JUnit 5

TDD

- Test-Driven Development (TDD) is a software development process which includes test-first development. It means that the developer first writes a fully automated test case before writing the production code to fulfil that test and refactoring.
- A unit test is a piece of code written by a developer that executes a specific functionality in the code to be tested and asserts a certain behavior or state.
- An integration test aims to test the behavior of a component or the integration between a set of components.

What is JUnit 5?

- JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage
- JUnit 5 requires Java 8 (or higher) at runtime

Maven Dependencies (pom.xml)


```
<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-api</artifactId>
<version>5.10.1</version>
<scope>test</scope>
</dependency>
```

```
<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-
engine</artifactId>
<version>5.10.1</version>
<scope>test</scope>
</dependency>s
```

Writing Tests

```
class MyFirstJUnitJupiterTests {
    private final Calculator calculator = new Calculator();
    @Test
    void addition() {
        assertEquals(2, calculator.add(1, 1));
    }
}
```

Annotations

- @Test
- @ParameterizedTest
- @RepeatedTest
- @DisplayName
- @BeforeEach
- @AfterEach
- @BeforeAll
- @AfterAll
- @Tag
- @Disabled
- @Timeout

```
Assertions.assertEquals(4, Calculator.add(2, 2));
Assertions.assertNotEquals(3, Calculator.add(2, 2));\
Assertions.assertArrayEquals(new int[]{1,2,3}, new int[]{1,2,3}, "Array Equal Test");

Iterable<Integer> listOne = new ArrayList<>(Arrays.asList(1,2,3,4));
Iterable<Integer> listTwo = new ArrayList<>(Arrays.asList(1,2,3,4));
assertions.assertIterableEquals(listOne, listTwo);


String nullString = null;
String notNullString = "java.com";
Assertions.assertNotNull(notNullString);
Assertions.assertNull(nullString);


String originalObject = "java.com";
String cloneObject = originalObject;
String otherObject = "example.com";
```

```
String originalObject = "java.com";
String cloneObject = originalObject;
String otherObject = "example.com";

//Test will pass
Assertions.assertNotSame(originalObject, otherObject);
//Test will fail
Assertions.assertNotSame(originalObject, cloneObject);
//Test will pass
Assertions.assertSame(originalObject, cloneObject);
// Test will fail
Assertions.assertSame(originalObject, otherObject);
```

```
Assertions.assertTimeout(Duration.ofMillis(100), () -> {
Thread.sleep(200);
return "result";
});


Assertions.assertTrue(trueBool);
Assertions.assertFalse(falseBool);

Throwable exception = Assertions.assertThrows(IllegalArgumentException.class, () -> {
throw new IllegalArgumentException("error message");
 });
```

Test Suites

- Suites help us run the tests spread into multiple classes and packages.

Maven Dependencies (pom.xml)

```
<dependency>
<groupId>org.junit.platform</groupId>
<artifactId>junit-platform-suite</artifactId>
<version>1.10.1</version>
<scope>test</scope>
</dependency>
```

Creating Test Suites

```
@Suite
@SuiteDisplayName("JUnit Platform Suite Demo")
@SelectPackages("example")
@IncludeClassNamePatterns(".*Tests")
class SuiteDemo {
}
```

Annotations

- @SelectClasses
- @SelectPackages
- @IncludePackages
- @ExcludePackages
- @IncludeClassNamePatterns
- @ExcludeClassNamePatterns
- @IncludeTags
- @ExcludeTags

@Timeout Annotation

- JUnit 5 uses the declarative way to define the timeout behavior of a given test using the @Timeout annotation
- A timeout configured test should fail if its execution time exceeds a given duration.

```
public class TestTimeOut {
@Timeout(3000)
@Test
public void test() {
try {
Thread.sleep(2000);
} catch (InterruptedException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
}
}
```

Mockito

- Mockito is a popular mocking framework used in Java to create and configure mock objects for unit tests. It's particularly helpful for writing tests for classes that have external dependencies (like databases, APIs, or other services).
- One of the fundamental requirements of making Unit testing work is isolation.
- During unit testing of the application, sometimes it is not possible to replicate exact production environment. Sometimes database is not available and sometimes network access is not allowed. There can be many more such restrictions. To deal with such limitations, we have to create mock for these unavailable resources.

Mockito Dependency (pom.xml)

```
<dependency>
<groupId>org.mockito</groupId>
<artifactId>mockito-junit-jupiter</artifactId>
<version>3.11.1</version>
</dependency>
<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-engine</artifactId>
<version>5.10.1</version>
<scope>test</scope>
</dependency>
```

- Stubs – is an object that has predefined return values to method executions made during the test.
- Spies – are objects that are similar to stubs, but they additionally record how they were executed.
- Mocks – are objects that have return values to method executions made during the test and has recorded expectations of these executions.

Verification

```
import static org.mockito.Mockito.*;
 //mock creation
 List mockedList = mock(List.class);
 //using mock object
 mockedList.add("one");
 mockedList.clear();

 //verification
 verify(mockedList).add("one");
 verify(mockedList).clear();
```

Stubbing

```
LinkedList mockedList = mock(LinkedList.class);

 //stubbing
 when(mockedList.get(0)).thenReturn("first");
 when(mockedList.get(1)).thenThrow(new RuntimeException());

 //following prints "first"
 System.out.println(mockedList.get(0));

 //following throws runtime exception
 System.out.println(mockedList.get(1));

 //following prints "null" because get(999) was not stubbed
 System.out.println(mockedList.get(999));
```

