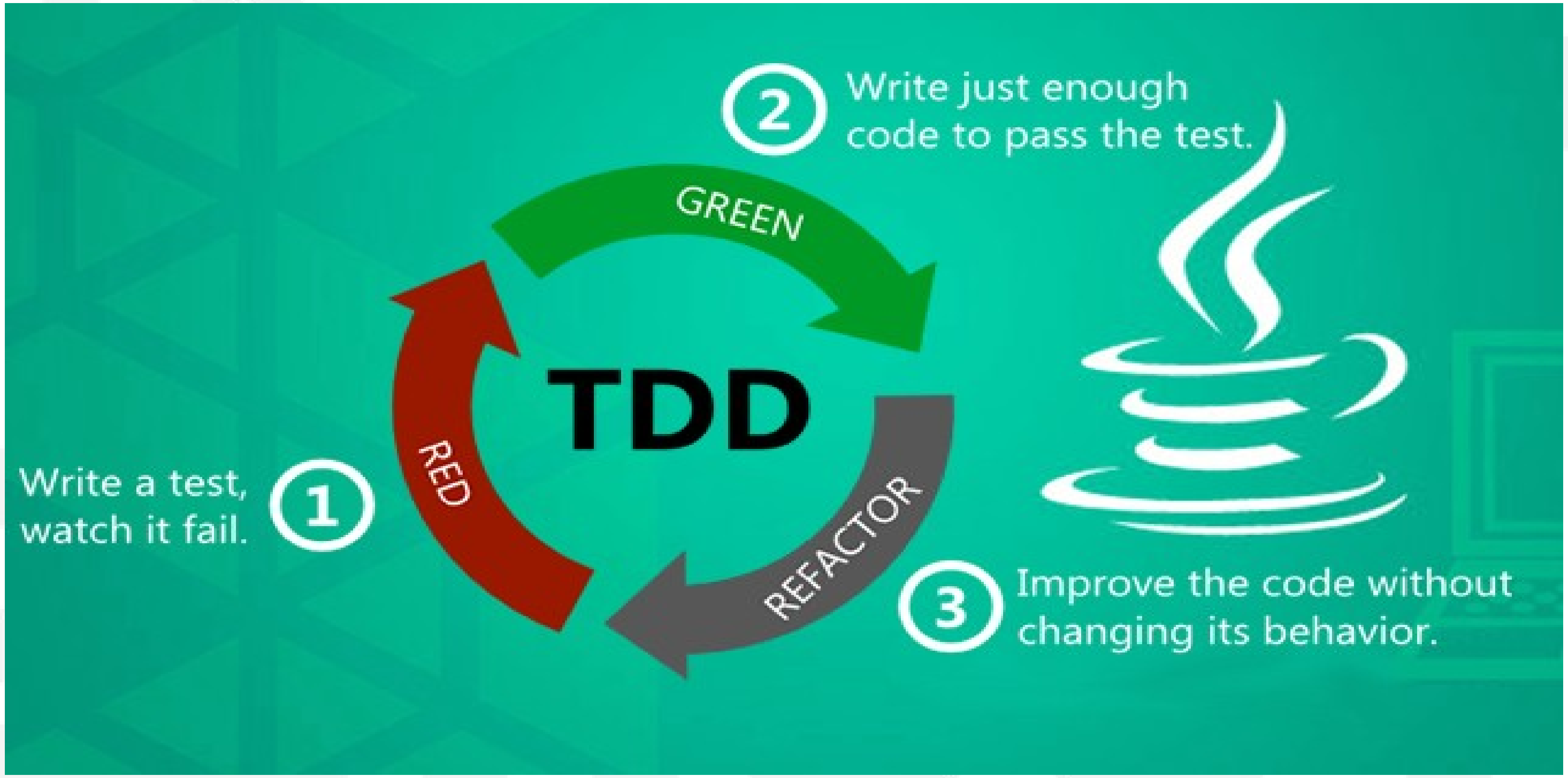
Unit Testing with JUnit

Session Goals

By the end of this session, you will be able to demonstrate how to:

- Implement unit testing in Java
- Use JUNIT test annotations
- Use assertion
- Ignore test cases
- Timeout test cases
- Handle testing exceptions
- Use parameterized test cases

Context Setting


2
Write just enough code to pass the test.
GREEN
TDD
RED
REFACTOR
1
Write a test, watch it fail.
3
Improve the code without changing its behavior.

