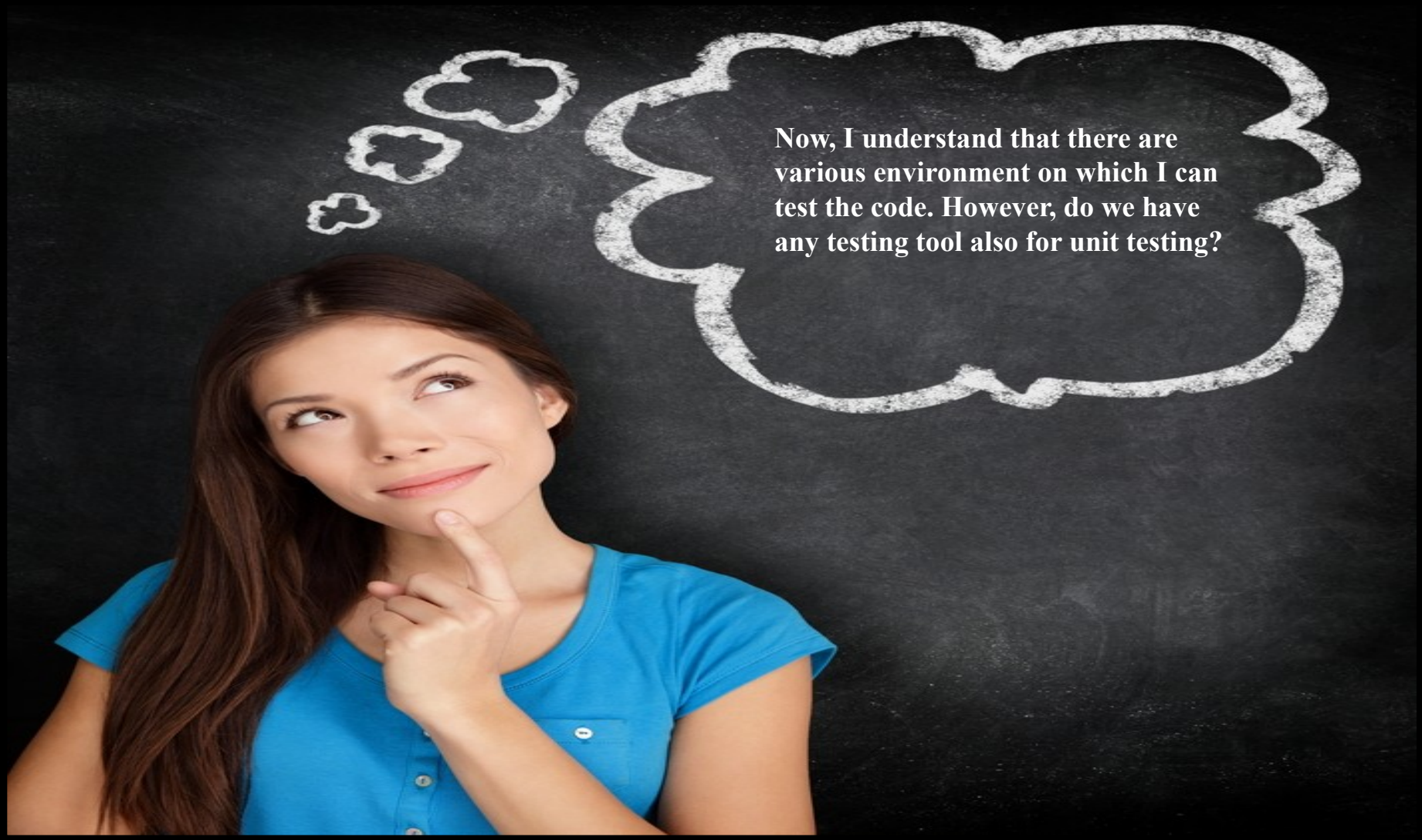
Testing JAVA Applications Using JUnit

Aesthetic: Tarkeshwar Barua

Identifying JUnit as a Testing Tool



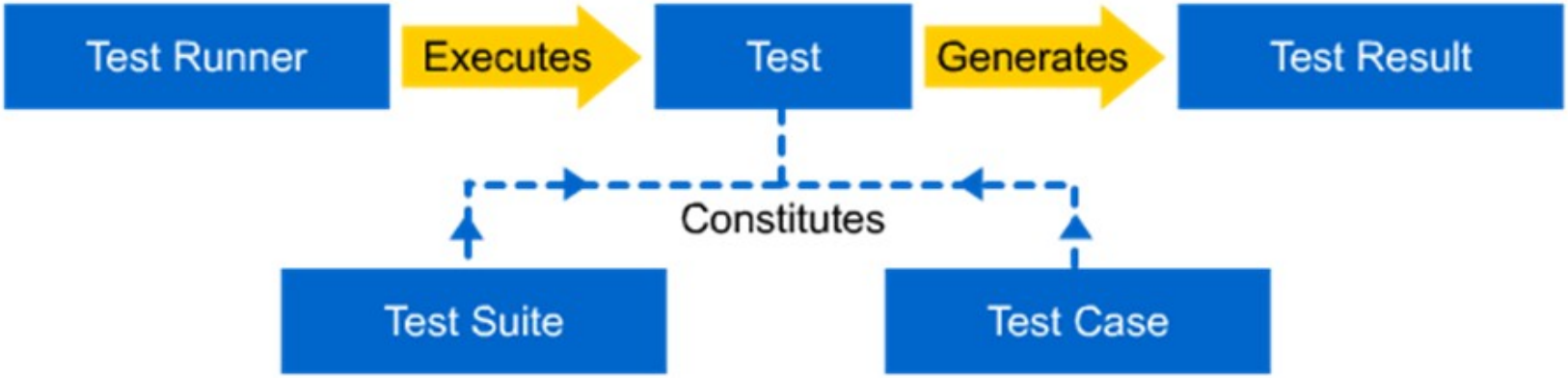
Identifying JUnit as a Testing Tool (Contd.)