Argument matchers

```
when(mockedList.get(anyInt())).thenReturn("element");

//stubbing using custom matcher (let's say isValid() returns your own matcher implementation):
when(mockedList.contains(argThat(isValid()))).thenReturn(true);

//following prints "element"
System.out.println(mockedList.get(999));

//you can also verify using an argument matcher
verify(mockedList).get(anyInt());

//argument matchers can also be written as Java 8 Lambdas
verify(mockedList).add(argThat(someString -> someString.length() > 5));
```

Verifying exact number of invocations

```
mockedList.add("once");
mockedList.add("twice");
mockedList.add("twice");
mockedList.add("three times");
mockedList.add("three times");
mockedList.add("three times");
//following two verifications work exactly the same - times(1) is used by default
verify(mockedList).add("once");
verify(mockedList, times(1)).add("once");
//exact number of invocations verification
verify(mockedList, times(2)).add("twice");
verify(mockedList, times(3)).add("three times");
//verification using never(). never() is an alias to times(0)
verify(mockedList, never()).add("never happened");
//verification using atLeast()/atMost()
verify(mockedList, atMostOnce()).add("once");
verify(mockedList, atLeastOnce()).add("three times");
verify(mockedList, atLeast(2)).add("three times");
verify(mockedList, atMost(5)).add("three times");
```

Spying on real objects

```
List list = new LinkedList();
 List spy = spy(list);
//optionally, you can stub out some methods:
  when(spy.size()).thenReturn(100);
  //using the spy calls *real* methods
  spy.add("one");
  spy.add("two");
  //prints "one" - the first element of a list
  System.out.println(spy.get(0));
  //size() method was stubbed - 100 is printed
  System.out.println(spy.size());
  //optionally, you can verify
  verify(spy).add("one");
  verify(spy).add("two");
```

Mockito Annotations

- @ExtendWith(MockitoExtension.**class)**
- @Mock is used for mock creation.
- @Spy is used to create a spy instance.
- @InjectMocks is used to instantiate the tested object automatically and inject all the @Mock

Mockito Behavior Verification

```
verify(calcService, atLeastOnce()).subtract(20.0, 10.0);
verify(calcService, atLeast(2)).add(10.0, 20.0);
verify(calcService, atMost(3)).add(10.0,20.0);
verify(calcService, timeout(100)).add(20.0,10.0);
```

Java Logging

Java Logging

- Java allows us to create and capture log messages and files through the process of logging.
- Java has a built-in logging framework in the java.util.logging package.

Logger

- The Logger class provides methods for logging.

```
Logger logger = Logger.getLogger(MyClass.class.getName());
```

Log Level

- SEVERE - serious failure
- WARNING - warning message, a potential problem
- INFO - general runtime information
- CONFIG - configuration information
- FINE - general developer information (tracing messages)
- FINER - detailed developer information (tracing messages)
- FINEST - highly detailed developer information(tracing messages)
- OFF - turn off logging for all levels (capture nothing)
- ALL - turn on logging for all levels (capture everything)

Logging the message

```
logger.setLevel(Level.FINE);
logger.log(Level.INFO, "This is INFO log level message");

logger.info( "This is INFO log level message");
logger.warning( "This is WARNING log level message");
```

Handlers(Appenders)

- The log handler receive the LogRecord and exports it to various targets.
  - StreamHandler - writes to an OutputStream
  - ConsoleHandler - writes to console
  - FileHandler - writes to file
  - SocketHandler - writes to remote TCP ports
  - MemoryHandler - writes to memory

```
ConsoleHandler handler = new ConsoleHandler();
FileHandler handler = new FileHandler(String pattern, int limit, int count);
FileHandler handler = new FileHandler("myapp-log%g.txt", true);
FileHandler handler = new FileHandler("myapp-log%g.txt",  1024 * 1024, 10);
```

Add/Remove Handler

```
Handler handler = new ConsoleHandler();
logger.addHandler(handler);
logger.removeHandler(handler);
```

Formatter

- A handler can also use a Formatter to format the LogRecord object into a string before exporting it to external systems.
  - SimpleFormatter - formats LogRecord to string
  - XMLFormatter - formats LogRecord to XML form

```
handler.setFormatter(new SimpleFormatter());
handler.setFormatter(new XMLFormatter());
```

Log4j 2

- Log4j 2 is the brand new version of log4j and they are a really common logging framework in many JAVA projects.
- Log4j 2 is a fast, reliable and flexible logging framework which is written in java.
- It is an open-source logging API for java.

Maven Dependency

```
<dependency>
<groupId>org.apache.logging.log4j</groupId>
<artifactId>log4j-api</artifactId>
<version>2.14.1</version>
<scope>test</scope>
</dependency>

<dependency>
<groupId>org.apache.logging.log4j</groupId>
<artifactId>log4j-core</artifactId>
<version>2.14.1</version>
</dependency>
```

Log4J 2 Configuration Properties File

```
name=Log4j2PropertiesConfig
appenders=console
appender.console.type=Console
appender.console.name=LogToConsole
appender.console.layout.type=PatternLayout
appender.console.layout.pattern=[%-15level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c-%msg%n
rootLogger.level=info
rootLogger.appenderRefs=stdout
rootLogger.appenderRef.stdout.ref=LogToConsole
```

Logger

```
private final Logger LOGGER =
LogManager.getLogger(GreetingService.class.getName());
```


```
property.filename=mylog.log
name=Log4j2PropertiesConfig
appenders=file
appender.file.type=File
appender.file.name=FileLogger
appender.file.filename=logs/${filename}
appender.file.layout.type=PatternLayout
appender.file.layout.pattern=[%-15level] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c-%msg%n

rootLogger.level=info
rootLogger.appenderRefs=file
rootLogger.appenderRef.file.ref=FileLogger
```