Unit Testing in Java

Advanced Programming, Dr. Mojataba Vahidi Asl

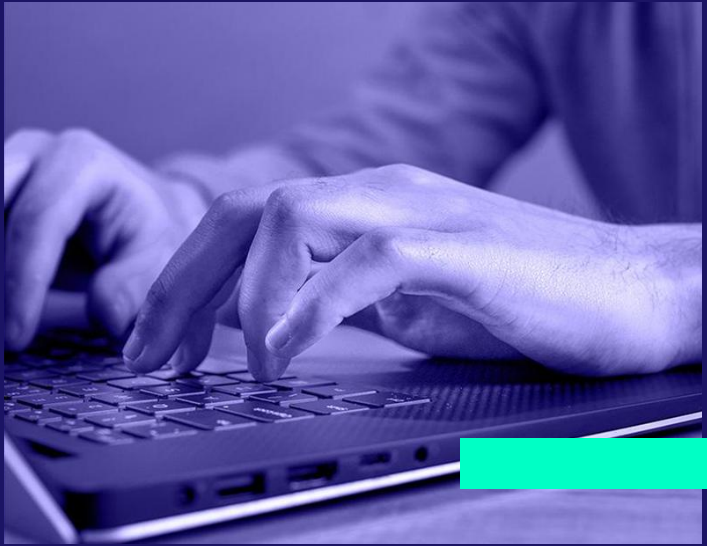
Seyyed Mohammad Hosseini, Spring 2025

TABLE OF CONTENTS

TABLE OF CONTENTS

INTRODUCTION

Unit Testing is a software testing method where individual components or functions of a program are tested in isolation to ensure they work as expected. It is typically automated and focuses on verifying the smallest testable parts of an application, such as functions, methods, or classes.



01

History Of Unit Testing

History

- **1. Early Days (1950s–1970s): Manual Testing Era**
- In the early days of computing, software testing was primarily manual.
- Debugging and testing were often done by the same developers without a structured approach.
- The need for systematic testing grew as software became more complex.

History

- **2. Structured Programming & Initial Automation (1970s–1980s)**
- The rise of structured programming (e.g., Pascal, C) led to better-defined coding practices.
- Developers began writing test cases to verify functions and modules.
- The concept of **unit testing** emerged as a formal technique to test individual components.

History

- **3. The Rise of Object-Oriented Programming (1990s)**
- With the popularity of object-oriented languages like **Java and C++**, unit testing frameworks started appearing.
- **JUnit (1997)** by Kent Beck and Erich Gamma became one of the first widely used unit testing frameworks, setting the standard for automated testing.

History

- **4. Agile and Test-Driven Development (2000s–2010s)**
- The Agile methodology and **Test-Driven Development (TDD)** popularized the idea of writing tests before writing code.
- Many unit testing frameworks emerged, including **NUnit (.NET), PyTest (Python), and RSpec (Ruby).**
- Continuous Integration (CI) tools began integrating unit testing into development workflows.

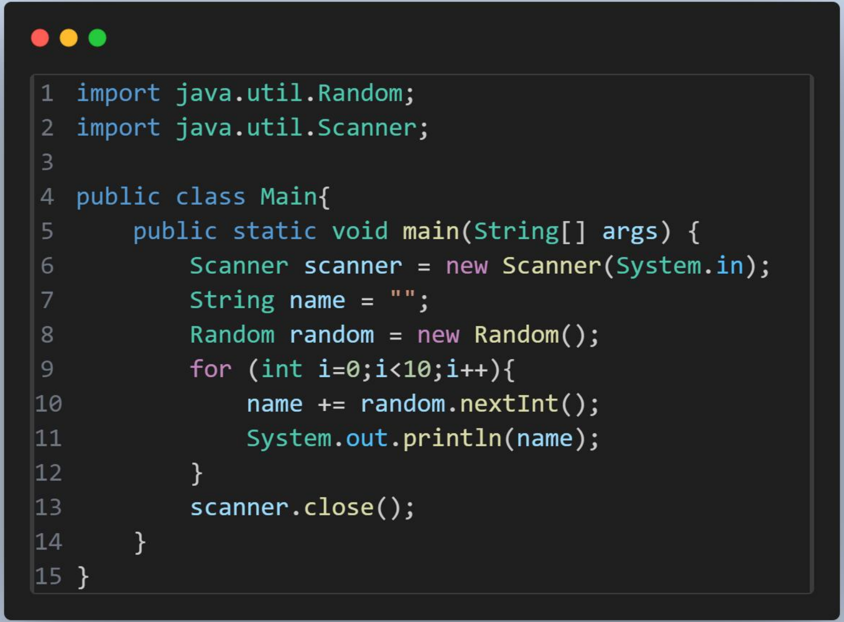
History

- **5. Modern Unit Testing (2010s–Present)**
- Unit testing is now an essential part of **DevOps, CI/CD pipelines, and modern software engineering.**
- Advanced tools like **Jest (JavaScript), Mocha, xUnit, and Google Test** continue to improve automated testing.
- **Mutation testing and AI-driven testing** are emerging to enhance test effectiveness.

02

Traditional Testing

Traditional Testing



```java
import java.util.Random;
import java.util.Scanner;

public class Main{
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String name = "";
        Random random = new Random();
        for (int i=0;i<10;i++){
            name += random.nextInt();
            System.out.println(name);
        }
        scanner.close();
    }
}
```

03

Weaknesses of the Traditional Approach

Weakness

Programmers should check the result manually.

This method goes against software reuse principles.

You may have a memory leak when you test your code manually.

This is not efficient because you have to check step by step instead of in parallel.

04

Key Features of Unit Testing

Key Features

This is performed automatically

Use can use it multiple times

You can see the result and the code that was written incorrectly in the main code.

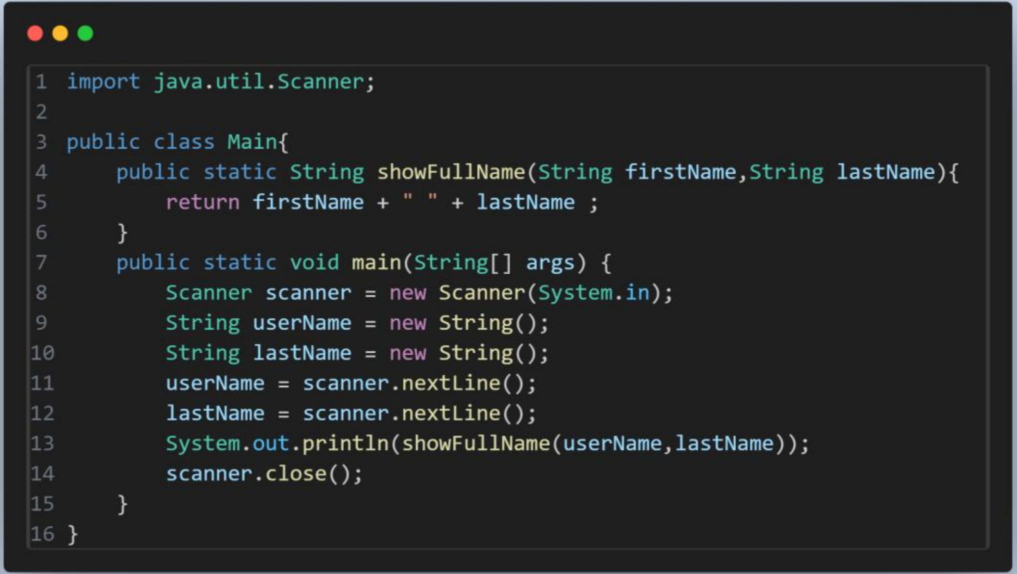
They are efficient because they execute in parallel and provide a clear log of your mistakes.

05

JUNIT Samples

Example

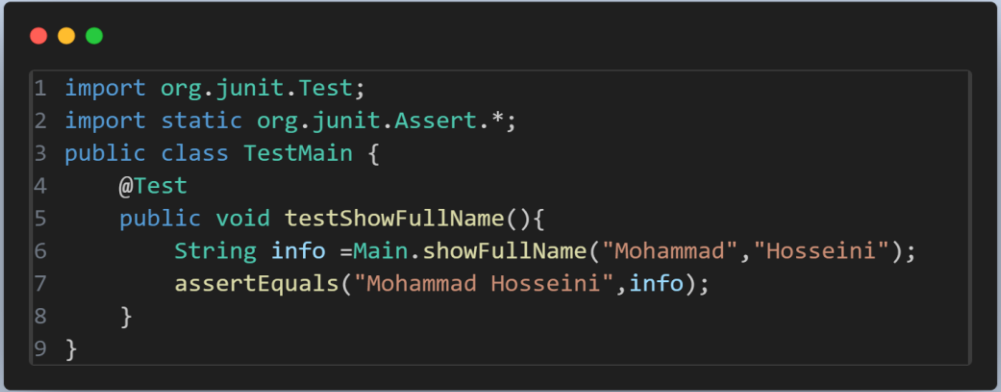
How we can test this code ?



```java
import java.util.Scanner;

public class Main{
    public static String showFullName(String firstName,String lastName){
        return firstName + " " + lastName ;
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String userName = new String();
        String lastName = new String();
        userName = scanner.nextLine();
        lastName = scanner.nextLine();
        System.out.println(showFullName(userName,lastName));
        scanner.close();
    }
}
```