JUnit is an open-source testing framework for Java that emphasizes the idea of testing the code before it is coded. The features of the JUnit framework are:

- Provides unit test case that verifies the output of the code under test
- Allows automatic execution of JUnit test cases individually
- Shows a progress bar that signifies the success or failure of test cases
- Allows organization of multiple test classes into a test suite
- Allows execution of test cases in a test suite one after the other automatically
- Saves time by allowing concurrent execution of multiple tests

Identifying JUnit as a Testing Tool (Contd.)

JUnit is primarily designed to perform unit testing. When you perform unit testing using JUnit, the JUnit framework performs certain tasks. The process used by the JUnit framework to execute a test and display the result constitutes the architecture of JUnit.



The JUnit Framework Architecture

Identifying JUnit as a Testing Tool (Contd.)

Elements of the JUnit framework architecture:

Test Case: This is the smallest unit of any JUnit test that verifies the functionality of a method being tested.

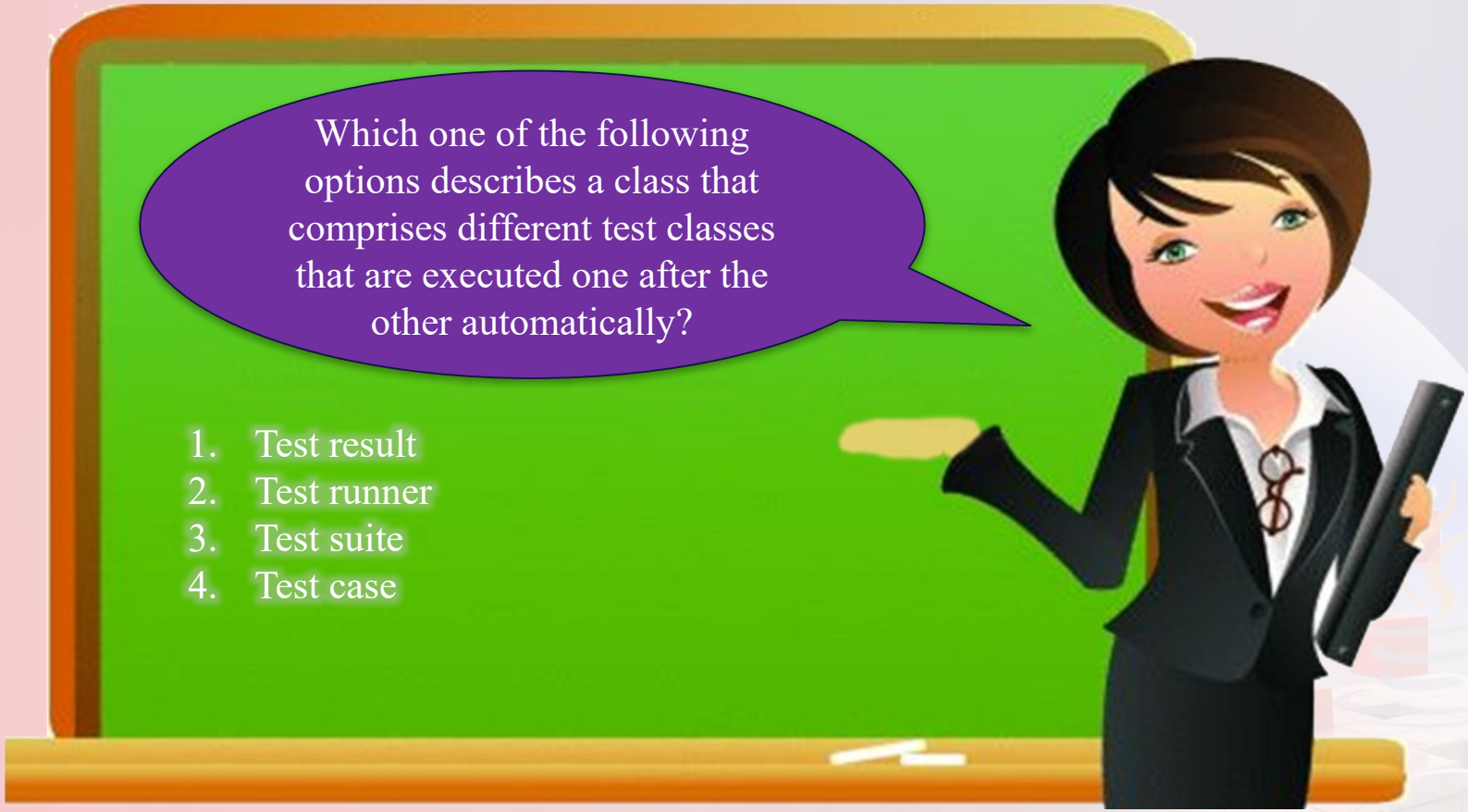
Test Suite: When you have multiple test classes and you want to automatically execute these classes one after the other, you can create a test suite.

Test: It is a collection of one or more test cases or test suites that are executed to perform the testing of an application.

Test Runner: When you execute a test, the test runner is executed in the background, which displays the result of the test in terms of pass or fail.

Test Result: It refers to the outcome of a test, which is displayed to the user. This test result is collected from the test class.

