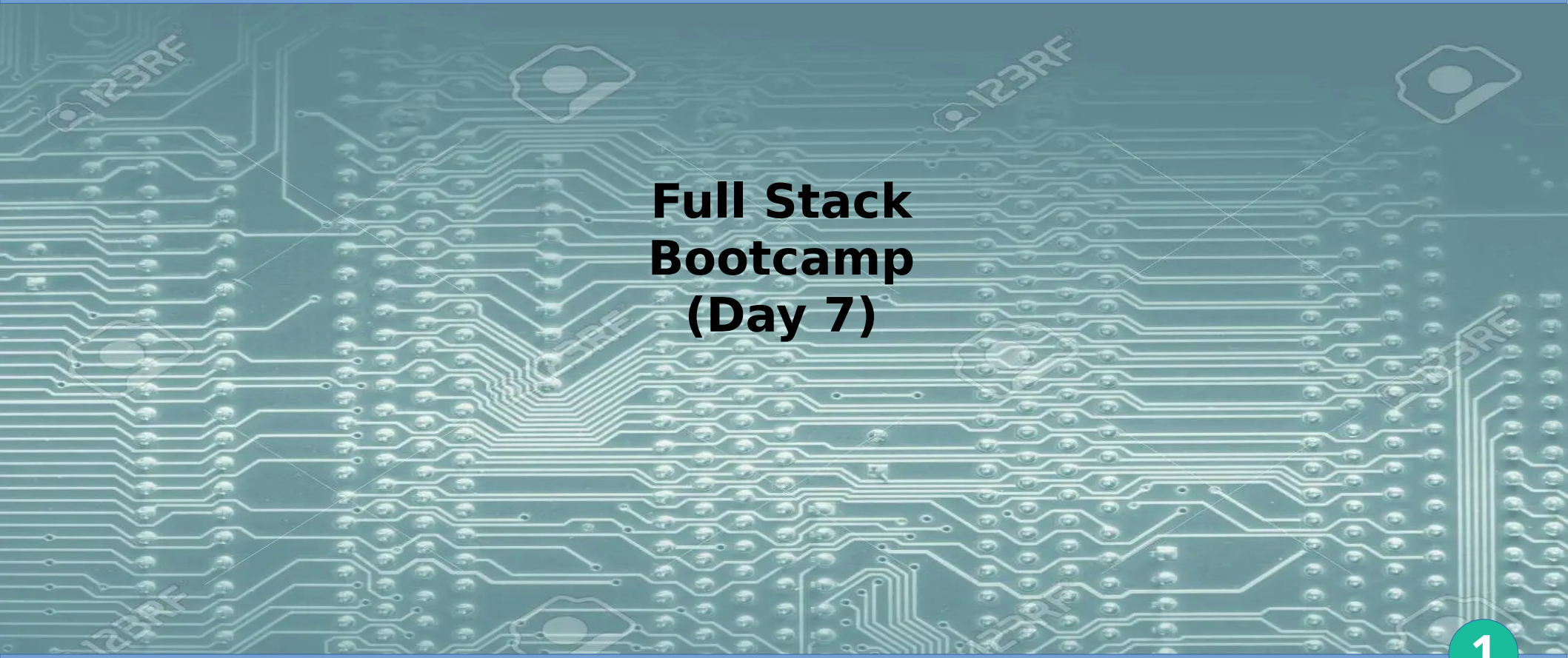
Testing Fundamentals



Agenda

- Day 1
  - ES6+
- Day 2
  - React Native
- Day 3
  - Angular
- Day 4
  - Springboot
  - SpringData
- Day 5
  - JSON
  - NoSQL
- Day 6
  - Relational
- Day 7
  - Junit
  - Mockito
- Day 8
  - Docker
- Day 9
  - Kubernetes
- Day 10
  - Images and tips

Test Driven Development

- Definition and explanation
- Test cases

JUNIT Overview

- JUnit has set a benchmark when it comes to testing Java applications. Since JUnit is compatible with almost all IDE's, organizations worldwide have adopted it to perform unit testing in the Java Programming Language. In this article titled “What is JUnit”, you will explore all the relevant topics needed to get a firm understanding of Junit.
- JUnit is a unit testing open-source framework for the Java programming language. Java Developers use this framework to write and execute automated tests. In Java, there are test cases that have to be re-executed every time a new code is added. This is done to make surethat nothing in the code is broken.

