Junit, Mockito and Logging – Saran V S

Agenda

- Junit
  - What is Junit?
  - Junit Features
  - How Junit Works
  - What is Unit Testing
  - Junit Test Methods
  - Basic Annotations
  - Exception Testing
  - Using Timeout
  - Assertions
  - Test Suits
  - Parametarized tests
- Mockito
  - Overview
  - Creating Mock Instances
  - Stubbing Method calls
  - Verifications
  - Argument matching
  - Stubbing Consecutive calls
  - Verification Order
  - Capturing arguments
  - Spying
  - Partial Mocks
- Logging (Log4j 2)
  - Overview
  - Features
  - Architecture
  - Configuration
  - Logging Methods
  - Logging Levels
  - Log Formatting

What is Junit?

- It is just a testing tool
- Open Source
- Easy to implement
- Increase the stability of software. (Wait what?)

Lets hold back for a second...

Junit Features

- JUnit is an open source framework, which is used for writing and running tests.
- Provides annotations to identify test methods.
- Provides assertions for testing expected results.
- Provides test runners for running tests.
- JUnit tests allow you to write codes faster, which increases quality.
- JUnit is elegantly simple. It is less complex and takes less time.
- JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.
- JUnit tests can be organized into test suites containing test cases and even other test suites.
- JUnit shows test progress in a bar that is green if the test is running smoothly, and it turns red when a test fails. (LOL.. When you desperately wanna increase the features)

How Junit Works?

You can only test and see... but we'll see this in some time (Trust me!)

What is Unit Testing?

- A Unit Test Case is a part of code, which ensures that another part of code (method) works as expected.
- A formal written unit test case is characterized by a known input and an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a post-condition.
- There must be at least two unit test cases for each requirement − one positive test and one negative test. If a requirement has sub-requirements, each sub-requirement must have at least two test cases as positive and negative.

Junit Testing methods

Let’s setup Junit

Live Action coming shortly!

Basic Annotations

- @Test
- @Before
  - @BeforeEach
  - @BeforeAll
- @After
  - @AfterEach
  - @AfterAll
- @BeforeClass
- @AfterClass
- @Ignore

Exception Testing

- Exception: Exceptions from methods can also be tested using the Junit
- Exceptions can be tested using the following the Assert Statement
- assertThrows(exceptionName, supplier);

Using Timeouts

- JUnit provides a handy option of Timeout.
- If a test case takes more time than the specified number of milliseconds, then JUnit will automatically mark it as failed.
- Usage: @Test(timeout = <time_in_milliseconds>)

Junit Testing methods (Assertions)

- What is an assertion? (Anyone?)
- Types
  - assertEqual()
  - assertTrue()
  - assertFalse()
  - assertNotNull()
  - assertNull()
  - assertSame()
  - assertNotSame()
  - assertArrayEquals()

Test Suits

- Test suite is used to bundle a few unit test cases and run them together. In JUnit, both @RunWith and @Suite annotations are used to run the suite tests.
- @RunWith(Suite.class)

```
@Suite.SuiteClasses({
        TestJunit1.class,
        TestJunit2.class
})
```

So why do we need this feature?

Parameterized tests

- JUnit 4 has introduced a new feature called parameterized tests. Parameterized tests allow a developer to run the same test over and over again using different values. There are five steps that you need to follow to create a parameterized test.
  - Annotate test class with @RunWith(Parameterized.class).
  - Create a public static method annotated with @Parameters that returns a Collection of Objects (as Array) as test data set.
  - Create a public constructor that takes in what is equivalent to one "row" of test data.
  - Create an instance variable for each "column" of test data.
  - Create your test case(s) using the instance variables as the source of the test data.

BREAK

Overview

- Mocking is a way to test the functionality of a class in isolation. Mocking does not require a database connection or properties file read or file server read to test a functionality. Mock objects do the mocking of the real service. A mock object returns a dummy data corresponding to some dummy input passed to it.
- Mockito facilitates creating mock objects seamlessly. It uses Java Reflection in order to create mock objects for a given interface. Mock objects are nothing but proxy for actual implementations.
- No Handwriting − No need to write mock objects on your own.
- Refactoring Safe − Renaming interface method names or reordering parameters will not break the test code as Mocks are created at runtime.
- Return value support − Supports return values.
- Exception support − Supports exceptions.
- Order check support − Supports check on order of method calls.
- Annotation support − Supports creating mocks using annotation.

Creating Mock Instances

- Two ways
  - Using @Mock Annotation
  - Using Mockito.mock() method
- All the methods of mocked instances return null by default unless stubbed (We'll look at stubbing in the next slide itself)
- Eg:
  - @Mock private ControllerClass mockedControllerclass;
  - Private ControllerClass mockedControllerclass = mock(ControllerClass.class);

Stubbing Method Calls

- Just the when-then mantra
- when(something_is_called).thenReturn(something)
- 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.