Just a Minute



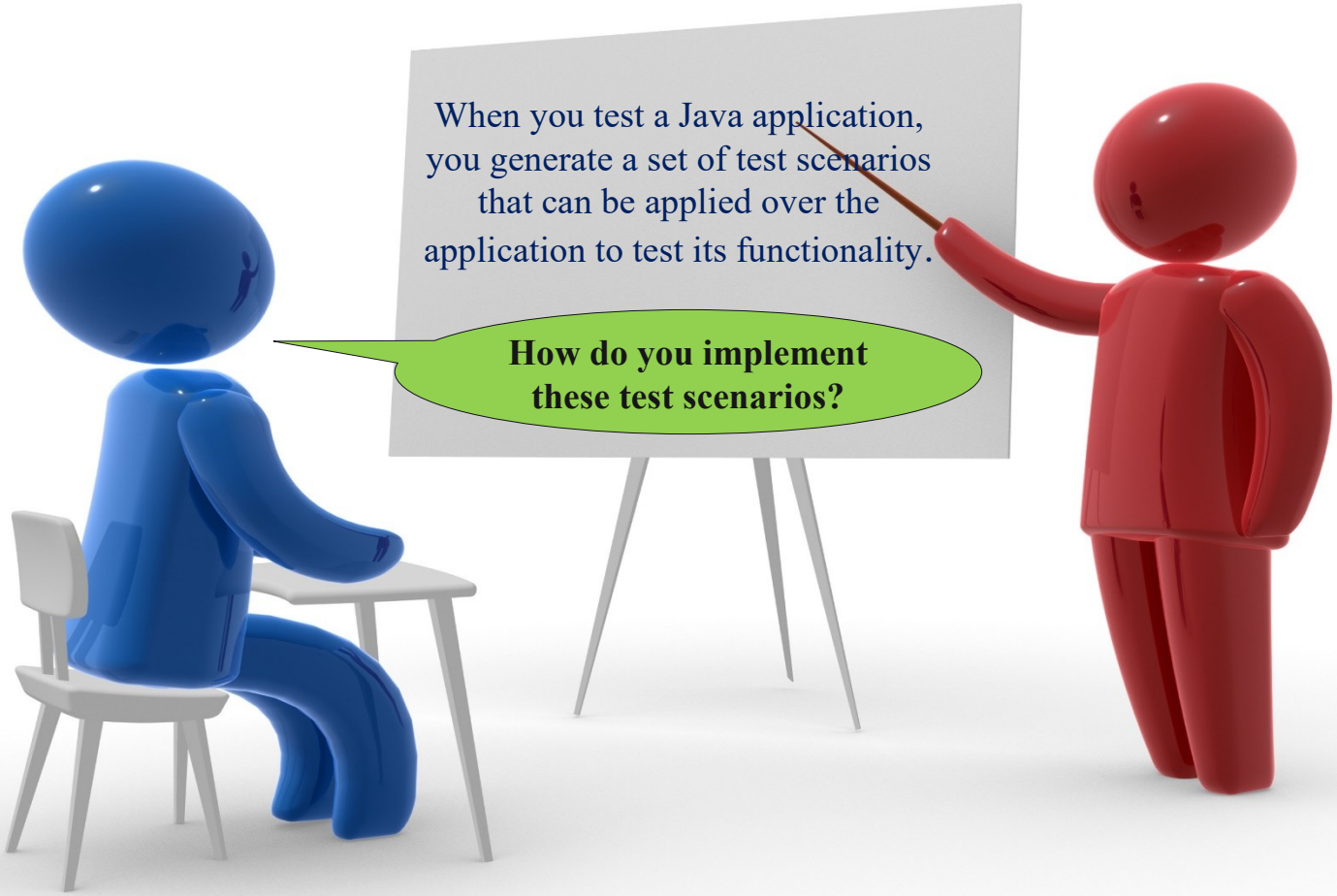
Answer: 3. Test suite

Demo: Executing Test Cases in JUnit

- **Problem Statement**
  - Explore the testing environment in NetBeans IDE using the JUnit framework.
  - **Prerequisite**: Ask your faculty to provide you with the **Activity1_2.zip** file required for completing this activity.

- **Solution**
  - To explore the testing environment in NetBeans IDE using the JUnit framework, you need to perform the following tasks:
    1. Open the **Activity1_2** project.
    2. Run the test file to execute the test.

Writing Test Cases in JUnit



Writing Test Cases in JUnit (Contd.)

Test case

Carries a set of instructions based on which you can test the functional aspect of the code being tested.

Consists of three parts:

- An input
- An event
- An expected response

Writing Test Cases in JUnit (Contd.)

In JUnit, you:

- Write test cases by creating test classes in a separate package to separate the test code from the application code.
- Can have multiple test cases inside a test class.
- Can have multiple test classes containing class-specific test cases.

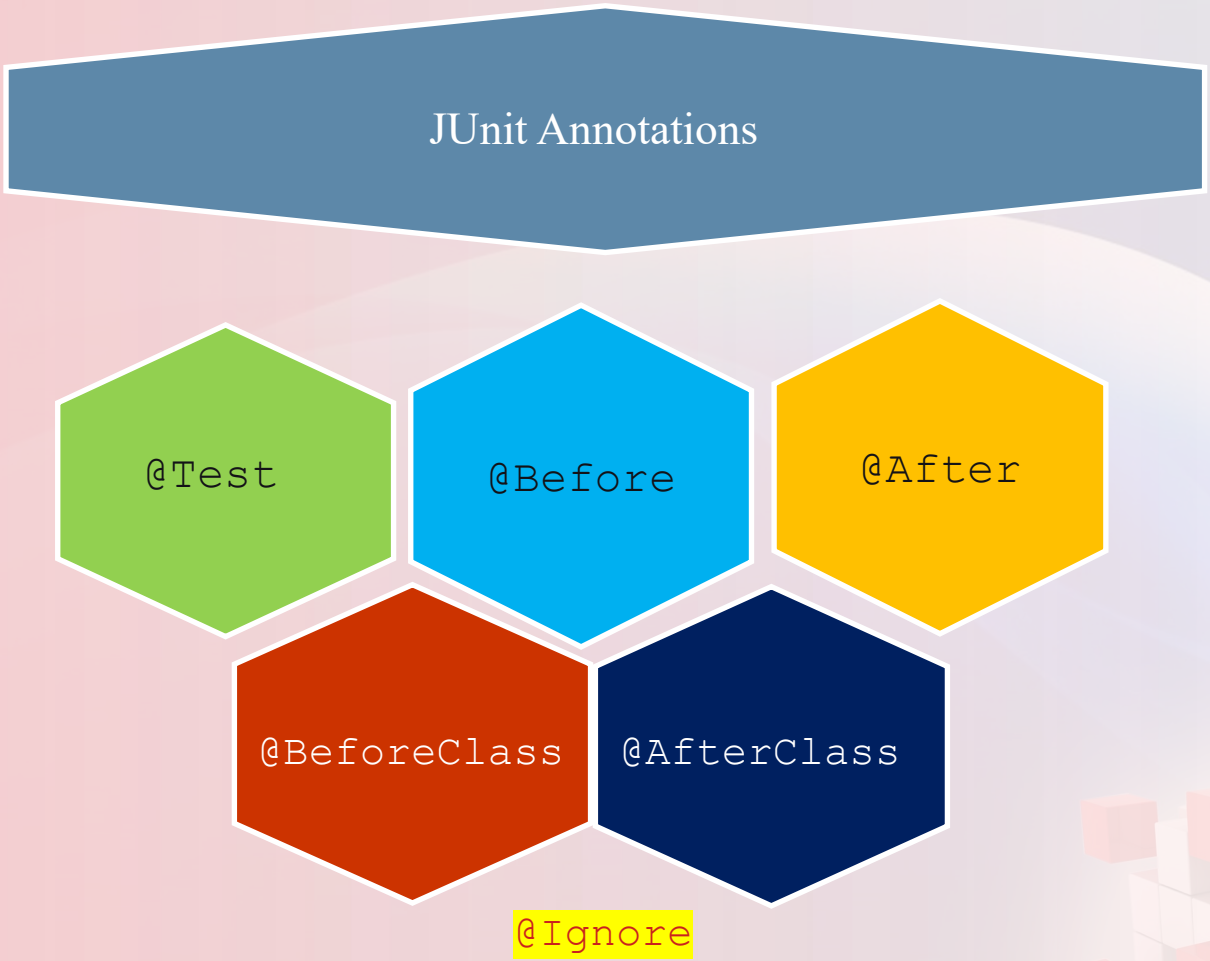
Writing Test Cases in JUnit (Contd.)

