Advanced JUnit Testing Exercises

Exercise 1: Parameterized Tests

Scenario:

You want to test a method that checks if a number is even. Instead of writing multiple test cases, you will use parameterized tests to run the same test with different inputs.

Steps:

1. Create a new Java class `EvenChecker` with a method `isEven(int number)`.

2. Write a parameterized test class `EvenCheckerTest` that tests the `isEven` method with different inputs.

3. Use JUnit's `@ParameterizedTest` and `@ValueSource` annotations.

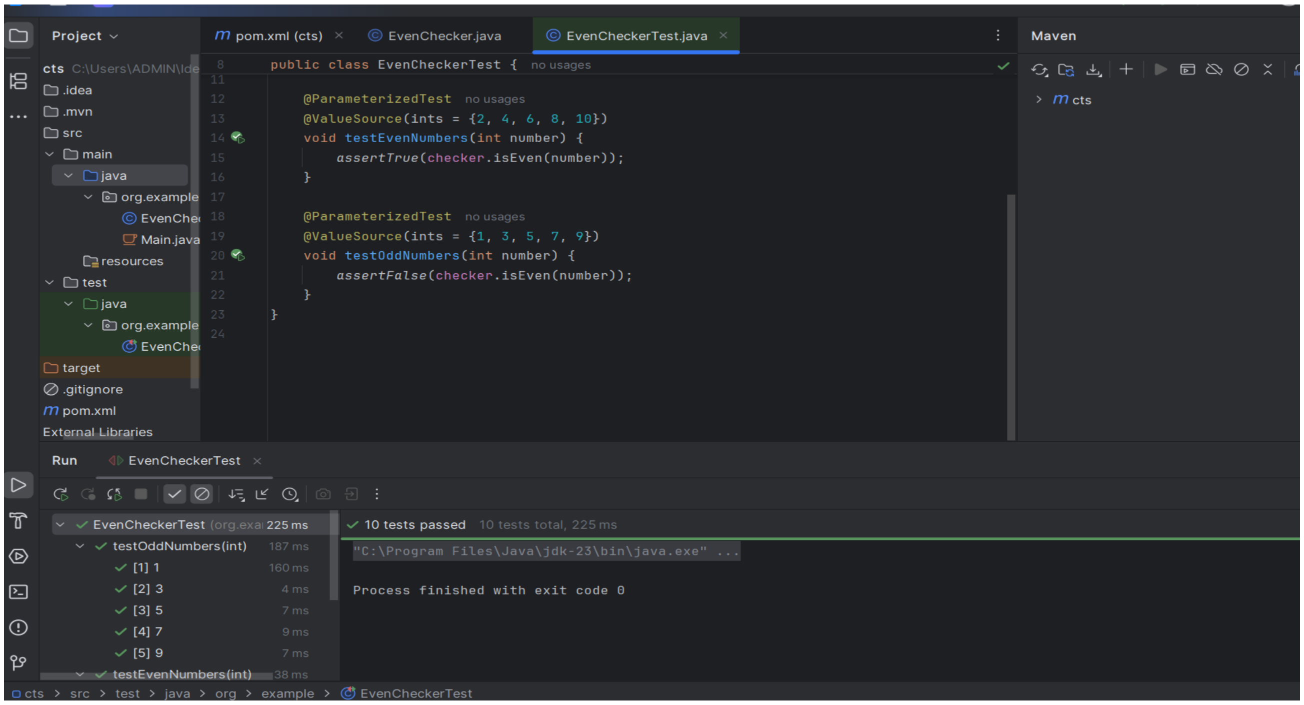
EvenChecker.java

```
package org.example;

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

EvenCheckerTest.java

```
package org.example;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

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

public class EvenCheckerTest {

    EvenChecker checker = new EvenChecker();
    @ParameterizedTest
    @ValueSource(ints = {2, 4, 6, 8, 10})
    void testEvenNumbers(int number) {
        assertTrue(checker.isEven(number));
    }

    @ParameterizedTest
    @ValueSource(ints = {1, 3, 5, 7, 9})
    void testOddNumbers(int number) {
        assertFalse(checker.isEven(number));
    }
}
```

OUTPUT:



```
public class EvenCheckerTest {  no usages

    @ParameterizedTest  no usages
    @ValueSource(ints = {2, 4, 6, 8, 10})
    void testEvenNumbers(int number) {
        assertTrue(checker.isEven(number));
    }