Let Us Try to Find Out

- **What is Testing?**
- **What is Unit Testing?**
- **Why do we need Testing?**
- **How does Unit Testing work in Java?**
- **How does JUnit help?**

Automated Software Testing

Automated software testing:

- **saves time and money**
- **increases test coverage**
- **Improves accuracy**
- **does what manual testing cannot**
- **helps developers catch errors quickly**
- **improves the morale of the QA and Dev team**

Functional vs Non Functional Testing

Functional Testing	Non Functional Testing
Verifies the operations and actions of the application	Verifies the behaviour of the application
Based on the requirements of the customers	Based on the expectations of the customers
Easy to execute manually	Hard to execute manually
Enhances the behaviour of the application	Improves the performance of the application
Describes what the product does	Describes how the product works
Based on business requirements	Based on performance
Ex. Unit testing, Smoke testing, Integration testing	Ex. Performance testing, Load Testing and Stress testing

Unit vs Integration Testing

Unit Testing	Integration Testing
Each module of the software is tested separately	All the modules of the software are combined for testing
Tester knows the internal design of the software	Tester does not know the internal design of the software
Checks a single component of an application	Here, the behaviour of the integration module is considered
Performed first of all testing processes	Performed after unit testing and before system testing
Performed by the developer	Performed by the tester
No dependencies on the code outside the unit tested	Dependent on outside system like databases, hardware allocated for them etc.

Environment Setup (Demo)

1. **Verify Java installation in the machine**
2. **Set JAVA_HOME environment variable**
3. **Download JUnit archive file**
4. **Set JUnit_HOME environment variable**
5. **Set CLASSPATH environment variable point to JUnit jar location**
6. **Create Java class file and verify the result by running**

JUnit 5 Annotations

- @BeforeAll
- @AfterAll
- @BeforeEach
- @AfterEach
- @Test
- @Ignores
- @Test(timeout=500)
- @Test(IllegalArgumentException.class)

Unit Testing (Demo)

```
public String toUpperCaseAndConcat(String
message1, String message 2){

String
concatedMessage=message1.toUppercase().concat
(message2.toUppercase());

}
```

Assertions

- **Assertions are utility methods to support asserting conditions in tests**
- **These methods are accessible through the *Assert* class, in JUnit 4, and the *Assertions* class, in JUnit 5.**
- **Assert class is in org.junit package**
- **Assertion class is in org.junit.jupiter.api package**

Common Methods in Assert and Assertion Class

- assertEquals
- assertArrayEquals
- assertNull
- assertNotNull
- assertSame
- assertNotSame
- assertTrue
- assertFalse
- fail
- assertThat

Common Methods in Assert and Assertion Class (Demo)

```
@Test
public void givenTwoSameValuesThenCheckForEquality(){
assertTrue(10==10,”Ten is equal to ten”);
assertFalse(10>11,”Ten is not greater than eleven”);
}
```

Additional Methods in Assertion Class

- assertAll
- assertIterableEquals
- assertLinesMatch
- assertThows
- assertTimeout

Common Methods in Assert and Assertion Class (Demo slide)

```java
@Test
public void givenMultipleAssertionWhenAssertingAllThenOK() {
   assertAll(
           "heading",
           () -> assertEquals(4, 2 * 2, "4 is 2 times 2"),
           () -> assertEquals("java", "JAVA".toLowerCase())
           );
}
```

Execution Procedure (Demo)

```
public class ExecutionProcedureTest {//execute only once, in the starting
   @BeforeAll
   public static void beforeAll() {
       System.out.println("Before all tests called");
   }//execute only once, in the end
   @AfterAll
   public static void  afterAll() {
       System.out.println("After all tests called");
   }//execute for each test, before executing test
   @BeforeEach
   public void beforeEach() {
       System.out.println("Before each test called");
   }//execute for each test, after executing test
   @AfterEach
   public void afterEach() {
       System.out.println("After each test called");
   }
```

Test Execution Order

We can use *@TestMethodOrder* to control the execution order of tests.

