**Topics**

Mass assignment

**Eclipse Shortcuts**

**Ctrl + Space –** make suggestions

**Ctrl + 1 –** create instance or rename classes

**Ctrl + shift + L -** opens all the shortcut available on eclipse

**Ctrl + shift + R -** open resource

**Ctrl + shift + T –** open class

**F3 –** Go to decleration

**F4 –** Open type hirarchey

**Save actions :**

you can use save actions to automat formatting, import organizing and other things when saving your file. Located on ***windows->preferences->save Actions***

**Alt + Shift + S –** Code generation (getters/setters/constructor and many more)

**Alt + shift + I -** inline value

**Maven Build Life Cycle**

Validate

Compile

Test

Package

Integration test

Verify

Install

Deploy

**They can use as maven commands** – e.g

---->mvn clean install

**Maven Commands -**

**mvn compile**

**mvn clean compile**

**help:effective-settings** ==> this command is helpful when testing code - it will pring the maven settings on your console

**dependency:tree** ==> it will print all the dependency hirarchey of your project including transitive dependencies.

**dependency:sources** ==> it will also pring the source code of dependency

**--debug** ==> by using this with your mvn command you can run your command in debug mode

e.g. mvn compile –debug

**Maven Repositories**

1. Local Repository

2. Remote Repository

**Terminologies**

**1. Beans -**

**2. Autowiring –** spring dynamically search for matching instances and initialize objects for you

**3. Dependency Injection –** Set the value of instances using constructor or setter. Basically it is a way by which we can make our application loosely coupled.

**4. Inversion of control –** To passing the control from component(a class who have @component annotation) to spring is known as inversion of control

**5. IOC container –** The entire life-cycle of Bean is maintained by Inversion of control (IOC) container

**6. Application Context –** ApplicationContext is created by spring when we run our application it carry all objects related to beans

**why Spring?**

**Enables Testable code –** The code written in spring is unit testable. And it also able to work with junit and others.

**No Plumbing code –** for example when you write any code which may generate exceptions you need to handle the exception. Those code are unnecesserly increase your code . Spring will handle those exception for you. So you can focus on business logic only.

**Flexiable Architecture –** Spring devided in module so they are flexible to each other. It is also able to work with different other frameworks like struts, hibernate and others.

**Stay Current –** It develop with time. So if any technology is present in market and it is popular. Spring create projects to connect with them like spring can connect with mysql, mongo, thymeleaf, etc.

**Spring Dependencies**

**Core -**

*provides core concepts of spring*

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

</dependency>

*provides application context*

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

To get the object of application context you need to use constructror of :

ApplicationContext context =

new AnnotationConfigApplicationContext(Class\_name.class);

**Logger-**

<!-- adding a dependency for Logger to log something in standard way -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

</dependency>

<!-- also add the service provider for logger -->

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

</dependency>

**Logging with SLF4J Logger**

Logger is used to log something in console. It is the standard way to logging in spring

**private** **static** Logger *LOGGER* = LoggerFactory.*getLogger*(Runner.**class**);

***LOGGER.info("{}", object);***

***LOGGER.info("{}", object.show());***

**Dependency Injection (DE)**

**Constructor based DE –** If the dependency is mendatory use constructor based dependency injection

-> setting value of dependency using constructor

**Setter based DE –** If the dependency is optional use constructor based dependency injection

-> setting value of dependency using setter

***ApplicationContext*** is created by spring. By using this, we can simply access our beans -

Here we are saying bean to the object of any paricular class -

e.g.