    @ParameterizedTest  no usages
    @ValueSource(ints = {1, 3, 5, 7, 9})
    void testOddNumbers(int number) {
        assertFalse(checker.isEven(number));
    }
}
```

```
✓ EvenCheckerTest (org.exa  225 ms
  ✓ testOddNumbers(int)     187 ms
    ✓ [1] 1                 160 ms
    ✓ [2] 3                   4 ms
    ✓ [3] 5                   7 ms
    ✓ [4] 7                   9 ms
    ✓ [5] 9                   7 ms
  ✓ testEvenNumbers(int)     38 ms
```

```
✓ 10 tests passed  10 tests total, 225 ms
"C:\Program Files\Java\jdk-23\bin\java.exe" ...

Process finished with exit code 0
```

Exercise 2: Test Suites and Categories

Scenario:

You want to group related tests into a test suite and categorize them.

Steps:

1. Create a new test suite class `AllTests`.

2. Add multiple test classes to the suite.

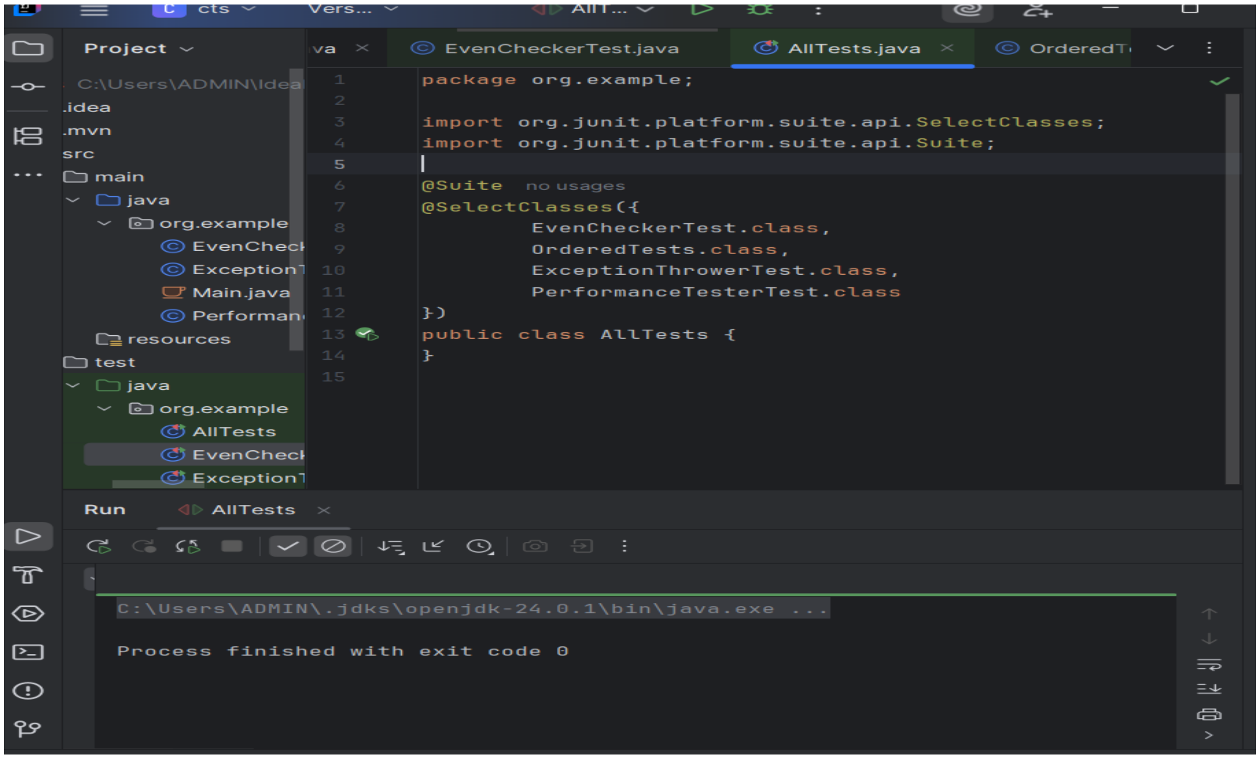
3. Use JUnit's `@Suite` and `@SelectClasses` annotations.

CODE:

AllTests.java

```
package org.example;

import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;

@Suite
@SelectClasses({
    EvenCheckerTest.class,
    OrderedTests.class,
    ExceptionThrowerTest.class,
    PerformanceTesterTest.class
})
public class AllTests {
}
```

OUTPUT:



```java
package org.example;

import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;

@Suite
@SelectClasses({
        EvenCheckerTest.class,
        OrderedTests.class,
        ExceptionThrowerTest.class,
        PerformanceTesterTest.class
})
public class AllTests {
}
```

```
C:\Users\ADMIN\.jdks\openjdk-24.0.1\bin\java.exe ...