1. We can select one of the three built-in orderers:
   a. *@Order* Annotation
   b. *Alphanumeric* Order
   c. Random Order
2. We can use our own custom order by implementing the *MethodOrderer* interface

Test Execution Order (Demo)

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class MethodOrderExecutionTest {
   private static StringBuilder output = new StringBuilder("");
   @Test
   @Order(1)
   public void firstMethod() {
       output.append("a");}
   @Test
   @Order(2)
   public void secondMethod() {
       output.append("b");
   }
   @Test
   @Order(3)
   public void thirdMethod() {
       output.append("c");
   }
   @AfterAll
   public static void assertOutput() {
       assertEquals(output.toString(), "abc");}}
```

Complete List of Annotations in JUnit 5

- @Test
- @ParameterizedTest
- @RepeatedTest
- @TestFactory
- @TestTemplate
- @TestMethodOrder
- @TestInstance
- @DisplayName
- @DisplayNameGeneration
- @BeforeEach
- @AfterEach
- @BeforeAll
- @AfterAll
- @Nested
- @Tag
- @Disabled
- @Timeout
- @ExtendWith
- @RegisterExtension
- @TempDir

Aggregating Tests in Suites (Demo)

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
  TestFeatureLogin.class,
  TestFeatureLogout.class,
  TestFeatureNavigate.class,
  TestFeatureUpdate.class
})
```

Ignore a Test (Demo)

```
@Disabled
public class AppTest {
//@Disabled
    @Test
    void testOnDev()
    {
        System.setProperty("ENV", "DEV");

Assumptions.assumeFalse("DEV".equals(System.getProperty("ENV")));
    }
    @Test
    void testOnProd()
    {
        System.setProperty("ENV", "PROD");

Assumptions.assumeFalse("DEV".equals(System.getProperty("ENV")));
    }}
```

Timeout for Tests

Tests that 'runaway' or take too long, can be automatically failed.It is implemented by:

- Timeout parameter on @Test Annotation (applies to the test method)
- Timeout Rule (applies to all test cases in the test class)

Timeout Test (Demo)

```
public class HasGlobalTimeout {
    public static String log;
    private final CountDownLatch latch = new CountDownLatch(1);

    @Rule
    public Timeout globalTimeout = Timeout.seconds(10); // 10
seconds max per method tested
    @Test
    public void testSleepForTooLong() throws Exception {
        log += "ran1";
        TimeUnit.SECONDS.sleep(100); // sleep for 100 seconds
    }
    @Test
    public void testBlockForever() throws Exception {
        log += "ran2";
        latch.await(); // will block    }}
```

Exception Testing

- JUnit provides the option of tracing the exception handling of a code
- The expected parameter is used along with @Test annotation to check whether the code throws a desired exception or not
- The method assertThrows has been added to the Assert class in version 4.13
- Apart from asserting that a particular function is throwing the desired exception, it returns the exception object to do further assertions.

Exception Test(Demo)

```
@Test
public void testExceptionMessage() {
  List<Object> list = new ArrayList<>();

  try {
    list.get(0);
    fail("Expected an IndexOutOfBoundsException to be 
thrown");
  } catch (IndexOutOfBoundsException 
anIndexOutOfBoundsException) {
    assertThat(anIndexOutOfBoundsException.getMessage(), 
is("Index: 0, Size: 0"));
  }
 }
```

Parameterized Test in JUnit 4

- Parameterized tests allow a developer to run the same test over and over again using different values.
- The custom runner Parameterized implements parameterized tests.
- When running a parameterized test class, instances are created for the cross-product of the test methods and the test data elements.

Parameterized Test in JUnit 5

- Parameterized tests make it possible to run a test multiple times with different arguments. They are declared just like regular @Test methods but use the @ParameterizedTest annotation instead.
- In addition, you must declare at least one *source* that will provide the arguments for each invocation and then *consume* the arguments in the test method.
- In order to use parameterized tests you need to add a dependency on the junit-jupiter-params artifact.

Paramterized Test (Demo)

```
@ParameterizedTest
@ValueSource(ints = {2, 4, 198, -120, 150})
public void givenValidNumberWhenEvenThenReturnTrue(int
number){
assertTrue(evenNumber.checkEvenNumber(number),”should
return true for even numbers”);
}
```

Key TakeAways

At the end of this session, you should be able to demonstrate how to:

- **implement unit testing in Java**
- **use JUNIT test annotations**
- **use assertion**
- **execute test suite**
- **Igore test**
- **timeout tests**
- **handle testing exceptions**
- **use parameterized test**

Thank You!