Example

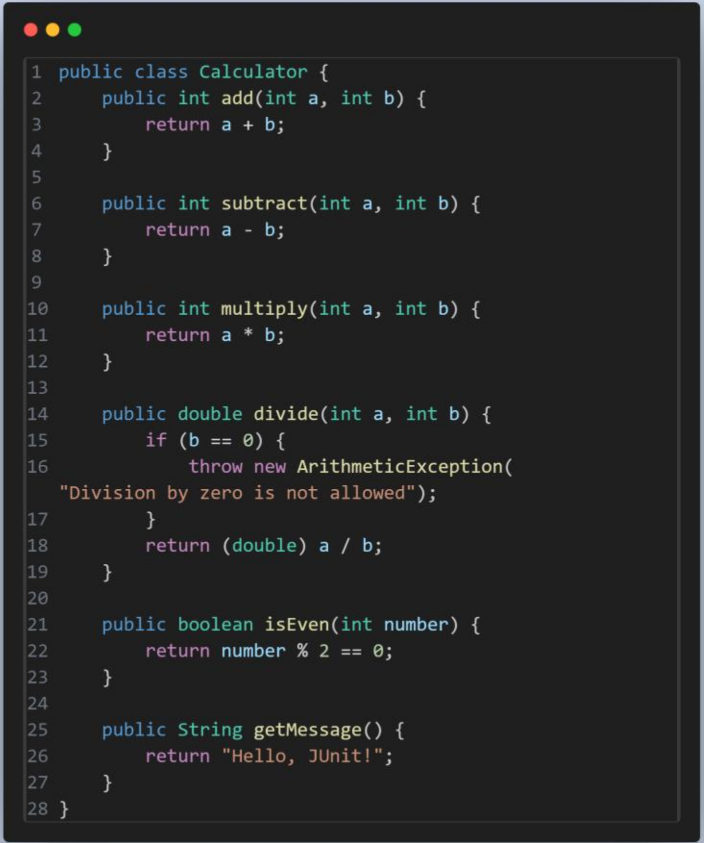
Your first JUNIT test !!!



```java
import org.junit.Test;
import static org.junit.Assert.*;
public class TestMain {
    @Test
    public void testShowFullName(){
        String info =Main.showFullName("Mohammad","Hosseini");
        assertEquals("Mohammad Hosseini",info);
    }
}
```

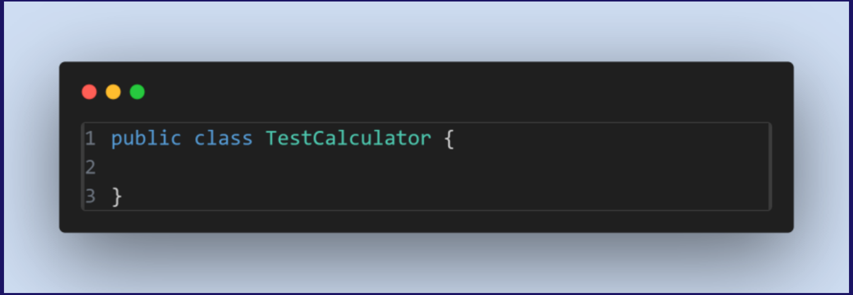
SecondExample

How we can test this code ?

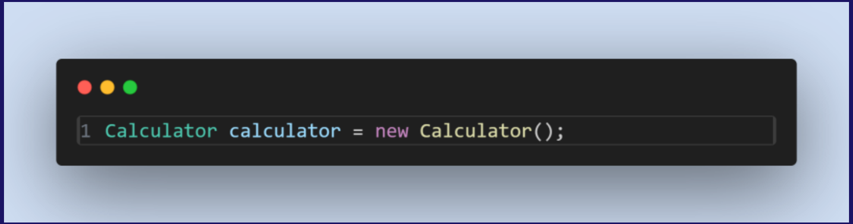


```java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

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

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

    public double divide(int a, int b) {
        if (b == 0) {
            throw new ArithmeticException(
"Division by zero is not allowed");
        }
        return (double) a / b;
    }

    public boolean isEven(int number) {
        return number % 2 == 0;
    }

    public String getMessage() {
        return "Hello, JUnit!";
    }
}
```

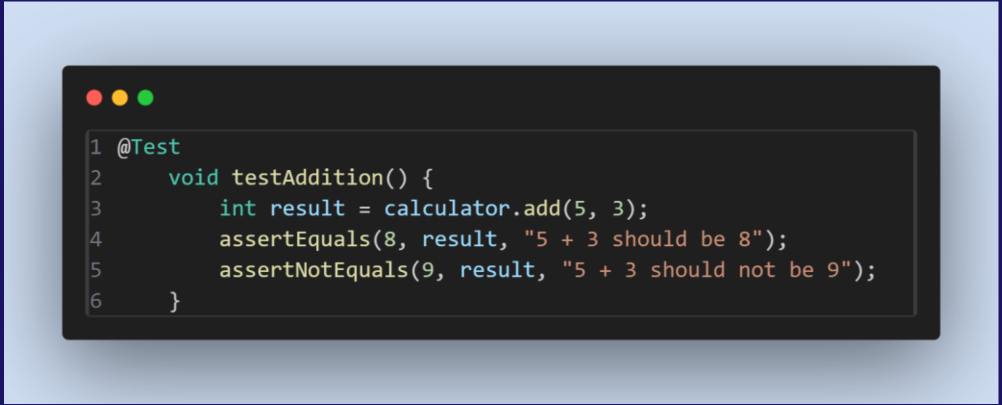
SecondExample Test



```
public class TestCalculator {

}
```

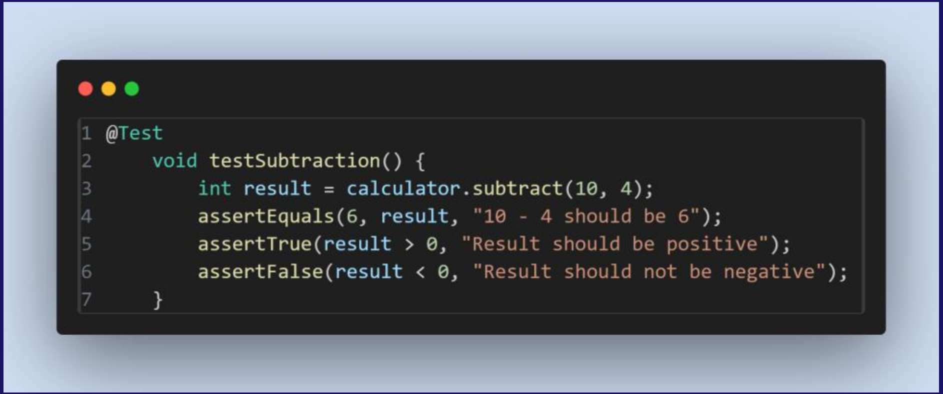


```
Calculator calculator = new Calculator();
```

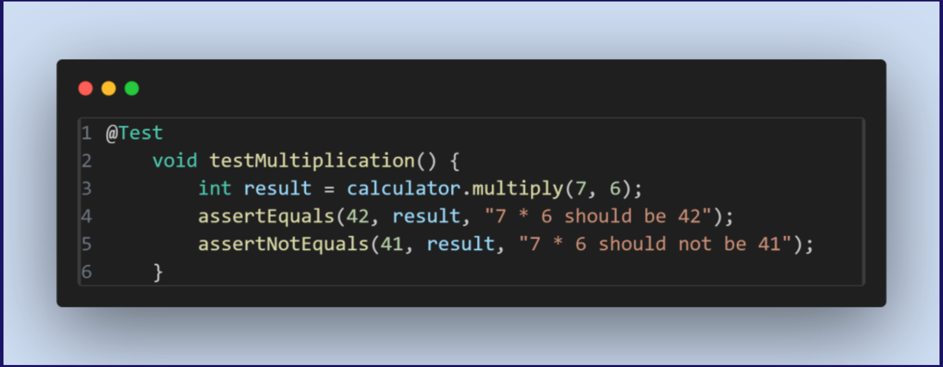


```
@Test
    void testAddition() {
        int result = calculator.add(5, 3);
        assertEquals(8, result, "5 + 3 should be 8");
        assertNotEquals(9, result, "5 + 3 should not be 9");
    }
```

