Owen Lindsey

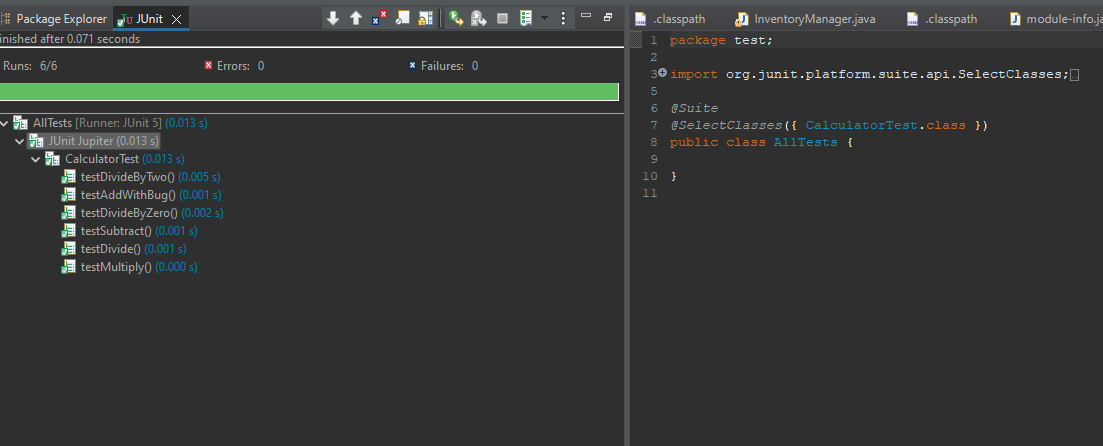
Professor Couch

Cst-239

Activity 7

Part 1:

Basic Unit Tests



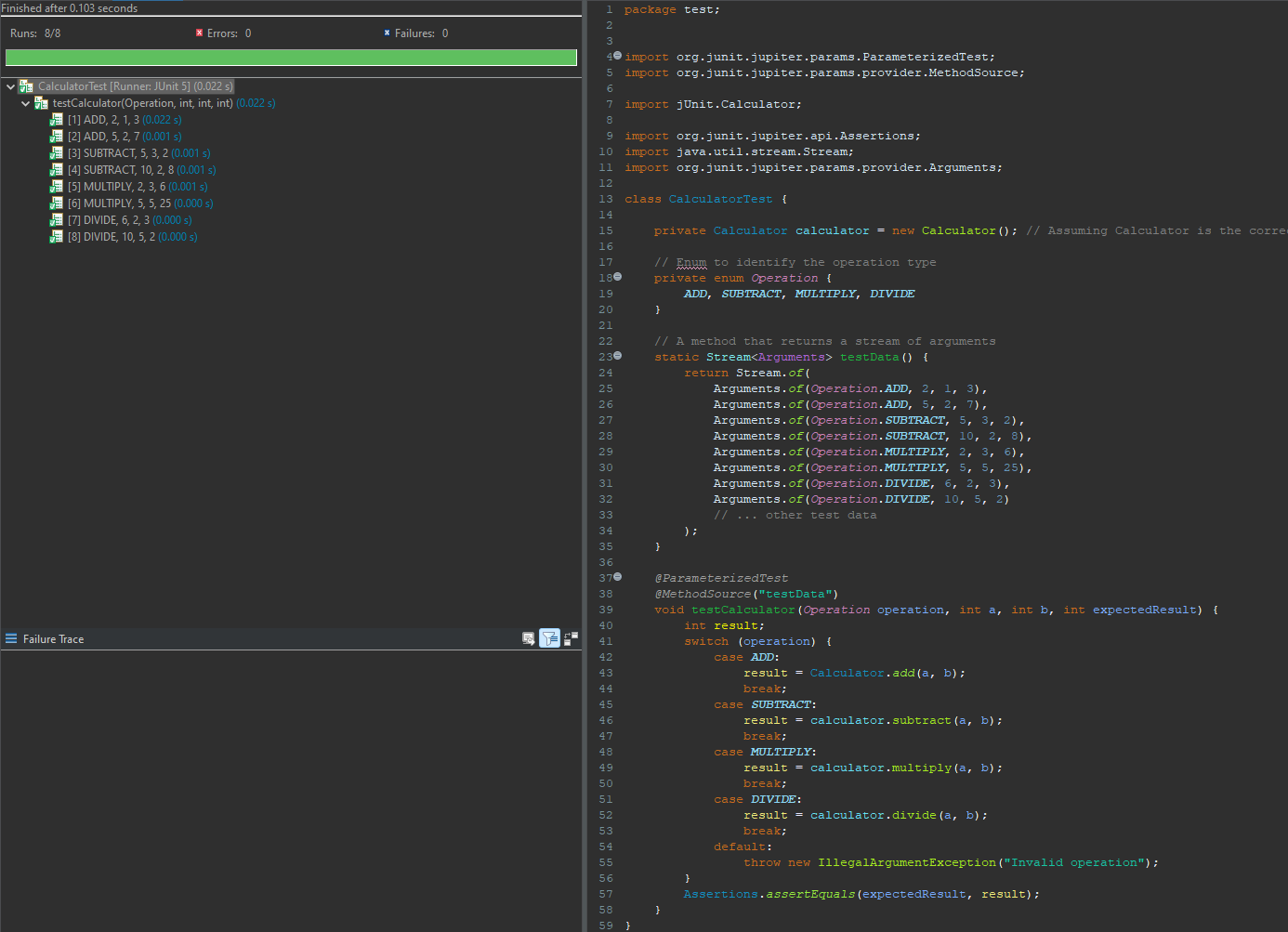
a. The number of test cases would depend on the complexity of the methods and the range of input values you expect. You would need to test edge cases, such as dividing by zero, to ensure the application can handle such scenarios gracefully.