What to test

Test definition and analysis

- Test cases vs use cases
- Agile stories and test cases
- Sprint tests

Junit Maven Artifacts

```
<properties>
   <project.build.sourceEncoding>
            UTF-8
   </project.build.sourceEncoding>

<maven.compiler.source>11</maven.compiler.source>
   <maven.compiler.target>11</maven.compiler.target>
</properties>
<build>
      <plugins>
          <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.22.2</version>
          </plugin>
     <plugin>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>2.22.2</version>
     </plugin>
  </plugins>
</build>
```

```
<dependencies>
  <dependency>
     <groupId>org.junit.jupiter</groupId>
     <artifactId>junit-jupiter-api</artifactId>
     <version>5.7.2</version>
     <scope>test</scope>
  </dependency>
  <dependency>
     <groupId>org.junit.jupiter</groupId>
     <artifactId>junit-jupiter-engine</artifactId>
     <version>5.7.2</version>
     <scope>test</scope>
  </dependency>
</dependencies>
```

Creating the first tests

```
package com.beisbolicos.testing.first;
import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.Test;
class AClassWithOneJUnitTest {
        @Test
        void demoTestMethod() {
                assertTrue(true);
                }
        }
```

Test classes and methods

Display names. Set the test name with spaces, special characters, and even emojis — that will be displayed in test reports and by test runners and IDEs.Display names generators

- Assertions:
  - assertEquals, assertTrue, assertTimeout,
  - assertThrows
  - assertTimeoutPreemptively
- Assumptions
  - assumeTrue, assumingThat

@Test Annotations

- 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**.

A “standard” test class

```
import static org.junit.jupiter.api.Assertions.fail;
import static
org.junit.jupiter.api.Assumptions.assumeTrue;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;
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() {
    }
}
```

Exercise 1 & 2

- My first test

@ParameterizedTest

- 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.
- The following example demonstrates a parameterized test that uses the @ValueSource annotation to specify a String array as the source of arguments.

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
        assertTrue(StringUtils.isPalindrome(candidate));
}
```

ParameterizedTest setup

- Parameterized test methods typically consume arguments directly from the configured source (see Sources of Arguments) following a one-to-one correlation between argument source index and method parameter index.
- Additional arguments may also be provided by a **ParameterResolver** (e.g., to obtain an instance of **TestInfo**, **TestReporter**, etc.).
- Specifically, a parameterized test method must declare formal parameters according to the following rules.
  - Zero or more indexed arguments must be declared first.
  - Zero or more aggregators must be declared next.
  - Zero or more arguments supplied by a **ParameterResolver** must be declared last.
- An aggregator is any parameter of type **ArgumentsAccessor** or any parameter annotated with **@AggregateWith**.

Exercise 3

- Testing math functions

Sources of Arguments

- @ValueSource
- Null and Empty Sources
- @EnumSource
- @MethodSource
- @CsvSource
- @CsvFileSource
- @ArgumentsSource

Sources of Arguments (2)

- Argument Conversion
- Widening Conversion
- Implicit Conversion
- Explicit Conversion

Argument Aggregation

- By default, each argument provided to a @ParameterizedTest method corresponds to a single method parameter. Consequently, argument sources which are expected to supply a large number of arguments can lead to large method signatures.
- In such cases, an ArgumentsAccessor can be used instead of multiple parameters. Using this API, you can access the provided arguments through a single argument passed to your test method. In addition, type conversion is supported as discussed in Implicit Conversion.
- Custom Aggregators

@RepeatedTest

- Unit Jupiter provides the ability to repeat a test a specified number of times by annotating a method with **@RepeatedTest** and specifying the total number of repetitions desired. Each invocation of a repeated test behaves like the execution of a regular **@Test** method with full support for the same lifecycle callbacks and extensions.