SecondExample Test

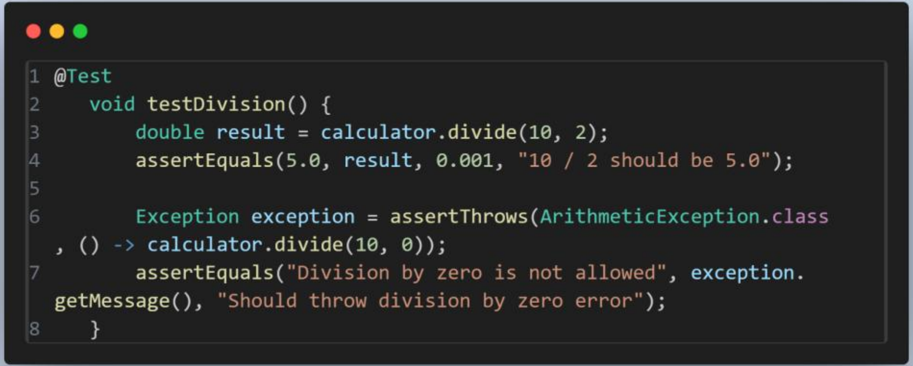


```java
@Test
    void testSubtraction() {
        int result = calculator.subtract(10, 4);
        assertEquals(6, result, "10 - 4 should be 6");
        assertTrue(result > 0, "Result should be positive");
        assertFalse(result < 0, "Result should not be negative");
    }
```

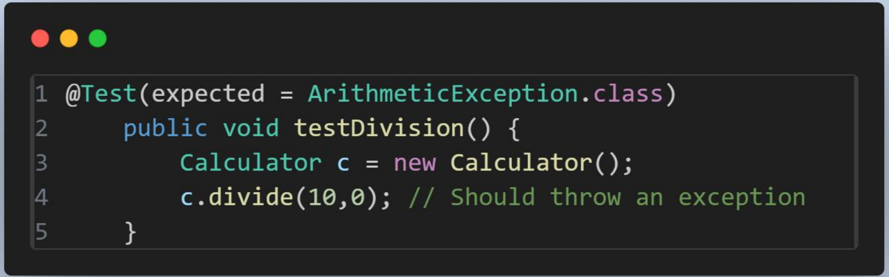


```java
@Test
    void testMultiplication() {
        int result = calculator.multiply(7, 6);
        assertEquals(42, result, "7 * 6 should be 42");
        assertNotEquals(41, result, "7 * 6 should not be 41");
    }
```

SecondExample Test(Optional)

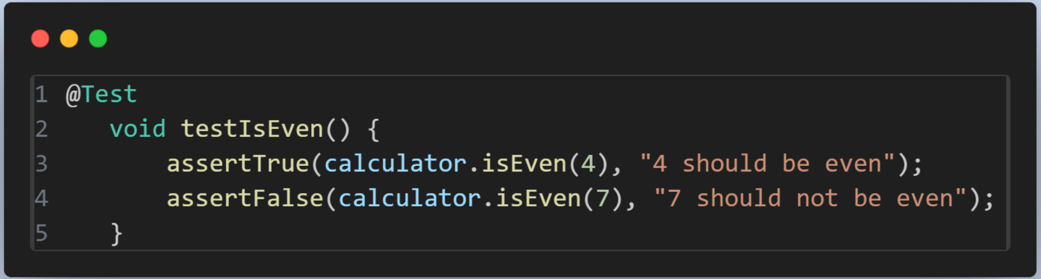


```java
@Test
   void testDivision() {
       double result = calculator.divide(10, 2);
       assertEquals(5.0, result, 0.001, "10 / 2 should be 5.0");

       Exception exception = assertThrows(ArithmeticException.class
, () -> calculator.divide(10, 0));
       assertEquals("Division by zero is not allowed", exception.
getMessage(), "Should throw division by zero error");
   }
```

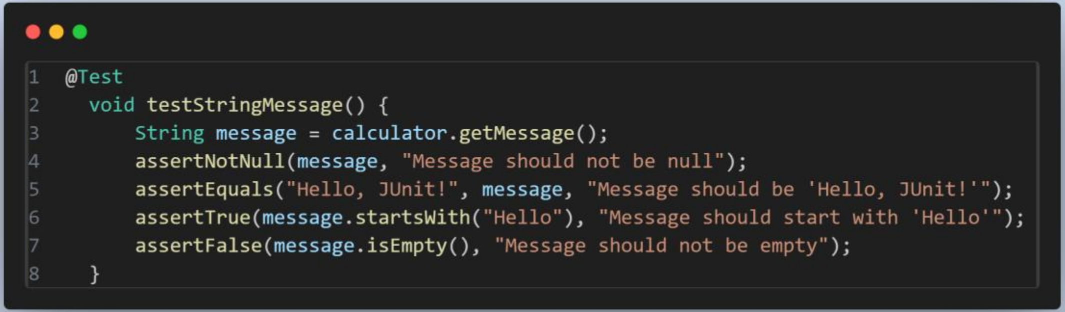


```java
@Test(expected = ArithmeticException.class)
    public void testDivision() {
        Calculator c = new Calculator();
        c.divide(10,0); // Should throw an exception
    }
```

SecondExample Test

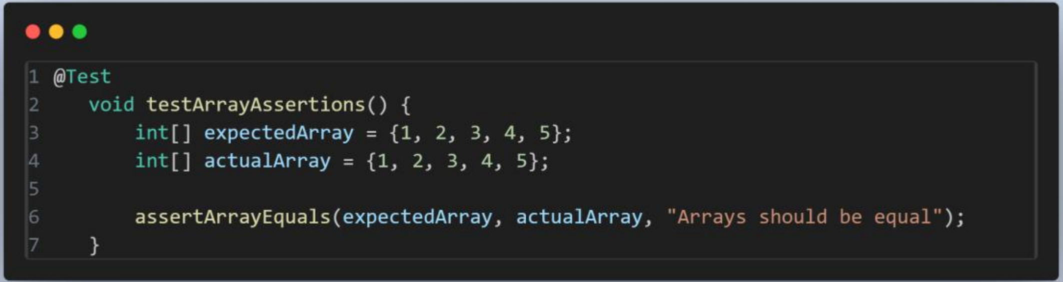


```java
@Test
  void testIsEven() {
      assertTrue(calculator.isEven(4), "4 should be even");
      assertFalse(calculator.isEven(7), "7 should not be even");
  }
```



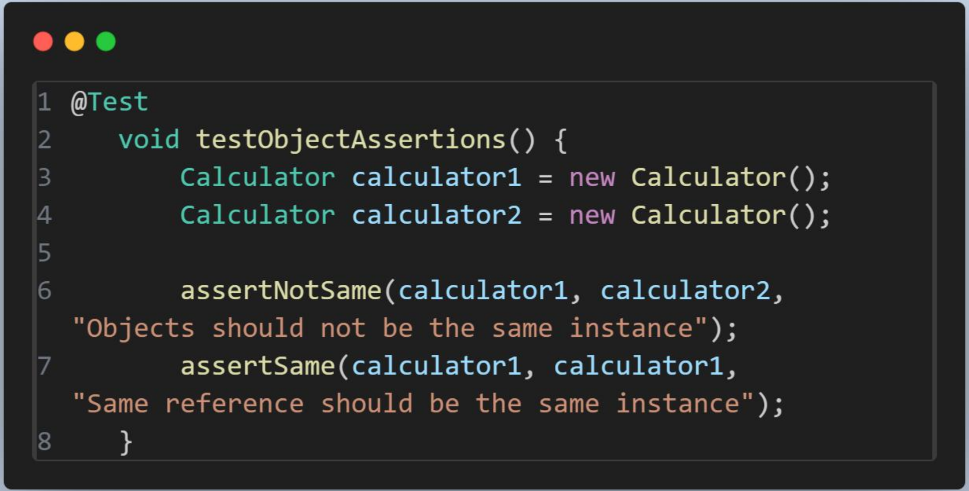
```java
@Test
  void testStringMessage() {
      String message = calculator.getMessage();
      assertNotNull(message, "Message should not be null");
      assertEquals("Hello, JUnit!", message, "Message should be 'Hello, JUnit!'");
      assertTrue(message.startsWith("Hello"), "Message should start with 'Hello'");
      assertFalse(message.isEmpty(), "Message should not be empty");
  }
```

SecondExample Test



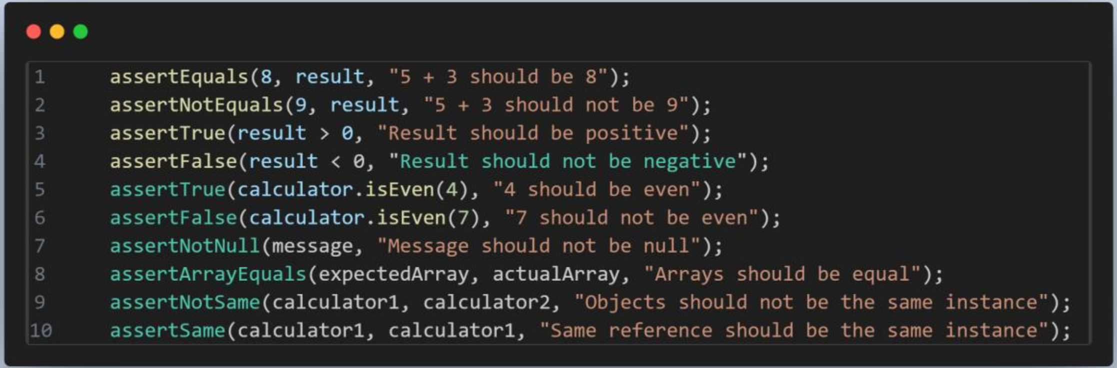
```java
@Test
    void testArrayAssertions() {
        int[] expectedArray = {1, 2, 3, 4, 5};
        int[] actualArray = {1, 2, 3, 4, 5};

        assertArrayEquals(expectedArray, actualArray, "Arrays should be equal");
    }
```

