Table of Contents

[What you will learn 1](#_Toc41899618)

[Spring Core Container 1](#_Toc41899619)

[Configuring 1](#_Toc41899620)

[Environment 1](#_Toc41899621)

[Autowiring 1](#_Toc41899622)

[Spring Boot 1](#_Toc41899623)

[Spring Boot starters 1](#_Toc41899624)

[Logging 1](#_Toc41899625)

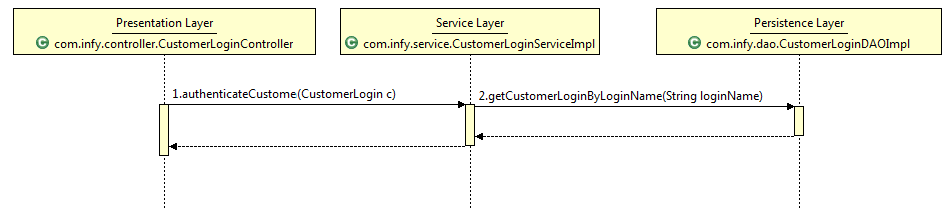
[Aspect Oriented Programming 1](#_Toc41899626)

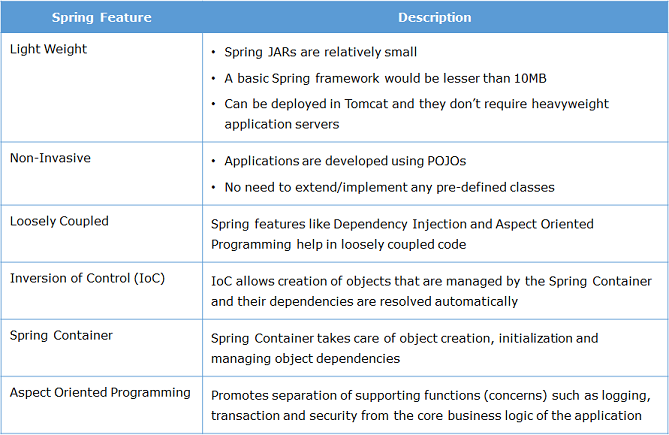
[Test cases with JUnit 1](#_Toc41899627)

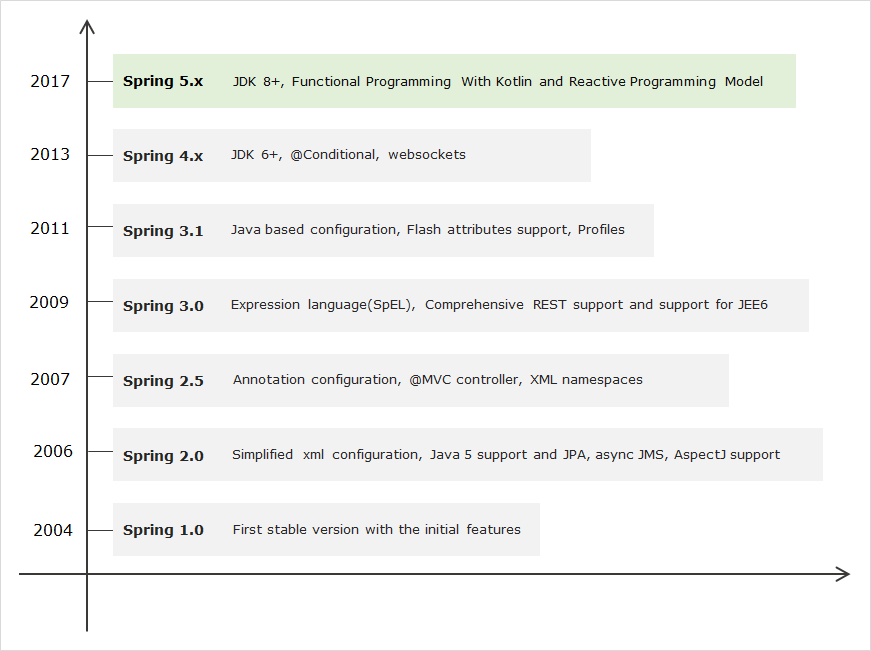
[Spring Caching 1](#_Toc41899628)

