SOFTWARE TESTING (IT-461)

PRACTICAL FILE

University School of Information, Communication and Technology
Guru Gobind Singh Indraprastha University, Delhi

Submitted by: YASH ARYAN (06816403220)
Batch: B. Tech (CSE) 7th Semester
Submitted to: Dr. ANJU SAHA

Index

5	Write a JUnit class to test stack and demonstrate the concepts of exception handling, ignore annotation, test suite annotation, timeout.
6	Write a TriangleTest class to check the types of triangle and demonstrate the concepts of exception handling, ignore annotation, testsuite annotation, timeout.
7	Write a java class to implement the recursive method fib, which generates the nth Fibonacci Number. Create test cases for this class using JUnit.
8	Convert an ASCII string to the sum of its characters represented as a Binary string i.e. adds up the values (ASCII) of strings , character by character, and show the sum as a binary digit. Write the JUnit class to test its functioanlity
9	Write junit class for vending machine.The functions are as given below: 1. Accepts coins of 1,5,10,20 Cents. 2. Allow user to select products Coke(25), Pepsi(35), Soda(45) 3. Allow user to take refund by canceling the request. 4. Return selected product and remaining change if any 5. Allow reset operation for vending machine supplier.

Program No. 1

Validation of a program to find the minimum and maximum element out of a given list

The program that determines the minimum and maximum element from a given list in C++ can be validated using the following test cases:

S. No.	Description of test case	Input	Output
1	Test case with a list containing both positive and negative numbers	[-5, -10, 20, -30, 40, 50]	(min: -30, max: 50)
2	Test case with a list containing all negative numbers	[-5, -10, -20, -30, -40, -50]	(min: -50, max: -5)
3	Test case with a list containing all positive numbers	[5, 10, 20, 30, 40, 50]	(min: 5, max: 50)
4	Test case with a list of length greater than 3	[5, 10, -5, 20, -30, 0, 15]	(min: -30, max: 20)
5	Test case with a list of length 3	[5, 10, -5]	(min: -5, max: 10)
6	Test case with a list of length 2	[5, 10]	(min: 5, max: 10)
7	Test case with a list of length 1	[5]	(min: 5, max: 5)
8	Test case with an empty list	[]	(min: 0, max: 0)

Program No. 2

Select a program with at least three inputs and design the test cases for the same using the following techniques:

i) **Equivalence class**
ii) **Decision table testing**
iii) **Boundary value testing**
iv) **Cause effect graph**

Code:

```
#include <iostream>

#include <cmath>

using namespace std;

// Function to calculate the area of a triangle

double calculate_area(double a, double b, double c)
{

    double s = (a + b + c) / 2.0;

    return sqrt(s * (s - a) * (s - b) * (s - c));

}

int main()
{
    double a, b, c;

    cout << "Enter the sides of the triangle: ";

    cin >> a >> b >> c;

    cout << "Area of the triangle: " << calculate_area(a, b, c) << endl;
    return 0;
}
```

Equivalence class testing:

- Test case 1: a = 3, b = 4, c = 5 (valid inputs - triangle inequality holds)
- Test case 2: a = 3, b = 4, c = 10 (invalid inputs - triangle inequality does not hold)
- Test case 3: a = 0, b = 4, c = 5 (invalid inputs - one side is 0)
- Test case 4: a = 3, b = -4, c = 5 (invalid inputs - one side is negative)

Decision table testing:

a	**b**	**c**	**area**
3	4	5	6
3	4	10	0
0	4	5	0
3	-4	5	0

Boundary value testing:

- Test case 1: a = 0, b = 0, c = 0 (minimum value for all sides)
- Test case 2: a = DBL_MAX, b = DBL_MAX, c = DBL_MAX (maximum value for all sides)
- Test case 3: a = DBL_MAX, b = DBL_MAX, c = 1 (maximum value for two sides and minimum value for one side)

Cause effect graph testing

- Test case 1: a = 3, b = 4, c = 5
- Test case 2: a = 5, b = 4, c = 3
- Test case 3: a = 5, b = 3, c = 4
- Test case 4: a = 4, b = 5, c = 3
- Test case 5: a = 4, b = 3, c = 5
- Test case 6: a = 3, b = 5, c = 4
- Test case 7: a = 0, b = 4, c = 5
- Test case 8: a = 3, b = -4, c = 5
- Test case 9: a = DBL_MAX, b = DBL_MAX, c = DBL_MAX
- Test case 10: a = DBL_MAX, b = DBL_MAX, c = 1

Program No. 3

Write a program to find cyclomatic complexity of a program

Code:

```
#include <iostream>
#include <vector>
#include <string>
#include <sstream>

using namespace std;

// Function to count edges, nodes, and regions in a program
void calculateCyclomaticComplexity(const string& sourceCode) {
    istringstream codeStream(sourceCode);
    string line;

    int edges = 0;
    int nodes = 0;
    int regions = 0;

    while (getline(codeStream, line)) {
        if (line.find("if") != string::npos || line.find("else") != string::npos ||
            line.find("for") != string::npos || line.find("while") != string::npos) {
            edges++; // Decision point
        }

        nodes++; // Each line represents a node

        if (line.find("}") != string::npos) {
            regions++; // Closing brace indicates the end of a region
        }
    }

    int complexity = edges - nodes + 2 * regions;

    cout << "Cyclomatic Complexity: " << complexity << endl;
}

int main() {
    // Example program (you can replace it with your own program)
    string sourceCode = R"(
        #include <iostream>

        int main() {
            int x = 5;
            int y = 10;

            if (x > y) {
                cout << "x is greater than y." << endl;
            } else {
                cout << "y is greater than or equal to x." << endl;
            }

            for (int i = 0; i < 5; ++i) {
                cout << "Iteration " << i << endl;
            }

            return 0;
        }
    )";

    calculateCyclomaticComplexity(sourceCode);

    return 0;
}
```

Output:

```
Cyclomatic Complexity: 5
```

Program No. 4.1

Conduct a Survey of (At least Five) different Testing Tools available. Prepare a Presentation of their advantages and disadvantages

1. **JUnit:** JUnit is a popular open-source testing tool for Java. It allows developers to write and run repeatable tests. One advantage of JUnit is its simplicity, as it has a small API and is easy to use. Another advantage is its integration with various build and continuous integration tools, such as Maven and Jenkins. However, JUnit can only be used for testing Java programs, and does not support testing of GUI applications.
2. **TestNG:** TestNG is another popular testing tool for Java, similar to JUnit. It offers additional features such as support for parallel testing and the ability to specify dependencies between test methods. TestNG also allows for the use of custom annotations and test groups. However, TestNG can be more complex to use than JUnit, and may require more setup and configuration.
3. **Selenium:** Selenium is a browser automation tool that can be used for testing web applications. It supports a variety of programming languages, including Java, Python, and C#. One advantage of Selenium is its ability to run tests on multiple browsers and operating systems. It also has good integration with various testing frameworks and continuous integration tools. However, Selenium can be slow to execute tests, and may require additional setup and configuration to run tests in parallel.
4. **Appium:** Appium is a mobile application testing tool that allows developers to test native, hybrid, and web apps on various mobile platforms, including iOS and Android. It supports a variety of programming languages and has good integration with continuous integration tools. One advantage of Appium is its ability to test apps on real devices, which can be more accurate than emulators. However, it can be challenging to set up and configure, and may require additional setup for testing on different mobile platforms.
5. **Postman:** Postman is a tool for testing APIs, allowing developers to send HTTP requests and validate the responses. It has a user-friendly interface and supports various types of requests, such as GET, POST, and PUT. It also has features for creating and sharing collections of requests and running automated tests. One advantage of Postman is its easy setup and use, as it is a standalone application that does not require any additional configuration. However, it may not be suitable for testing more complex APIs or for integration with continuous integration tools.

Program No. 4.2

Conduct at least four experiments to explore the strength of any of the following testing tools for automated testing

a. Rational robo

b. NUnit

c. JUNIT

1. **Test execution time:** One experiment that could be conducted is to measure the execution time of a set of test cases using JUnit. This could help to determine how efficiently JUnit is able to run tests.
2. **Test coverage:** Another experiment could be to measure the test coverage achieved by JUnit for a given codebase. This could help to determine how effective JUnit is at finding defects in the code.
3. **Integration with build and continuous integration tools:** Another experiment could be to evaluate the integration of JUnit with various build and continuous integration tools, such as Maven or Jenkins. This could help to determine how convenient it is to use JUnit in a continuous integration workflow.
4. **Ease of use:** A final experiment could be to evaluate the ease of use of JUnit, for example by comparing the learning curve for the tool or the number of lines of code required to write and run tests. This could help to determine how user-friendly JUnit is.