ApplicationContext context =

SpringApplication.run(SpringExampleApplication.class, args);

We can take the object of any calss who is using @Component and @autowired by using

context.getBean(ExampleClass.class);

*Example* -

Animal animal = context.getBean(Dog.class);

**Annotations -**

**@Components -** It is used to say to spring - you need to start managing the instances of those class who carry @component annotation

Note: *if you put this annotation spring start managing the instances of that class*

**@Autowired -** spring start looking for this dependency, on which instance you use autowired.

**@Primary –** if you have more than one component matching to a specific type , you can use @Primary to give more importance to one of those components;

***Autowiring -***

***@Primary -*** It is used to give precedence between components , when an autowired instance need value

***@Qualifier***(“name”) - You can call the component you want for @Autowired by giveng the save annotations with the same name to both the component and your @Autowired instance

**Autowiring by name**

you can also give the name of implementing class where you need to Autowiring.

e.g. @Autowired

SortAlgorithm bubbleSortAlgoritm; // this should use bubble sort

***Debug mode\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

*To put our application into debug mode, you need to write this line* into your application.properites file :

***Logging.level.org.springframework = debug***

In this not all the application goes into debug mode but the spring framework goes into debug mode.

**@ComponentScan()**

*It is used to tell to spring where to find components*

By default the @SpringBootApplication search for components on it’s parent package and it’s subpackages

@ComponentScan("com.practice3.proxy")

When you not pass path of any package it would seach for component on it’s parent package and subpackges

@ComponentScan()

**@PostConstruct() / @PreConstruct**

**@PostConstruct()** = it is used to execute or perform some kind of operation just after the Bean creation

**@PreDestroy()** = it is used to execute or perform some kind of operation just before the Bean have been destroy

**CDI –** context and dependency injection

**CDI** - components and dependency injection

- it provides same annotation like spring but different name

- spring implements CDI or we simply say it uses CDI

- Java EE Dependency injection standard(JSR-330)

- spring supporst most annotations

annotations -

Spring CDI

@Component | @Named

@Autowired | @Inject

@Qualifier | @Qualifier

@Scop | @Scope

@Singleton | @Singleton

@Provider

Dependency for use CDI -

<dependency>

<groupId>javax.inject</groupId> <artifactId>javax.inject</artifactId>

<version>1</version>

</dependency>

**XML Configuration**

In the past all the dependenciy injection for beans is configured by xml configuration

XML configuration is written on resources forlder

XML name-sapces : xml uses name-sapces which defines this XML-file can create beans and use core-features of spring

Example -

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<bean id=*"xmlJdbcConnection"* class=*"com.practice7.xml.XmlJdbcConnection"*>

</bean>

<bean id=*"xmlPersonDao"* class=*"com.practice7.xml.XmlPersonDao"*>

<property name=*"xmlJdbcConnection"* ref=*"xmlJdbcConnection"* />

<!-- the first parameter of property is the name of the field on which we want to inject the value and the second is the value id, simply what we want to inject -->

</bean>

</beans>

In XML based configuration to create application context we need to use ClassPathXmlApplicationContext(“file\_name.xml”);

example -

ClassPathXmlApplicationContext context =

**new** ClassPathXmlApplicationContext("applicationContext8.xml");

Component Scan in xml is done through the given way

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*>

<context:component-scan base-package=*"com.practice1.autowiring"* />

<!-- here we simply search for components in the specific package -->

</beans>

in the name-space two thing are added

xmlns:context=*"*[*http://www.springframework.org/schema/context*](http://www.springframework.org/schema/context)*"*

*and on the schema location -*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

**getBeanDefinitionNames()-** *method gives the name of used beans on component scan the*

this method return an array of defined bean whether xml-config or java-config when we want to print all elements we need to type cast this as an object

LOGGER.info("{}", (Object)context.getBeanDefinitionNames());

**DISAMBIGUATION -**

**1. IOC Container –** In classical way objects are created by bean itself, But in spring we are shifting the responsibility to spring to do that for us, It called Inversion of controll or the program/framework who do this is know as IOC conatainer. We are shifting our control to spring and ask him to create/manage objects.

IOC generate beans and manages objects, autowire instances. Its a generic concept not framework specific.

**2. Bean Fatory -** The bean factory provides basic management for beans and autowirng of them, it comes with spring core package. When you need to work with a system where memory are limited or less you go with bean factory otherwise use application context.