Process finished with exit code 0
```

Exercise 3: Test Execution Order

Scenario:

You want to control the order in which tests are executed.

Steps:

1. Create a test class `OrderedTests`.

2. Use JUnit's `@TestMethodOrder` and `@Order` annotations.

CODE:

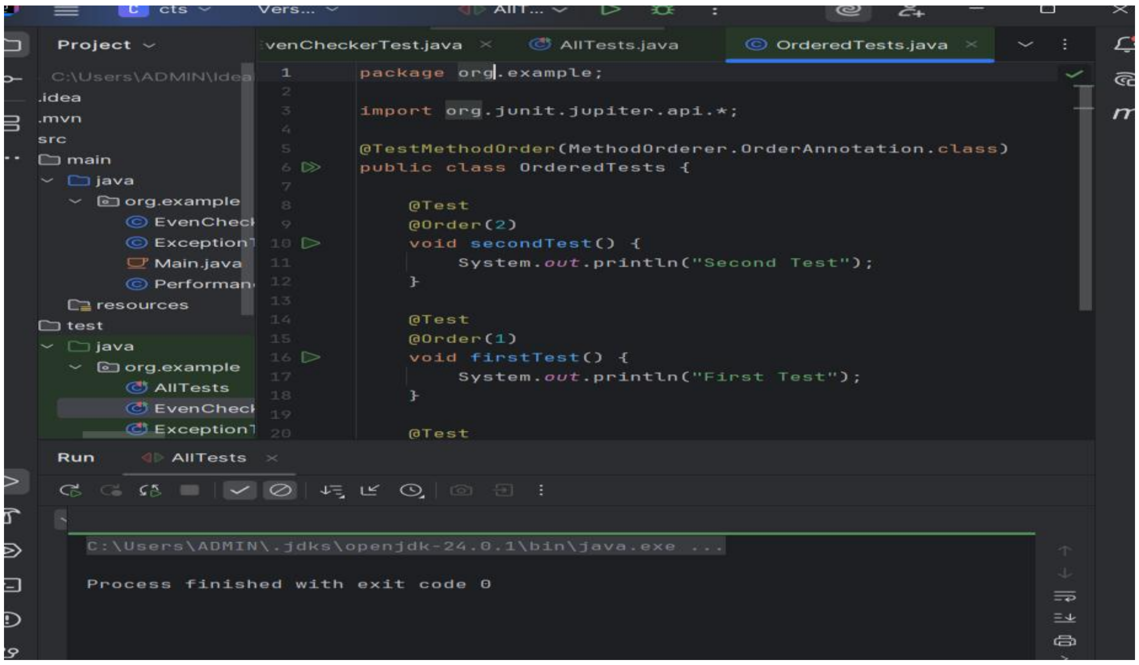
OrderedTests.java

package org.example;

import org.junit.jupiter.api.*;

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class OrderedTests {

    @Test
    @Order(2)
    void secondTest() {
        System.out.println("Second Test");
    }

    @Test
    @Order(1)
    void firstTest() {
        System.out.println("First Test");
    }

    @Test
    @Order(3)
    void thirdTest() {
        System.out.println("Third Test");
    }
}
```

OUTPUT:



```
package org.example;

import org.junit.jupiter.api.*;

@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class OrderedTests {

    @Test
    @Order(2)
    void secondTest() {
        System.out.println("Second Test");
    }

    @Test
    @Order(1)
    void firstTest() {
        System.out.println("First Test");
    }

    @Test
```

```
C:\Users\ADMIN\.jdks\openjdk-24.0.1\bin\java.exe ...

Process finished with exit code 0
```