Program No. 5

Write a program to perform mutation testing of a given program

Code:

```cpp
#include <iostream>

// The function to be tested
int add(int a, int b) {
    return a + b;
}

using namespace std;

int main() {
    // Test cases
    int test_cases[][2] = {
        {3, 4},     // Test Case 1
        {-1, 5},    // Test Case 2
        {0, 0},     // Test Case 3
        {10, -3},   // Test Case 4
        {-8, -2},   // Test Case 5
    };

    // Loop over test cases
    for (int i = 0; i < sizeof(test_cases) / sizeof(test_cases[0]); ++i) {
        int test_a = test_cases[i][0];
        int test_b = test_cases[i][1];

        // Original function
        int original_result = add(test_a, test_b);
        cout << "Test Case " << i + 1 << " - Original Result: " << original_result <<
endl;

        // Mutated function: Replace + with -
        int mutated_result = test_a - test_b;
        cout << "Test Case " << i + 1 << " - Mutated Result: " << mutated_result << endl;

        // Check if the test case detects the mutation
        if (original_result != mutated_result) {
            cout << "Test Case " << i + 1 << " - Mutation detected: Original function and
mutated function have different results.\n";
        } else {
            cout << "Test Case " << i + 1 << " - Mutation not detected: Test case passed
for both original and mutated functions.\n";
        }

        cout << endl;
    }

    return 0;
}
```

Output:

```
Test Case 1 - Original Result: 7

Test Case 1 - Mutated Result: -1

Test Case 1 - Mutation detected: Original function and mutated function have different
results.


Test Case 2 - Original Result: 4

Test Case 2 - Mutated Result: -6

Test Case 2 - Mutation detected: Original function and mutated function have different
results.


Test Case 3 - Original Result: 0

Test Case 3 - Mutated Result: 0

Test Case 3 - Mutation not detected: Test case passed for both original and mutated
functions.


Test Case 4 - Original Result: 7

Test Case 4 - Mutated Result: 13

Test Case 4 - Mutation detected: Original function and mutated function have different
results.


Test Case 5 - Original Result: -10

Test Case 5 - Mutated Result: -6

Test Case 5 - Mutation detected: Original function and mutated function have different
results.
```

Program No. 6

Write a program to perform slice based testing of a given program.

Code:

```
#include <iostream>

// Function to be tested
int calculateSumProduct(int a, int b, int c) {
    int sum = a + b;
    int product = sum * c;
    return product;
}

using namespace std;

int main() {
    // Test cases
    int testCases[][3] = {
        {1, 2, 3},
        {-1, -2, 5},
        {0, 0, 7},
        {4, 5, 2},
    };

    // Slices
    int slices[][2] = {
        {0, 1},  // Slice 1: a + b
        {1, 2},  // Slice 2: sum * c
        {0, 2}   // Slice 3: a + b * c
    };

    // Perform slice-based testing
    for (int i = 0; i < sizeof(testCases) / sizeof(testCases[0]); ++i) {
        int a = testCases[i][0];
        int b = testCases[i][1];
        int c = testCases[i][2];

        cout << "Test Case " << i + 1 << ":\n";
        for (int j = 0; j < sizeof(slices) / sizeof(slices[0]); ++j) {
            int start = slices[j][0];
            int end = slices[j][1];

            // Create a slice
            int sliceResult = calculateSumProduct(a, b, c);

            // Print slice information
            cout << "  Slice " << j + 1 << ": ";
            for (int k = start; k <= end; ++k) {
                cout << (k == 0 ? "a" : (k == 1 ? "+ b" : "* c"));
            }
            cout << " = " << sliceResult << "\n";
        }
        cout << "\n";
    }

    return 0;
}
```

Output:

```
Test Case 1:

  Slice 1: a + b = 3

  Slice 2: + b* c = 9

  Slice 3: a + b* c = 9


Test Case 2:

  Slice 1: a + b = -3

  Slice 2: + b* c = -15

  Slice 3: a + b* c = -15


Test Case 3:

  Slice 1: a + b = 0

  Slice 2: + b* c = 0

  Slice 3: a + b* c = 0


Test Case 4:

  Slice 1: a + b = 9

  Slice 2: + b* c = 18

  Slice 3: a + b* c = 9
```

Program No. 1

Download eclipse and create a JUnit class for Scientific Calculator java class.

Code:

Calculator.java

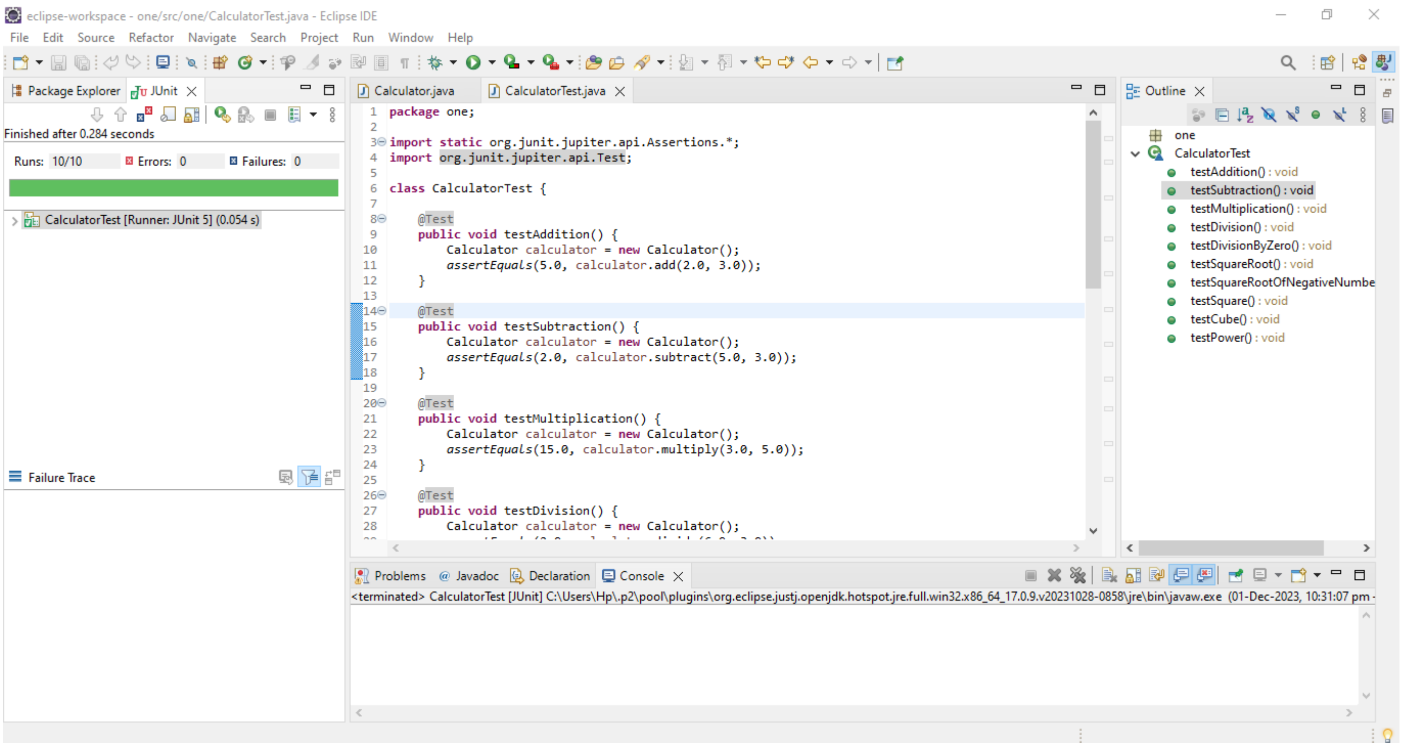
```
package one;

public class Calculator {
	public double add(double a, double b) {
        return a + b;
    }

    public double subtract(double a, double b) {
        return a - b;
    }

    public double multiply(double a, double b) {
        return a * b;
    }

    public double divide(double a, double b) {
        if (b == 0) {
            throw new IllegalArgumentException("Cannot divide by zero");
        }
        return a / b;
    }

    public double squareRoot(double a) {
        if (a < 0) {
            throw new IllegalArgumentException("Cannot calculate square root of a negative 
number");
        }
        return Math.sqrt(a);
    }
    public double square(double a) {
        return Math.pow(a, 2);
    }

    public double cube(double a) {
        return Math.pow(a, 3);
    }

    public double power(double base, double exponent) {
        return Math.pow(base, exponent);
    }

}
```

CalculatorTest.java

```
package one;

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

```java
class CalculatorTest {

     @Test
    public void testAddition() {
        Calculator calculator = new Calculator();
        assertEquals(5.0, calculator.add(2.0, 3.0));
    }

    @Test
    public void testSubtraction() {
       Calculator calculator = new Calculator();
        assertEquals(2.0, calculator.subtract(5.0, 3.0));
    }

    @Test
    public void testMultiplication() {
       Calculator calculator = new Calculator();
        assertEquals(15.0, calculator.multiply(3.0, 5.0));
    }

    @Test
    public void testDivision() {
       Calculator calculator = new Calculator();
        assertEquals(2.0, calculator.divide(6.0, 3.0));
    }

    @Test
    public void testDivisionByZero() {
       Calculator calculator = new Calculator();
        assertThrows(IllegalArgumentException.class, () -> calculator.divide(5.0, 0.0));
    }

    @Test
    public void testSquareRoot() {
       Calculator calculator = new Calculator();
        assertEquals(2.0, calculator.squareRoot(4.0));
    }

    @Test
    public void testSquareRootOfNegativeNumber() {
       Calculator calculator = new Calculator();
        assertThrows(IllegalArgumentException.class, () -> calculator.squareRoot(-4.0));
    }
    @Test
    public void testSquare() {
       Calculator calculator = new Calculator();
        assertEquals(4.0, calculator.square(2.0));
    }

    @Test
    public void testCube() {
       Calculator calculator = new Calculator();
        assertEquals(8.0, calculator.cube(2.0));
    }

    @Test
    public void testPower() {
       Calculator calculator = new Calculator();
        assertEquals(8.0, calculator.power(2.0, 3.0));
    }

}
```

Output:



eclipse-workspace - one/src/one/CalculatorTest.java - Eclipse IDE

Finished after 0.284 seconds

Runs: 10/10 Errors: 0 Failures: 0

CalculatorTest [Runner: JUnit 5] (0.054 s)

```java
package one;

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

class CalculatorTest {

    @Test
    public void testAddition() {
        Calculator calculator = new Calculator();
        assertEquals(5.0, calculator.add(2.0, 3.0));
    }

    @Test
    public void testSubtraction() {
        Calculator calculator = new Calculator();
        assertEquals(2.0, calculator.subtract(5.0, 3.0));
    }

