JUNIT

Exercise 1: Setting Up JUnit

Scenario: You need to set up JUnit in your Java project to start writing unit tests.

Steps:

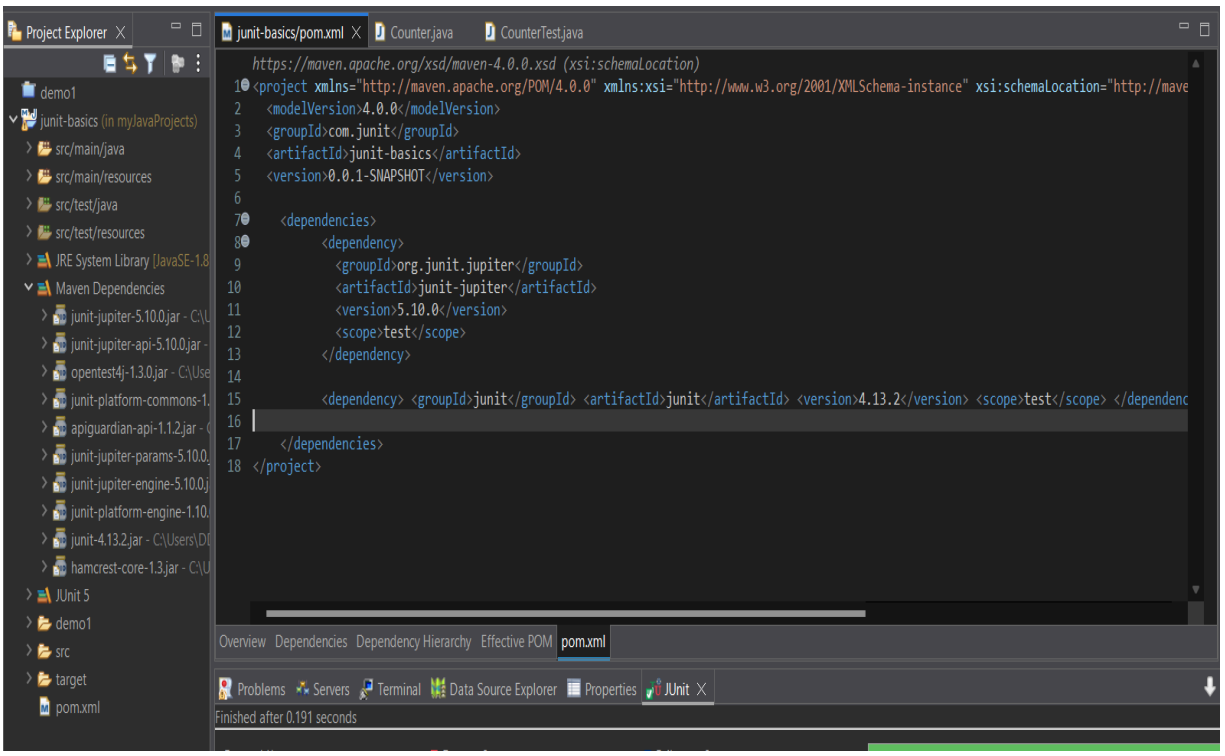
1. Create a new Java project in your IDE (e.g., IntelliJ IDEA, Eclipse).

2. Add JUnit dependency to your project. If you are using Maven, add the following to your pom.xml: <dependency> <groupId>junit</groupId> <artifactId>junit</artifactId> <version>4.13.2</version> <scope>test</scope> </dependency>

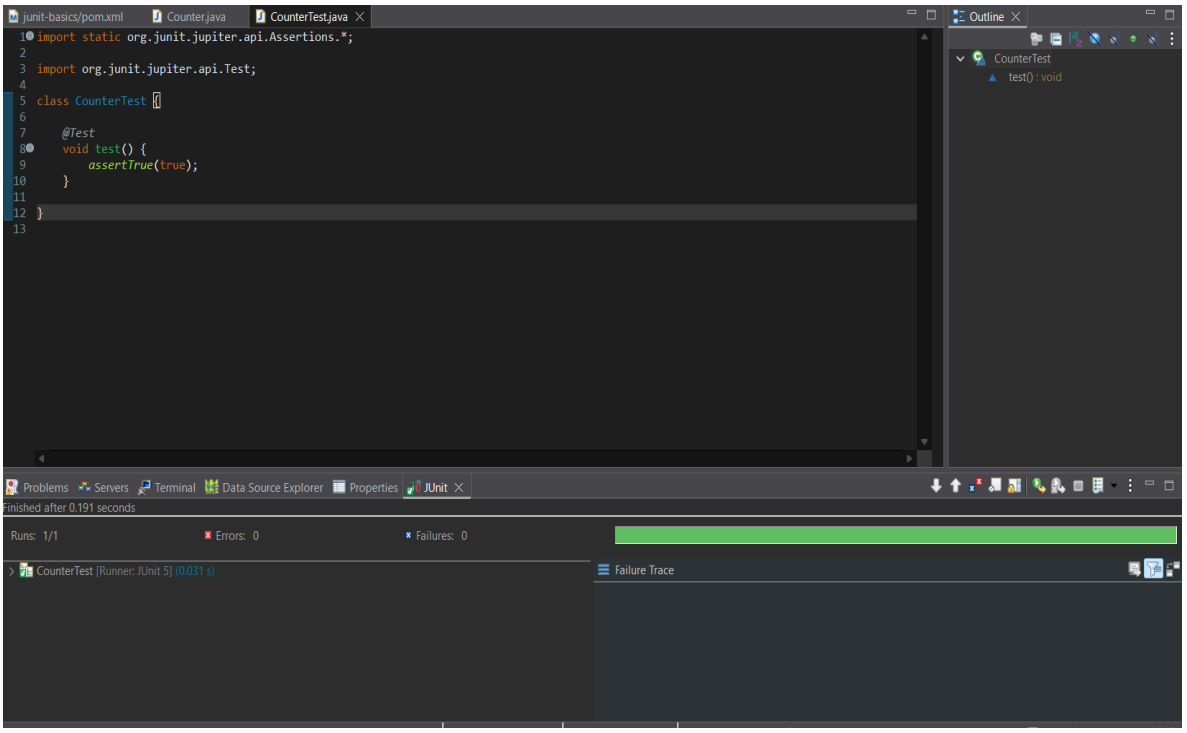
3. Create a new test class in your project.

