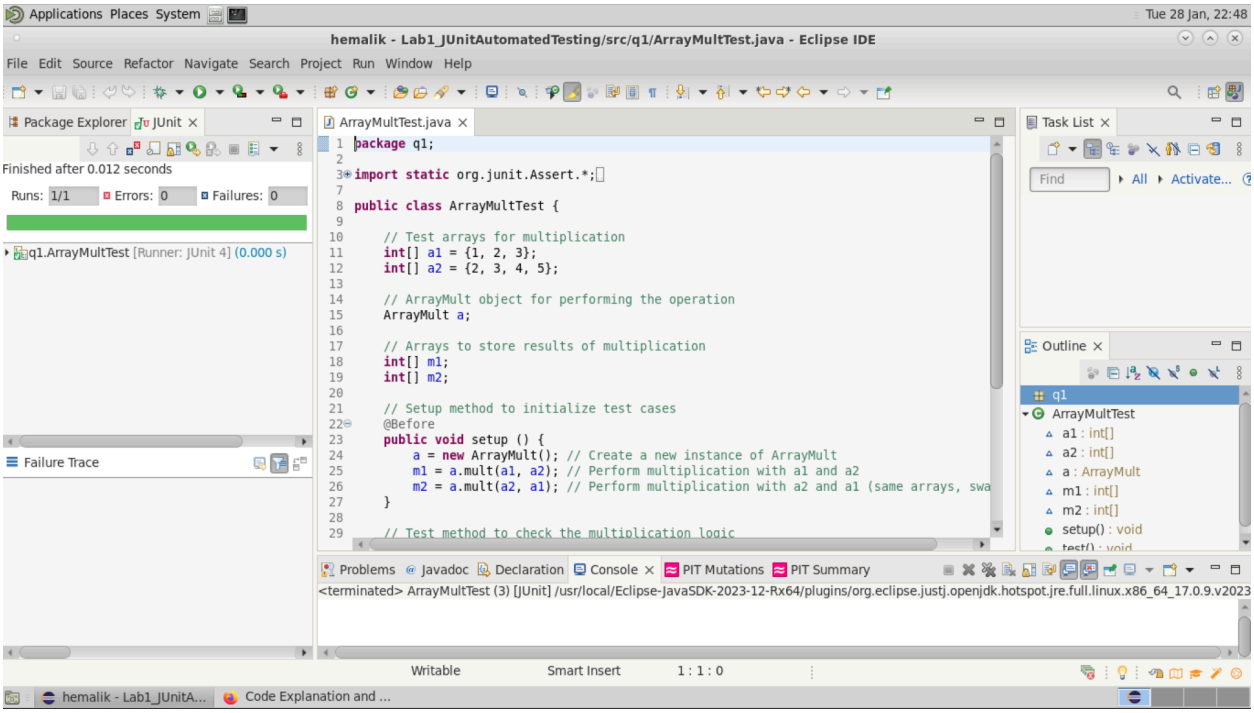
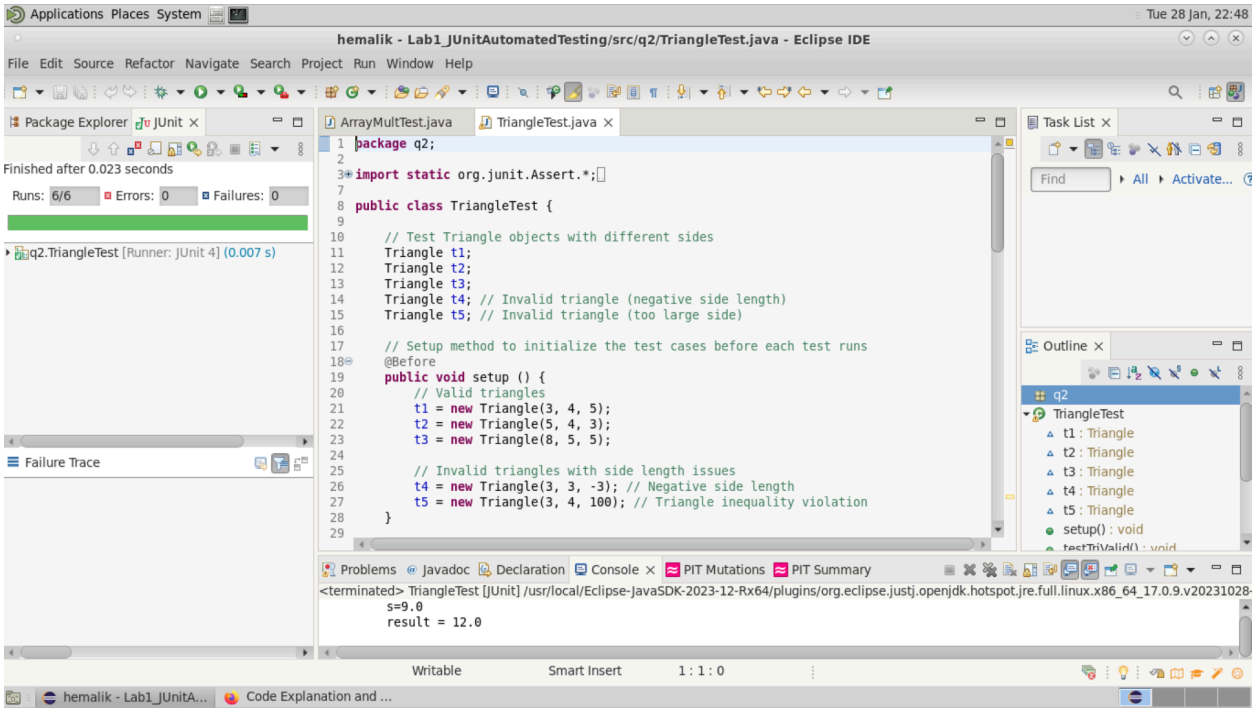
COE891 Lab I: JUnit: Automated Testing

Hamza Malik
501112545

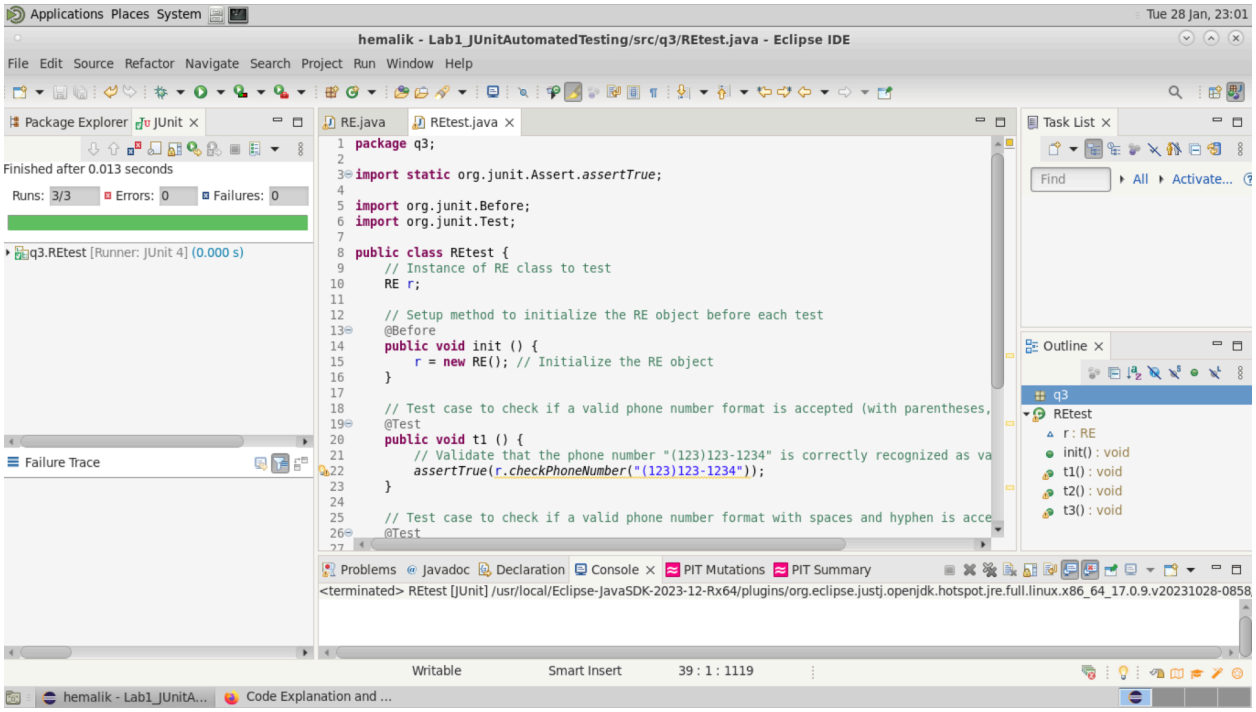
Jan 28, 2025



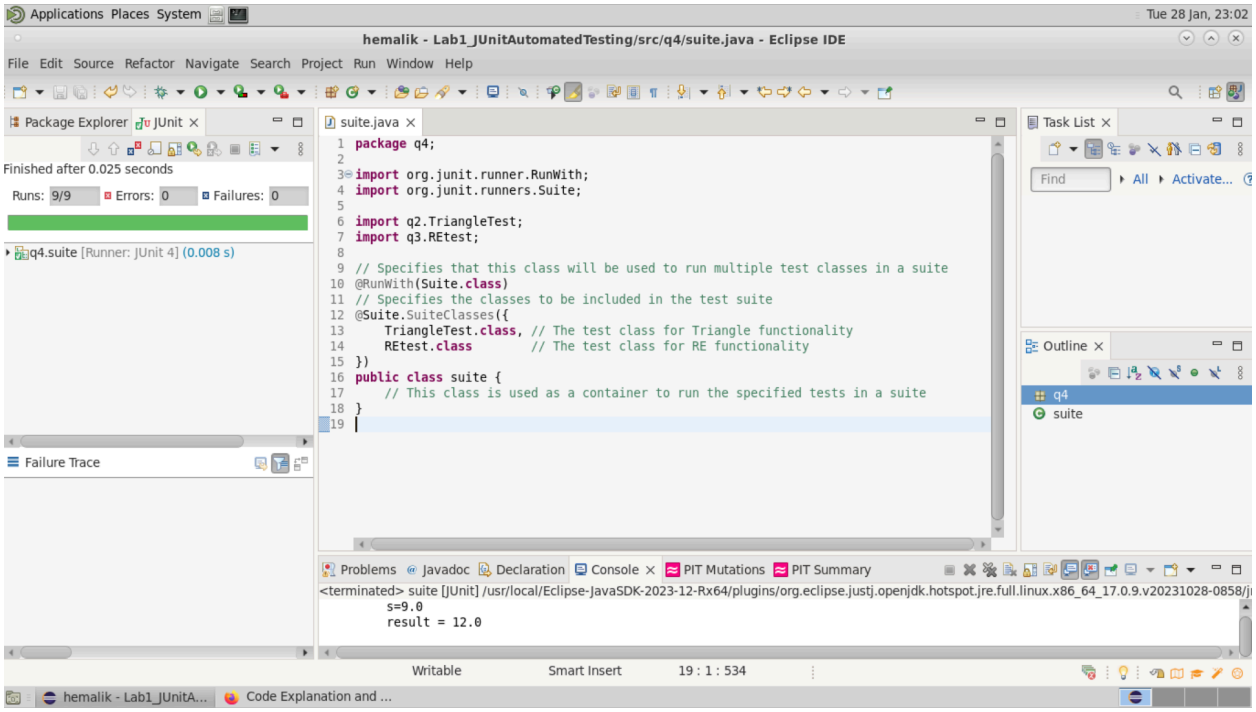
Q1:In this task, we implemented the mult(int[] array1, int[] array2) method, which performs element-wise multiplication of two integer arrays while accommodating different lengths. The resulting output array takes the length of the longer input array. Where both arrays have values at the same index, their corresponding elements are multiplied. If one array is shorter, any remaining elements from the longer array are directly copied into the output. A JUnit test class is used to verify the correctness of the ArrayMult implementation, ensuring it handles different input sizes and produces accurate results.