    @Test
    public void testMultiplication() {
        Calculator calculator = new Calculator();
        assertEquals(15.0, calculator.multiply(3.0, 5.0));
    }

    @Test
    public void testDivision() {
        Calculator calculator = new Calculator();
```

```
one
CalculatorTest
    testAddition() : void
    testSubtraction() : void
    testMultiplication() : void
    testDivision() : void
    testDivisionByZero() : void
    testSquareRoot() : void
    testSquareRootOfNegativeNumbe
    testSquare() : void
    testCube() : void
    testPower() : void
```

```
<terminated> CalculatorTest [JUnit] C:\Users\Hp\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.9.v20231028-0858\jre\bin\javaw.exe (01-Dec-2023, 10:31:07 pm -
```

Program No. 2

Write all the assert statements with the meaning

Assert Statements:

1. **assertEquals:** Asserts that two values are equal.

   Syntax:

   assertEquals(expected, actual, message);

2. **assertNotEquals:** Asserts that two values are not equal.

   Syntax:

   assertNotEquals(expected, actual, message);

3. **assertTrue:** Asserts that a boolean condition is true.

   Syntax:

   assertTrue(condition, message);

4. **assertFalse:** Asserts that a boolean condition is false.

   Syntax:

   assertFalse(condition, message);

5. **assertNull:** Asserts that a value is null.

   Syntax:

   assertNull(value, message);

6. **assertNotNull:** Asserts that a value is not null.

   Syntax:

   assertNotNull(value, message);

7. assertSame: Asserts that two references refer to the same object.

Syntax:

```
assertSame(expected, actual, message);
```

8. assertNotSame: Asserts that two references do not refer to the same object.

Syntax:

```
assertNotSame(expected, actual, message);
```

9. assertArrayEquals: Asserts that two arrays are equal.

Syntax:

```
assertArrayEquals(expectedArray, actualArray, message);
```

10. assertThrows: Asserts that a specific exception is thrown.

Syntax:

```
assertThrows(ExpectedException.class, () -> someMethod(), message);
```

11. fail: Causes the test to fail with the given message.

Syntax:

```
fail(message);
```

Program No. 3

Write a JUnit Class to find the minimum out of given n numbers

Code:

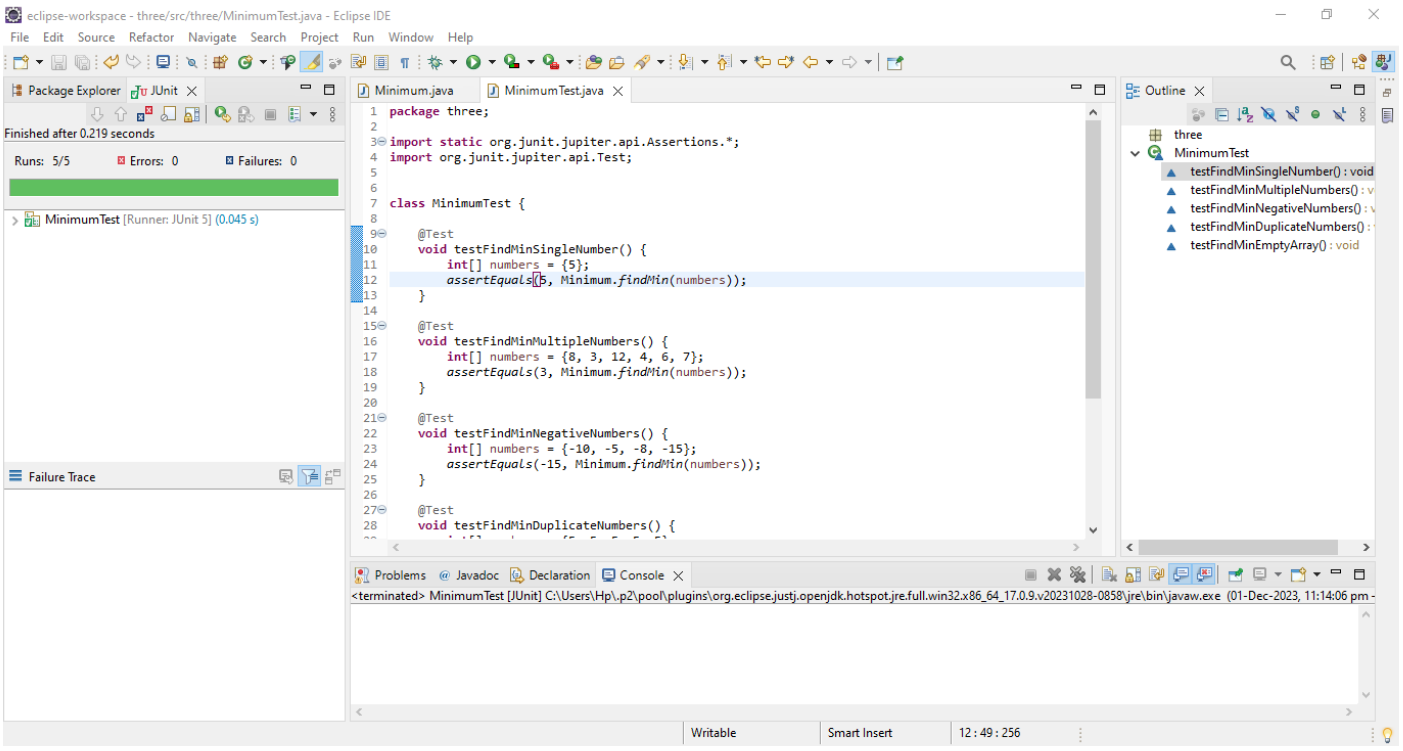
Minimum.java

```java
package three;

public class Minimum {
	public static int findMin(int[] numbers) {
	  if (numbers.length == 0) {
	      throw new IllegalArgumentException("Array cannot be empty");
	  }

	  int min = numbers[0];
	  for (int i = 1; i < numbers.length; i++) {
	      if (numbers[i] < min) {
	          min = numbers[i];
	      }
	  }

	  return min;
    }
}
```

MinimumTest.java

```java
package three;

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


class MinimumTest {

	@Test
    void testFindMinSingleNumber() {
        int[] numbers = {5};
        assertEquals(5, Minimum.findMin(numbers));
    }

    @Test
    void testFindMinMultipleNumbers() {
        int[] numbers = {8, 3, 12, 4, 6, 7};
        assertEquals(3, Minimum.findMin(numbers));
    }

    @Test
    void testFindMinNegativeNumbers() {
        int[] numbers = {-10, -5, -8, -15};
        assertEquals(-15, Minimum.findMin(numbers));
    }

    @Test
    void testFindMinDuplicateNumbers() {
        int[] numbers = {5, 5, 5, 5, 5};
        assertEquals(5, Minimum.findMin(numbers));
    }
    @Test
    void testFindMinEmptyArray() {
        int[] numbers = {};
        assertThrows(IllegalArgumentException.class, () -> Minimum.findMin(numbers));
    }


}
```

Output:



eclipse-workspace - three/src/three/MinimumTest.java - Eclipse IDE

Finished after 0.219 seconds

Runs: 5/5 Errors: 0 Failures: 0

MinimumTest [Runner: JUnit 5] (0.045 s)

```java
package three;

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


class MinimumTest {

    @Test
    void testFindMinSingleNumber() {
        int[] numbers = {5};
        assertEquals(5, Minimum.findMin(numbers));
    }

    @Test
    void testFindMinMultipleNumbers() {
        int[] numbers = {8, 3, 12, 4, 6, 7};
        assertEquals(3, Minimum.findMin(numbers));
    }

    @Test
    void testFindMinNegativeNumbers() {
        int[] numbers = {-10, -5, -8, -15};
        assertEquals(-15, Minimum.findMin(numbers));
    }