SecondExample Test



```java
@Test
    void testObjectAssertions() {
        Calculator calculator1 = new Calculator();
        Calculator calculator2 = new Calculator();

        assertNotSame(calculator1, calculator2,
 "Objects should not be the same instance");
        assertSame(calculator1, calculator1,
 "Same reference should be the same instance");
    }
```

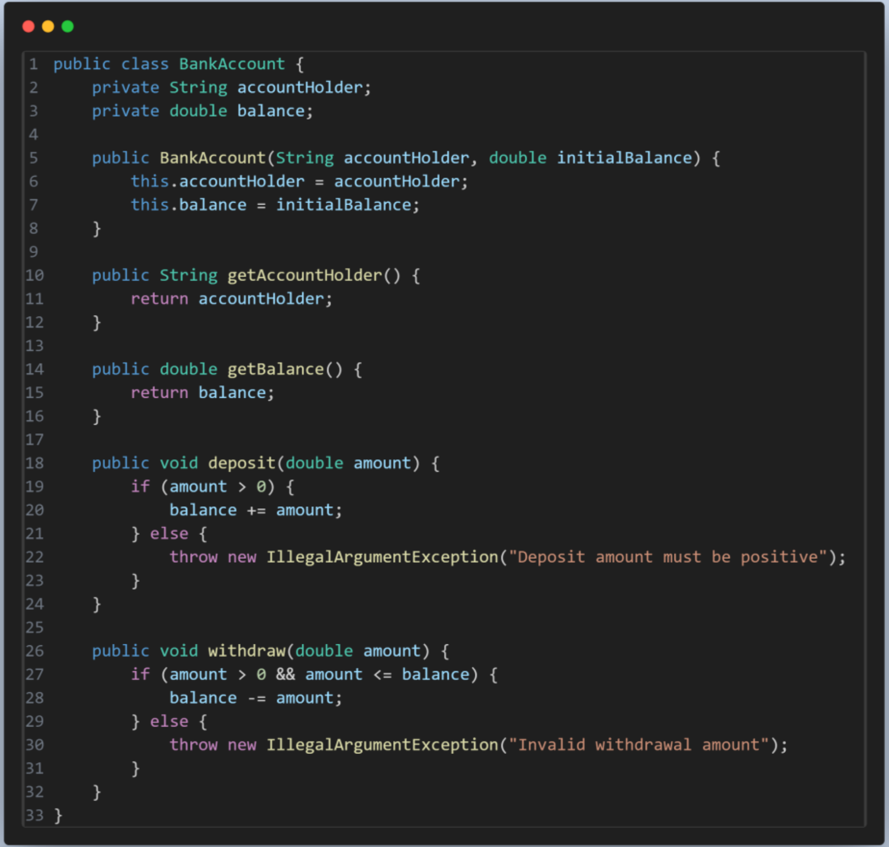
Summary



```
assertEquals(8, result, "5 + 3 should be 8");
assertNotEquals(9, result, "5 + 3 should not be 9");
assertTrue(result > 0, "Result should be positive");
assertFalse(result < 0, "Result should not be negative");
assertTrue(calculator.isEven(4), "4 should be even");
assertFalse(calculator.isEven(7), "7 should not be even");
assertNotNull(message, "Message should not be null");
assertArrayEquals(expectedArray, actualArray, "Arrays should be equal");
assertNotSame(calculator1, calculator2, "Objects should not be the same instance");
assertSame(calculator1, calculator1, "Same reference should be the same instance");
```

ThirdExample

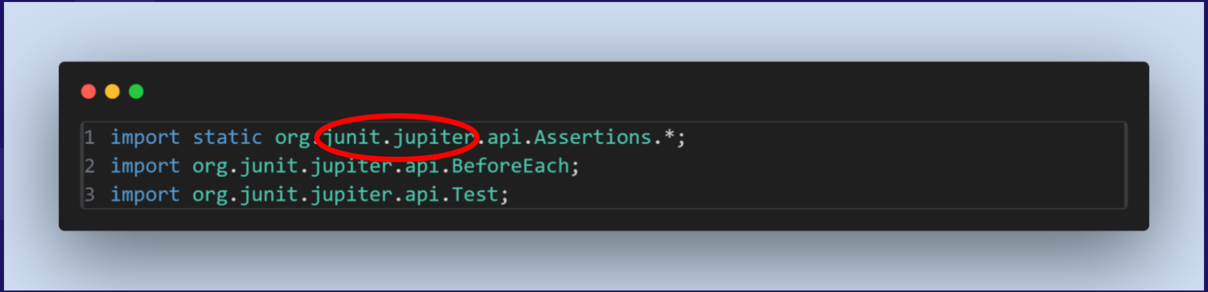
How we can test this code ?



```java
public class BankAccount {
    private String accountHolder;
    private double balance;

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

    public String getAccountHolder() {
        return accountHolder;
    }

    public double getBalance() {
        return balance;
    }

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

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        } else {
            throw new IllegalArgumentException("Invalid withdrawal amount");
        }
    }
}
```

ThirdExample Test

JUnit Jupiter is the testing framework introduced in JUnit 5 for writing and running unit tests in Java. It provides a modern API with annotations like **@Test**, **@BeforeEach**, and **@AfterEach**, along with advanced features like parameterized tests and test lifecycle control. It improves upon JUnit 4 by **offering better extensibility**, improved assertions, and integration with other tools.

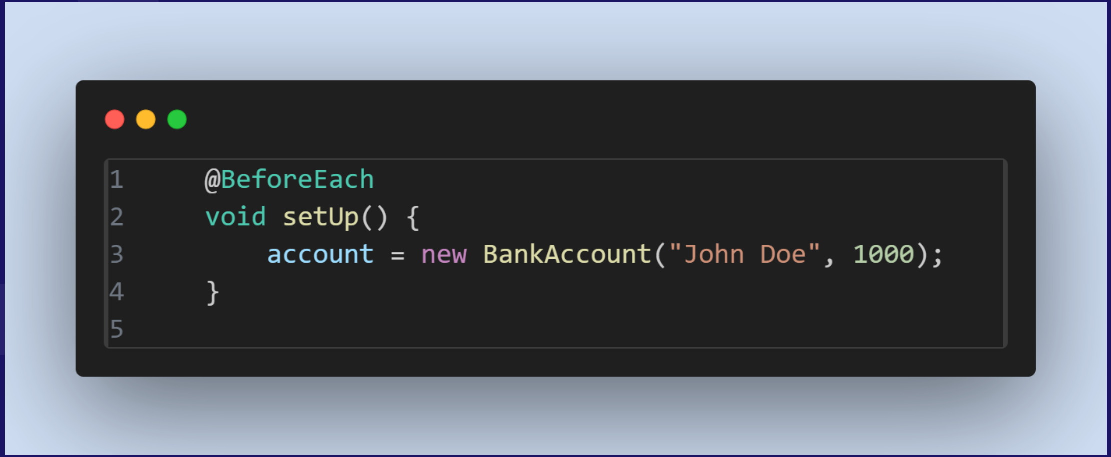


```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
```

ThirdExample Test

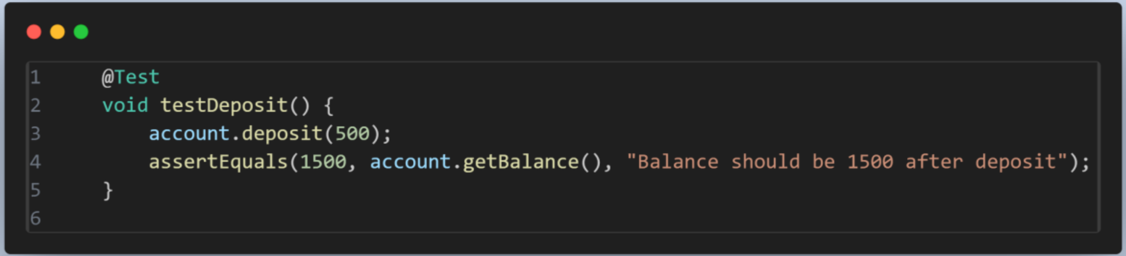
1. @BeforeEach

   Runs before each test method in the test class.
   Used to set up common test data or resources.



```java
    @BeforeEach
    void setUp() {
        account = new BankAccount("John Doe", 1000);
    }