Output Screenshots

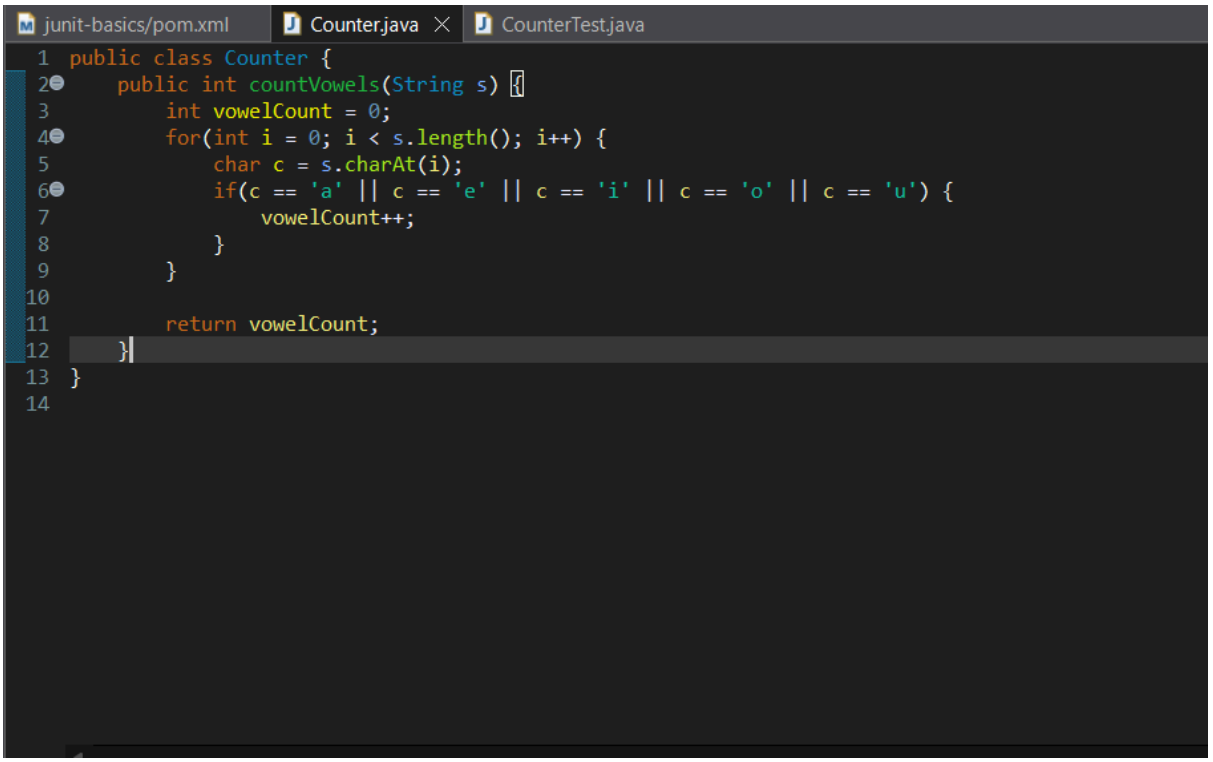
pom.xml



Test class



Created a counter class and a method for counting vowels in a string



Exercise 2: Writing Basic JUnit Tests

Scenario: You need to write basic JUnit tests for a simple Java class.

Steps:

1. Create a new Java class with some methods to test.

2. Write JUnit tests for these methods.

Solution

- Created a Counter class that has methods like countVowels(string) and countConsonants(string)
- Ran the tests and obtained a failure
- Corrected the countConsonants code to check whether the character is part of the English alphabet.
- Re-ran the tests and all tests have passed