A test case, written to test an application, can either pass or fail.

- If a test case passes, it is indicated by a green signal.
- If a test case fails, it is indicated by a red signal.
- If a test class contains multiple test cases, some of them may pass, while others may fail.

The Results of Test Cases

Identifying JUnit Annotations



Identifying JUnit Annotations

Annotation	Description
@Test	Denotes that a method is a test method. Unlike JUnit 4's @Test annotation, this annotation does not declare any attributes, since test extensions in JUnit Jupiter operate based on their own dedicated annotations. Such methods are inherited unless they are overridden.
@ParameterizedTest	Denotes that a method is a parameterized test. Such methods are inherited unless they are overridden.
@RepeatedTest	Denotes that a method is a test template for a repeated test. Such methods are inherited unless they are overridden.
@TestFactory	Denotes that a method is a test factory for dynamic tests. Such methods are inherited unless they are overridden.
@TestTemplate	Denotes that a method is a template for test cases designed to be invoked multiple times depending on the number of invocation contexts returned by the registered providers. Such methods are inherited unless they are overridden.

Identifying JUnit Annotations

@TestClassOrder	Used to configure the test class execution order for @Nested test classes in the annotated test class. Such annotations are inherited.
@TestMethodOrder	Used to configure the test method execution order for the annotated test class; similar to JUnit 4's @FixMethodOrder. Such annotations are inherited.
@TestInstance	Used to configure the test instance lifecycle for the annotated test class. Such annotations are inherited.
@DisplayName	Declares a custom display name for the test class or test method. Such annotations are not inherited.
@DisplayNameGeneration	Declares a custom display name generator for the test class. Such annotations are inherited.
@BeforeEach	Denotes that the annotated method should be executed before each @Test, @RepeatedTest, @ParameterizedTest, or @TestFactory method in the current class; analogous to JUnit 4's @Before. Such methods are inherited unless they are overridden.

@AfterEach	Denotes that the annotated method should be executed after each @Test, @RepeatedTest, @ParameterizedTest, or @TestFactory method in the current class; analogous to JUnit 4's @After. Such methods are inherited unless they are overridden.
@BeforeAll	Denotes that the annotated method should be executed before all @Test, @RepeatedTest, @ParameterizedTest, and @TestFactory methods in the current class; analogous to JUnit 4's @BeforeClass. Such methods are inherited (unless they are hidden or overridden) and must be static (unless the "per-class" test instance lifecycle is used).
@AfterAll	Denotes that the annotated method should be executed after all @Test, @RepeatedTest, @ParameterizedTest, and @TestFactory methods in the current class; analogous to JUnit 4's @AfterClass. Such methods are inherited (unless they are hidden or overridden) and must be static (unless the "per-class" test instance lifecycle is used).
@Nested	Denotes that the annotated class is a non-static nested test class. @BeforeAll and @AfterAll methods cannot be used directly in a @Nested test class unless the "per-class" test instance lifecycle is used. Such annotations are not inherited.
@Tag	Used to declare tags for filtering tests, either at the class or method level; analogous to test groups in TestNG or Categories in JUnit 4. Such annotations are inherited at the class level but

Identifying JUnit Annotations

@Disabled	Used to disable a test class or test method; analogous to JUnit 4's @Ignore. Such annotations are not inherited.
@Timeout	Used to fail a test, test factory, test template, or lifecycle method if its execution exceeds a given duration. Such annotations are inherited.
@ExtendWith	Used to register extensions declaratively. Such annotations are inherited.
@RegisterExtension	Used to register extensions programmatically via fields. Such fields are inherited unless they are shadowed.
@TempDir	Used to supply a temporary directory via field injection or parameter injection in a lifecycle method or test method; located in the org.junit.jupiter.api.io package.

Identifying JUnit Annotations (Contd.)