```

ThirdExample Test

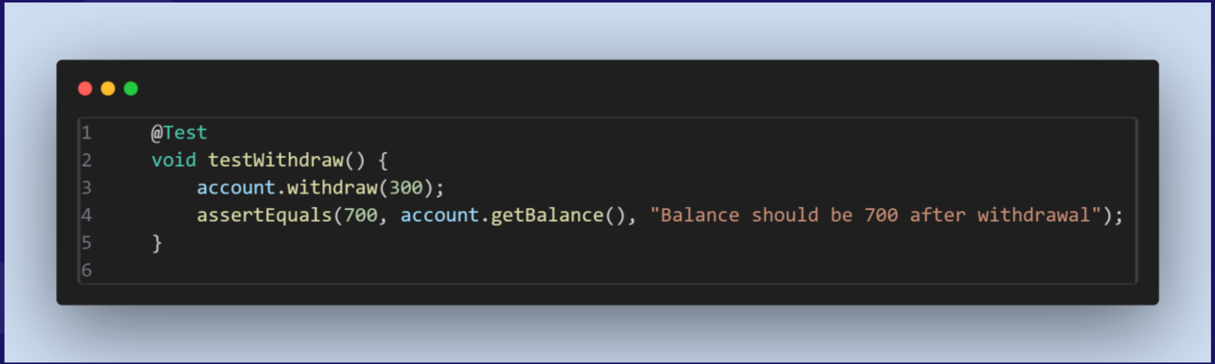


```java
    @Test
    void testDeposit() {
        account.deposit(500);
        assertEquals(1500, account.getBalance(), "Balance should be 1500 after deposit");
    }

```

ThirdExample Test

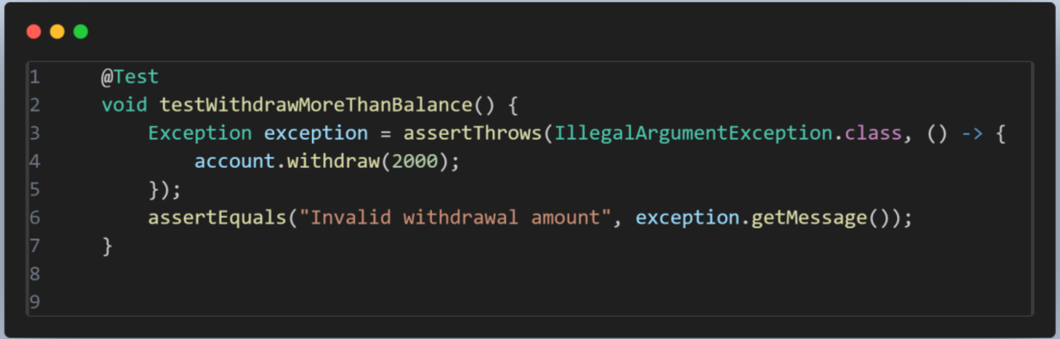
Usage of @BeforeEach :

We instantiate a user with 1000 dollar.
The 500 dollar in previous slide doesn't effect on this test.



```java
    @Test
    void testWithdraw() {
        account.withdraw(300);
        assertEquals(700, account.getBalance(), "Balance should be 700 after withdrawal");
    }

```

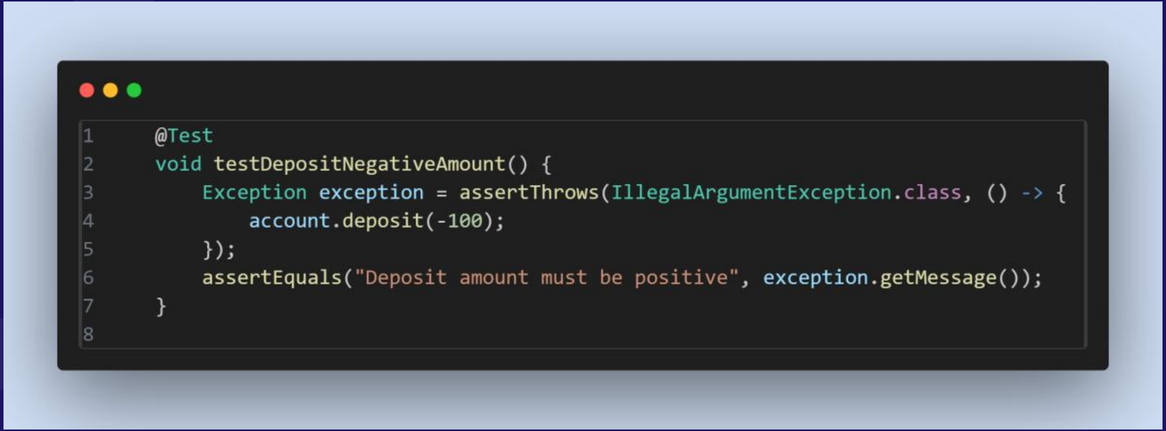
ThirdExample Test



```java
    @Test
    void testWithdrawMoreThanBalance() {
        Exception exception = assertThrows(IllegalArgumentException.class, () -> {
            account.withdraw(2000);
        });
        assertEquals("Invalid withdrawal amount", exception.getMessage());
    }
```

ThirdExample Test

Testing an exception

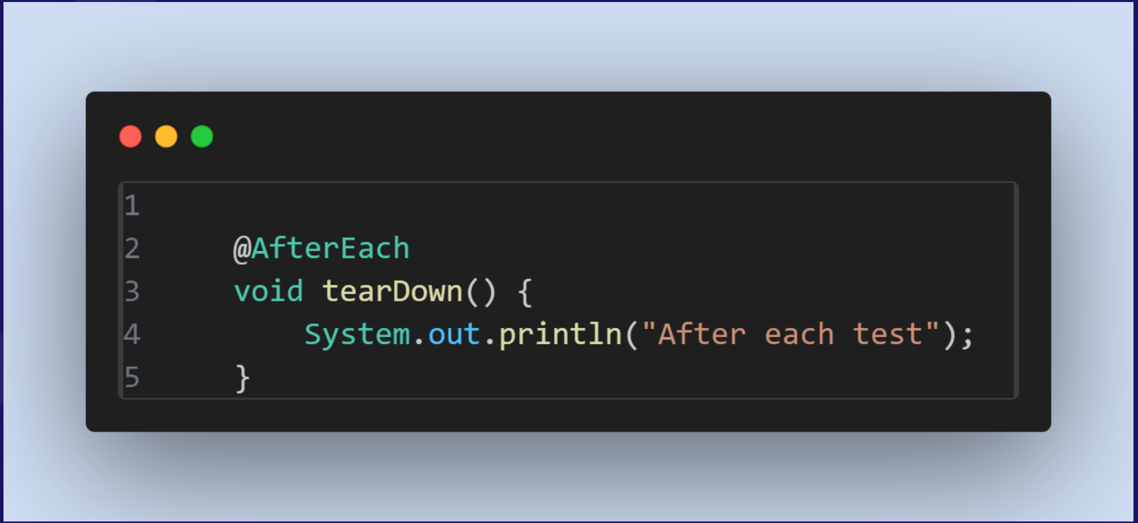


```java
@Test
void testDepositNegativeAmount() {
    Exception exception = assertThrows(IllegalArgumentException.class, () -> {
        account.deposit(-100);
    });
    assertEquals("Deposit amount must be positive", exception.getMessage());
}
```

Supplementary point

2. @AfterEach

Runs after each test method in the test class.
Used to clean up resources or reset states.

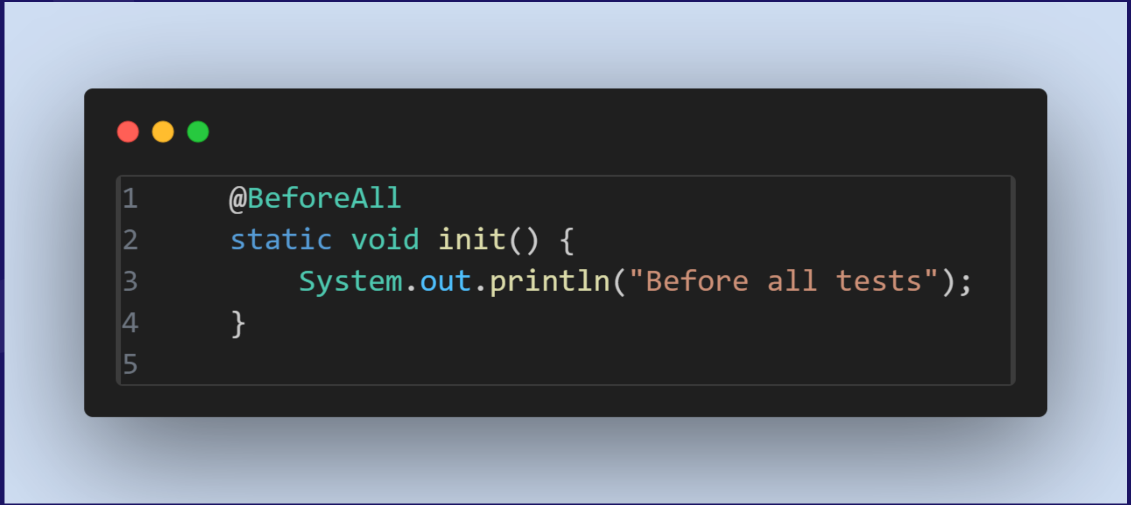


```java

    @AfterEach
    void tearDown() {
        System.out.println("After each test");
    }
```

Supplementary point

3. @BeforeAll

Runs once before all test methods in the class.
Typically used for expensive setup (e.g., database connections).
Must be static or in a non-static method inside a
@TestInstance(Lifecycle.PER_CLASS).