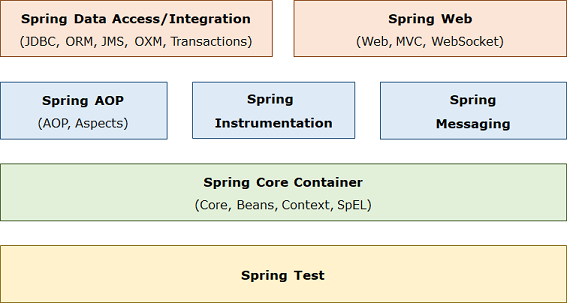
# What you will learn

* Explain need of Spring Framework.
* A **dependency injection** and its different types.
* Implement dependency injection using **@Autowired** annotation.
* Create Spring Boot application using **Spring Initlizr**.
* Explain what is **AOP** and demostrate how to implement different types of advices using Spring AOP.
* Write code for logging using **Log4j** in a Spring Boot application.
* Write code for testing service layer of Spring Boot application using **JUnit** and **Mockito**.



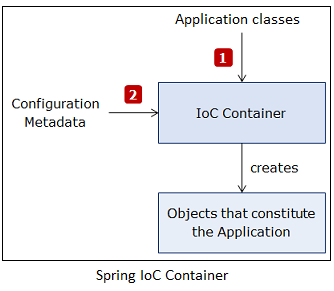
* **Dependency Injection** (DI) is used as a technique in which the responsibility of creating, assembling, and wiring the dependencies of a dependent class is externalized to the external framework or library called *dependency injection (DI) frameworks*
* Reversal of responsibilities is sometimes also known as **Inversion of Control(IoC).** Dependency injection frameworks are also called **IoC containers**
  + Examples: Spring Framework, Google Guice, Play Framework



Spring is organized in a modular fashion. The developer can pick and choose the modules as per the need for building an enterprise application. Spring Framework 5.x has the following key module groups:

* **Spring** **Core Container**: This module provides the Spring IoC container and Dependency Injection features.
* **Spring** **Data Access/Integration**: This module provide support to access database in Spring applications.
* **Spring Web**: This modules provide support to create web applications.
* **Spring AOP**: This module support aspect oriented programming implementation.
* **Spring Messaging**: This module provides support for integrating Spring applications with messaging systems.
* **Spring Test**: This module supports testing of Spring applications using JUnit or TestNG.

# Spring Core Container

* **Core**: This is the key module of Spring Framework which provides fundamental support on which all other modules of framework are dependent.
* **Bean**: This module provide basic Spring IoC container called BeanFactory.
* **Context**: This module provides Spring IoC container called ApplicationContext which extends the features of BeanFactory container and provides additional features for enterprise application development.
* **Spring Expression Language (SpEL):** This module is used for querying/manipulating object value.
* It creates, initializes, and injects the required objects.
* Objects whose life cycle is managed by Spring are called as Spring beans or **beans**
* The Spring Container needs information about objects of which class to create and *how to wire them together*.
  + This information is called as configuration metadata.
  + It can be provided in following ways:
    - XML configuration
    - Java-based configuration
    - Java Annotation-based configuration
* The Spring IoC container is represented by following interfaces:
  + **BeanFactory interface**:
    - It represents container which provides basic functionalities.
    - It instantiates bean whenever asked for by the client application.
    - Using its getBean() method you can get instances of beans.
    - It instantiates bean objects only when getBean() method is called.
  + **ApplicationContext interface**:
    - It extends BeanFactory interface and provides additional functionalities to support enterprise application development.
    - It instantiates *all beans* when container is loaded.
    - There are many implementations of this interface. Some commonly used implementation classes are as follows:
      * **ClassPathXmlApplicationContext** : It is used to process XML-based configuration metadata.
      * **AnnotationConfigApplicationContext** : It is used to process Java‐based configuration metadata