Exercise 4: Exception Testing

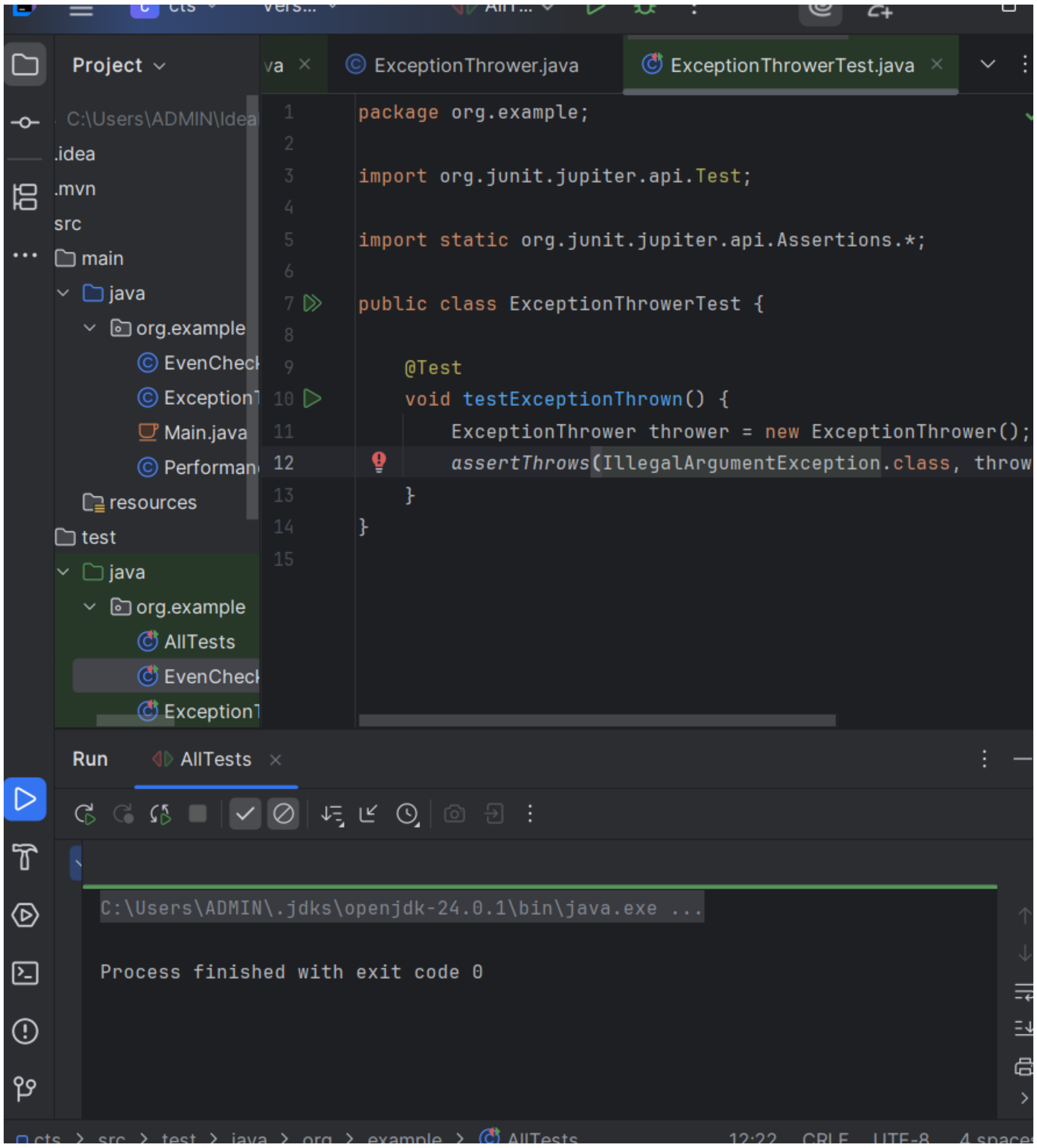
Scenario:

You want to test that a method throws the expected exception.

Steps:

1. Create a class `ExceptionThrower` with a method `throwException`.

2. Write a test class `ExceptionThrowerTest` that tests the method for the expected exception.



```java
package org.example;

import org.junit.jupiter.api.Test;

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

public class ExceptionThrowerTest {

    @Test
    void testExceptionThrown() {
        ExceptionThrower thrower = new ExceptionThrower();
        assertThrows(IllegalArgumentException.class, throw
    }
}
```

```
C:\Users\ADMIN\.jdks\openjdk-24.0.1\bin\java.exe ...

Process finished with exit code 0
```

Exercise 5: Timeout and Performance Testing

Scenario:

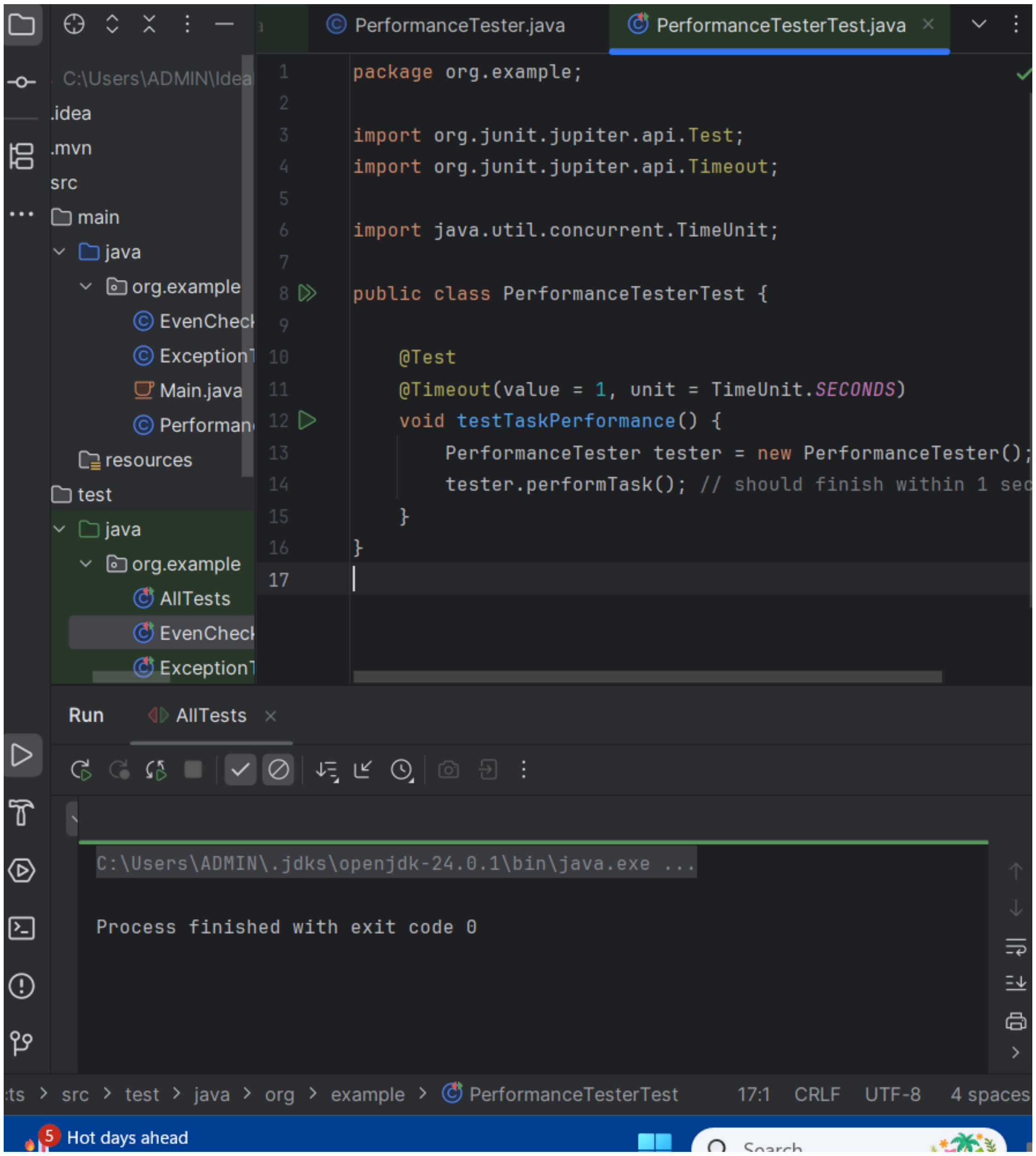
You want to ensure that a method completes within a specified time limit.

Steps:

1. Create a class `PerformanceTester` with a method `performTask`.

2. Write a test class `PerformanceTesterTest` that tests the method for timeout

OUTPUT:


PerformanceTester.java   PerformanceTesterTest.java

```java
package org.example;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Timeout;

import java.util.concurrent.TimeUnit;

public class PerformanceTesterTest {

    @Test
    @Timeout(value = 1, unit = TimeUnit.SECONDS)
    void testTaskPerformance() {
        PerformanceTester tester = new PerformanceTester();
        tester.performTask(); // should finish within 1 sec
    }
}
```

Run   AllTests

```
C:\Users\ADMIN\.jdks\openjdk-24.0.1\bin\java.exe ...

Process finished with exit code 0
```