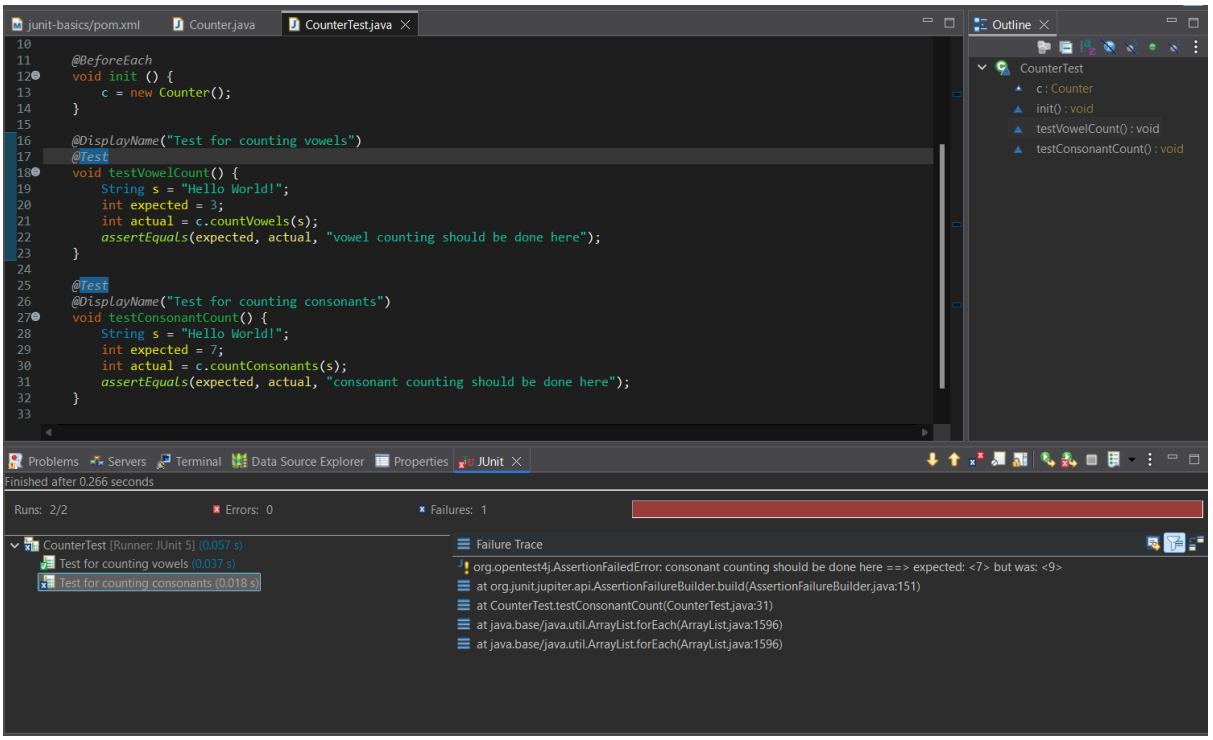
`Counter.java`

```
public class Counter {
	public int countVowels(String s) {
		int vowelCount = 0;
		for(int i = 0; i < s.length(); i++) {
			char c = s.charAt(i);
			if(c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
{
				vowelCount++;
			}
		}

		return vowelCount;
	}

	public int countConsonants(String s) {
		int consonantsCount = 0;
		for(int i = 0; i < s.length(); i++) {
			char c = s.charAt(i);
			if(Character.isAlphabetic(c) && c != 'a' && c != 'e' && c !=
'i' && c != 'o' && c != 'u') {
				consonantsCount++;
			}
		}

		return consonantsCount;
	}
}
```

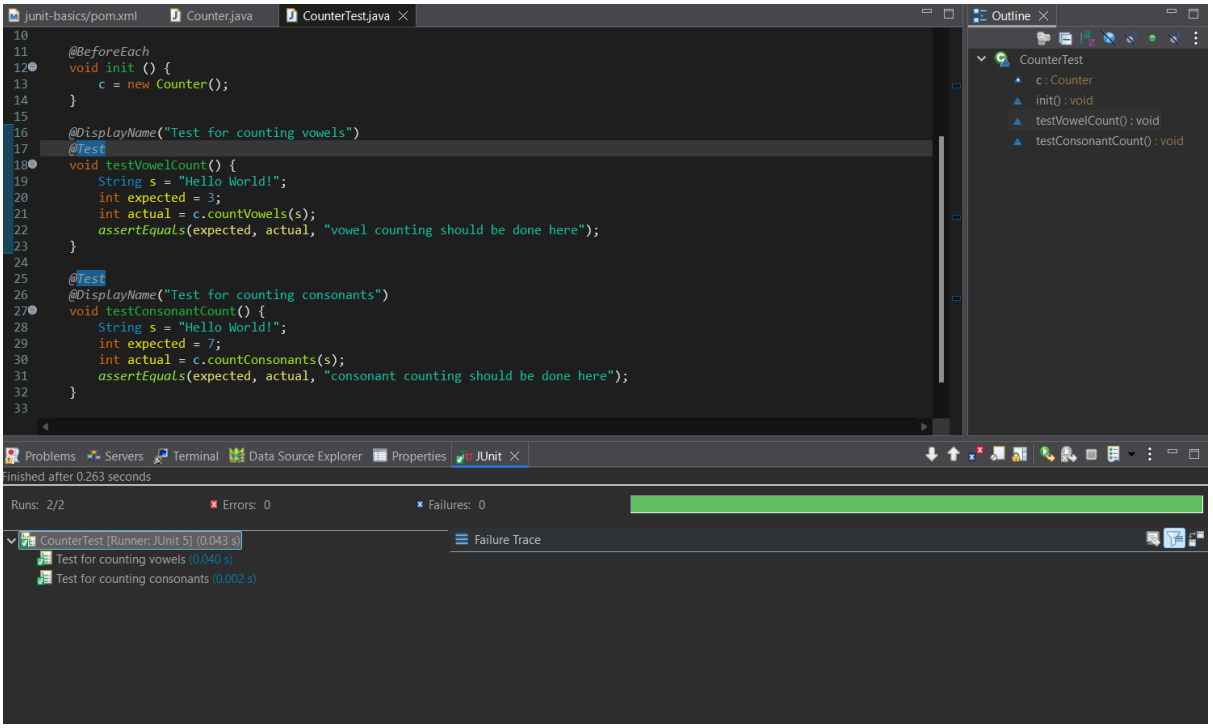
CounterTest.java

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
class CounterTest {

	Counter c;

	@BeforeEach
	void init () {
		c = new Counter();
	}

	@DisplayName("Test for counting vowels")
	@Test
	void testVowelCount() {
		String s = "Hello World!";
		int expected = 3;
		int actual = c.countVowels(s);
		assertEquals(expected, actual, "vowel counting should be done
here");
	}