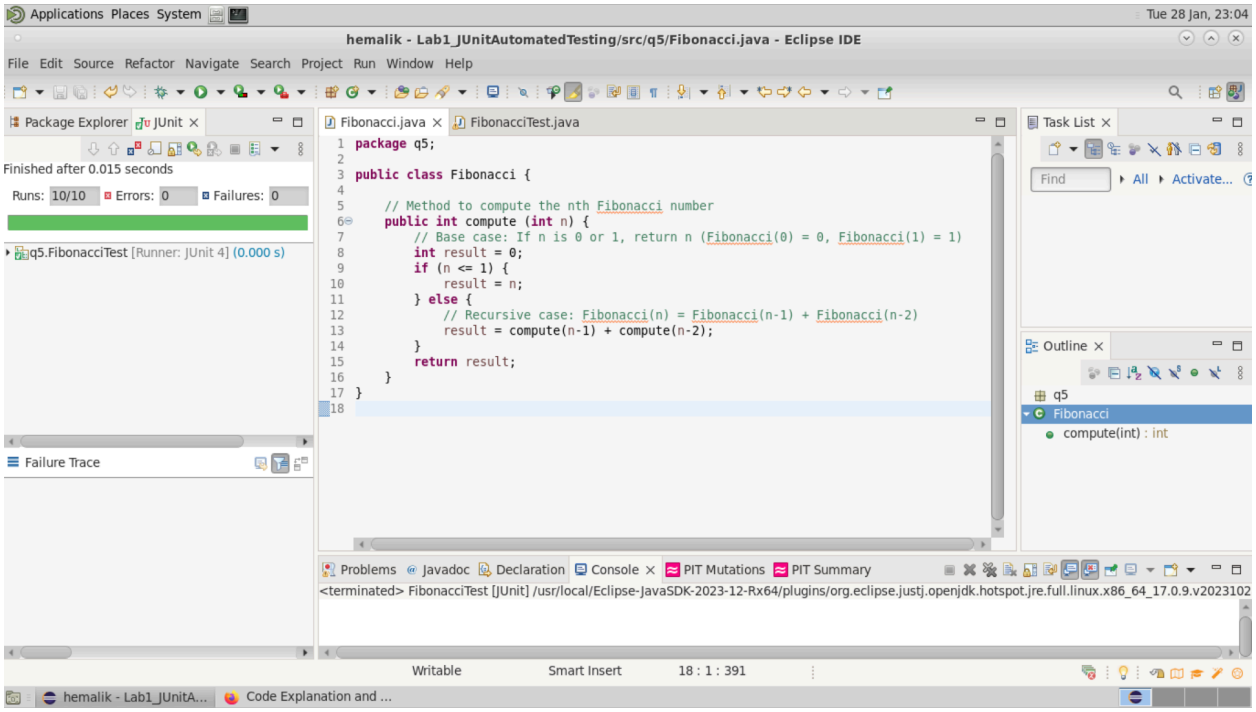
Q2: In this task, we were given a Triangle class with a calculateArea method that incorrectly implemented Heron's formula. Our first objective was to identify and fix the error in the mathematical equation. Once corrected, we were required to use JUnit testing to validate the implementation. This included verifying whether a triangle with sides (3, 4, 100) is valid, writing three test methods to compute the area of triangles t1, t2, and t3, and checking for both positive and negative test cases. Additionally, we tested whether the area of t1 was equal to t2 to ensure consistency in our calculations. The JUnit tests provided comprehensive validation for the corrected calculateArea method and ensured its accuracy in handling various triangle inputs.



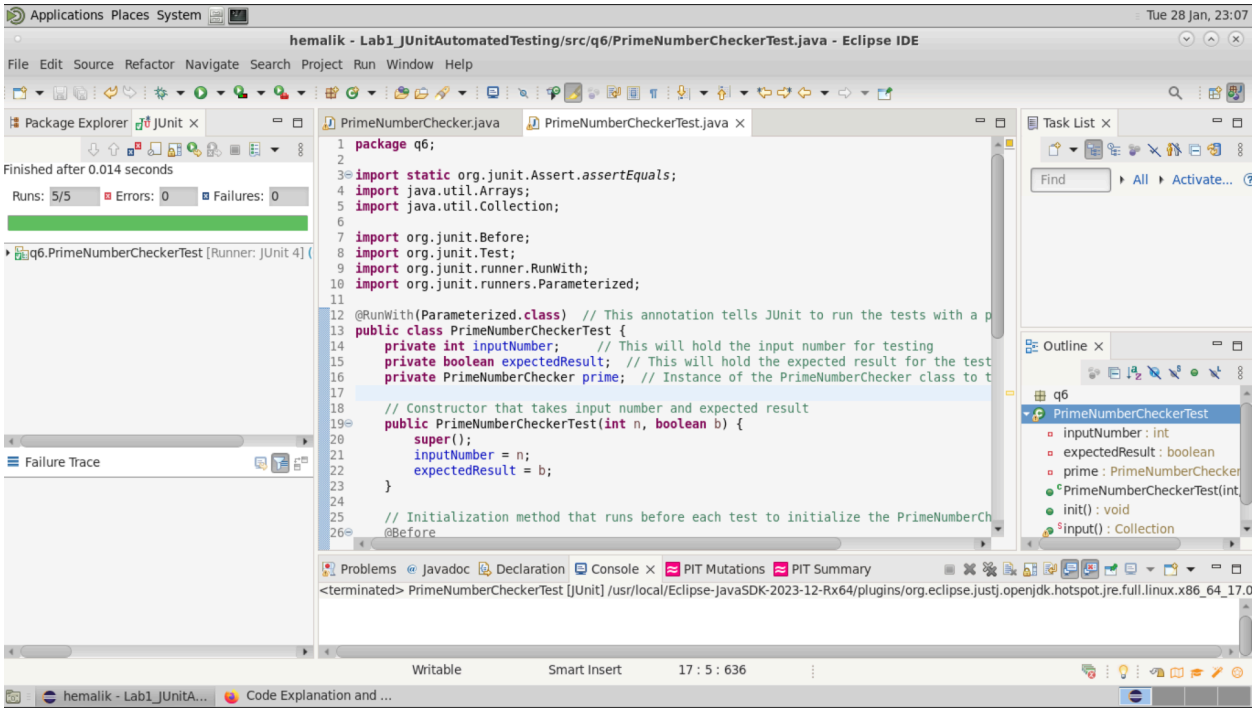
Q3: In this task, we implemented a class with a method to validate phone numbers using regular expressions. The `checkPhoneNumber` method uses the `matches` function to verify whether an input follows the expected pattern: three digits, a space, three digits, a hyphen, and