## Configuring

* The Java-based configuration metadata is provided in Java class using following annotations:
  + **@Configuration** : The Java configuration class is marked with this annotation.
    - This annotation identifies this as a configuration class, and it’s expected to contain details on beans that are to be created in the Spring application context.
  + **@Bean** : This annotation is used to declare a bean.
    - The methods of configuration class that creates an instance of the desired bean are annotated with this annotation.
    - These method are called by Spring container during bootstrap and the values returned by these methods are treated as Spring beans.
    - By default, only one bean instance is created for a bean definition by the Spring Container, and that instance is used by the container for the whole application lifetime.
  + Wiring a dependency in configuration class:
    - @Configuration
    - public class SpringConfig
    - @Bean
    - public CustomerLoginService customerLoginService()
    - return new CustomerLoginServiceImpl(customerLoginDao());
    - }
    - @Bean(name="customerDao")
    - public CustomerLoginDAO customerLoginDao()
    - return new CustomerLoginDAOImpl
* Java Annotation based configuration:
  + Java annotation-based configuration there is no need to explicitly configure the bean.
  + Spring automatically scans, detects and instantiates the beans from the specified package through component scanning.
  + It looks for classes annotated with following annotations:
    - ​**@Component** // General purpose annotation to mark a class as Spring bean.
    - public class CustomerLoginController
    - //rest of the code
    - **@Service** // used to define a service layer Spring bean. It is specialization of the @Component annotation for the service layer.
    - public class CustomerLoginSeviceImpl implements CustomerLoginService {
    - **@Repository** // It is used to define a persistence layer Spring bean. It is specialization of the @Component annotation for the persistence layer.
    - public class CustomerLoginDAOImpl implements CustomerLoginDAO
    - **@Controller** // It is used to defined a web component. It is specialization of the @Component annotation for the presentation layer.
    - public class CustomerLoginController
  + By default, the bean name is same as class name with a lowercase initial character.
    - Specific names can be given with following annotation:
      * @Repository(value=”customerLogin”)
  + The component scanning isn’t enabled by default. You have to annotate configuration class with **@ComponentScan** annotation to enable component scanning and add basePackages attribute to annotation to specify a different or multiple packages.
    - @Configuration
    - @ComponentScan (basePackages = "com.infy.service com.infy.dao")
    - Public class SpringConfig
  + If bean in your application are not having any dependencies then component scanning alone works.
  + If beans have dependencies then you have to inject dependencies in your *component-scanned* beans.
  + For this you have to do **autowiring** of dependencies.
* Using IoC Container
  + **ApplicationContext** can be loaded using **AnnotationConfigApplicationContext** class as follows:
    - ApplicationContext ctx = new AnnotationConfigApplicationContext(SpringConfig.class);
  + The **getBean()** method is used to access a particular bean by specifying its class as follows:
    - UserService userService = ctx.getBean(CustomerLoginServiceImpl.class);
  + The bean name which is the name of the method in which bean is configured can also be used to access a particular bean as follows:
    - UserService userService = (UserService) ctx.getBean("customerLoginServiceImpl");
  + **Java based configuration:**
  + Step 1: Create a new Maven project using Eclipse.
  + Step 2: Add the following Spring dependency in pom.xml:
  + <dependency>
  + <groupId>org.springframework</groupId>
  + <artifactId>spring-context</artifactId>
  + <version>5.0.4.RELEASE</version>
  + </dependency>
  + Step 3: Create the following Spring bean in com.infy.bean package:
  + public class WelcomeBean {
  + public void printWelcome() {
  + System.out.println("Welcome to Spring");
  + }
  + }
  + Step 4: Create the following configuration file in com.infy.configuration package:
  + @Configuration
  + public class SpringConfig {
  + @Bean
  + public WelcomeBean welcomeBean() {
  + return new WelcomeBean();
  + }
  + }
  + Step 5: Create the following UserInterface class in com.infy.ui package with the main method and instantiate the Spring container using configuration class you created in the previous step as the constructor argument. Retrieve the WelcomeBean via getBean() method and use it.
  + public class UserInterface {
  + public static void main(String[] args) {
  + ApplicationContext ctx = new AnnotationConfigApplicationContext(SpringConfig.class);
  + WelcomeBean welcomeBean = ctx.getBean(WelcomeBean.class);
  + welcomeBean.printWelcome();
  + }
  + }
  + Step 6: Run the UserInterface class created in the previous step. You will get the following output:
  + Welcome to Spring

Environment

A class to access values in properties file

* Environment environment = applicationContext.getEnvironment();

Using Environment, property value can be fetched using getProperty() method as follows:

* environment.getProperty("SUCCESS");

The location of properties file is mentioned in Spring configuration file using @PropertySource annotation along with @Configuration annotation.