	@Test
	@DisplayName("Test for counting consonants")
	void testConsonantCount() {
		String s = "Hello World!";
		int expected = 7;
		int actual = c.countConsonants(s);
		assertEquals(expected, actual, "consonant counting should be done
here");
	}
}
```

Incorrect Result



Correct Result



Exercise 3: Assertions in JUnit

Scenario: You need to use different assertions in JUnit to validate your test results.

Steps:

1. Write tests using various JUnit assertions.

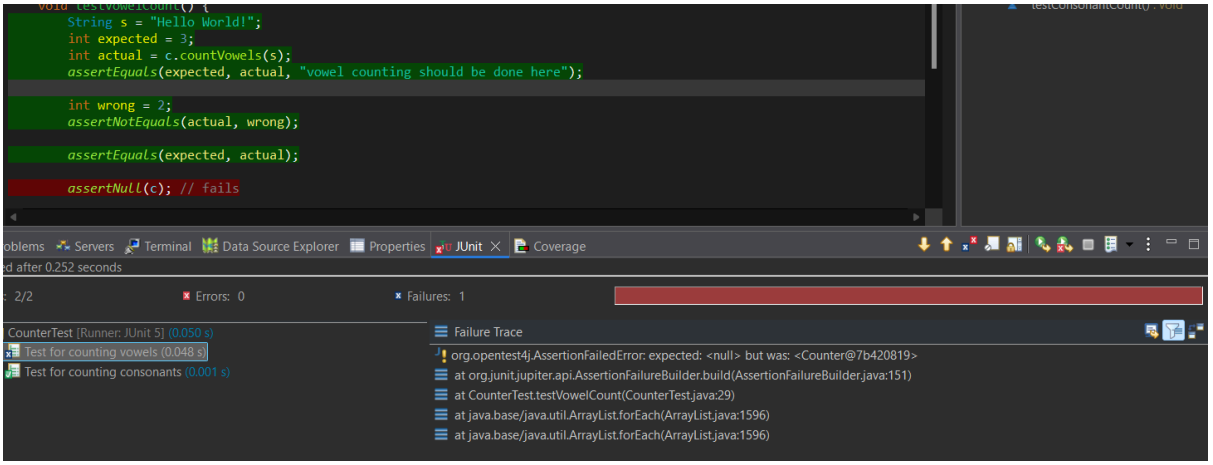
Solution

`testVowelCount() method`

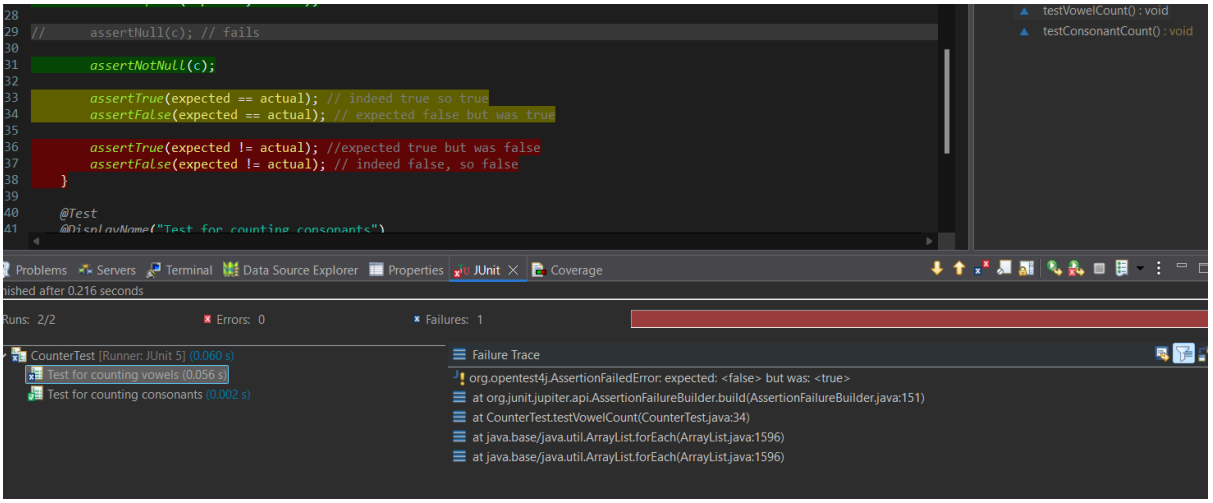
```
@DisplayName("Test for counting vowels")
	@Test
	void testVowelCount() {
		String s = "Hello World!";
		int expected = 3;
		int actual = c.countVowels(s);
		assertEquals(expected, actual, "vowel counting should be done
here");

		int wrong = 2;
		assertNotEquals(actual, wrong);

		assertEquals(expected, actual);

		assertNull(c); // fails

		assertNotNull(c);

		assertTrue(expected == actual); // indeed true so true
		assertFalse(expected == actual); // expected false but was true