```
@RepeatedTest(10)
void repeatedTest() {
        // ...
}
```

@TestFactory

- Dynamic test methods are annotated with @TestFactory and allow you to create multiple tests of type DynamicTest with your code. They can return:
  - an Iterable
  - a Collection
  - a Stream
- JUnit 5 creates and runs all dynamic tests during test execution.
- Methods annotated with @BeforeEach and @AfterEach are not called for dynamic tests.

@TestFactory

```
class DynamicTestCreationTest {
@TestFactory
Stream<DynamicTest> testDifferentMultiplyOperations() {
    MyClass tester = new MyClass();
    int[][] data = new int[][] { { 1, 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };
    return Arrays.stream(data).map(entry -> {
         int m1 = entry[0];
         int m2 = entry[1];
         int expected = entry[2];
        return dynamicTest(m1 + " * " + m2 + " = " + expected, () → {
             assertEquals(expected, tester.multiply(m1, m2));
        });
   });
}
```

```
// class to be tested
class MyClass {
    public int multiply(int i, int j) {
          return i * j;
     }
  }
}
```

@BeforeEach

```
class CalculatorTest {
  Calculator calculator;
  @BeforeEach
  void setUp() {
    calculator = new Calculator();
  }
  @Test
  @DisplayName("Simple multiplication should work")
  void testMultiply() {
      assertEquals(20, calculator.multiply(4, 5),
            "Regular multiplication should work");
  }
  @RepeatedTest(5)
  @DisplayName("Ensure correct handling of zero")
  void testMultiplyWithZero() {
      assertEquals(0, calculator.multiply(0, 5), "Multiple with zero should be zero");
      assertEquals(0, calculator.multiply(5, 0), "Multiple with zero should be zero");
  }
}
```

1. The method annotated with @BeforeEach runs before each test
2. A method annotated with @Test defines a test method
3. @DisplayName can be used to define the name of the test which is displayed to the user
4. This is an assert statement which validates that expected and actual value is the same, if not the message at the end of the method is shown
5. @RepeatedTest defines that this test method will be executed multiple times, in this example 5 times

@AfterEach

- Methods annotated with **@BeforeEach** and **@AfterEach** are not called for dynamic tests. This means, that you can’t use them to reset the test object, if you change it’s state in the lambda expression for a dynamic test (**@TestFactory**)

- **@BeforeTest** and **@AfterText** can be overriden or generalized using callbacks.

BeforeTest and AfterTest callbacks

```
public class TimingExtension implements BeforeTestExecutionCallback,
AfterTestExecutionCallback {
  private static final Logger logger =
          Logger.getLogger(TimingExtension.class.getName());
  }
  private static final String START_TIME = "start time";
  @Override
  public void beforeTestExecution(ExtensionContext context) throws Exception
{
      getStore(context).put(START_TIME, System.currentTimeMillis());
  }
   @Override
   public void afterTestExecution(ExtensionContext context) throws Exception {
         Method testMethod = context.getRequiredTestMethod();
         long startTime = getStore(context).remove(START_TIME, long.class);
         long duration = System.currentTimeMillis() - startTime;
         logger.info(() ->
         String.format("Method [%s] took %s ms.", testMethod.getName(),
duration));
  }
   private Store getStore(ExtensionContext context) {
      return context.getStore(Namespace.create(getClass(),
               context.getRequiredTestMethod()));
   }
}
```

- Since the TimingExtensionTests class registers the TimingExtension via **@ExtendWith**, its tests will have this timing applied when they execute.
- A test class that uses the example TimingExtension