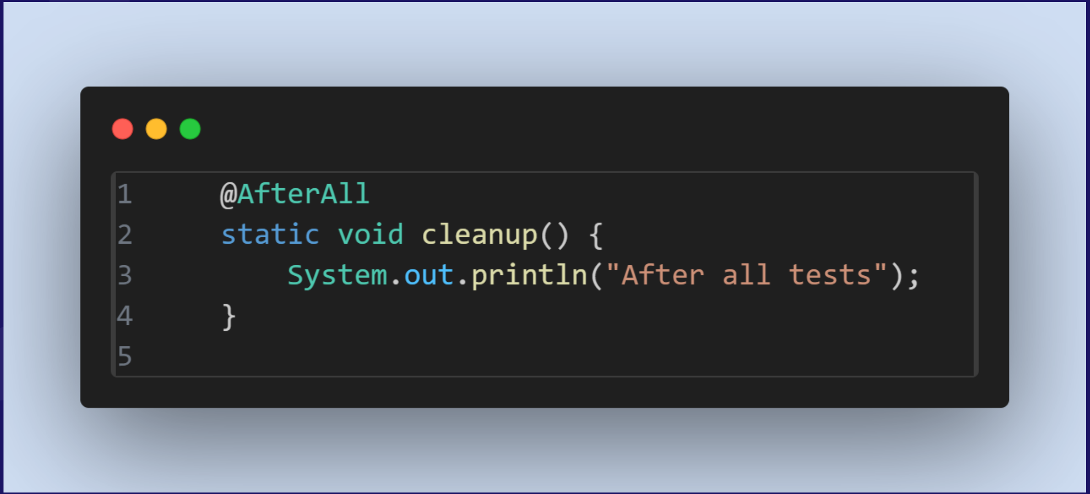


```java
@BeforeAll
static void init() {
    System.out.println("Before all tests");
}

```

Supplementary point

4. @AfterAll

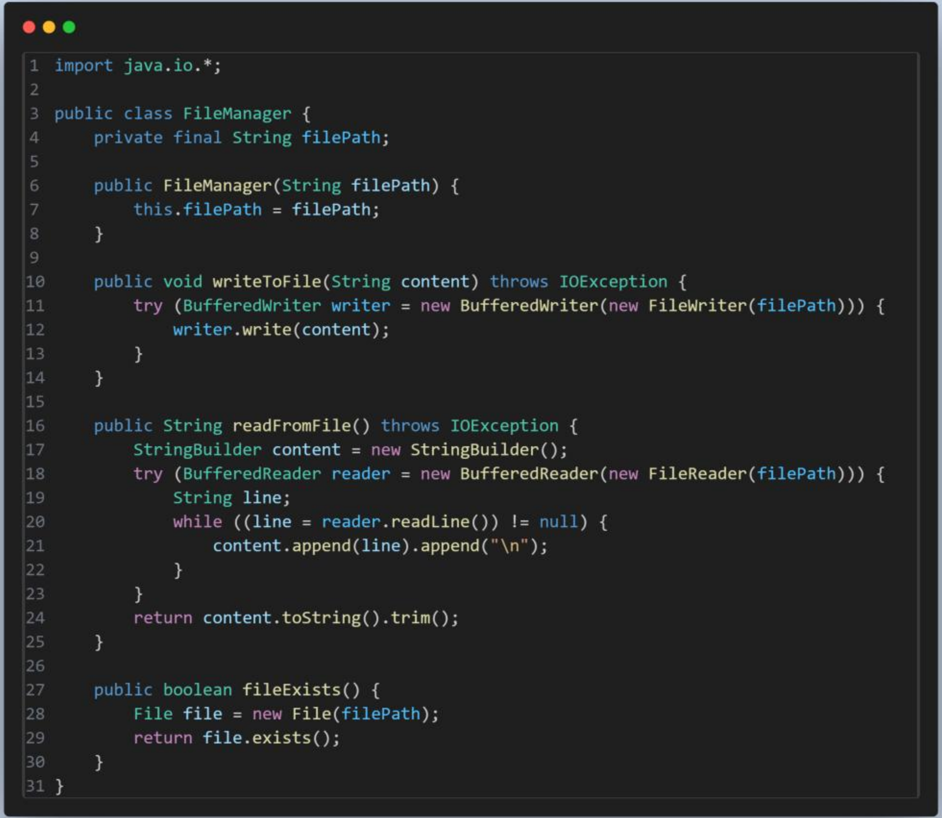
Runs once after all test methods in the class.
Used for cleanup tasks (e.g., closing database connections).
Must be static or in a non-static method inside a TestInstance(Lifecycle.PER_CLASS).



```java
@AfterAll
static void cleanup() {
    System.out.println("After all tests");
}

```

FourthExample (Optional)

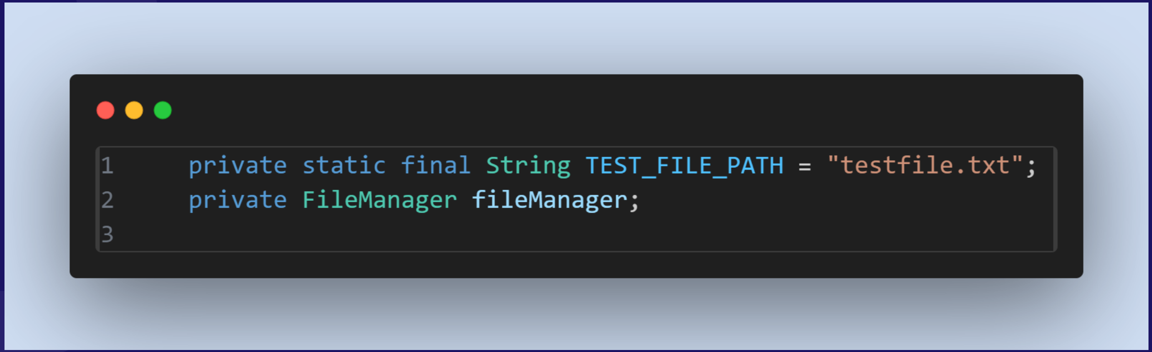
How we can test this code ?



```java
import java.io.*;

public class FileManager {
    private final String filePath;

    public FileManager(String filePath) {
        this.filePath = filePath;
    }

    public void writeToFile(String content) throws IOException {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath))) {
            writer.write(content);
        }
    }

    public String readFromFile() throws IOException {
        StringBuilder content = new StringBuilder();
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = reader.readLine()) != null) {
                content.append(line).append("\n");
            }
        }
        return content.toString().trim();
    }

    public boolean fileExists() {
        File file = new File(filePath);
        return file.exists();
    }
}
```

FourthExample Test(Optional)

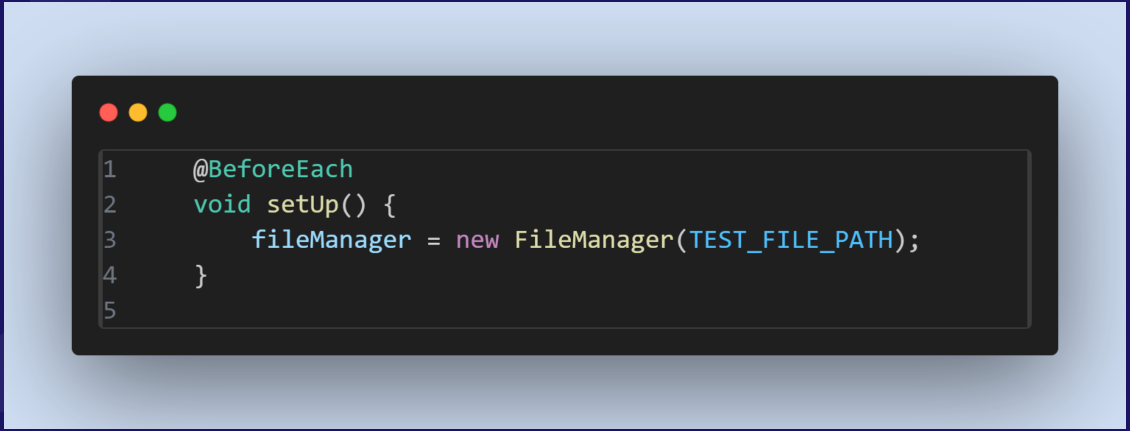
We don't put these two lines of code in @BeforeEach because they are the same in every test. For instance, the file path is unique and doesn't change for each test.



```java
    private static final String TEST_FILE_PATH = "testfile.txt";
    private FileManager fileManager;

```

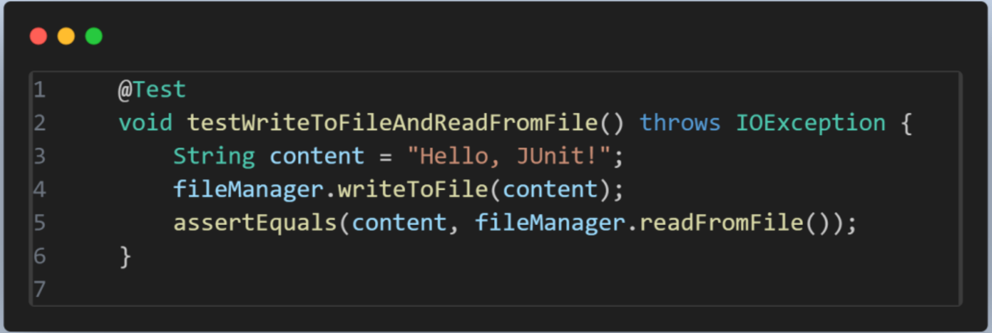
FourthExample Test(Optional)

We open a specific file right before each test.