		assertTrue(expected != actual); //expected true but was false
		assertFalse(expected != actual); // indeed false, so false
	}
```

Output Screenshots

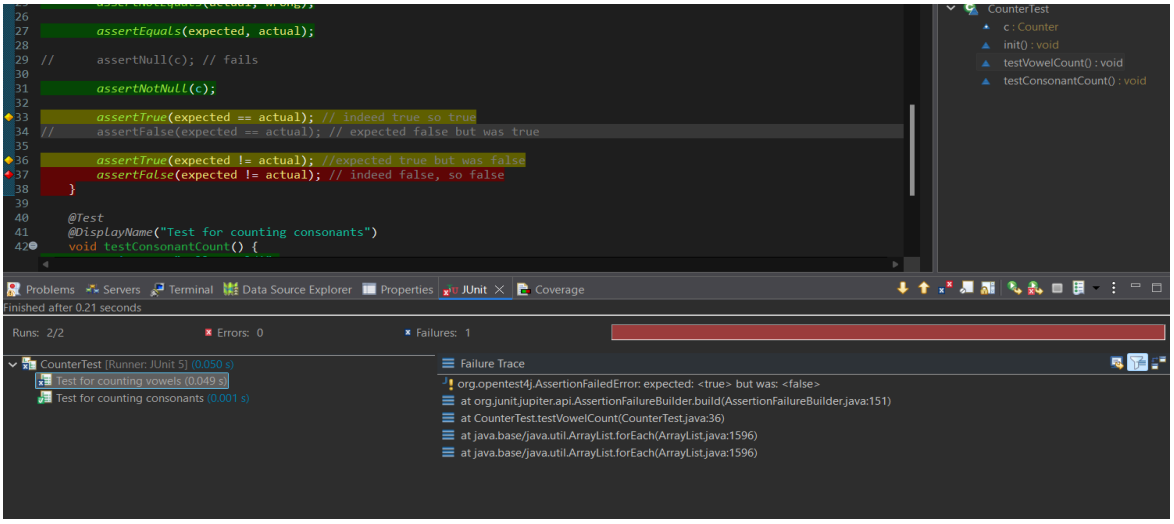
assertNull(c);



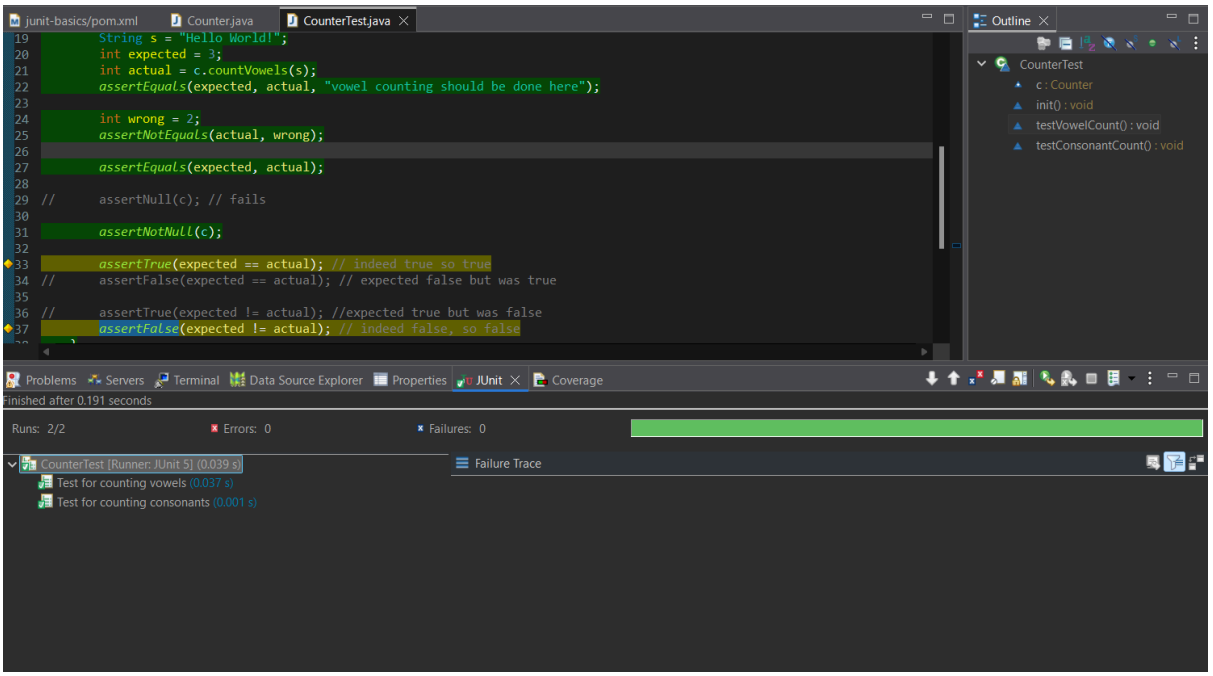
assertFalse(expected == actual);



assertTrue(expected != actual);



Success


junit-basics/pom.xml
Counter.java
CounterTest.java
Outline
CounterTest
c : Counter
init() : void
testVowelCount() : void
testConsonantCount() : void
Problems
Servers
Terminal
Data Source Explorer
Properties
JUnit
Coverage
Finished after 0.191 seconds
Runs: 2/2
Errors: 0
Failures: 0
CounterTest [Runner: JUnit 5] (0.039 s)
Test for counting vowels (0.037 s)
Test for counting consonants (0.001 s)
Failure Trace

Exercise 4: Arrange-Act-Assert (AAA) Pattern, Test Fixtures, Setup and Teardown Methods in JUnit

Scenario: You need to organize your tests using the Arrange-Act-Assert (AAA) pattern and use setup and teardown methods.

Steps:

1. Write tests using the AAA pattern.

2. Use @Before and @After annotations for setup and teardown methods.

Solution

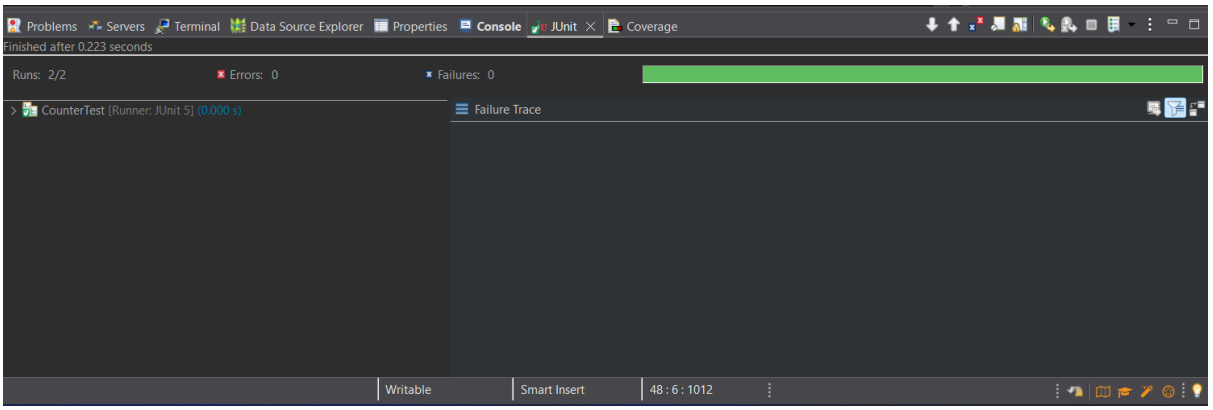
CounterTest.java

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
class CounterTest {

	Counter c;

	@BeforeEach
	void init () {
		c = new Counter();
	}

	@AfterEach
	void cleanUp() {
		System.out.println("well done! cleaning up..!");
	}


	@DisplayName("Test for counting vowels")
	@Test
	void testVowelCount() {
		// Arrange
		String s = "Hello World!";
		int expected = 3;
		int wrong = 2;

		// Act
		int actual = c.countVowels(s);

		// Assert
		assertEquals(expected, actual, "vowel counting should be done
```