```
@ExtendWith(TimingExtension.class)
class TimingExtensionTests {
   @Test
   void sleep20ms() throws Exception {
         Thread.sleep(20);
   }
   @Test
   void sleep50ms() throws Exception {
      Thread.sleep(50);
   }}}
```

@BeforeAll

- Only non-static nested classes (i.e. inner classes) can serve as @**Nested** test classes.
- Nesting can be arbitrarily deep, and those inner classes are subject to full lifecycle support with one exception: @**BeforeAll** and @**AfterAll** methods do not work by default. The reason is that Java does not allow static members in inner classes.
- However, this restriction can be circumvented by annotating a @**Nested** test class with @TestInstance(Lifecycle.PER_CLASS) (see Test Instance Lifecycle).

@Nested

```
class UsingNestedTests {
  private List<String> list;
  @BeforeEach
  void setup() {
    list = Arrays.asList("JUnit 5", "Mockito");
  }
  @Test
  void listTests() {
  }
  assertEquals(2, list.size());
 }
// TODO define inner class with @Nestled write one tests named
// checkFirstElement() to check that the first list element is "JUnit
4"
```

```
// write one tests named checkSecondElement() to
check that the first list element is "JUnit 4"
  @DisplayName("Grouped tests for checking
members")
  @Nested
  class CheckMembers {
  @Test
  void checkFirstElement() {
     assertEquals(("JUnit 5"), list.get(0));
  }
  @Test
  void checkSecondElement() {
     assertEquals(("Mockito"), list.get(1));
  }
}
}
```

Exercise 5

- Testing, testing, testing

Meta 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.
  - For example, instead of copying and pasting @Tag("fast") throughout your code base (see Tagging and Filtering), you can create a custom composed annotation named @Fast as follows. @Fast can then be used as a drop-in replacement for @Tag("fast").

Disabling Tests

- Entire test classes or individual test methods may be disabled via the **@Disabled** annotation, via one of the annotations discussed in Conditional Test Execution, or via a custom ExecutionCondition.

```
@Disabled("Disabled until bug #99 has been fixed")
class DisabledClassDemo {
    @Test
    void testWillBeSkipped() {
    }
}
```

Conditional Test Execution

- The **ExecutionCondition** extension API in JUnit Jupiter allows developers to either enable or disable a container or test based on certain conditions programmatically. The simplest example of such a condition is the built-in **DisabledCondition** which supports the **@Disabled** annotation (see Disabling Tests). In addition to **@Disabled**, JUnit Jupiter also supports several other annotation-based conditions in the **org.junit.jupiter.api.condition** package that allow developers to enable or disable containers and tests declaratively. When multiple **ExecutionCondition** extensions are registered, a container or test is disabled as soon as one of the conditions returns disabled. If you wish to provide details about why they might be disabled, every annotation associated with these built-in conditions has a **disabledReason** attribute available for that purpose.

```
@Test
@EnabledOnOs({OS.WINDOWS, OS.MAC})
public void shouldRunBothWindowsAndMac() {
//...
}
```

```
@Test
@DisabledOnOs(OS.LINUX)
public void shouldNotRunAtLinux() {
//...
}
```

Execution conditions

- Operating System Conditions
- Java Runtime Environment
  - @EnabledOnJre({JRE.JAVA_10, JRE.JAVA_11})
- Conditions
  - @EnabledIf("'FR' == systemProperty.get('user.country')")
  - @DisabledIf("java.lang.System.getProperty('os.name').toLowerCase().contains('mac')")
- System Property Conditions
  - @EnabledIfSystemProperty(named = "java.vm.vendor", matches = "Oracle.*")
- Environment Variable Conditions
  - @EnabledIfEnvironmentVariable(named = "GDMSESSION", matches = "ubuntu")
- Custom Conditions

Test Execution Order

- By default, test classes and methods will be ordered using an algorithm that is deterministic but intentionally non obvious. This ensures that subsequent runs of a test suite execute test classes and test methods in the same order, thereby allowing for repeatable builds.

Method Order

- Although true unit tests typically should not rely on the order in which they are executed, there are times when it is necessary to enforce a specific test method execution order — for example, when writing integration tests or functional tests where the sequence of the tests is important, especially in conjunction with @**TestInstance**(Lifecycle.PER_CLASS).
- To control the order in which test methods are executed, annotate your test class or test interface with @**TestMethodOrder** and specify the desired **MethodOrderer** implementation. You can implement your own custom **MethodOrderer** or use one of the following built-in **MethodOrderer** implementations.

Class order