Verifications

- Mockito can ensure whether a mock method is being called with reequired arguments or not.
- It is done using the verify() method.
- verify(emailSender).sendEmail(sampleCustomer); -> Check if it is called once
- verify(emailSender, times(0)).sendEmail(sampleCustomer); -> Check if this method is not called
- Can also use
  - atLeast (int min) − expects min calls
  - atLeastOnce () − expects at least one call
  - atMost (int max) − expects max calls.

Argument Matching

- Argument matchers are mainly used for performing flexible verification and stubbing in Mockito.
- It extends ArgumentMatchers class to access all the matcher functions.
- Mockito uses equal() as a legacy method for verification and matching of argument values. In some cases, we need more flexibility during the verification of argument values, so we should use argument matchers instead of equal() method.
- The ArgumentMatchers class is available in org.mockito package.

```
//Here any integer can be passed to mocklist.get()
when(mocklist.get(Mockito.anyInt())).thenReturn("Mockito");
```

Stubbing Consecutive calls

- Particularly useful when we have some retry logic

```
UpdateUtils mockUU = mock(UpdateUtils.class);
when(mockUU.update("Data"))
.thenThrow(new RuntimeException())
.thenReturn("DATA");
assertThrows(RuntimeException.class, () ->
mockUU.update("Data"));
assertEquals("DATA", mockUU.update("Data"));

//further calls will return the last mocked output
assertEquals("DATA", mockUU.update("Data"));
```

Verification Order

- Mockito provides Inorder class which takes care of the order of method calls that the mock is going to make in due course of its action.

```
//create an inOrder verifier for a single mock
InOrder inOrder = inOrder(calcService);


//following will make sure that add is first called
then subtract is called.
inOrder.verify(calcService).add(20.0,10.0);
inOrder.verify(calcService).subtract(20.0,10.0);
```

Capturing Arguments

- ArgumentCaptor allows us to capture an argument passed to a method in order to inspect it.
- This is especially useful when we can't access the argument outside of the method we'd like to test.

```
public void lateInvoiceEvent(){
     when(invoiceStorage.hasOutstandingInvoice(sampleCustomer)).thenReturn(t
 rue);
    lateInvoiceNotifier.notifyIfLate(sampleCustomer);
    verify(emailSender).sendEmail(sampleCustomer);
    ArgumentCaptor<Event> myCaptor = ArgumentCaptor.forClass(Event.class);
    verify(eventRecorder).recordEvent(myCaptor.capture());
    Event eventThatWasSent = myCaptor.getValue();
    assertNotNull(eventThatWasSent.getTimestamp());
    assertEquals(Event.Type.REMINDER_SENT, eventThatWasSent.getType());
    assertEquals("Susan Ivanova",eventThatWasSent.getCustomerName());
}
```

Spying

- Mockito provides option to create spy on real objects. When spy is called, then actual method of real object is called.
- @Spy or Mockito.spy() can be used to spy

```
@Spy
List<String> spyList = new ArrayList<String>();
@Test
public void whenUsingTheSpyAnnotation_thenObjectIsSpied() {
    spyList.add("one");
    spyList.add("two");
    Mockito.verify(spyList).add("one");
    Mockito.verify(spyList).add("two");
    assertEquals(2, spyList.size());
}
```

Partial Mocks

- Stubbing with Spy causes partial mocks
- Here spyList.size() is mocked while spyList.add() calls the real method

```
@Spy
List<String> spyList = new ArrayList<String>();
@Test
public void whenUsingTheSpyAnnotation_thenObjectIsSpied() {
    spyList.add("one");
    spyList.add("two");
    Mockito.verify(spyList).add("one");
    Mockito.verify(spyList).add("two");
    Mockito.doReturn(100).when(spyList).size();
    assertEquals(100, spyList.size());
}
```