four digits (e.g., `123 456-7890`). The program prompts the user to enter a phone number, then calls the `checkPhoneNumber` method to determine its validity and prints the result. The issue with the initial regular expression was that it did not account for potential variations in spacing around parentheses and the hyphen, leading to incorrect validation. The regex `(\d{3}) \d{3} - \d{4}` enforced a strict pattern with exactly one space between the area code and the first segment, as well as before the hyphen. To fix this, we needed to modify the regex to allow flexible spacing while ensuring the correct structure of the phone number.



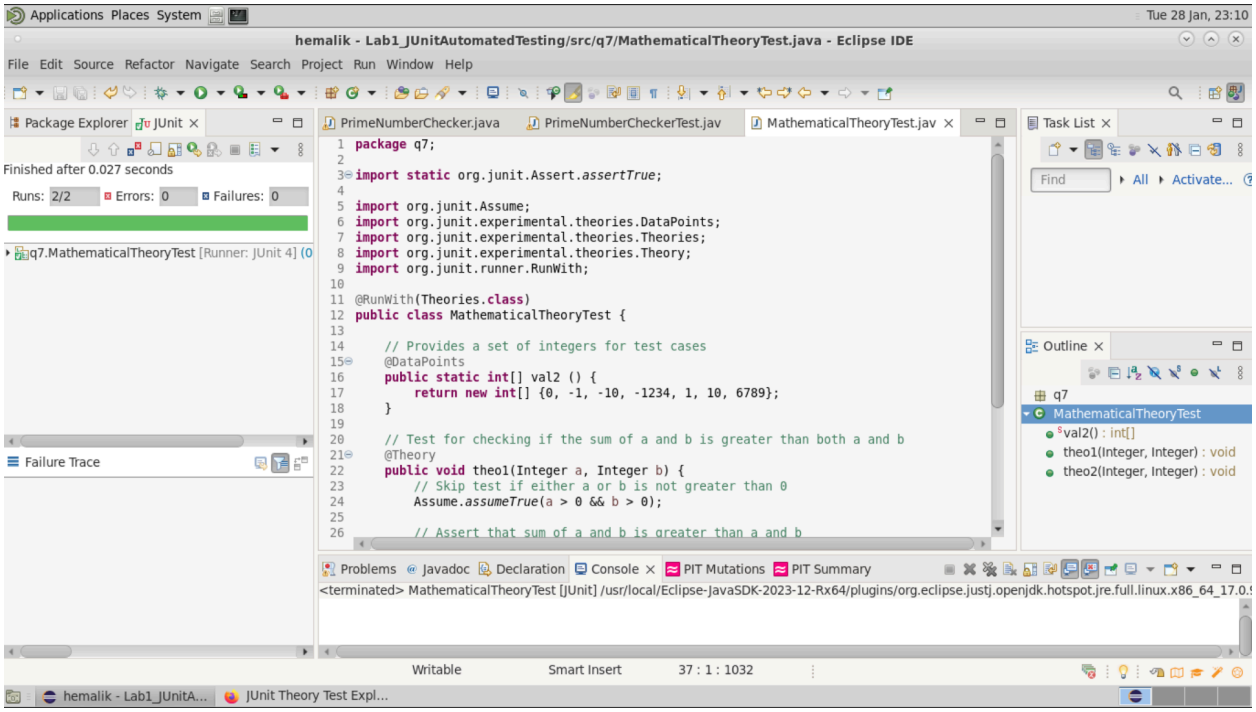
Q4: In this task, we created a JUnit test suite to run tests from Questions 2 and 3 in suite mode. This involved defining a new test class that utilized JUnit annotations to group together the test cases from both programs. By doing so, we were able to execute multiple test programs simultaneously, making it more efficient to run and manage the tests for the Triangle class and the phone number validation class within a single suite.