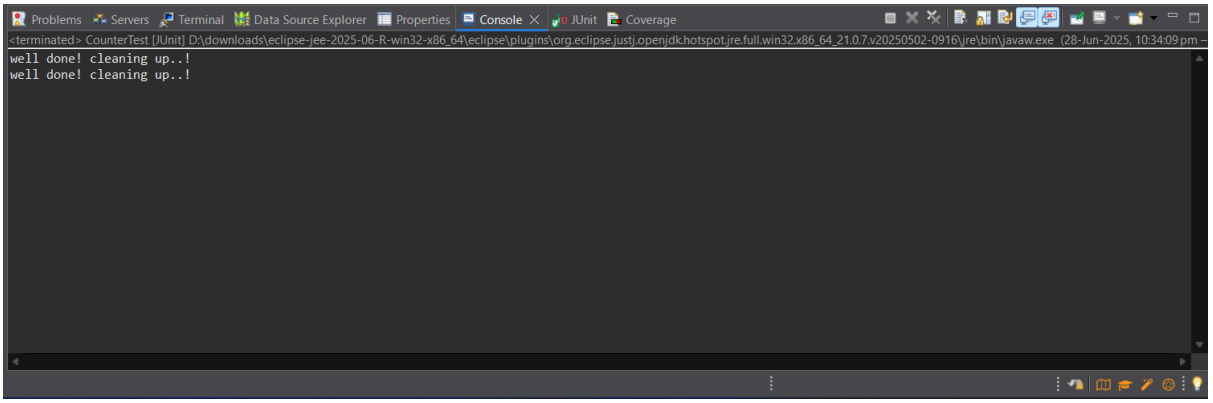
```
here");
            assertNotEquals(actual, wrong);
        }

        @Test
        @DisplayName("Test for counting consonants")
        void testConsonantCount() {
            String s = "Hello World!";
            int expected = 7;

            int actual = c.countConsonants(s);

            assertEquals(expected, actual, "consonant counting should be done
here");
        }
}
```

Output Screenshots

Successful Tests



Clean up console statements



MOCKITO

Exercise 1: Mocking and Stubbing

Scenario: You need to test a service that depends on an external API. Use Mockito to mock the external API and stub its methods.

Steps:

1. Create a mock object for the external API.

2. Stub the methods to return predefined values.

3. Write a test case that uses the mock object.

Solution

URLClient.java

```
package moc.bas.basMoc;
public interface URLClient {
	String shortenURL(String URL);
}
```

ShortenerService.java

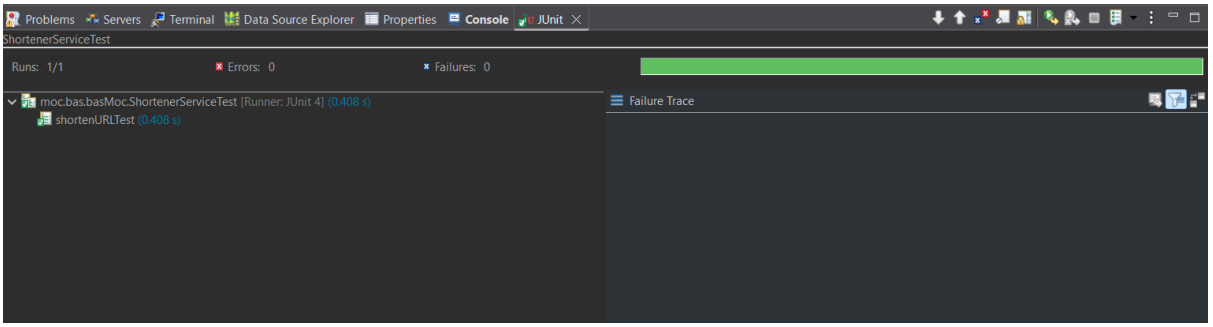
```
package moc.bas.basMoc;
public class ShortenerService {
	private URLClient urlclient;

	public ShortenerService(URLClient urlclient) {
		this.urlclient = urlclient;
	}

	public String shortenURL(String url) {
		return urlclient.shortenURL(url);
	}
}
```