For example, messages.properties which is present in classpath can be loaded using @PropertySource as follows:

* @Configuration
* @PropertySource("classpath:messages.properties")
* public class SpringConfig { //code for configuring other beans }

## Autowiring

* If one bean class is dependent on another bean class then the bean dependencies needs to be explicitly defined in your configuration class.
* You can let the Spring IoC container inject the dependencies into dependent bean classes without it being defined in your configuration class. This is called as **autowiring**.
* To do autowiring, you can use **@Autowired** annotation.
* This annotation allows Spring IoC container to resolve and inject dependencies into your bean.
* It can be applied to attributes, constructors, setter methods of a bean class.
* If the container does not find a bean for autowiring, it will throw NoSuchBeanDefinitionException exception.
* If more than one beans of the same type are available in the container, then the framework throws an exception indicating that more than one bean is available for autowiring.
  + To handle this **@Qualifier** annotation

# Spring Boot

* Helps reduce time done in configuring
  + Configuring different environments to override for production, development, testing,…
* Helps project dependency management for compatible versions
* Spring Boot is a framework built on the top Spring framework that helps developers build Spring-based applications quickly and easily.
* Without requiring developers to write the same boilerplate configuration again and again.

1. Spring Boot is *opinionated framework*
   * Spring Boot has some sensible defaults which you can use to quickly build your application.
     + For example, Spring Boot uses embedded **Tomcat** as the default web container.
2. Spring Boot is customizable
   * You can easily customize it at any time during your development based on your need.
     + For example, if you prefer **log4j** for logging over Spring Boot built-in logging support then you can easily make dependency change in your pom.xml file to replace the default logger with log4j dependencies.

* Spring Boot features are as follows:
  + Starter Dependencies
  + Automatic Configuration
  + Spring Boot Actuator
  + Embedded Servlet Container

Spring Boot starters

* Pre-configured dependency descriptors with most commonly used libraries that you can add in your application.
* You don't need to search for compatible libraries and configure them manually.
* Spring Boot will ensure that the necessary libraries are added to the build.
* To use these **starters**, you have to add them in the pom.xml file.
  + For example, to use spring-boot-starter following dependency needs to be added in pom.xml:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

* Spring Boot comes with many starters. Some popular starters which we are going to use in this course are as follows:
  + **spring-boot-starter** - This is the core starter which includes support for auto-configuration, logging and **YAML**.
  + **spring-boot-starter-aop** - This starter is used for aspect-oriented programming with **Spring AOP** and **AspectJ**.
  + **spring-boot-starter-data-jdbc** - This starter is used for **Spring Data JDBC**.
  + **spring-boot-starter-data-jpa** - This starter is used for **Spring Data JPA** with **Hibernate**.
  + **spring-boot-starter-web** - This starter is used for building web application using **Spring MVC** and **Spring REST**
    - It also provides Tomcat as the default embedded container.
  + **spring-boot-starter-test** - This starter provides support for testing Spring Boot applications using libraries such as **JUnit**, **Hamcrest** and **Mockito**.
  + **spring-boot-starter-log4j2** - This starter provides support for using **Log4j2** for logging. It is an alternative to spring-boot-starter-logging.
  + **spring-boot-starter-actuator** - This starter provides support for using **Spring Boot Actuator**.
* Spring Boot uses auto-configuration based on the jar dependencies available on the classpath. This eliminates the need of manually configuring Spring applicati
  + For example, if you are using JDBC in your Spring Boot application then there is no need to configure any database connection beans.
  + As soon as Spring Boot detects that you have JDBC library in application’s classpath it will automatically configure database connection beans.
* Spring Boot Start Parent:

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.3.RELEASE</version>

<relativePath/>

</parent>

* + Creating Spring Boot Application: with **Spring Initializr, Using the Spring Tool Suite (STS), Using Spring Boot CLI, Using Spring Maven Project**
* Spring Boot Runners
  + Interfaces to implement for actions performed after SpringApplication.run() is invoked.
    - CommandLineRunner – Override this interfaces run(String… args) method to perform actions
    - ApplicationRunner
* Application.properties
  + Read in properties fromt his file using Spring’s Environment.getProperty() method.