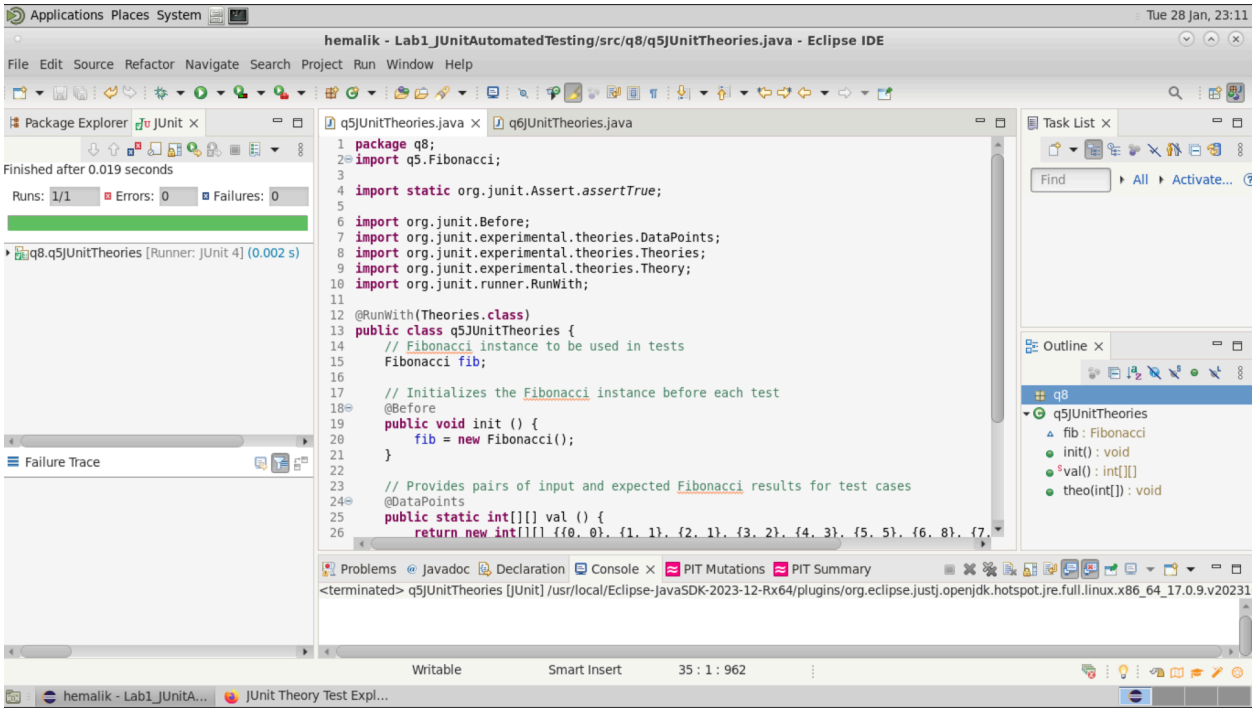
Q5: In this task, we implemented a JUnit testing program for a given Fibonacci sequence generator. The `compute(int n)` method recursively calculates the Fibonacci number for a given index `n`, with base cases for $n \leq 1$. To validate its correctness, I created a parameterized JUnit test that checks the computed Fibonacci numbers for indices 0 to 9. Using parameterized testing with a constructor, we passed both the input index and the expected Fibonacci value, ensuring that the method correctly produces the expected sequence while avoiding redundant test cases. This approach streamlined the testing process and effectively verified the accuracy of the Fibonacci computation.