ShortenerServiceTest.java

```
package moc.bas.basMoc;
import org.junit.Test;
import static org.junit.Assert.*;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;
public class ShortenerServiceTest {
```

```
	@Test
	public void shortenURLTest() {
		URLClient urlclientmock = mock(URLClient.class);
		String longurl =
"http://chart.apis.google.com/chart?chs=500x500&chma=0,0,100,100&cht=p&chco=FF00
00%2CFFFF00%7CFF8000%2C00FF00%7C00FF00%2C0000FF&chd=t%3A122%2C42%2C17%2C10%2C8%2
C7%2C7%2C7%2C7%2C6%2C6%2C6%2C6%2C5%2C5&chl=122%7C42%7C17%7C10%7C8%7C7%7C7%7C7%7C
7%7C6%7C6%7C6%7C6%7C5%7C5&chdl=android%7Cjava%7Cstack-trace%7Cbroadcastreceiver%
7Candroid-ndk%7Cuser-agent%7Candroid-webview%7Cwebview%7Cbackground%7Cmultithrea
ding%7Android-source%7Csms%7Cadb%7Csollections%7Cactivity|Chart";
		String expected = "http://chart.google.com";

		when(urlclientmock.shortenURL(longurl)).thenReturn(expected);

		ShortenerService ss = new ShortenerService(urlclientmock);

		String actual = ss.shortenURL(longurl);

		assertEquals(expected, actual);

	}
}
```

Output Screenshots



Exercise 2: Verifying Interactions

Scenario: You need to ensure that a method is called with specific arguments.

Steps:

1. Create a mock object.

2. Call the method with specific arguments.

3. Verify the interaction.

Solution

URLClient.java

```
package moc.bas.basMoc;
public interface URLClient {
	String shortenURL(String URL);
}
```

ShortenerService.java

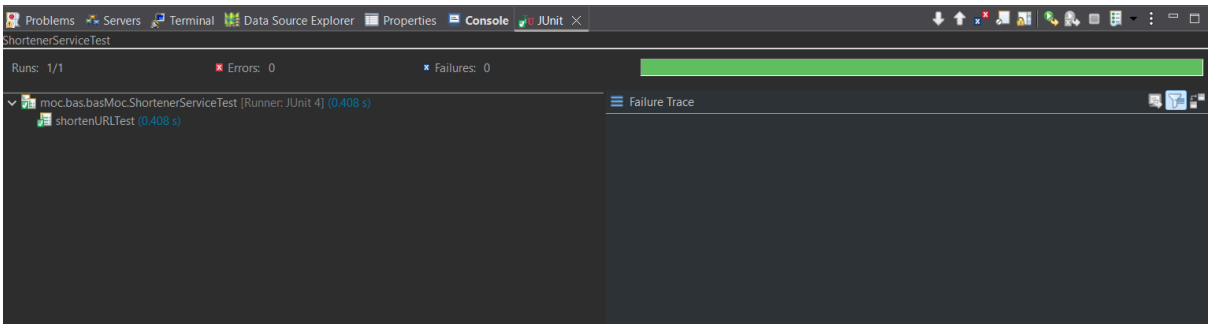
```
package moc.bas.basMoc;
public class ShortenerService {
	private URLClient urlclient;

	public ShortenerService(URLClient urlclient) {
		this.urlclient = urlclient;
	}

	public String shortenURL(String url) {
		return urlclient.shortenURL(url);
	}
}
```

ShortenerServiceTest.java

```
package moc.bas.basMoc;
import org.junit.Test;
import static org.junit.Assert.*;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;
public class ShortenerServiceTest {

	@Test
	public void shortenURLTest() {
```

```
            URLClient urlclientmock = mock(URLClient.class);
            String longurl =
"http://chart.apis.google.com/chart?chs=500x500&chma=0,0,100,100&cht=p&chco=FF00
00%2CFFFF00%7CFF8000%2C00FF00%7C00FF00%2C0000FF&chd=t%3A122%2C42%2C17%2C10%2C8%2
C7%2C7%2C7%2C7%2C6%2C6%2C6%2C6%2C5%2C5&chl=122%7C42%7C17%7C10%7C8%7C7%7C7%7C7%7C
7%7C6%7C6%7C6%7C6%7C5%7C5&chdl=android%7Cjava%7Cstack-trace%7Cbroadcastreceiver%
7Candroid-ndk%7Cuser-agent%7Candroid-webview%7Cwebview%7Cbackground%7Cmultithrea
ding%7Android-source%7Csms%7Cadb%7Csollections%7Cactivity|Chart";
            String expected = "http://chart.google.com";

            when(urlclientmock.shortenURL(longurl)).thenReturn(expected);

            ShortenerService ss = new ShortenerService(urlclientmock);

            String actual = ss.shortenURL(longurl);

            assertEquals(expected, actual);

            verify(urlclientmock).shortenURL(longurl);
        }
}
```

Output Screenshots



SLF4J

Exercise 1: Logging Error Messages and Warning Levels

Task:

Write a Java application that demonstrates logging error messages and warning levels using SLF4J.