    @Test
    void testFindMinDuplicateNumbers() {
```

```
<terminated> MinimumTest [JUnit] C:\Users\Hp\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.9.v20231028-0858\jre\bin\javaw.exe  (01-Dec-2023, 11:14:06 pm -
```

Outline:
- three
- MinimumTest
  - testFindMinSingleNumber() : void
  - testFindMinMultipleNumbers() : v
  - testFindMinNegativeNumbers() : v
  - testFindMinDuplicateNumbers() :
  - testFindMinEmptyArray() : void

Writable | Smart Insert | 12 : 49 : 256

Program No. 4

Write a JUnit Class to test a bank account class.

Code:

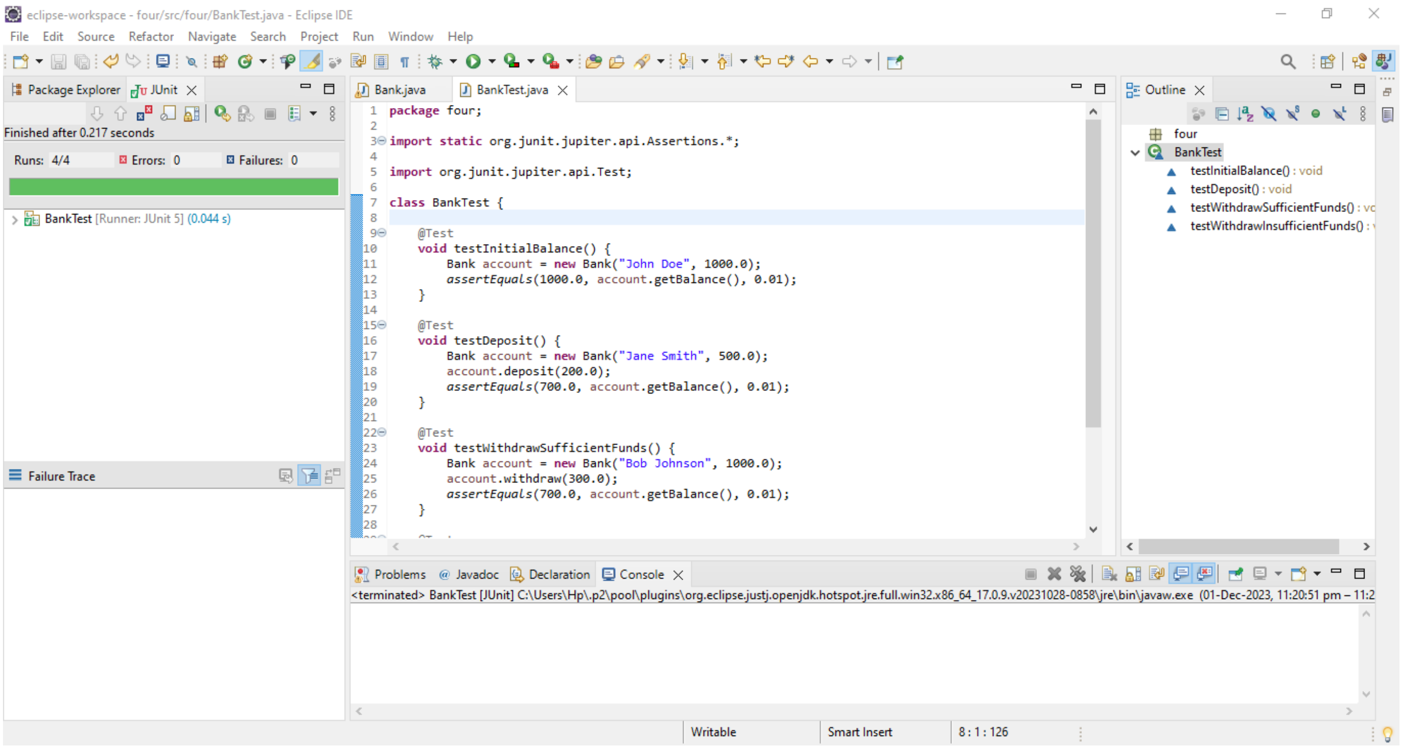
Bank.java

```
package four;

public class Bank {
	    private String accountHolder;
    private double balance;

    public Bank(String accountHolder, double initialBalance) {
        this.accountHolder = accountHolder;
        this.balance = initialBalance;
    }

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        } else {
            throw new IllegalArgumentException("Deposit amount must be greater than zero");
        }
    }

    public void withdraw(double amount) {
        if (amount > 0) {
            if (amount <= balance) {
                balance -= amount;
            } else {
                throw new RuntimeException("Insufficient funds for withdrawal");
            }
        } else {
            throw new IllegalArgumentException("Withdrawal amount must be greater than zero");
        }
    }

}
```

BankTest.java

```
package four;

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

import org.junit.jupiter.api.Test;

class BankTest {

	    @Test
    void testInitialBalance() {
        Bank account = new Bank("John Doe", 1000.0);
        assertEquals(1000.0, account.getBalance(), 0.01);
    }

    @Test
    void testDeposit() {
       Bank account = new Bank("Jane Smith", 500.0);
         account.deposit(200.0);
         assertEquals(700.0, account.getBalance(), 0.01);
    }

    @Test
    void testWithdrawSufficientFunds() {
       Bank account = new Bank("Bob Johnson", 1000.0);
         account.withdraw(300.0);
         assertEquals(700.0, account.getBalance(), 0.01);
    }

    @Test
    void testWithdrawInsufficientFunds() {
       Bank account = new Bank("Alice Brown", 200.0);
         assertThrows(RuntimeException.class, () -> account.withdraw(300.0));
    }


}
```

Output:



```
Finished after 0.217 seconds
Runs: 4/4    Errors: 0    Failures: 0

BankTest [Runner: JUnit 5] (0.044 s)
```

```java
package four;

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

import org.junit.jupiter.api.Test;

class BankTest {

    @Test
    void testInitialBalance() {
        Bank account = new Bank("John Doe", 1000.0);
        assertEquals(1000.0, account.getBalance(), 0.01);
    }

    @Test
    void testDeposit() {
        Bank account = new Bank("Jane Smith", 500.0);
        account.deposit(200.0);
        assertEquals(700.0, account.getBalance(), 0.01);
    }

    @Test
    void testWithdrawSufficientFunds() {
        Bank account = new Bank("Bob Johnson", 1000.0);
        account.withdraw(300.0);
        assertEquals(700.0, account.getBalance(), 0.01);
    }
```

```
four
BankTest
    testInitialBalance() : void
    testDeposit() : void
    testWithdrawSufficientFunds() : vo
    testWithdrawInsufficientFunds() :
```

```
<terminated> BankTest [JUnit] C:\Users\Hp\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.9.v20231028-0858\jre\bin\javaw.exe (01-Dec-2023, 11:20:51 pm – 11:2
```

Problem No. 5

Write a JUnit class to test stack and demonstrate the concepts of exception handling, ignore annotation, test suite annotation, timeout

Code:

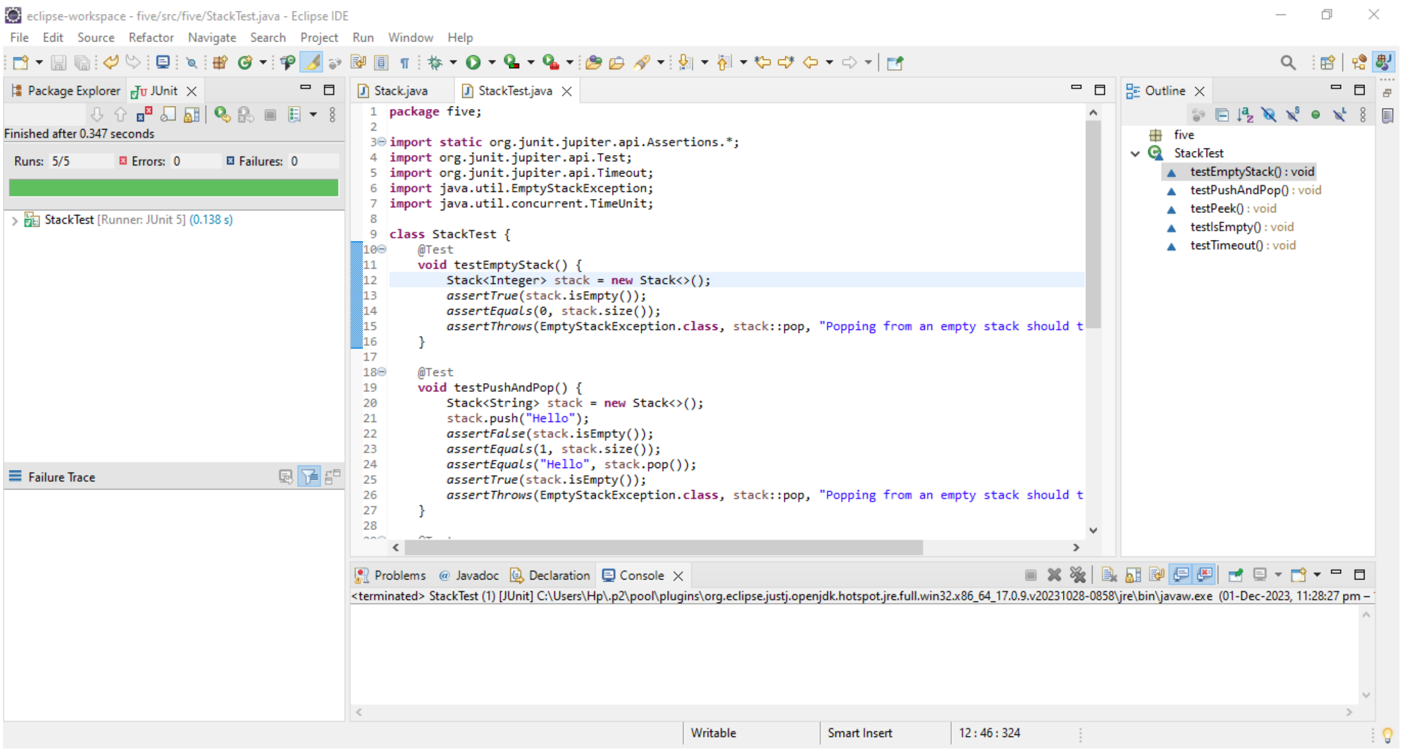
Stack.java

```
package five;
import java.util.EmptyStackException;
import java.util.LinkedList;

public class Stack<T> {
	private LinkedList<T> list = new LinkedList<>();

    public void push(T item) {
        list.addLast(item);
    }

    public T pop() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return list.removeLast();
    }

    public T peek() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return list.getLast();
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public int size() {
        return list.size();
    }

}
```

StackTest.java

```
package five;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Timeout;
import java.util.EmptyStackException;
import java.util.concurrent.TimeUnit;

class StackTest {
	@Test
    void testEmptyStack() {
        Stack<Integer> stack = new Stack<>();
        assertTrue(stack.isEmpty());
        assertEquals(0, stack.size());
        assertThrows(EmptyStackException.class, stack::pop, "Popping from an empty stack should
throw EmptyStackException");
    }

    @Test
    void testPushAndPop() {
       Stack<String> stack = new Stack<>();
        stack.push("Hello");
        assertFalse(stack.isEmpty());
        assertEquals(1, stack.size());
        assertEquals("Hello", stack.pop());
        assertTrue(stack.isEmpty());
        assertThrows(EmptyStackException.class, stack::pop, "Popping from an empty stack should
throw EmptyStackException");
    }