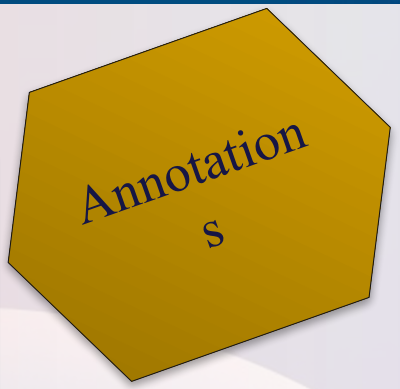
@Test Annotation

- Informs JUnit that the method, it has been attached to, needs to be run as a test case.
- Runs the method by first building a new instance of the class, and then invoking the annotated method.
- Syntax:

```
@Test
public void testXXX()
{
.......................
.......................
}
```

Identifying JUnit Annotations (Contd.)

```
public class JunitAnnotation {
  //execute before class
  @BeforeClass
  public static void beforeClass() {
    System.out.println("in before class");
  }
  //execute after class
  @AfterClass
  public static void  afterClass() {
    System.out.println("in after class");
  }
  //execute before test
  @Before
  public void before() {
    System.out.println("in before");
  }
  //execute after test
  @After
  public void after() {
    System.out.println("in after");
  }
  //test case
  @Test
  public void test() {
    System.out.println("in test");
  }
  //test case ignore and will not execute
  @Ignore
  public void ignoreTest() {
    System.out.println("in ignore test");
  }
}
```



Identifying JUnit Annotations (Contd.)

Runner

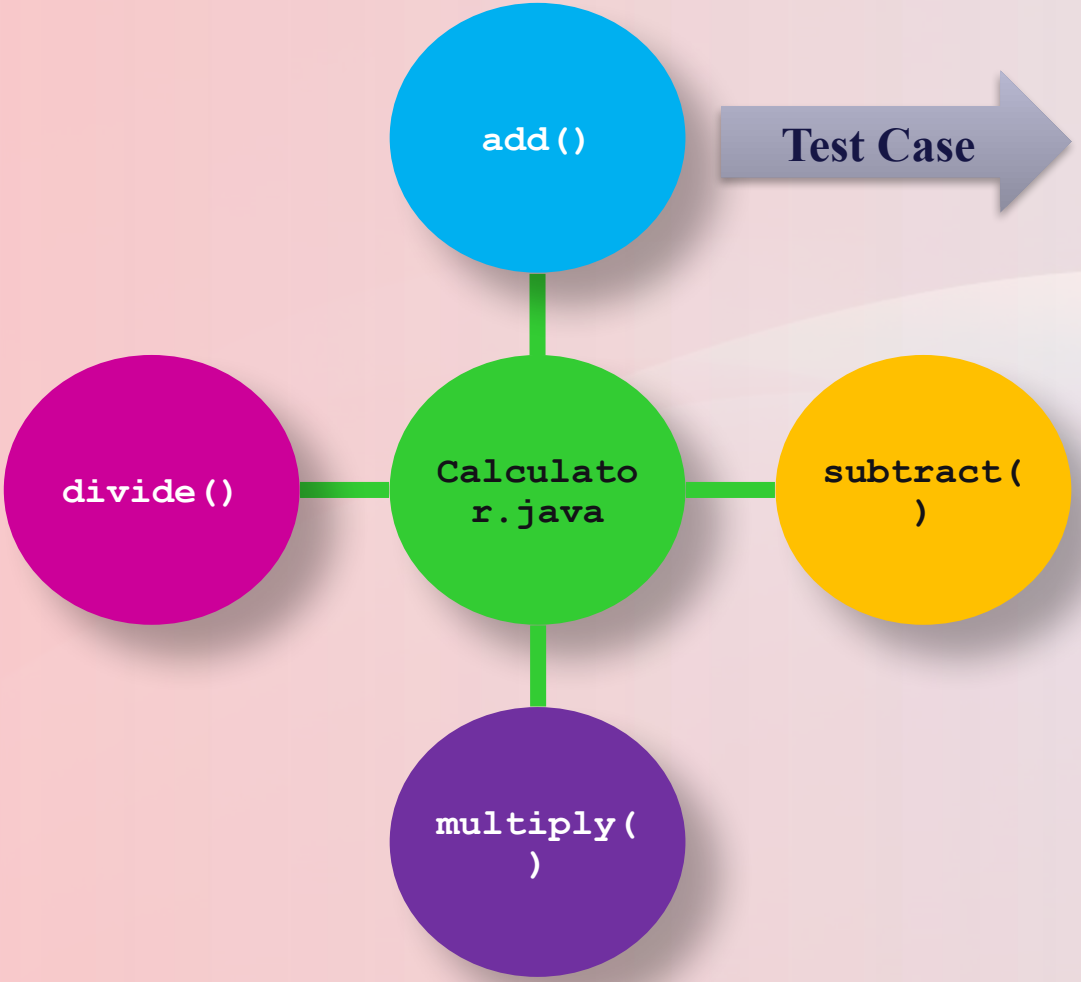
```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(JunitAnnotation.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }

      System.out.println(result.wasSuccessful());
   }
}
C:\JUNIT_WORKSPACE>javac JunitAnnotation.java TestRunner.java
C:\JUNIT_WORKSPACE>java TestRunner
```

Identifying JUnit Annotations (Contd.)



Test Case

```
@Test
public void testAdd() {
Calculator c1=new
Calculator();
int expectedresult=8;
assertEquals(expectedresult
,c1.add(3,5));
}
```

Identifying JUnit Annotations (Contd.)

@Before Annotation

- Causes the method, it has been attached to, run before the test method.
- Is ideally used when several tests require similar objects to be created before they are executed.
- Must be a public method and should not return any value.
- Syntax:

```
@Before
public void setUp(){
//runs before every test
method …
}
```

Initializing variables

```
@Before
  public void setUp()
  {
  value1=3;
  value2=5;
  }
```

Identifying JUnit Annotations (Contd.)

- Causes the method, it has been attached to, run after the test method.
- Must be a public method and should not return any value.
- Syntax:

```
@After
public void tearDown(){
//runs after every test
…
}
```

@After Annotation

```
@After
   public void tearDown()
   {
   value1=0;
   value2=0;
   }
```

Identifying JUnit Annotations (Contd.)

@BeforeClass Annotation

- Causes the method, it has been attached to, run once before any of the test methods in the class.
- Should be a public method, declared as static.
- Should neither contain any arguments nor return any value.
- Syntax:

```
@BeforeClass
public static void setUpClass(){
//runs once before all the test
  cases
…
}
```

```
@BeforeClass
public static void setUpClass(){
System.out.println("@BeforeClass - oneTimeSetUp");
}
```

Identifying JUnit Annotations (Contd.)

