scenario. This complexity necessitates prioritized, intelligent testing strategies to effectively manage resources and risk.

Part 4:

1.

JUnit and TestNG are both extensive testing frameworks in Java, used for facilitating high-quality software through rigorous testing.

Similarities:

1. Java-based: Both JUnit and TestNG are written in Java and primarily used for Java projects.

2. Annotation Use: They utilize annotations to signify test methods, making it easy to set up tests.

3. IDE Support: Both are supported and can be integrated with Eclipse and IntelliJ IDEA.

4. Build Tool Support: Maven and Gradle can integrate with both JUnit and TestNG for managing project builds.

5. Assertion Libraries: They provide powerful libraries for asserting conditions in the testing methods.

6. Test Execution Control: Both frameworks offer mechanisms to control test execution flow through various means.

7. Mocking and Stubbing: Integration with mocking frameworks like Mockito is possible with both.

8. Parallel Execution: They support the parallel execution of tests, reducing the overall test execution time.

9. Extensible: They can be extended through custom runners and listeners.

10. Documentation: Both have extensive and well-maintained documentation to assist developers in getting started with testing.

Differences:

1. Scope and Flexibility: JUnit is primarily designed for unit testing and follows a more rigid structure, whereas TestNG is more flexible, designed for a broader range of tests including unit, integration, and end-to-end.

2. Grouping and Dependencies: TestNG supports grouping of test methods and specifying method dependencies; JUnit does not support this natively.

3. Parameterization: TestNG supports parameterized testing through XML configurations or DataProviders; JUnit requires a more verbose approach using Parameterized Classes.

4. Test Execution Model: TestNG allows for executing test methods in parallel, enabling more efficient execution of tests; JUnit executes test methods in isolation by default.

5. Suite Test Configuration: TestNG allows for more extensive suite-wide setup configurations through XML; JUnit primarily relies on Annotations.

6. Reporting: TestNG has more extensive and flexible reporting capabilities out of the box compared to JUnit.

7. Annotations: TestNG has a wider range of Annotations, allowing for more detailed test configurations.

8. Data-driven Testing: TestNG has more built-in support for data-driven testing compared to JUnit.

9. Thread Safe: TestNG is more thread-safe compared to JUnit due to its inherent design for parallel execution.

10. Community and Popularity: JUnit has a larger community and is more popular, primarily due to its longevity in the market.

Resources:

https://junit.org/junit5/docs/current/user-guide/

https://testng.org/doc/documentation-main.html

2.

Code coverage is a metric used in software development to measure the extent to which the source code of a program is executed when a particular test suite runs. It is often expressed as a percentage, with higher percentages indicating more extensive test coverage. A code coverage tool can usually provide insights into which parts of the codebase are covered by tests and which are not, making it an invaluable tool in identifying areas of the code that could potentially harbor undetected bugs.

There are different types of code coverage, such as:

- Function Coverage : Measures whether each function or method in the codebase has been executed.

- Statement Coverage : Measures whether each statement in the codebase has been executed.

- Branch Coverage : Measures whether each branch (e.g., `if` and `else` paths) in the codebase has been executed.

- Condition Coverage : Measures whether each boolean expression has been evaluated to both `true` and `false`.

Code coverage is used in the development process to improve software quality and reliability. While a high code coverage percentage is generally beneficial, it does not guarantee the absence of software defects. It mainly indicates the extent to which the codebase has been tested, not the quality of the tests or whether they have been tested under all possible conditions. Developers and testers typically aim for high code coverage but also need to consider the depth and breadth of the tests and the various test scenarios and edge cases.

In summary, code coverage is a significant aspect of software testing, as it helps in identifying untested parts of the codebase, thus enabling developers to improve software quality and reliability by adding necessary tests and refining existing ones. However, it is crucial to approach this metric with a nuanced perspective, considering it as part of a wider testing strategy and focusing on writing meaningful tests, rather than merely striving for a high coverage percentage.

https://www.lambdatest.com/learning-hub/code-coverage