    @Test
    void testPeek() {
       Stack<Double> stack = new Stack<>();
        stack.push(3.14);
        assertEquals(3.14, stack.peek(), 0.01);
        assertFalse(stack.isEmpty());
        assertEquals(1, stack.size());
    }

    @Test
    void testIsEmpty() {
       Stack<Character> stack = new Stack<>();
        assertTrue(stack.isEmpty());
        stack.push('A');
        assertFalse(stack.isEmpty());
    }

    @Test
    @Timeout(value = 100, unit = TimeUnit.MILLISECONDS)
    void testTimeout() throws InterruptedException {
        // This test will pass if it completes within 100 milliseconds
        Thread.sleep(50);
        assertTrue(true);
    }

}
```

Output:



eclipse-workspace - five/src/five/StackTest.java - Eclipse IDE

Finished after 0.347 seconds

Runs: 5/5 Errors: 0 Failures: 0

StackTest [Runner: JUnit 5] (0.138 s)

```java
package five;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Timeout;
import java.util.EmptyStackException;
import java.util.concurrent.TimeUnit;

class StackTest {
    @Test
    void testEmptyStack() {
        Stack<Integer> stack = new Stack<>();
        assertTrue(stack.isEmpty());
        assertEquals(0, stack.size());
        assertThrows(EmptyStackException.class, stack::pop, "Popping from an empty stack should t
    }

    @Test
    void testPushAndPop() {
        Stack<String> stack = new Stack<>();
        stack.push("Hello");
        assertFalse(stack.isEmpty());
        assertEquals(1, stack.size());
        assertEquals("Hello", stack.pop());
        assertTrue(stack.isEmpty());
        assertThrows(EmptyStackException.class, stack::pop, "Popping from an empty stack should t
    }

```

Outline: five, StackTest, testEmptyStack() : void, testPushAndPop() : void, testPeek() : void, testIsEmpty() : void, testTimeout() : void

```
<terminated> StackTest (1) [JUnit] C:\Users\Hp\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.9.v20231028-0858\jre\bin\javaw.exe  (01-Dec-2023, 11:28:27 pm –
```

Writable | Smart Insert | 12 : 46 : 324

Problem No. 6

Write a triangletest class to check the types of triangle and demonstrate the concepts of exception handling, ignore annotation, testsuite annotation, timeout.

Code:

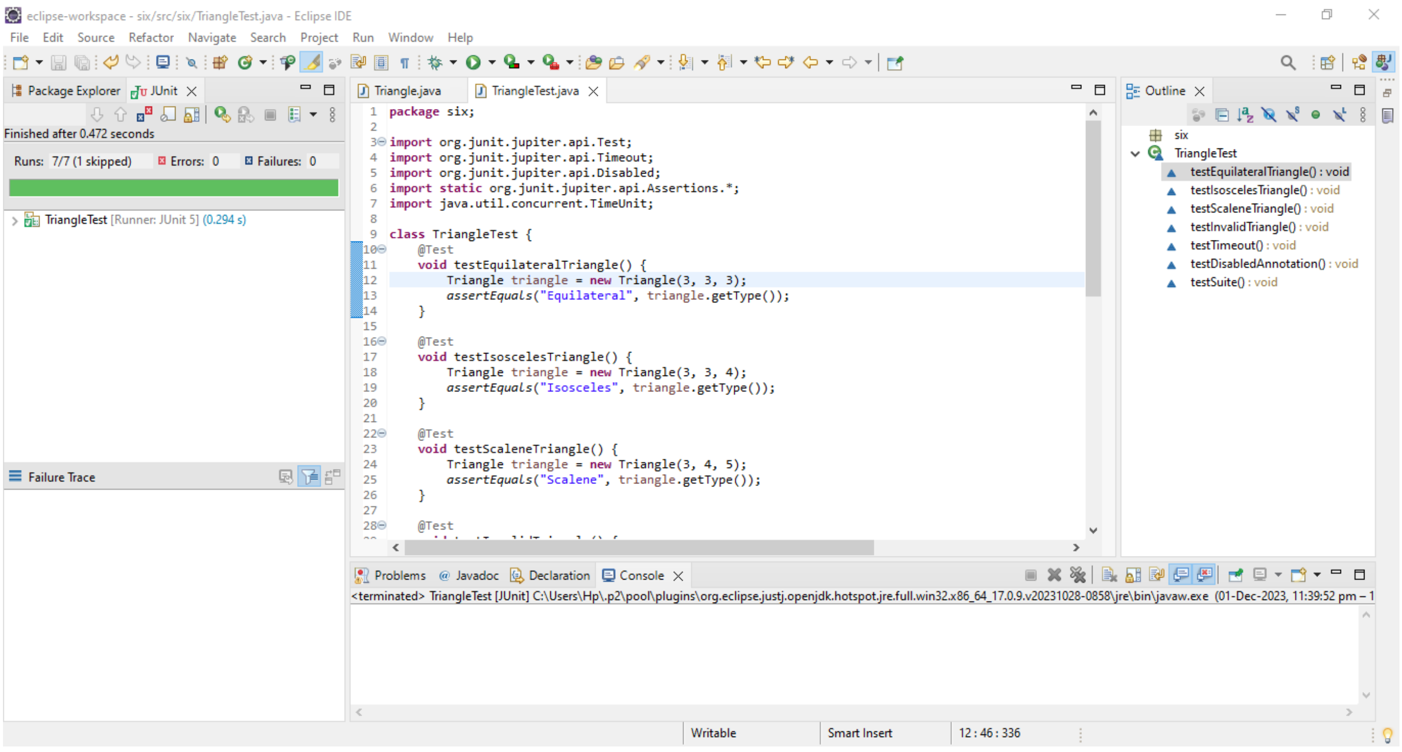
Triangle.java

```java
package six;

public class Triangle {
	private int side1;
    private int side2;
    private int side3;

    public Triangle(int side1, int side2, int side3) {
        validateSides(side1, side2, side3);
        this.side1 = side1;
        this.side2 = side2;
        this.side3 = side3;
    }

    public String getType() {
        if (side1 == side2 && side2 == side3) {
            return "Equilateral";
        } else if (side1 == side2 || side1 == side3 || side2 == side3) {
            return "Isosceles";
        } else {
            return "Scalene";
        }
    }

    private void validateSides(int side1, int side2, int side3) {
        if (side1 <= 0 || side2 <= 0 || side3 <= 0 || (side1 + side2 <= side3) || (side1 +
side3 <= side2) || (side2 + side3 <= side1)) {
            throw new IllegalArgumentException("Invalid sides for a triangle");
        }
    }

}
```

TriangleTest.java

```java
package six;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Timeout;
import org.junit.jupiter.api.Disabled;
import static org.junit.jupiter.api.Assertions.*;
import java.util.concurrent.TimeUnit;

class TriangleTest {
	@Test
    void testEquilateralTriangle() {
        Triangle triangle = new Triangle(3, 3, 3);
        assertEquals("Equilateral", triangle.getType());
    }

    @Test
    void testIsoscelesTriangle() {
       Triangle triangle = new Triangle(3, 3, 4);
        assertEquals("Isosceles", triangle.getType());
    }

    @Test
    void testScaleneTriangle() {
       Triangle triangle = new Triangle(3, 4, 5);
        assertEquals("Scalene", triangle.getType());
    }

    @Test
    void testInvalidTriangle() {
        assertThrows(IllegalArgumentException.class, () -> new Triangle(1, 2, 5), "Invalid 
sides should throw IllegalArgumentException");
    }

    @Test
    @Timeout(value = 100, unit = TimeUnit.MILLISECONDS)
    void testTimeout() throws InterruptedException {
        // This test will pass if it completes within 100 milliseconds
        Thread.sleep(50);
        assertTrue(true);
    }


    @Test
    @Disabled("This test should be disabled")
    void testDisabledAnnotation() {
        // This test will be disabled
        // No assertions or fail statements here
    }


    // Test suite
    @Test
    void testSuite() {
        assertAll(
            () -> testEquilateralTriangle(),
            () -> testIsoscelesTriangle(),
            () -> testScaleneTriangle(),
            () -> testInvalidTriangle(),
            () -> testTimeout(),
            () -> testDisabledAnnotation()
        );
    }


}
```

Output:



```
Finished after 0.472 seconds
Runs: 7/7 (1 skipped)   Errors: 0   Failures: 0
TriangleTest [Runner: JUnit 5] (0.294 s)
```

```java
package six;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Timeout;
import org.junit.jupiter.api.Disabled;
import static org.junit.jupiter.api.Assertions.*;
import java.util.concurrent.TimeUnit;

class TriangleTest {
    @Test
    void testEquilateralTriangle() {
        Triangle triangle = new Triangle(3, 3, 3);
        assertEquals("Equilateral", triangle.getType());
    }

    @Test
    void testIsoscelesTriangle() {
        Triangle triangle = new Triangle(3, 3, 4);
        assertEquals("Isosceles", triangle.getType());
    }

    @Test
    void testScaleneTriangle() {
        Triangle triangle = new Triangle(3, 4, 5);
        assertEquals("Scalene", triangle.getType());
    }

    @Test
```

```
six
TriangleTest
    testEquilateralTriangle() : void
    testIsoscelesTriangle() : void
    testScaleneTriangle() : void
    testInvalidTriangle() : void
    testTimeout() : void
    testDisabledAnnotation() : void
    testSuite() : void
```