`@AfterClass`
Annotation

- Causes the method, it has been attached to, run after all the methods in the class.
- Should be a public method, declared as static.
- Should neither contain any arguments nor return any value.
- Syntax:

```
@AfterClass
public static void tearDownClass(){
//runs once after all the test
  cases
…
}
```

```
@AfterClass
public static void
tearDownClass()
{
System.out.println("@
AfterClass -
oneTimeTearDown");
}
```

Identifying JUnit Annotations (Contd.)

```
public class ExecutionProcedureJunit {

  //execute only once, in the starting
  @BeforeClass
  public static void beforeClass() {
    System.out.println("in before class");
  }

  //execute only once, in the end
  @AfterClass
  public static void  afterClass() {
    System.out.println("in after class");
  }
  //execute for each test, before executing test
  @Before
  public void before() {
    System.out.println("in before");
  }
  //execute for each test, after executing test
  @After
  public void after() {
    System.out.println("in after");
  }
  //test case 1
  @Test
  public void testCase1() {
    System.out.println("in test case 1");
  }
  //test case 2
  @Test
  public void testCase2() {
    System.out.println("in test case 2");
  }
}
```

```
public class TestRunner {
   public static void main(String[] args) {
      Result result =
JUnitCore.runClasses(ExecutionProcedureJunit.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }

      System.out.println(result.wasSuccessful());
   }
}
```

```
$javac ExecutionProcedureJunit.java TestRunner.java
$java TestRunner
```

Identifying JUnit Assert Statements

JUnit Assert Statements

- Are written to test the functionality of a function, once you have annotated a particular method as a test case using the `@Test` annotation.
- Allows you to verify the output of the method being tested with the expected output.

Identifying JUnit Assert Statements (Contd.)

Static methods in the `Assert` class

- Verify the expected and actual results.
- Compare the expected value with the actual value returned by the test.
- Throws the `AssertionException` exception, if the comparison fails.

Identifying JUnit Assert Statements (Contd.)

Assert Statements

- `assertEquals([message], expected, actual)`
- `assertEquals([message], expected, actual, tolerance)`
- `assertNull([message], object)`
- `assertNotNull([message], object)`
- `assertFalse([message], boolean condition)`
- `assertSame([message], expected, actual)`
- `assertTrue([message], boolean condition)`
- `assertNotSame([message], expected, actual)`

Identifying JUnit Assert Statements (Contd.)

Let us see an example to make you more familiar with the assert statements.

An application calculating the square root of a number

```
public class Arithmetic
{
public double
findSquareroot(double num)
          {
    return Math.sqrt(num);
}
}
```

Identifying JUnit Assert Statements (Cont

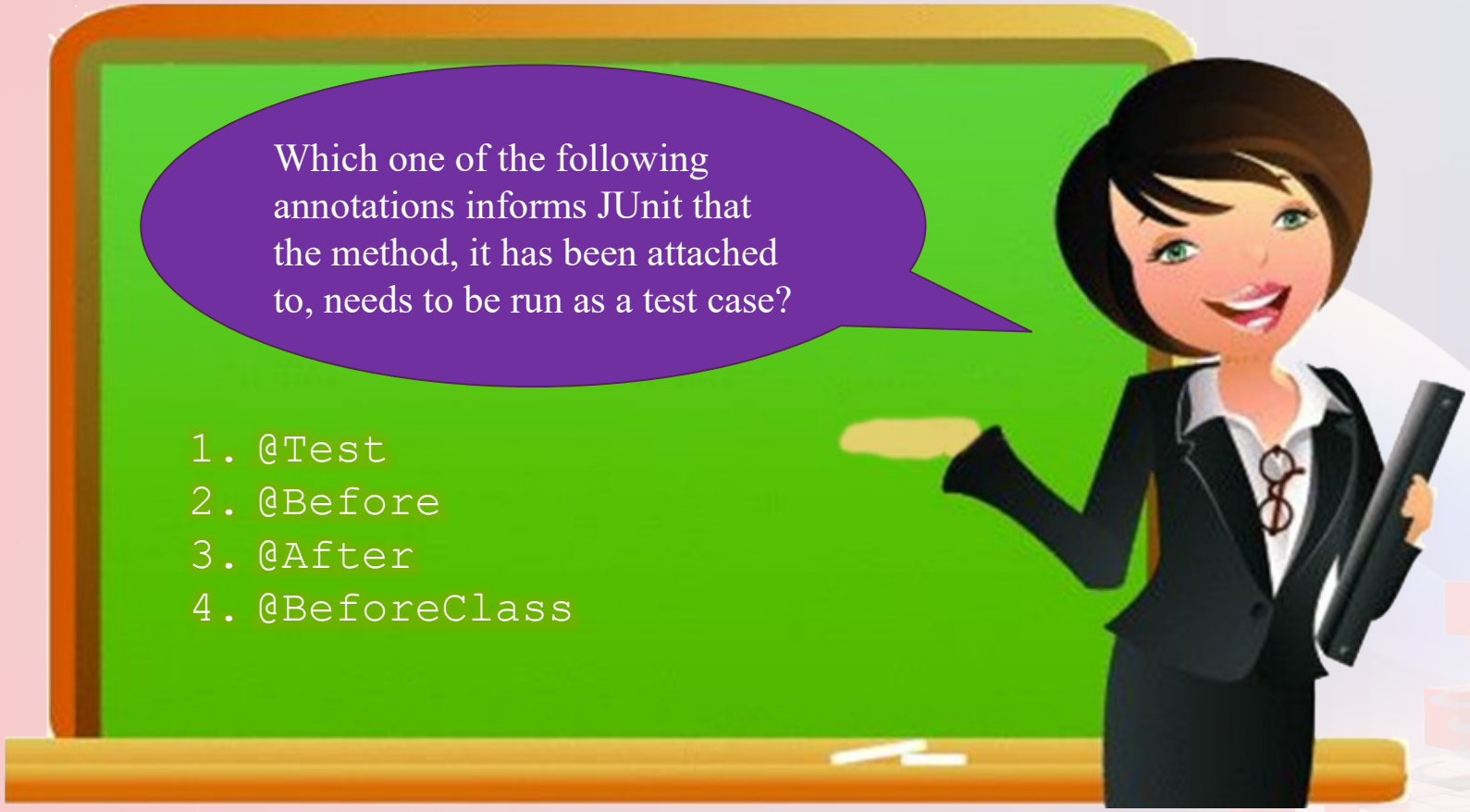
A test class that verifies the `findSquareroot()` method and compares the expected square root of a number to be equal to the actual value.

```
public class Arithmetic
{
public double
findSquareroot(double num)
                {
   return Math.sqrt(num);

}
}
```

```
public class ArithmeticTest {
    public ArithmeticTest() { }
    @Test
    public void testFindSquareroot() {
       Arithmetic instance = new       Arithmetic();
       double expResult = 2.5;
       double result = instance.findSquareroot(6.25);
   assertEquals("findSquareroot",expResult, result, 0.0);
    }
}
```