Features of Bean factory

- Bean Management

- Autowiring

**3. Application Context –** Spring recomends to use application context on 99% scenarios. Application context provides more features than bean factory that is typically needed by enterprise application. Application context provides features of bean factroy as well as it’s own features.

Features of application context

- Spring AOP Features

- Internationalization

- WebApplicationContext for web applications etc.

**Component Annotation -**

**@Component –** Generic Component

**@Repository –** @Repository is related to getting data.

- Encapsulating storage, retrieval and search behaviour typically from a relational database

**@Service –** Business service facade / Business logic

**@Controller –** Controller in MVC pattern

**NOTE –** always use the specific annotation on all components rather than using the generic annotiation.

**Property files -**

Property files are configuration files which contains configuration of software we made. It contains key value so we can access any value by it’s key.

You can access your properties file due to relative path of your classpath with indicates the src/main/rosources/Meta-INF folder

**step 1:** create a file with .property extension

example – app.property

**step 2:** you can write your properties in app.propery file example -

app\_name=MY\_Application\_name

**step 3:** Define Property source on the runner class which contains main method

@PropertySource(“classpath:file\_location”)

**step 4:** use @value() to get the value from your property file

@Value(“${key}”)

private String app\_name;

**Unit Testing**

unit testing is defines testing of specifc method or group of methods of any class.

It is different from sytem testing where we the test the entire system, here we test some method of software.

Unite tesing is to write different tests for each method or group of methods or for a class.

*JUnit* – Junit is a framework that which will help you to call any specific method and check the output as expected.

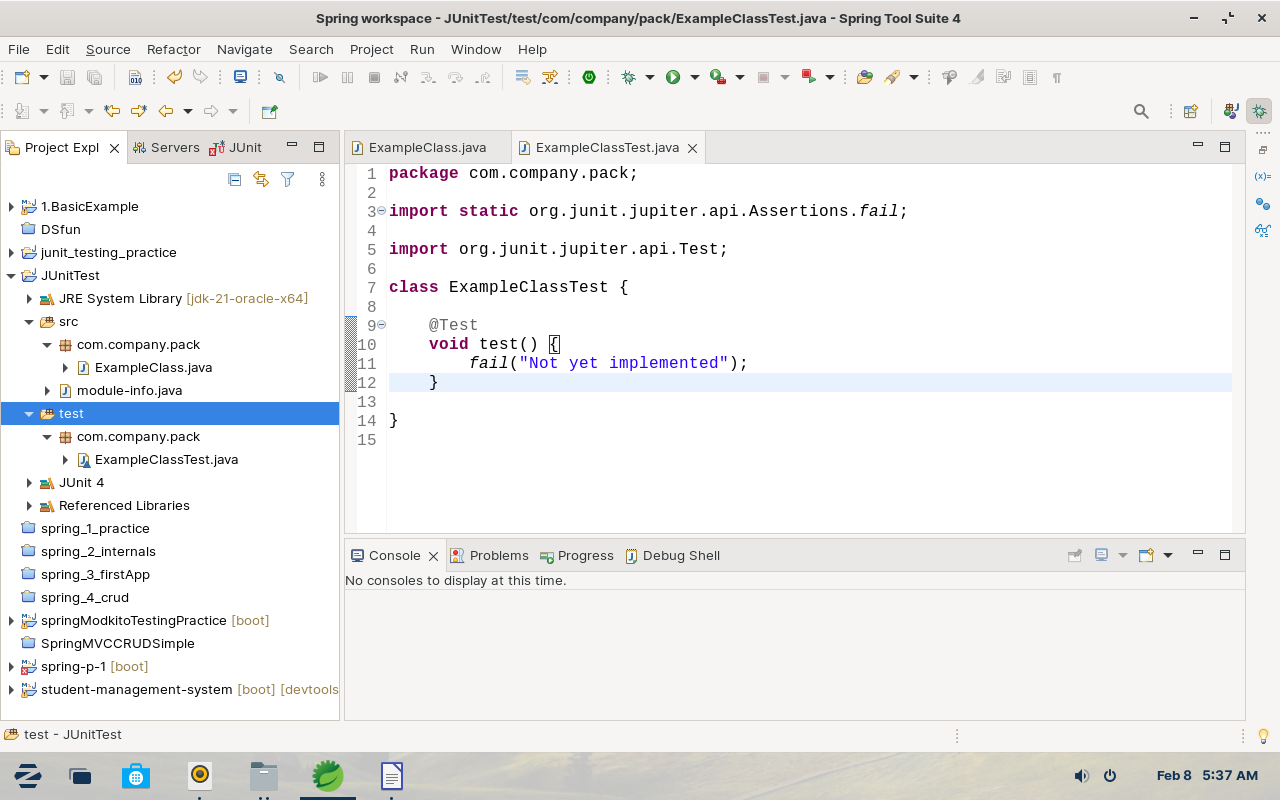
\* Junit tests are automated tests.