```
<terminated> TriangleTest [JUnit] C:\Users\Hp\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.9.v20231028-0858\jre\bin\javaw.exe  (01-Dec-2023, 11:39:52 pm – 1
```

Program No. 7

Write a java class to implement the recursive method fib, which generates the nth Fibonacci Number. Create test cases for this class using Junit

Code:

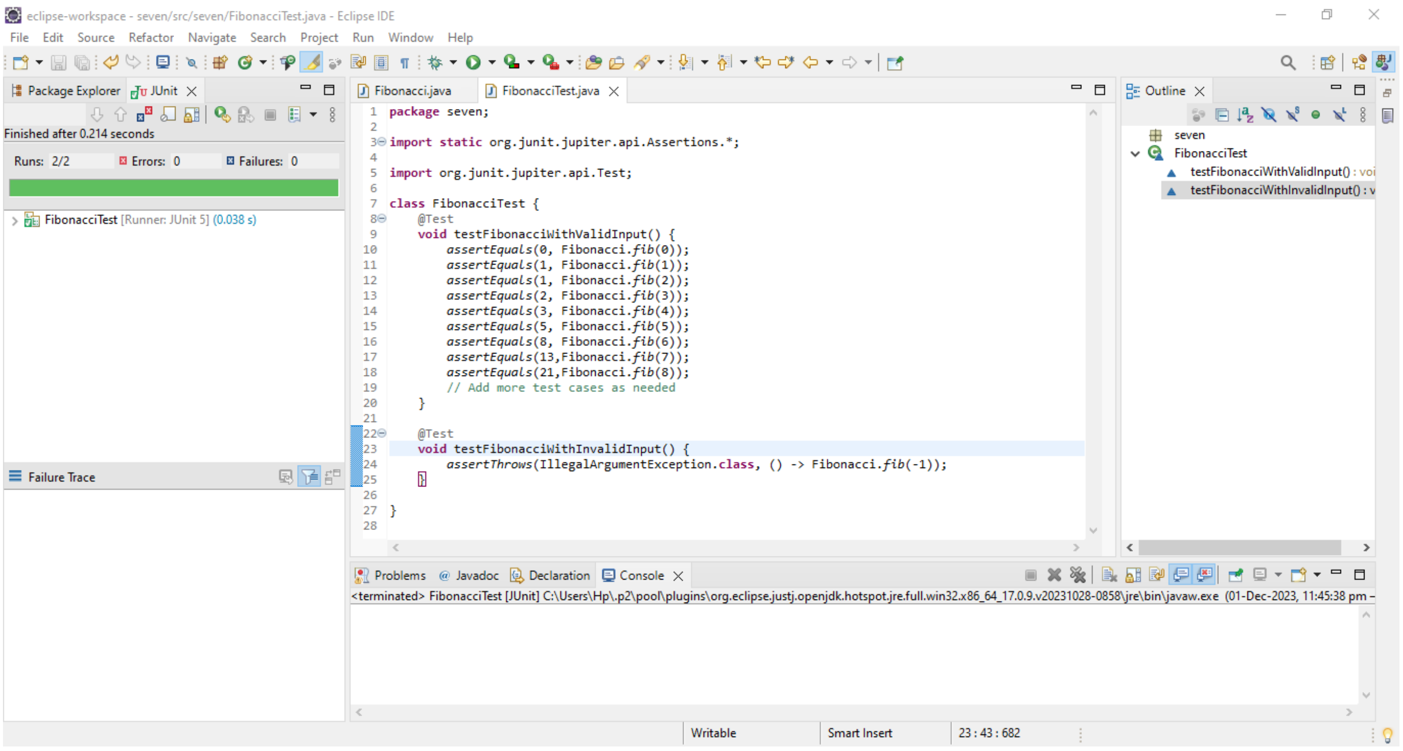
Fibonacci.java

```java
package seven;

public class Fibonacci {
	public static int fib(int n) {
	    if (n < 0) {
	        throw new IllegalArgumentException("Input should be a non-negative integer");
	    }
	    if (n <= 1) {
	        return n;
	    }
	    return fib(n - 1) + fib(n - 2);
    }

}
```

FibonacciTest.java

```java
package seven;

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

import org.junit.jupiter.api.Test;

class FibonacciTest {
	@Test
    void testFibonacciWithValidInput() {
        assertEquals(0, Fibonacci.fib(0));
        assertEquals(1, Fibonacci.fib(1));
        assertEquals(1, Fibonacci.fib(2));
        assertEquals(2, Fibonacci.fib(3));
        assertEquals(3, Fibonacci.fib(4));
        assertEquals(5, Fibonacci.fib(5));
        assertEquals(8, Fibonacci.fib(6));
        assertEquals(13,Fibonacci.fib(7));
        assertEquals(21,Fibonacci.fib(8));
        // Add more test cases as needed
    }

    @Test
    void testFibonacciWithInvalidInput() {
        assertThrows(IllegalArgumentException.class, () -> Fibonacci.fib(-1));
    }

}
```

Output:



eclipse-workspace - seven/src/seven/FibonacciTest.java - Eclipse IDE

```java
package seven;

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

import org.junit.jupiter.api.Test;

class FibonacciTest {
    @Test
    void testFibonacciWithValidInput() {
        assertEquals(0, Fibonacci.fib(0));
        assertEquals(1, Fibonacci.fib(1));
        assertEquals(1, Fibonacci.fib(2));
        assertEquals(2, Fibonacci.fib(3));
        assertEquals(3, Fibonacci.fib(4));
        assertEquals(5, Fibonacci.fib(5));
        assertEquals(8, Fibonacci.fib(6));
        assertEquals(13,Fibonacci.fib(7));
        assertEquals(21,Fibonacci.fib(8));
        // Add more test cases as needed
    }

    @Test
    void testFibonacciWithInvalidInput() {
        assertThrows(IllegalArgumentException.class, () -> Fibonacci.fib(-1));
    }

}
```

```
Finished after 0.214 seconds
Runs: 2/2    Errors: 0    Failures: 0
FibonacciTest [Runner: JUnit 5] (0.038 s)
```

```
<terminated> FibonacciTest [JUnit] C:\Users\Hp\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.9.v20231028-0858\jre\bin\javaw.exe  (01-Dec-2023, 11:45:38 pm –
```

Writable   Smart Insert   23 : 43 : 682

File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
JUnit
Fibonacci.java
FibonacciTest.java
Outline
seven
FibonacciTest
testFibonacciWithValidInput() : voi
testFibonacciWithInvalidInput() : v
Failure Trace
Problems
Javadoc
Declaration
Console

Problem No. 8

Convert an ASCII string to the sum of its characters represented as a Binary string i.e. adds up the values (ASCII) of strings, character by character, and show the sum as a binary digit. Write the JUnit class to test its functionality.

Code:

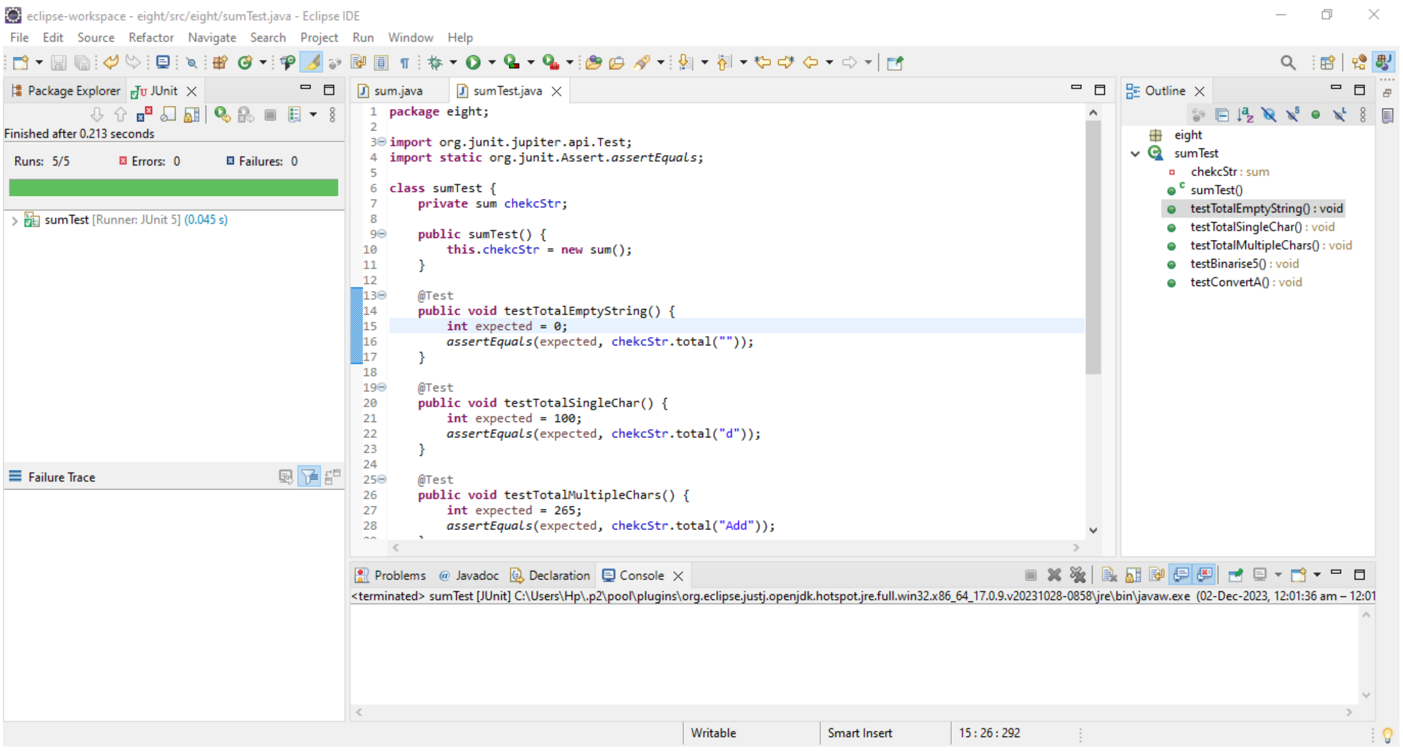
sum.java