b. White box testing is a method of testing the software's internal structures or workings. Black box testing involves testing the software's functionality without knowing the internal workings.

c. A Test Suite is useful when you have a collection of test cases that you want to execute together. This could be when you are grouping similar tests or conducting regression testing to ensure that new changes haven't broken existing functionality.

Part 2:

Parameterized Unit Tests

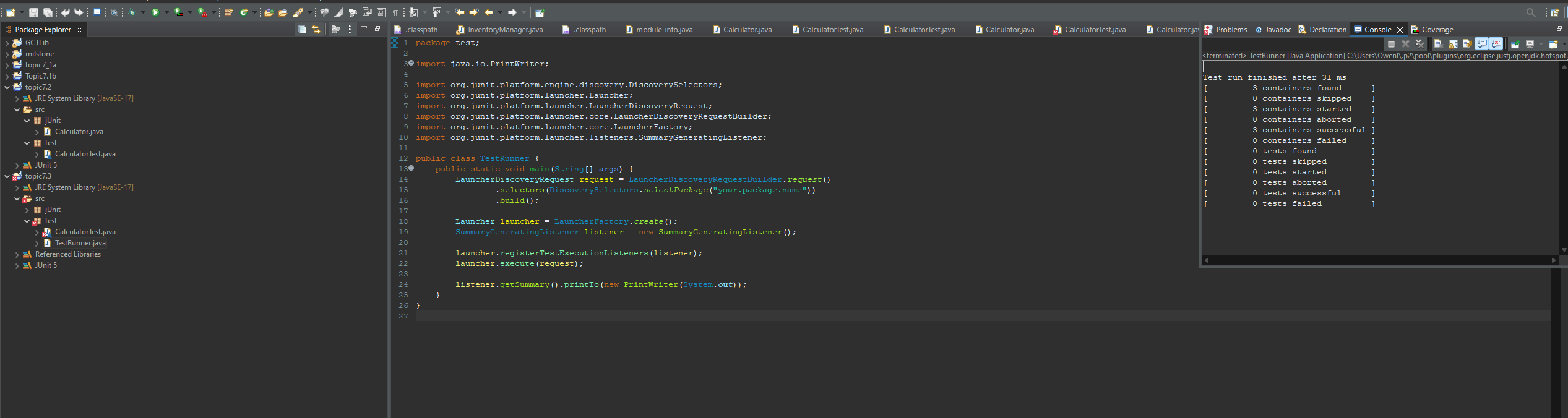


a.

Parameterized tests help to minimize code duplication by allowing a single test method to run with multiple sets of data, thereby upholding the DRY principle and simplifying code maintenance. They enhance test coverage by systematically checking function behavior across a vast range of input scenarios, including edge cases. This method ensures a thorough evaluation of functions without the need to write multiple test methods, making the tests more manageable and less error prone. By adding new parameter sets, developers can easily extend test cases, making it a time-efficient way to ensure software reliability. Overall, parameterized testing is an effective strategy for creating efficient, extensive, and maintainable test suites.

Part 3:

Advanced Unit Tests



a. How can you test that an exception is thrown in your code?

In JUnit, to test that an exception is thrown, you can use the assertThrows method, which is designed to assert that a particular type of exception is thrown during the execution of a block of code. Here's a basic example of its usage:

@Test

public void whenExceptionThrown\_thenAssertionSucceeds() {

Exception exception = assertThrows(NumberFormatException.class, () -> {

Integer.parseInt("One");

});

assertEquals("For input string: \"One\"", exception.getMessage());

}

Part 3 continued:

Advanced Unit Tests

b. What challenges can you think of that will make testing for all error conditions and exceptions in your code even possible?

The main challenges in testing for all error conditions and exceptions include:

Complexity: Complex systems with many components and interactions can produce a vast number of paths and states, making it difficult to cover all scenarios.

External Dependencies: When a system interacts with external services, databases, or APIs, it may encounter unpredictable behaviors or outages that are difficult to simulate in tests.

Edge Cases: Some error conditions are rare or involve edge cases that are easily overlooked during testing.

Resource Limitations: Constraints such as time, budget, and access to skilled testers can limit the extent of testing, especially in covering less likely error scenarios.

Concurrent Operations: Multi-threaded environments and race conditions can be challenging to replicate consistently in a test environment.

Changing Requirements: Evolving requirements can lead to a moving target for testing, where new features or changes can introduce unanticipated error conditions.

To address these challenges, robust test planning, continuous integration, use of test coverage tools, stress testing, and employing a combination of testing types (unit, integration, system, acceptance) are often required to increase the likelihood of catching as many error conditions and exceptions as possible.