```
@TestClassOrder(ClassOrderer.OrderAnnotation.class)
class OrderedNestedTestClassesDemo {
    @Nested
    @Order(1)
    class PrimaryTests {
        @Test
        void test1() {
        }
    }
    @Nested
    @Order(2)
    class SecondaryTests {
```

Exercise 6

- All in one

Changing the Default Test

- If a test class or test interface is not annotated with **@TestInstance,** JUnit Jupiter will use a default lifecycle mode. The standard default mode is PER_METHOD; however, it is possible to change the default for the execution of an entire test plan. To change the default test instance lifecycle mode, set the junit.jupiter.testinstance.lifecycle.default Configuration parameter to the name of an enum constant defined in **TestInstance**.Lifecycle, ignoring case. This can be supplied as a JVM system property, as a configuration parameter in the **LauncherDiscoveryRequest** that is passed to the Launcher, or via the JUnit Platform configuration file

Dependency Injection for Constructors and Methods

- In all prior JUnit versions, test constructors or methods were not allowed to have parameters (at least not with the standard Runner implementations). As one of the major changes in JUnit Jupiter, both test constructors and methods are now permitted to have parameters. This allows for greater flexibility and enables Dependency Injection for constructors and methods.
- ParameterResolver defines the API for test extensions that wish to dynamically resolve parameters at runtime. If a test class constructor, a test method, or a lifecycle method (see Test Classes and Methods) accepts a parameter, the parameter must be resolved at runtime by a registered ParameterResolver.

Test Templates

- A **@TestTemplate** method is not a regular test case but rather a template for test cases. As such, it is designed to be invoked multiple times depending on the number of invocation contexts returned by the registered providers. Thus, it must be used in conjunction with a registered **TestTemplateInvocationContextProvider** extension. Each invocation of a test template method behaves like the execution of a regular **@Test** method with full support for the same lifecycle callbacks and extensions.

Dynamic Tests

- Dynamic Test Examples
  - https://www.youtube.com/watch?v=0n-qspcJfrc
- URI Test Sources for Dynamic Tests

Parallel Execution

- Configuration
- Synchronization
- Built-in Extensions

MOCKITO

- What’s a mock?
- MockitoOverview

Seting up

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.21.0</version>
    <scope>test</scope>
</dependency>
<plugin>
     <artifactId>maven-surefire-plugin</artifactId>
     <version>2.19.1</version>
<dependencies>
<dependency>
     <groupId>org.junit.platform</groupId><artifactId>junit-platform-surefire-provider</artifactId>
     <version>1.0.1</version>
</dependency>
</dependencies>
</plugin>
```

Junit integration

- Mockito provides an implementation for JUnit5 extensions in the library — mockito-junit-jupiter.
- We'll include this dependency in our pom.xml:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>2.23.0</version>
    <scope>test</scope>
</dependency>
```

Junit and Mockito

- Step 1 − Create an interface called CalculatorService to provide mathematical functions
- File: CalculatorService.java

```
public interface CalculatorService {
   public double add(double input1, double input2);
   public double subtract(double input1, double input2);
   public double multiply(double input1, double input2);
   public double divide(double input1, double input2);
}
```

Junit and Mockito

- Step 2 − Create a JAVA class to represent MathApplication
- File: MathApplication.java