Solution

pom.xml

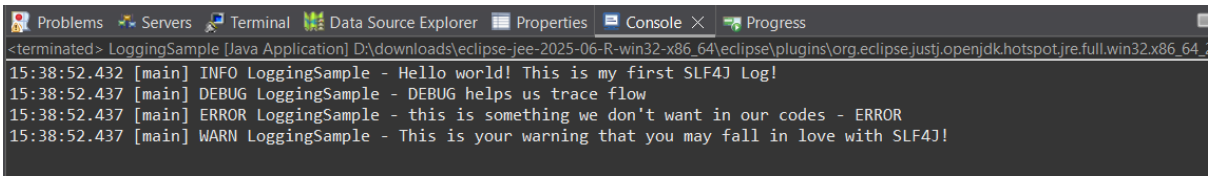
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>slf4j-pratice</groupId>
 <artifactId>slf4j-basics</artifactId>
 <version>0.0.1-SNAPSHOT</version>
  <dependencies>
       <!-- SLF4J API -->
       <dependency>
         <groupId>org.slf4j</groupId>
         <artifactId>slf4j-api</artifactId>
         <version>1.7.30</version>
       </dependency>
<!-- Logback implementation (backend) -->
       <dependency>
         <groupId>ch.qos.logback</groupId>
         <artifactId>logback-classic</artifactId>
         <version>1.2.3</version>
       </dependency>
 </dependencies>
</project>
```

LoggingSample.java

```
import org.slf4j.LoggerFactory;
import org.slf4j.Logger;
public class LoggingSample {

       private static final Logger logger =
LoggerFactory.getLogger(LoggingSample.class);
       public static void main(String[] args) {
```

```
            logger.info("Hello world! This is my first SLF4J Log!");
            logger.debug("DEBUG helps us trace flow");
            // logger.trace("Trace is DEBUG but more detailed");
            logger.error("this is something we don't want in our codes -
ERROR");
            logger.warn("This is your warning that you may fall in love with
SLF4J!");

        }
}
```

Output Screenshots



Exercise 2: Parameterized Logging

Task:

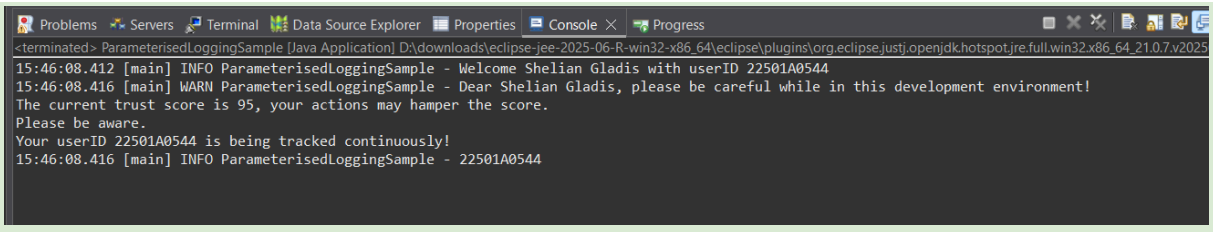
Write a Java application that demonstrates parameterized logging using SLF4J.

Solution

ParameterisedLoggingSample.java

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
public class ParameterisedLoggingSample {
	private static final Logger logger =
LoggerFactory.getLogger(ParameterisedLoggingSample.class);
	public static void main(String[] args) {
		String userName = "Shelian Gladis";
		String userID = "22501A0544";
		int trustScore = 95;

		logger.info("Welcome {} with userID {}", userName, userID);
		logger.warn("Dear {}, please be careful while in this development
environment!\nThe current trust score is {}, your actions may hamper the
score.\nPlease be aware.\nYour userID {} is being tracked continuously!",
userName, trustScore, userID);
		logger.info(userID);
	}
}
```

Output Screenshots



Exercise 3: Using Different Appenders

Task:

Write a Java application that demonstrates using different appenders with SLF4J

Solution

LoggingSample.java

```
import org.slf4j.LoggerFactory;
import org.slf4j.Logger;
public class LoggingSample {

	private static final Logger logger =
LoggerFactory.getLogger(LoggingSample.class);
	public static void main(String[] args) {
		logger.info("Hello world! This is my first SLF4J Log!");
		logger.debug("DEBUG helps us trace flow");
		logger.trace("Trace is DEBUG but more detailed");
		logger.error("this is something we don't want in our codes -
ERROR");
		logger.warn("This is your warning that you may fall in love with
SLF4J!");
	}
}
```

logback.xml

```
<configuration>
	<appender name="console" class = "ch.qos.logback.core.ConsoleAppender">
		<encoder>
			<pattern>
				%green([%thread]) %yellow(%d{HH:mm:ss.SSS}) %magenta(
%logger)- %highlight(%-5level) → %msg%n
			</pattern>
		</encoder>
	</appender>

	<appender name="file" class="ch.qos.logback.core.FileAppender">
		<file>app.log</file>
		<encoder>
			<pattern>
				[%thread] %d{HH:mm:ss} %logger{36} %-5level %msg%n
			</pattern>
		</encoder>
	</appender>

	<root level = "trace">
```

```
            <appender-ref ref="console" />
            <appender-ref ref="file" />
        </root>
</configuration>
```

Output Screenshots

Console output

```
<terminated> LoggingSample [Java Application] D:\downloads\eclipse-jee-2025-06-R-win32-x86_64\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.7.
[main] 19:17:38.964 LoggingSample - INFO  → Hello world! This is my first SLF4J Log!
[main] 19:17:38.966 LoggingSample - DEBUG → DEBUG helps us trace flow
[main] 19:17:38.966 LoggingSample - TRACE → Trace is DEBUG but more detailed
[main] 19:17:38.966 LoggingSample - ERROR → this is something we don't want in our codes - ERROR
[main] 19:17:38.966 LoggingSample - WARN  → This is your warning that you may fall in love with SLF4J!
```

app.log

```
[main] 19:17:38 LoggingSample INFO  Hello world! This is my first SLF4J Log!
[main] 19:17:38 LoggingSample DEBUG DEBUG helps us trace flow
[main] 19:17:38 LoggingSample TRACE Trace is DEBUG but more detailed
[main] 19:17:38 LoggingSample ERROR this is something we don't want in our codes - ERROR
[main] 19:17:38 LoggingSample WARN  This is your warning that you may fall in love with SLF4J!
```