Just a Minute



Which one of the following annotations informs JUnit that the method, it has been attached to, needs to be run as a test case?

1. `@Test`
2. `@Before`
3. `@After`
4. `@BeforeClass`

Answer: 1. `@Test`

Meta-Annotations and Composed Annotations

JUnit Jupiter annotations can be used as meta-annotations. That means that you can define your own composed annotation that will automatically inherit the semantics of its meta-annotations.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.junit.jupiter.api.Tag;

@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Tag("fast")
public @interface Fast {
}
```

```
@Fast
@Test
void myFastTest() {
    // ...
}
```

Meta-Annotations and Composed Annotations

```
@Fast
@Test
void myFastTest() {
    // ...
}
```

Meta-Annotations and Composed Annotations

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

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

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Tag("fast")
@Test
public @interface FastTest {
}
```

```
@FastTest
void myFastTest() {
    // ...
}
```

Meta-Annotations and Composed Annotations

```
@FastTest
void myFastTest() {
    // ...
}
```

Test Class

any top-level class, static member class, or **@Nested** class that contains at least one test method.

Test classes must not be abstract and must have a single constructor.

Test Method

any instance method that is directly annotated or meta-annotated with

@Test,

@RepeatedTest,

@ParameterizedTest,

@TestFactory, or

@TestTemplate.

Lifecycle Method

any method that is directly annotated or meta-annotated with

@BeforeAll,

@AfterAll,

@BeforeEach, or

@AfterEach.

Test methods and lifecycle methods may be declared locally within the current test class, inherited from superclasses, or inherited from interfaces (see Test Interfaces and Default Methods).

Test methods and lifecycle methods must not be abstract and must not return a value (except *@TestFactory* methods which are required to return a value).

Special Note

Test classes, test methods, and lifecycle methods are not required to be **public**, but they must not be **private**.

It is generally recommended to omit the public modifier for test classes, test methods, and lifecycle methods unless there is a technical reason for doing so –

for example, when a test class is extended by a test class in another package.

Another technical reason for making classes and methods public is to simplify testing on the module path when using the Java Module System.

Special Note

```
class StandardTests {
    @BeforeAll
    static void initAll() {
    }
    @BeforeEach
    void init() {
    }
    @Test
    void succeedingTest() {
    }
    @Test
    void failingTest() {
        fail("a failing test");
    }
    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        // not executed
    }
    @Test
    void abortedTest() {
        assumeTrue("abc".contains("Z"));
        fail("test should have been aborted");
    }
    @AfterEach
    void tearDown() {
    }
    @AfterAll
    static void tearDownAll() {
    }
}
```

Display Names

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

@DisplayName("A special test case")
class DisplayNameDemo {

    @Test
    @DisplayName("Custom test name containing spaces")
    void testWithDisplayNameContainingSpaces() {
    }

    @Test
    @DisplayName("डॉ. तारकेश्वर बरुआ")
    void testWithDisplayNameContainingSpecialCharacters() {
    }

    @Test
    @DisplayName("😱")
    void testWithDisplayNameContainingEmoji() {
    }
```

Display Names

```
@DisplayName("A special test case")
class DisplayNameDemo {

    @Test
    @DisplayName("Custom test name containing spaces")
    void testWithDisplayNameContainingSpaces() {
        Assert.assertEquals(false, false);
    }

    @Test
    @DisplayName("डॉ. तारकेश्वर बरुआ")
    void testWithDisplayNameContainingSpecialCharacters() {
        List<String> list=new ArrayList();
        list.add("Tarkeshwar");
        list.add("Barua");
        List<String> list1=list;
        Assert.assertEquals(list, list1);
    }

    @Test
    @DisplayName("💀")
    void testWithDisplayNameContainingEmoji() {
        String[] name= {"Tarkeshwar", "Barua"};
        String[] name1 = name;
        assertEquals(name, name1);
    }

}
```

Custom Display Names

```
class DisplayNameGeneratorDemo {
    @Nested
    @DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
    class A_year_is_not_supported {
        @Test
        void if_it_is_zero() {
        }
        @DisplayName("A negative value for year is not supported by the leap year computation.")
        @ParameterizedTest(name = "For example, year {0} is not supported.")
        @ValueSource(ints = { -1, -4 })
        void if_it_is_negative(int year) {
        }
    }
    @Nested
    @IndicativeSentencesGeneration(separator = " -> ", generator = DisplayNameGenerator.ReplaceUnderscores.class)
    class A_year_is_a_leap_year {
        @Test
        void if_it_is_divisible_by_4_but_not_by_100() {
        }
        @ParameterizedTest(name = "Year {0} is a leap year.")
        @ValueSource(ints = { 2016, 2020, 2048 })
        void if_it_is_one_of_the_following_years(int year) {
        }
    }
}
```

Assertions