\* Junit can be run in continuous iterations.

\* Junit tests should be in different folder compared to production code.

To create junit test first you need to create new source folder where all the test should be present. ***To create source folder -> right click on your project -> new -> source folder***

To create a new test first make the same package structure for your test, it will be same as the package where your project class available.

**

Once you create the folder structure you are ready to write tests

@Test is an annotation who defines this methos is a test method

\* By default in junit all test are passed whether they are not implemented.

\* They fails when any output is not as expected.

**Assert Methods -**

\* assert method are use to check test cases.

\* there are many assert methods are available, use them as per your need some of them are given in example.

package com.company.pack;

import static org.junit.Assert.assertArrayEquals;

import static org.junit.Assert.assertFalse;

import static org.junit.Assert.assertNotNull;

import static org.junit.Assert.assertNull;

import static org.junit.Assert.assertTrue;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

class AssertMethods2 {

@Test

void test() {

boolean condition = true;

assertEquals(true, condition);

assertTrue(condition);

assertFalse(false);

assertNull(null);

assertNotNull(23);

assertArrayEquals(new int[] {12, 23, 34}, new int[] {12, 23, 34});

}

}

**Important Annotations of Junit -**

In the following example we uses some annotations

@BeforeEach - it run's before every test

@AfterEach - it runs after every test

@BeforeAll - it will run before all tests and @Before method

@AfterAll - this will run after all tests and @After method

@Test - in JUnit there is no garenty of test will run in which order, they can be run on any order

Note : one thing which is need to remember when using @BeforeAll and @AfterAll you need to use static methods.

*example:*

import static org.junit.Assert.assertEquals;

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.Test;

class ImportantAnnotations3 {

MyMath myMath = new MyMath();

@BeforeEach

public void before() {

System.out.println("Before each");

}

@AfterEach

public void after() {

System.out.println("After each");

}

@BeforeAll

public static void beforeClass() {

System.out.println("Before All");

}

@AfterAll

public static void afterClass() {

System.out.println("After All");

}

@Test

public void test1() {

System.out.println("test 1");

}

@Test

public void test2() {

System.out.println("test 2");

}

}

**Output:**

Before All

Before each

test 1

After each

Before each

test 2

After each

*After All*

**Mockito –** mockito is a framework who provides us facility to mock(), simpley create anonymous objects.

**example :** in this exampl**e DataService** is an interface and  **SomeBusinessImpl** uses the refterence of dataservice to initalize local reference variable via constructor.