QUESTIONS / DOUBTS

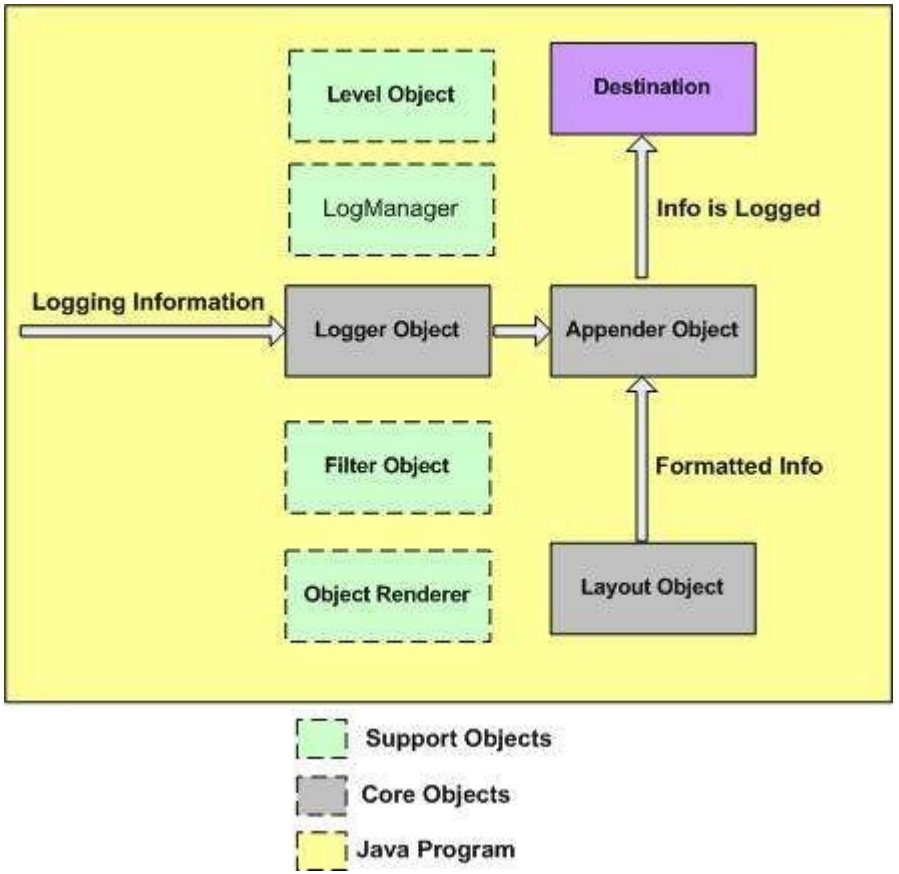
Overview (Log4J)

- log4j is a reliable, fast and flexible logging framework (APIs) written in Java, which is distributed under the Apache Software License.
- log4j has been ported to the C, C++, C#, Perl, Python, Ruby, and Eiffel language
- log4j has three main components:
- loggers: Responsible for capturing logging information.
- appenders: Responsible for publishing logging information to various preferred destinations.
- layouts: Responsible for formatting logging information in different styles.

Features (Log4J)

- It is thread-safe.
- It is optimized for speed.
- It is based on a named logger hierarchy.
- It supports multiple output appenders per logger.
- It supports internationalization.
- It is not restricted to a predefined set of facilities.
- Logging behavior can be set at runtime using a configuration file.
- It is designed to handle Java Exceptions from the start.
- It uses multiple levels, namely ALL, TRACE, DEBUG, INFO, WARN, ERROR and FATAL.
- The format of the log output can be easily changed by extending the Layout class.
- The target of the log output as well as the writing strategy can be altered by implementations of the Appender interface.
- It is fail-stop. However, although it certainly strives to ensure delivery, log4j does not guarantee that each log statement will be delivered to its destination.

Architecture



Logging Methods

- To get logger object:
  - public static Logger getRootLogger();
  - public static Logger getLogger(String name);
- Logging methods:
  - public void debug(Object message)
  - public void error(Object message)
  - public void fatal(Object message)
  - public void info(Object message)
  - public void warn(Object message)
  - public void trace(Object message)

Logging Levels

- ALL - All levels including custom levels.
- DEBUG - Designates fine-grained informational events that are most useful to debug an application.
- INFO - Designates informational messages that highlight the progress of the application at coarse-grained level.
- WARN - Designates potentially harmful situations.
- ERROR - Designates error events that might still allow the application to continue running.
- FATAL - Designates very severe error events that will presumably lead the application to abort.
- OFF - The highest possible rank and is intended to turn off logging.
- TRACE - Designates finer-grained informational events than the DEBUG.
- Priority: ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF
- SetLevel() filters the logs lesser than the level

Log Formatting

- Apache log4j provides various Layout objects, each of which can format logging data according to various layouts.
- The top-level class in the hierarchy is the abstract class org.apache.log4j.Layout. This is the base class for all other Layout classes in the log4j API.
- The Layout class is defined as abstract within an application, we never use this class directly; instead, we work with its subclasses which are as follows:
  - DateLayout
  - HTMLLayout
  - PatternLayout.
  - SimpleLayout
  - XMLLayout
- Layout Methods:
  - public abstract boolean ignoresThrowable()
  - public abstract String format(LoggingEvent event)
  - public String getContentType()
  - public String getFooter()
  - public String getHeader()

QUESTIONS

ASSIGNMENT

THE CUSTOMARY THANK YOU SLIDE