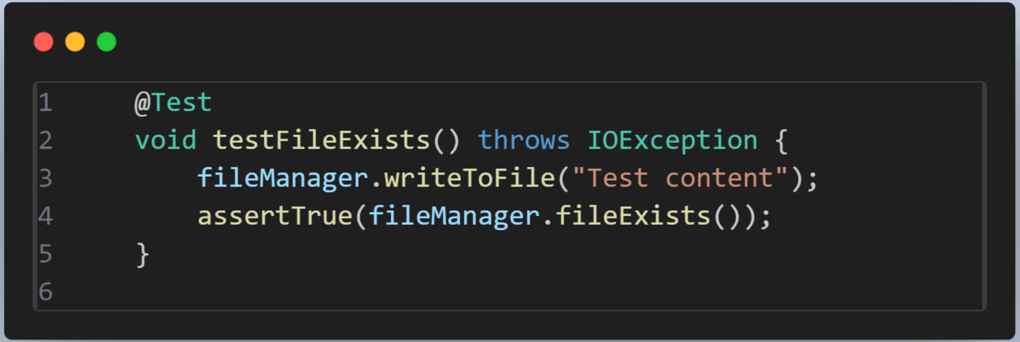
```
@BeforeEach
void setUp() {
    fileManager = new FileManager(TEST_FILE_PATH);
}

```

FourthExample Test(Optional)



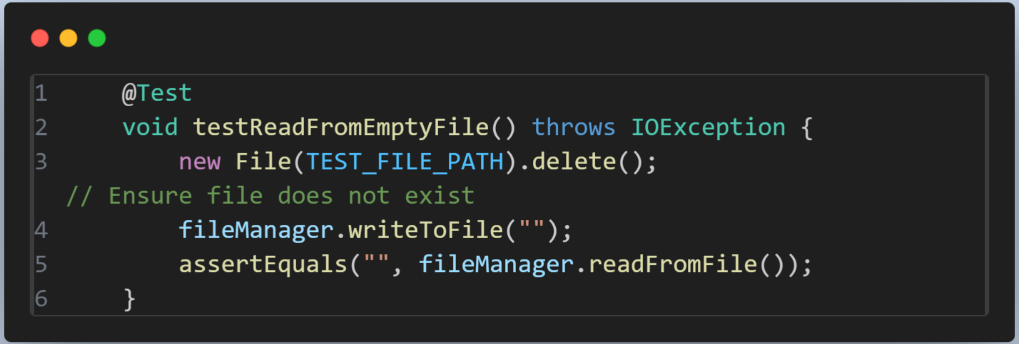
```java
    @Test
    void testWriteToFileAndReadFromFile() throws IOException {
        String content = "Hello, JUnit!";
        fileManager.writeToFile(content);
        assertEquals(content, fileManager.readFromFile());
    }

```

FourthExample Test(Optional)



```
@Test
void testFileExists() throws IOException {
    fileManager.writeToFile("Test content");
    assertTrue(fileManager.fileExists());
}
```

FourthExample Test(Optional)



```
    @Test
    void testReadFromEmptyFile() throws IOException {
        new File(TEST_FILE_PATH).delete();
 // Ensure file does not exist
        fileManager.writeToFile("");
        assertEquals("", fileManager.readFromFile());
    }
```

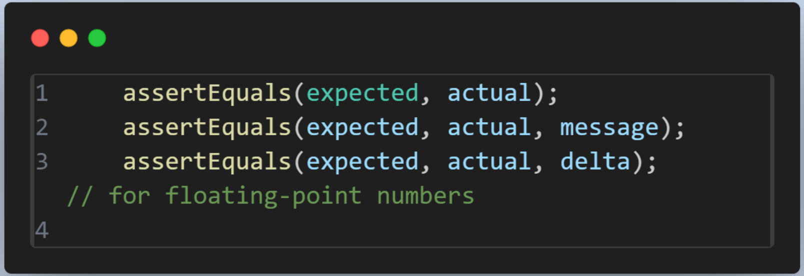
05-1

JUNIT Assertions

1. assertEquals(expected, actual)
2. assertNotEquals(expected, actual)

Description:

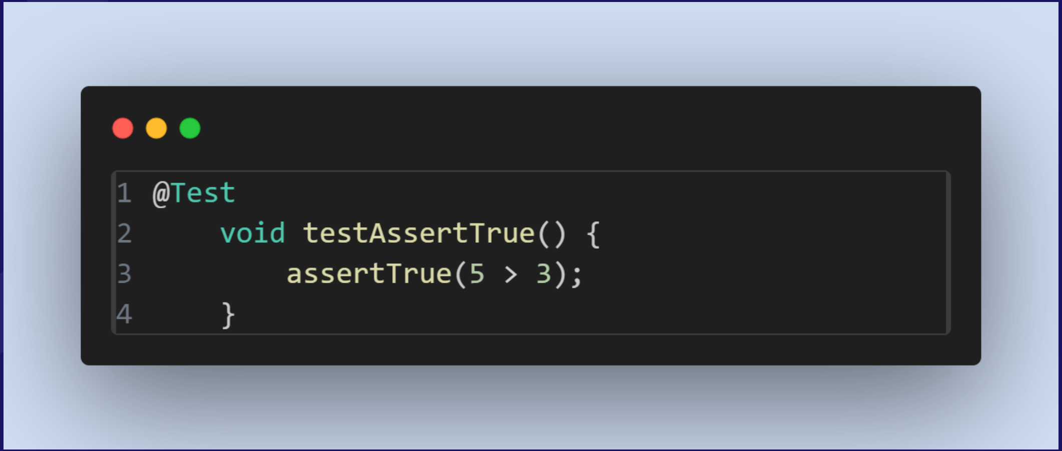
This assertion checks whether two values are equal. If they are not, the test fails.



```
    assertEquals(expected, actual);
    assertEquals(expected, actual, message);
    assertEquals(expected, actual, delta);
 // for floating-point numbers

```

3. assertTrue(condition)
4. assertFalse(condition)

Description:
Checks if the given condition is true. If false, the test fails. It's in opposite side for assertFalse.

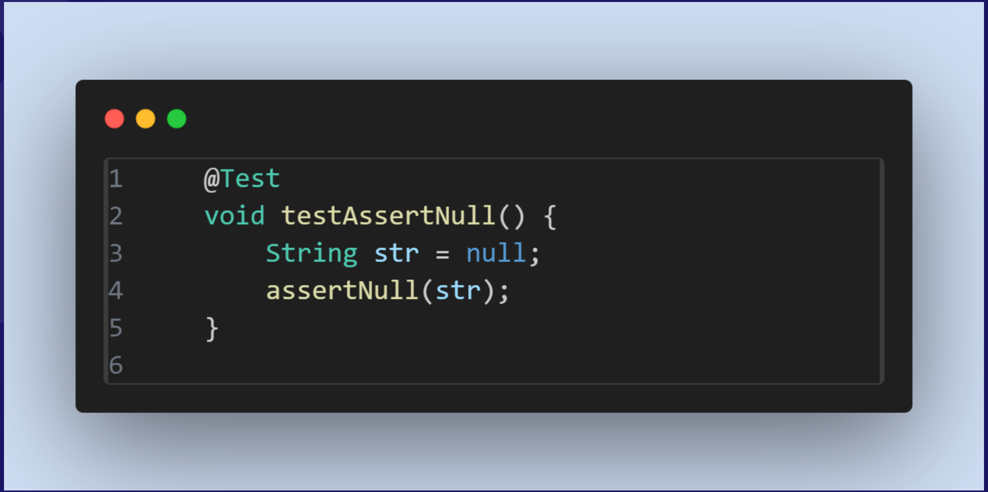


```
@Test
    void testAssertTrue() {
        assertTrue(5 > 3);
    }
```

5. assertNull(object)
6. assertNotNull(object)

Description:

Checks if the given object is null. If it's not, the test fails.