So mocking actually gives us the freedom where we can make-tests for that class and mockito provides us the created object. Throught mock() method .

package com.springInternals;

/\*

\* BY using mock we don't need to create any specific class who provide us the value

\* we can simple create mock with return value on any specific methods

\*

\* Here we are creating mock with interface

\* you can create mocks for classes as well as interfaces

\*

\* \*/

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.when;

import org.junit.jupiter.api.Test;

class SomeBusinessMokito2 {

@Test

void testfindTheGreatestFromAllData1() {

DataService dataService = mock(DataService.class);

when(dataService.retrieveAllData()).thenReturn(new int[] {23, 34, 45});

SomeBusinessImpl someBusinessImpl = new SomeBusinessImpl(dataService);

int result = someBusinessImpl.findTheGreatestFromAllData();

assertEquals(45, result);

System.out.println("Passed");

}

}

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

**mock() -** mock method takes one parameter which is the name of the class or interface which object you want to create.

**when() -**  a static method which defines a condition when you call the specified function what to do?

**thenReturn() -** then return use to return something when we call the specified method on when().

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Example -**

**import** **static** org.junit.Assert.*assertEquals*;

**import** **static** org.mockito.Mockito.*when*;

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

**import** org.junit.runner.RunWith;

**import** org.mockito.InjectMocks;

**import** org.mockito.Mock;

**import** org.mockito.junit.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.**class**)

**class** SomeBusinessMokitoAnnotations3 {

@Mock

DataService dataService;

@InjectMocks

SomeBusinessImpl someBusinessImpl;

@Test

**void** testfindTheGreatestFromAllData\_ThreeValues() {

*when*(dataService.retrieveAllData())

.thenReturn(**new** **int**[] {45, 34, 21});

*assertEquals*(45, someBusinessImpl.findTheGreatestFromAllData());

System.***out***.println("Passed with three values");

}

}

Example -

import static org.junit.Assert.assertEquals;

import static org.mockito.Mockito.mock;

import static org.mockito.Mockito.when;

import java.util.List;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

class ListTest\_4 {

@Test

void test\_Size() {

List listMock = mock(List.class);

when(listMock.get(0)).thenReturn("something");

assertEquals("something", listMock.get(0));

}

@Test

void testSize\_mulitpleReturns() {

List listMock = mock(List.class);

when(listMock.get(0)).thenReturn("something").thenReturn(123);

assertEquals("something", listMock.get(0));

assertEquals(123, listMock.get(0));

}

@Test

void test\_SpecificParameter() {

List listMock = mock(List.class);

when(listMock.get(1)).thenReturn("something");

assertEquals("something", listMock.get(1));

assertEquals(null, listMock.get(5));

}

@Test

void test\_GenericParameter() {

List listMock = mock(List.class);

when(listMock.get(Mockito.anyInt())).thenReturn("something");

assertEquals("something", listMock.get(0));

assertEquals("something", listMock.get(5));

}

}

**Note** - one thing you need to know when you make any parameter generic, all parameters of that method must be generic you can not use combination of generic and specific parameters

**Mass assignment**

**Def – Mass assignment vulnerabilities occur when the database model that is being assigned contains security-relevant fields, and the application user can supply values in the POST request that are saved to those fields, even though they are not present in the HTML form.**

**Software frameworks sometime allow developers to automatically bind HTTP request parameters into program code variables or objects to make using that framework easier on developers. This can sometimes cause harm.**

**Attackers can sometimes use this methodology to create new parameters that the developer never intended which in turn creates or overwrites new variable or objects in program code that was not intended.**

**This is called a Mass Assignment vulnerability.**

**Mass Assignment: Ruby on Rails, NodeJS.**

**Autobinding: Spring MVC, ASP NET MVC.**

**Object injection: PHP.**

***Allow listening :***

**@Controller**

**public class UserController**

**{**

**@InitBinder**

**public void initBinder(WebDataBinder binder, WebRequest request)**

**{**

**binder.setAllowedFields(["userid","password","email"]);**

**}**

**...**

**}**

***Block listening :***

**@Controller**

**public class UserController**

**{**

**@InitBinder**

**public void initBinder(WebDataBinder binder, WebRequest request)**

**{**

**binder.setDisallowedFields(["isAdmin"]);**

**}**

**...**

**}**