```
class AssertionsDemo
    private final Calculator calculator = new Calculator();
    private final Person person = new Person("Jane", "Doe");
    @Test
    void standardAssertions() {
        assertEquals(2, calculator.add(1, 1));
        assertEquals(4, calculator.multiply(2, 2), "The optional failure message is now the last parameter");
        assertTrue('a' < 'b',
            () -> "Assertion messages can be lazily evaluated -- "  + "to avoid constructing complex messages
unnecessarily.");
    }
    @Test
    void groupedAssertions() {
        // In a grouped assertion all assertions are executed, and all
        // failures will be reported together.
        assertAll("person",
            () -> assertEquals("Jane", person.getFirstName()),
            () -> assertEquals("Doe", person.getLastName())
        );
    }
}
```

Assertions

```
@Test
  void dependentAssertions() {
    // Within a code block, if an assertion fails the    // subsequent code in the same block will be skipped.
    assertAll("properties",
      () -> {
        String firstName = person.getFirstName();
        assertNotNull(firstName);

        // Executed only if the previous assertion is valid.
        assertAll("first name",  () -> assertTrue(firstName.startsWith("J")),  () -> assertTrue(firstName.endsWith("e"))
        );
      },
      () -> {
        // Grouped assertion, so processed independently  of results of first name assertions.
        String lastName = person.getLastName();
        assertNotNull(lastName);
        // Executed only if the previous assertion is valid.
        assertAll("last name", () -> assertTrue(lastName.startsWith("D")),  () -> assertTrue(lastName.endsWith("e"))
        );
      }
    );
  }
```

Assertions

```
@Test
  void exceptionTesting() {
    Exception exception = assertThrows(ArithmeticException.class, () -> calculator.divide(1, 0));
    assertEquals("/ by zero", exception.getMessage());
  }
  @Test
  void timeoutNotExceeded() {
    // The following assertion succeeds.
    assertTimeout(ofMinutes(2), () -> {    // Perform task that takes less than 2 minutes.     });
  }
  @Test
  void timeoutNotExceededWithResult() {
    // The following assertion succeeds, and returns the supplied object.
    String actualResult = assertTimeout(ofMinutes(2), () -> {
      return "a result";
    });
    assertEquals("a result", actualResult);
  }
  @Test
  void timeoutNotExceededWithMethod() {
    // The following assertion invokes a method reference and returns an object.
    String actualGreeting = assertTimeout(ofMinutes(2), AssertionsDemo::greeting);
    assertEquals("Hello, World!", actualGreeting);
  }
```

Assertions

```
@Test
    void timeoutExceeded() {
        // The following assertion fails with an error message similar to:
        // execution exceeded timeout of 10 ms by 91 ms
        assertTimeout(ofMillis(10), () -> {
            // Simulate task that takes more than 10 ms.
            Thread.sleep(100);
        });
    }

    @Test
    void timeoutExceededWithPreemptiveTermination() {
        // The following assertion fails with an error message similar to:
        // execution timed out after 10 ms
        assertTimeoutPreemptively(ofMillis(10), () -> {
            // Simulate task that takes more than 10 ms.
            new CountDownLatch(1).await();
        });
    }

    private static String greeting() {
        return "Hello, World!";
    }
}
```

Special Note

- Spring's testing support binds transaction state to the current thread (via a ThreadLocal) before a test method is invoked.
- If an executable or supplier provided to assertTimeoutPreemptively() invokes Spring-managed components that participate in transactions, any actions taken by those components will not be rolled back with the test-managed transaction.
- On the contrary, such actions will be committed to the persistent store (e.g., relational database) even though the test-managed transaction is rolled back.

Kotlin Assertion Support

```
class KotlinAssertionsDemo {
    private val person = Person("Jane", "Doe")
    private val people = setOf(person, Person("John", "Doe"))
    @Test
    fun `exception absence testing`() {
        val calculator = Calculator()
        val result = assertDoesNotThrow("Should not throw an exception") {
            calculator.divide(0, 1)
        }
        assertEquals(0, result)
    }
    @Test
    fun `expected exception testing`() {
        val calculator = Calculator()
        val exception = assertThrows<ArithmeticException> ("Should throw an exception") {
            calculator.divide(1, 0)
        }
        assertEquals("/ by zero", exception.message)
    }
```

Kotlin Assertion Support

```
@Test
  fun `grouped assertions`() {
    assertAll("Person properties",
      { assertEquals("Jane", person.firstName) },
      { assertEquals("Doe", person.lastName) }
    )
  }
  @Test
  fun `grouped assertions from a stream`() {
    assertAll("People with first name starting with J",
      people
        .stream()
        .map {
          // This mapping returns Stream<() -> Unit>
          { assertTrue(it.firstName.startsWith("J")) }
        }
    )
  }
  @Test
  fun `grouped assertions from a collection`() {
    assertAll("People with last name of Doe",
      people.map { { assertEquals("Doe", it.lastName) } }
    )
  }
```

Kotlin Assertion Support

```
@Test
    fun `timeout not exceeded testing`() {
        val fibonacciCalculator = FibonacciCalculator()
        val result = assertTimeout(Duration.ofMillis(1000)) {
            fibonacciCalculator.fib(14)
        }
        assertEquals(377, result)
    }

    @Test
    fun `timeout exceeded with preemptive termination`() {
        // The following assertion fails with an error message similar to:
        // execution timed out after 10 ms
        assertTimeoutPreemptively(Duration.ofMillis(10)) {
            // Simulate task that takes more than 10 ms.
            Thread.sleep(100)
        }
    }
}
```

Demo: Executing JUnit Test on Command Prompt

- **Problem Statement**
  - You have developed a Java application that multiplies two numbers. Now, you need to create a test case and verify this application. You need to display the test results on the command prompt window. For this, you have been provided with the `TestRunner.java` file. If the test cases pass, the message, `true`, is displayed on the command prompt window. How will you accomplish this task?
  - **Prerequisite**: To perform this activity, you need to use the **Multiply.zip** file. Extract the **Multiply** project folder and save it at an appropriate location on your system.