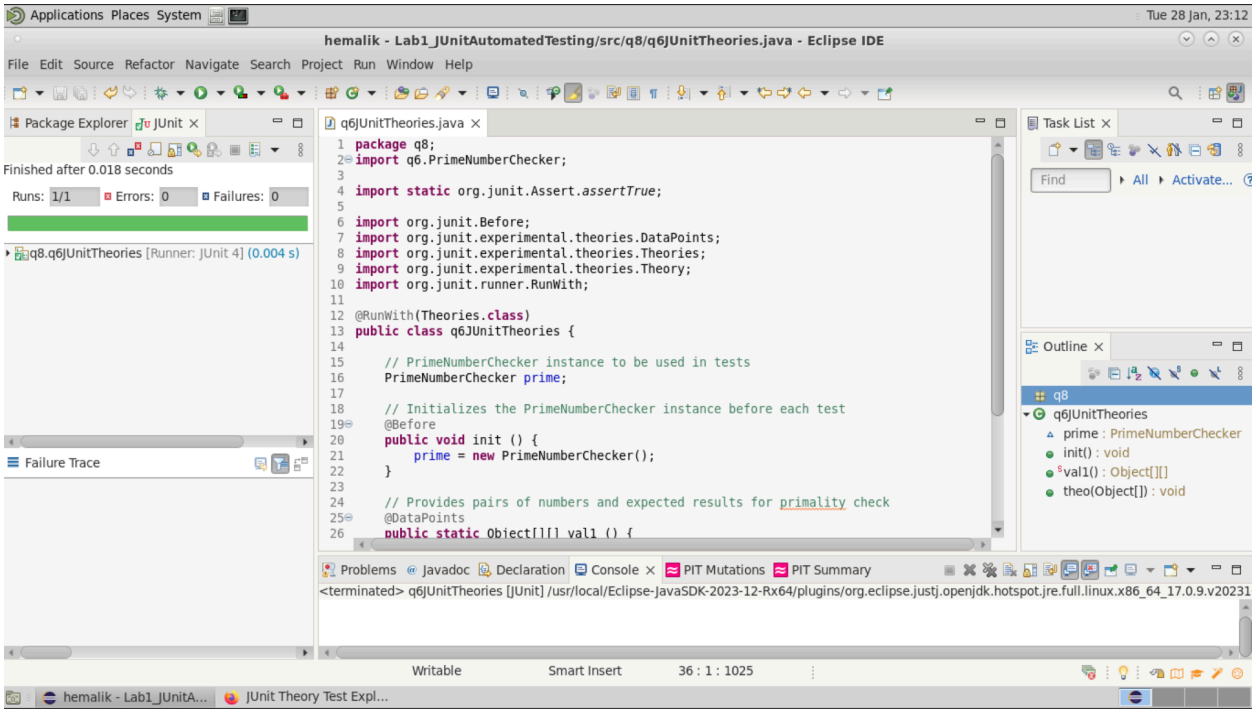


Q6: In this task, we implemented a PrimeNumberChecker class to determine whether a given integer is prime. The method returns `true` if the number is prime and `false` otherwise. To validate its functionality, I created a parameterized JUnit test that checks whether the numbers 2, 6, 19, 22, and 23 are prime. Using parameterized testing with a constructor, I passed each number along with its expected result, ensuring that the method correctly identifies prime and non-prime numbers. This approach made the testing process efficient and systematic, verifying the accuracy of the PrimeNumberChecker implementation.



Q7: In this task, we implemented JUnit Theories to test the mathematical properties a + b > a and a + b > b for different input values. Theories allow us to test multiple data points dynamically rather than writing individual test cases. If the program assumes both `a` and `b` are positive, pairs like `(0,0)`, `(-1,-1)`, and `(-10,-1234)` fail due to negative or zero values, while valid pairs like `(1,10)` or `(6789,6789)` successfully pass. This approach efficiently verifies the correctness of the mathematical relationships across a range of values while identifying edge cases where the conditions do not hold.





In this task, we were required to rewrite the JUnit tests from Questions 5 (Fibonacci) and 6 (Prime Number Checker) using JUnit Theories. Instead of using traditional parameterized testing, we leveraged the `@DataPoints` annotation to define multiple test values. This allowed JUnit to automatically generate and execute test cases for all

possible input combinations. By applying theory-based testing, we made the tests more dynamic, efficient, and flexible, ensuring the correctness of both the Fibonacci sequence calculator and the Prime Number Checker across a wider range of values without manually defining individual test cases.