```java
package eight;

public class sum {
	public sum() {}
	public String convert(String str) {
	return binarise(total(str));
	}
	public int total(String str) {
	if(str=="") return 0;
	if(str.length()==1) return ((int)(str.charAt(0)));
	return ((int)(str.charAt(0)))+total(str.substring(1));
	}
	public String binarise(int givenstrvalue) {
	if(givenstrvalue==0) return "";
	if(givenstrvalue %2==1) return "1"+binarise(givenstrvalue/2);
	return "0"+binarise(givenstrvalue/2);
	}

}
```

sumTest.java

```java
package eight;

import org.junit.jupiter.api.Test;
import static org.junit.Assert.assertEquals;

class sumTest {
	private sum chekcStr;

    public sumTest() {
        this.chekcStr = new sum();
    }

    @Test
    public void testTotalEmptyString() {
        int expected = 0;
        assertEquals(expected, chekcStr.total(""));
    }

    @Test
    public void testTotalSingleChar() {
        int expected = 100;
        assertEquals(expected, chekcStr.total("d"));
    }

    @Test
    public void testTotalMultipleChars() {
        int expected = 265;
        assertEquals(expected, chekcStr.total("Add"));
    }

    @Test
    public void testBinarise5() {
        String expected = "101";
        assertEquals(expected, chekcStr.binarise(5));
    }

    @Test
    public void testConvertA() {
        String expected = "1000001";
        assertEquals(expected, chekcStr.convert("A"));
    }

}
```

Output:



eclipse-workspace - eight/src/eight/sumTest.java - Eclipse IDE

Finished after 0.213 seconds

Runs: 5/5   Errors: 0   Failures: 0

sumTest [Runner: JUnit 5] (0.045 s)

```java
package eight;

import org.junit.jupiter.api.Test;
import static org.junit.Assert.assertEquals;

class sumTest {
    private sum chekcStr;

    public sumTest() {
        this.chekcStr = new sum();
    }

    @Test
    public void testTotalEmptyString() {
        int expected = 0;
        assertEquals(expected, chekcStr.total(""));
    }

    @Test
    public void testTotalSingleChar() {
        int expected = 100;
        assertEquals(expected, chekcStr.total("d"));
    }

    @Test
    public void testTotalMultipleChars() {
        int expected = 265;
        assertEquals(expected, chekcStr.total("Add"));
```

```
<terminated> sumTest [JUnit] C:\Users\Hp\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.9.v20231028-0858\jre\bin\javaw.exe  (02-Dec-2023, 12:01:36 am – 12:01
```

Program No. 9

Write junit class for vending machine. The functions are as given below:

1. Accepts coins of 1,5,10,20 Cents.

2. Allow user to select products Coke (25), Pepsi (35), Soda (45)

3. Allow user to take refund by cancelling the request.

4. Return selected product and remaining change if any

5. Allow reset operation for vending machine supplier.

Code:

Bucket.java

```
package vending;

public class Bucket<E1, E2> {
    private E1 first;
    private E2 second;

    public Bucket(E1 first, E2 second){
        this.first = first;
        this.second = second;
    }

    public E1 getFirst(){
        return first;
    }

    public E2 getSecond(){
        return second;
    }
}
```

Coin.java

```
package vending;

public enum Coin {
    PENNY(1), NICKLE(5), DIME(10), QUARTER(25);

    private int denomination;

    private Coin(int denomination){
        this.denomination = denomination;
    }
```

```java
    public int getDenomination(){
        return denomination;
    }
}
```

Item.java

```java
package vending;

public enum Item{
    COKE("Coke", 25), PEPSI("Pepsi", 35), SODA("Soda", 45);

    private String name;
    private int price;

    private Item(String name, int price){
        this.name = name;
        this.price = price;
    }

    public String getName(){
        return name;
    }

    public long getPrice(){
        return price;
    }
}
```

NotFullPaidException.java

```java
package vending;
public class NotFullPaidException extends RuntimeException {
    /**
       *
       */
      private static final long serialVersionUID = 1L;
      private String message;
    private long remaining;

    public NotFullPaidException(String message, long remaining) {
        this.message = message;
        this.remaining = remaining;
    }

    public long getRemaining(){
        return remaining;
    }

    @Override
    public String getMessage(){
        return message + remaining;
    }

}
```

NotSufficientChangeException.java

```java
package vending;

import java.io.Serializable;

public class NotSufficientChangeException extends RuntimeException implements
Serializable {

    private static final long serialVersionUID = 1L;

    private String message;

    public NotSufficientChangeException(String message) {
        this.message = message;
    }

    @Override
    public String getMessage() {
        return message;
    }
}
```

Inventory.java

```java
package vending;
import java.util.HashMap;
import java.util.Map;


public class Inventory<T> {
    private Map<T, Integer> inventory = new HashMap<T, Integer>();

    public int getQuantity(T item){
        Integer value = inventory.get(item);
        return value == null? 0 : value ;
    }

    public void add(T item){
        int count = inventory.get(item);
        inventory.put(item, count+1);
    }

    public void deduct(T item) {
        if (hasItem(item)) {
            int count = inventory.get(item);
            inventory.put(item, count - 1);
        }
    }

    public boolean hasItem(T item){
        return getQuantity(item) > 0;
    }

    public void clear(){
        inventory.clear();
    }
    public void put(T item, int quantity) {
        inventory.put(item, quantity);
    }
}
```

SoldOutException.java

```java
package vending;
public class SoldOutException extends RuntimeException {
    /**
       *
       */
      private static final long serialVersionUID = 1L;
      private String message;

    public SoldOutException(String string) {
        this.message = string;
    }

    @Override
    public String getMessage(){
        return message;
    }

}
```

VendingMachine.java

```java
package vending;

import java.util.List;

public interface VendingMachine {
    public long selectItemAndGetPrice(Item item);
    public void insertCoin(Coin coin);
    public List<Coin> refund();
    public Bucket<Item, List<Coin>> collectItemAndChange();
    public void reset();
}
```

VendingMachineFactory.java

```java
package vending;


public class VendingMachineFactory {
    public static VendingMachine createVendingMachine() {
        return new VendingMachineImpl();
    }
}
```

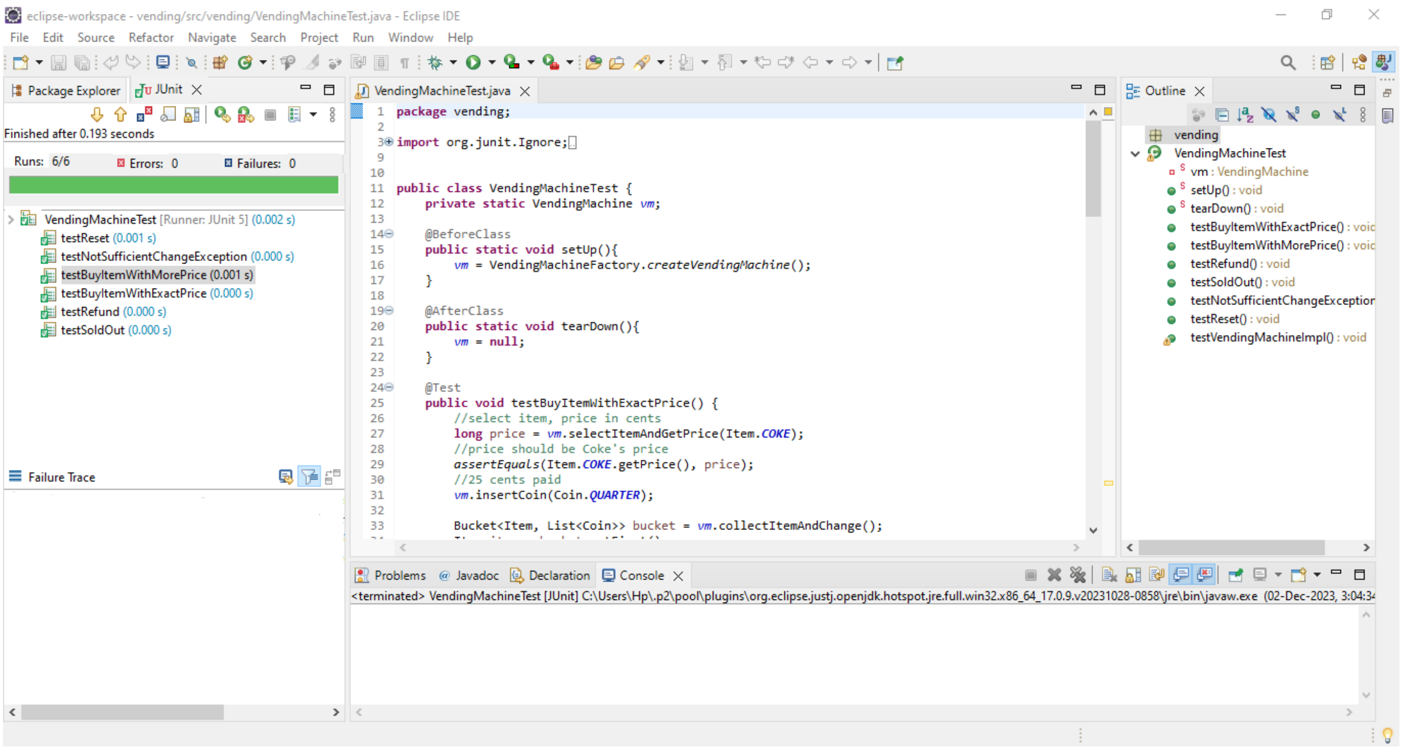
VendingMachineIml.java