```
public class MathApplication {
   private CalculatorService calcService;
   public void setCalculatorService(CalculatorService calcService){
      this.calcService = calcService;
   }

   public double add(double input1, double input2){
      return calcService.add(input1, input2);
   }

   public double subtract(double input1, double input2){
      return calcService.subtract(input1, input2);
   }

   public double multiply(double input1, double input2){
      return calcService.multiply(input1, input2);
   }

   public double divide(double input1, double input2){
      return calcService.divide(input1, input2);
   }
}
```

Junit and Mockito

- Step 3 − Test the MathApplication class
- Let's test the MathApplication class, by injecting in it a mock of calculatorService. Mock will be created by Mockito.

Junit and Mockito

```
import static org.mockito.Mockito.when;
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;
// @RunWith attaches a runner with the test class to initialize the test data
@RunWith(MockitoJUnitRunner.class)
public class MathApplicationTester {

   //@InjectMocks annotation is used to create and inject the mock object
   @InjectMocks
   MathApplication mathApplication = new MathApplication();
   //@Mock annotation is used to create the mock object to be injected
   @Mock
   CalculatorService calcService;
   @Test
   public void testAdd(){
      //add the behavior of calc service to add two numbers
      when(calcService.add(10.0,20.0)).thenReturn(30.00);

      //test the add functionality
      Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);
   }
}
```

Initializing

New Mocks

Generate Mock

Junit and Mockito

- Step 4 − Create a class to execute to test cases
- Create a java class file named TestRunner in C> Mockito_WORKSPACE to execute Test case(s).

JUnit and Mockito File: TestRunner.java

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

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

Junit and Mockito

- Step 5 − Verify the Result
  - Compile the classes using javac compiler as follows −
  - C:\Mockito_WORKSPACE>javac CalculatorService.java MathApplication.
  - java MathApplicationTester.java TestRunner.java
- Now run the Test Runner to see the result
  - C:\Mockito_WORKSPACE>java TestRunner
- Verify the output.
  - true

Exercise 7

- Generate a test with Mockito

Mockito - Adding behavior

- Mockito adds a functionality to a mock object using the methods when(). Take a look at the following code snippet.

```
//add the behavior of calc service to add two numbers
when(calcService.add(10.0,20.0)).thenReturn(30.00);
```

- Here we've instructed Mockito to give a behavior of adding 10 and 20 to the add method of calcService and as a result, to return the value of 30.00.
- At this point of time, Mock recorded the behavior and is a working mock object.

```
//add the behavior of calc service to add two numbers
when(calcService.add(10.0,20.0)).thenReturn(30.00);
```

Mockito-Adding behiavor - Example

```
import static org.mockito.Mockito.when;
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;
// @RunWith attaches a runner with the test class to initialize the test data
@RunWith(MockitoJUnitRunner.class)
public class MathApplicationTester {

   //@InjectMocks annotation is used to create and inject the mock object
   @InjectMocks
   MathApplication mathApplication = new MathApplication();
   //@Mock annotation is used to create the mock object to be injected
   @Mock
   CalculatorService calcService;
   @Test
   public void testAdd(){
      //add the behavior of calc service to add two numbers
      when(calcService.add(10.0,20.0)).thenReturn(30.00);

      //test the add functionality
      Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);
   }
}
```

Mockito – Verifying behavior

- Mockito can ensure whether a mock method is being called with reequired arguments or not. It is done using the verify() method. Take a look at the following code snippet.

```
//test the add functionality
Assert.assertEquals(calcService.add(10.0, 20.0),30.0,0);
```

```
//verify call to calcService is made or not with same arguments.
verify(calcService).add(10.0, 20.0);
```

Day 7 summary

- Test Driven Development
- Unit Test frameworks
- JUnit
- Mockito