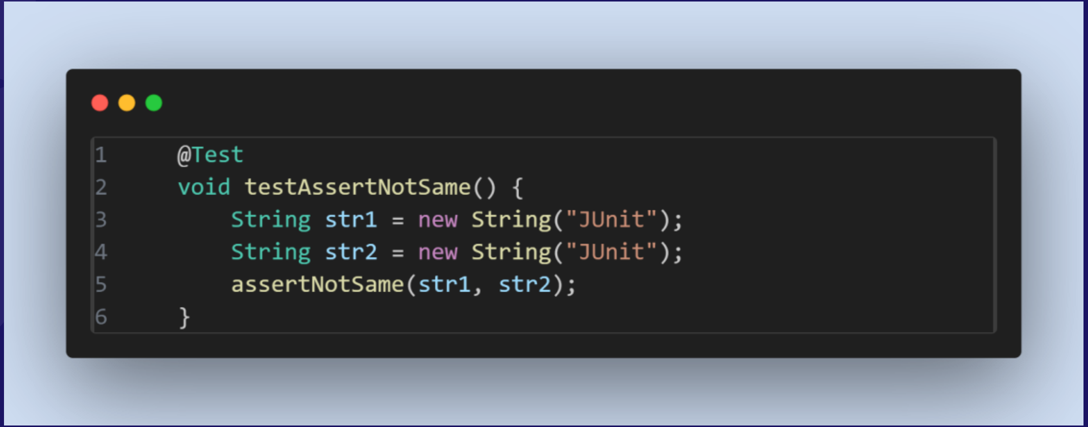
```
@Test
void testAssertNull() {
    String str = null;
    assertNull(str);
}

```

ual)

7. assertSame(expected, actual)
8. assertNotSame(expected, actual)

Description:

Checks if two object references do not point to the same object.

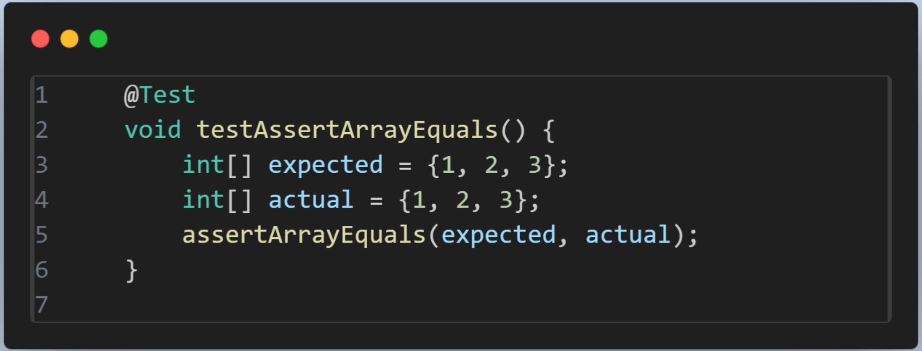


```java
@Test
void testAssertNotSame() {
    String str1 = new String("JUnit");
    String str2 = new String("JUnit");
    assertNotSame(str1, str2);
}
```

9. assertArrayEquals(expectedArray, actualArray)

Description:

Checks if two arrays are equal (i.e., have the same length and elements).



```java
@Test
void testAssertArrayEquals() {
    int[] expected = {1, 2, 3};
    int[] actual = {1, 2, 3};
    assertArrayEquals(expected, actual);
}
```

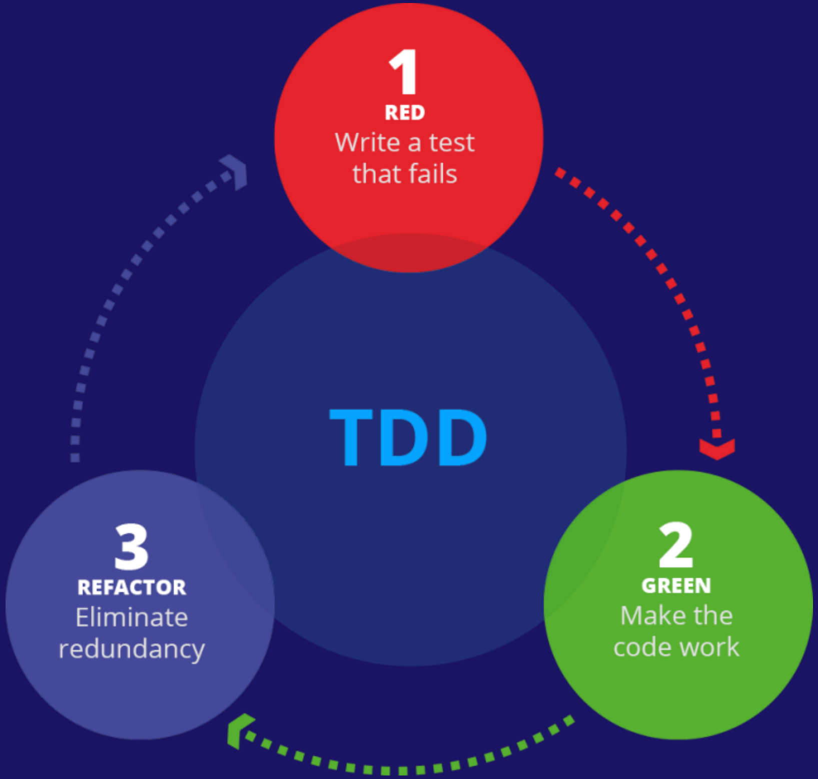
06

Test-Driven Development Method(TDD)

TDD

Test-Driven Development (TDD) in Java is a software development approach where tests are written **before the actual implementation**. The process follows a **Red-Green-Refactor** cycle

TDD


1
RED
Write a test that fails
TDD
2
GREEN
Make the code work
3
REFACTOR
Eliminate redundancy

TDD

TDD helps ensure better code quality, reduces bugs, and encourages modular and maintainable code. In Java, TDD is commonly implemented **using JUnit or TestNG** frameworks.

07

Test your code in GitHub

Testing in GitHub

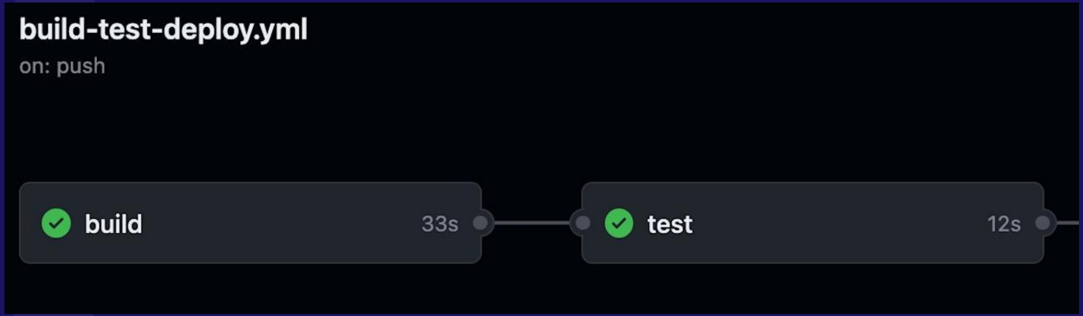
Click Commit changes....

Enter “**add test**” as the commit message and click Commit changes in the dialog box.

Select the “**Actions**” tab and click on the “**add test**” workflow run that lists the build-test-deploy workflow below it to see the status of both the build and test jobs.

You’ll see that the build and test jobs are visually linked together, implying that one needs the other.

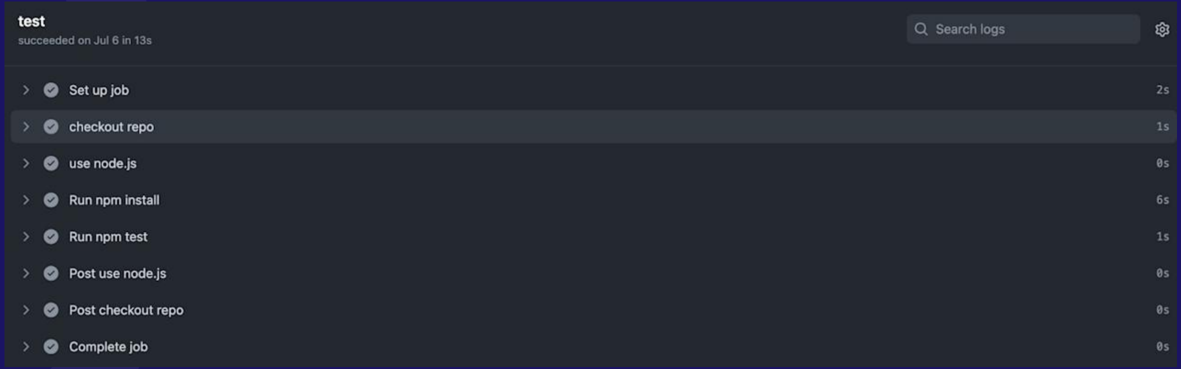
Testing in GitHub



build-test-deploy.yml

on: push

Testing in GitHub



```
test
succeeded on Jul 6 in 13s

> Set up job            2s
> checkout repo         1s
> use node.js           0s
> Run npm install       6s
> Run npm test          1s
> Post use node.js      0s
> Post checkout repo    0s
> Complete job          0s
```

Resources

https://www.tutorialspoint.com/junit

https://www.geeksforgeeks.org/introduction-to-junit-5/

https://www.baeldung.com/junit

Thanks for your attention