```java
package vending;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;


public class VendingMachineImpl implements VendingMachine {
    private Inventory<Coin> cashInventory = new Inventory<Coin>();
    private Inventory<Item> itemInventory = new Inventory<Item>();
    private long totalSales;
    private Item currentItem;
    private long currentBalance;

    public VendingMachineImpl(){
        initialize();
    }

    private void initialize(){
        //initialize machine with 5 coins of each denomination
        //and 5 cans of each Item
        for(Coin c : Coin.values()){
            cashInventory.put(c, 5);
        }

        for(Item i : Item.values()){
            itemInventory.put(i, 5);
        }

    }

   @Override
    public long selectItemAndGetPrice(Item item) {
        if(itemInventory.hasItem(item)){
            currentItem = item;
            return currentItem.getPrice();
        }
        throw new SoldOutException("Sold Out, Please buy another item");
    }

    @Override
    public void insertCoin(Coin coin) {
        currentBalance = currentBalance + coin.getDenomination();
        cashInventory.add(coin);
    }

    @Override
    public Bucket<Item, List<Coin>> collectItemAndChange() {
        Item item = collectItem();
        totalSales = totalSales + currentItem.getPrice();

        List<Coin> change = collectChange();

        return new Bucket<Item, List<Coin>>(item, change);
    }

    private Item collectItem() throws NotSufficientChangeException,
            NotFullPaidException{
        if(isFullPaid()){
            if(hasSufficientChange()){
                itemInventory.deduct(currentItem);
                return currentItem;
            }
            throw new NotSufficientChangeException("Not Sufficient change in
Inventory");

        }
        long remainingBalance = currentItem.getPrice() - currentBalance;
        throw new NotFullPaidException("Price not full paid, remaining : ",
                                          remainingBalance);
    }

    private List<Coin> collectChange() {
        long changeAmount = currentBalance - currentItem.getPrice();
        List<Coin> change = getChange(changeAmount);
        updateCashInventory(change);
        currentBalance = 0;
        currentItem = null;
        return change;
    }

    @Override
    public List<Coin> refund(){
        List<Coin> refund = getChange(currentBalance);
        updateCashInventory(refund);
        currentBalance = 0;
        currentItem = null;
        return refund;
    }


    private boolean isFullPaid() {
        if(currentBalance >= currentItem.getPrice()){
            return true;
        }
        return false;
    }


    private List<Coin> getChange(long amount) throws NotSufficientChangeException{
        List<Coin> changes = Collections.emptyList();

        if(amount > 0){
            changes = new ArrayList<Coin>();
            long balance = amount;
            while(balance > 0){
                if(balance >= Coin.QUARTER.getDenomination()
                            && cashInventory.hasItem(Coin.QUARTER)){
                    changes.add(Coin.QUARTER);
                    balance = balance - Coin.QUARTER.getDenomination();
                    continue;

                }else if(balance >= Coin.DIME.getDenomination()
                                 && cashInventory.hasItem(Coin.DIME)) {
                    changes.add(Coin.DIME);
                    balance = balance - Coin.DIME.getDenomination();
                    continue;
                }else if(balance >= Coin.NICKLE.getDenomination()
                                 && cashInventory.hasItem(Coin.NICKLE)) {
                    changes.add(Coin.NICKLE);
                    balance = balance - Coin.NICKLE.getDenomination();
                    continue;

                }else if(balance >= Coin.PENNY.getDenomination()
                                 && cashInventory.hasItem(Coin.PENNY)) {
                    changes.add(Coin.PENNY);
                    balance = balance - Coin.PENNY.getDenomination();
                    continue;

                }else{
                    throw new NotSufficientChangeException("NotSufficientChange,Please
try another product");
                }
            }
        }

        return changes;
    }

    @Override
    public void reset(){
        cashInventory.clear();
        itemInventory.clear();
        totalSales = 0;
        currentItem = null;
        currentBalance = 0;
    }

    public void printStats(){
        System.out.println("Total Sales : " + totalSales);
        System.out.println("Current Item Inventory : " + itemInventory);
        System.out.println("Current Cash Inventory : " + cashInventory);
    }


    private boolean hasSufficientChange(){
        return hasSufficientChangeForAmount(currentBalance - currentItem.getPrice());
    }

    private boolean hasSufficientChangeForAmount(long amount){
        boolean hasChange = true;
        try{
            getChange(amount);
        }catch(NotSufficientChangeException nsce){
            return hasChange = false;
        }

        return hasChange;
    }

    private void updateCashInventory(List<Coin> change) {
        for (Coin c : change) {
            cashInventory.deduct(c);
        }
    }
    public long getTotalSales(){
        return totalSales;
    }

}
```

VendingMachineTest.java

```java
package vending;

import org.junit.Ignore;
import java.util.List;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;


public class VendingMachineTest {
    private static VendingMachine vm;

    @BeforeClass
    public static void setUp(){
        vm = VendingMachineFactory.createVendingMachine();
    }

    @AfterClass
    public static void tearDown(){
        vm = null;
    }

    @Test
    public void testBuyItemWithExactPrice() {
        //select item, price in cents
        long price = vm.selectItemAndGetPrice(Item.COKE);
        //price should be Coke's price
        assertEquals(Item.COKE.getPrice(), price);
        //25 cents paid
        vm.insertCoin(Coin.QUARTER);

        Bucket<Item, List<Coin>> bucket = vm.collectItemAndChange();
        Item item = bucket.getFirst();
        List<Coin> change = bucket.getSecond();

        //should be Coke
        assertEquals(Item.COKE, item);
        //there should not be any change
        assertTrue(change.isEmpty());
    }
    @Test
    public void testBuyItemWithMorePrice(){
        long price = vm.selectItemAndGetPrice(Item.SODA);
        assertEquals(Item.SODA.getPrice(), price);

        vm.insertCoin(Coin.QUARTER);
        vm.insertCoin(Coin.QUARTER);
        Bucket<Item, List<Coin>> bucket = vm.collectItemAndChange();
        Item item = bucket.getFirst();
        List<Coin> change = bucket.getSecond();

        //should be Coke
        assertEquals(Item.SODA, item);
        //there should not be any change
        assertTrue(!change.isEmpty());
        //comparing change
        assertEquals(50 - Item.SODA.getPrice(), getTotal(change));

    }


    @Test
    public void testRefund(){
        long price = vm.selectItemAndGetPrice(Item.PEPSI);
        assertEquals(Item.PEPSI.getPrice(), price);
        vm.insertCoin(Coin.DIME);
        vm.insertCoin(Coin.NICKLE);
        vm.insertCoin(Coin.PENNY);
        vm.insertCoin(Coin.QUARTER);

        assertEquals(41, getTotal(vm.refund()));
    }

    @Test(expected=SoldOutException.class)
    public void testSoldOut(){
        for (int i = 0; i < 5; i++) {
            vm.selectItemAndGetPrice(Item.COKE);
            vm.insertCoin(Coin.QUARTER);
            vm.collectItemAndChange();
        }

    }

    @Test(expected=NotSufficientChangeException.class)
    public void testNotSufficientChangeException(){
        for (int i = 0; i < 5; i++) {
            vm.selectItemAndGetPrice(Item.SODA);
            vm.insertCoin(Coin.QUARTER);
            vm.insertCoin(Coin.QUARTER);
            vm.collectItemAndChange();

            vm.selectItemAndGetPrice(Item.PEPSI);
            vm.insertCoin(Coin.QUARTER);
            vm.insertCoin(Coin.QUARTER);
            vm.collectItemAndChange();
        }
    }


    @Test(expected=SoldOutException.class)
    public void testReset(){
        VendingMachine vmachine = VendingMachineFactory.createVendingMachine();
        vmachine.reset();

        vmachine.selectItemAndGetPrice(Item.COKE);
    }

    @Ignore
    public void testVendingMachineImpl(){
        VendingMachineImpl vm = new VendingMachineImpl();
    }

    private long getTotal(List<Coin> change){
        long total = 0;
        for(Coin c : change){
            total = total + c.getDenomination();
        }
        return total;
    }
}
```

Output:



eclipse-workspace - vending/src/vending/VendingMachineTest.java - Eclipse IDE

```
Finished after 0.193 seconds
Runs: 6/6    Errors: 0    Failures: 0

VendingMachineTest [Runner: JUnit 5] (0.002 s)
    testReset (0.001 s)
    testNotSufficientChangeException (0.000 s)
    testBuyItemWithMorePrice (0.001 s)
    testBuyItemWithExactPrice (0.000 s)
    testRefund (0.000 s)
    testSoldOut (0.000 s)
```

```java
package vending;

import org.junit.Ignore;


public class VendingMachineTest {
    private static VendingMachine vm;

    @BeforeClass
    public static void setUp(){
        vm = VendingMachineFactory.createVendingMachine();
    }

    @AfterClass
    public static void tearDown(){
        vm = null;
    }

    @Test
    public void testBuyItemWithExactPrice() {
        //select item, price in cents
        long price = vm.selectItemAndGetPrice(Item.COKE);
        //price should be Coke's price
        assertEquals(Item.COKE.getPrice(), price);
        //25 cents paid
        vm.insertCoin(Coin.QUARTER);

        Bucket<Item, List<Coin>> bucket = vm.collectItemAndChange();
```

```
vending
VendingMachineTest
    vm : VendingMachine
    setUp() : void
    tearDown() : void
    testBuyItemWithExactPrice() : void
    testBuyItemWithMorePrice() : void
    testRefund() : void
    testSoldOut() : void
    testNotSufficientChangeException
    testReset() : void
    testVendingMachineImpl() : void
```

```
<terminated> VendingMachineTest [JUnit] C:\Users\Hp\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.9.v20231028-0858\jre\bin\javaw.exe  (02-Dec-2023, 3:04:34
```