# Logging

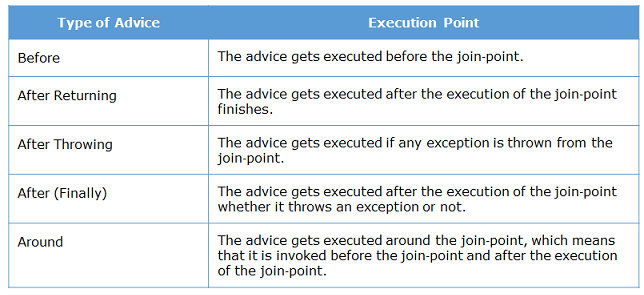
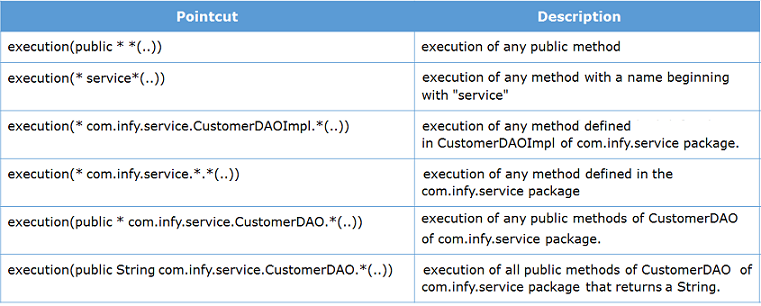
Spring Boot uses **SLF4J(**Simple Logging Facade for Java) along with **Logback** implementations for logging.

By default it logs messages in console at **ERROR**, **WARN**, and **INFO** level.

You can easily configure logging levels, logging format, and log file location by setting logging related properties in **application.properties** file.

* Example: logging.level.com.infy = ERROR
* Follows by format: logging.level.<logger-name>=<logging-level>
  + Logging-level: TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF
* The root logger can be configured by setting logging.level.root property in application.properties file.
* # Logging pattern for the console
  + logging.pattern.console = %d{yyyy-MMM-dd HH:mm:ss a} [%t] %-5level %logger{36} - %msg%n
* # Logging pattern for file
  + logging.pattern.file = %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n
* To support logging to file add the following in application.properties
  + logging.file = error.log
  + logging.path will also set the location for such a log file.
* Using SLF4J once configured:
  + import org.slf4j.Logger;
  + import org.slf4j.LoggerFactory;
  + Logger logger=LoggerFactory.getLogger(this.getClass());
  + logger.info("This is an info level log.");
  + logger.debug("This is debug level log.");
* Sprint Boot uses SLF4J by default. Exclude spring-boot-starter-logging and include spring-boot-starter-log4j2 to use log4j instead

# Aspect Oriented Programming

* There are few functionalities which are common to all layers of enterprise software; such as logging, security, and transaction management, etc.
* These functionalities are called as **cross-cutting concerns**.
* But this makes the code more difficult to maintain as the code of cross-cutting concerns is tightly coupled with the business logic code of each layer.
* So it is better to **keep all of cross cutting concerns’ code in one place** and use it in multiple places wherever required.
* This is where **Aspect Oriented Programming (AOP)** kicks in.
* It provides a way to separate the code of cross-cutting concern from business logic code and define them in one place so that it can be reused in all the layers of the application.
* Advantages of AOP:
  + It ensures that cross cutting concerns are kept separate from the core business logic.
  + It allows to create a more loosely coupled application wherein you can change the cross cutting concerns code without affecting the business code.
  + Spring framework provides AOP support by default.
  + It uses **AspectJ**, which is one of the most popular AOP frameworks.
  + AspectJ provides an easy way for implementing AOP with the help of annotations.
* **Aspect** is a class that implements the cross-cutting concerns.
  + To declare a class as an aspect it should be annotated with the **@Aspect** annotation.
  + It should be applied to the class which is annotated with **@Component** annotation or with derivatives of it.
* **Join** **point** is a specific point in the application such as method execution, exception handling, changing object variable values, etc during its execution.
  + In Spring AOP a join point is always the execution of a method.
* **Advice** is a method of the aspect class that provides the implementation for the cross-cutting concern.
  + It gets executed at the selected join point(s).
  + The following table shows the different types of advice along with the execution point they have
  + General order: Around-Before, Before, Around-After, After, AfterReturning.
* **Pointcut** represents an expression used to identify join points.
  + It that evaluates to the method name before or after which the advice needs to be executed.
  + **execution(<modifiers> <return-type> <fully qualified class name>.<method-name>(parameters))** 
    - execution is called as pointcut designator.
    - It tells Spring that join point is the execution of the matching method.
  + **<modifiers>** determines the access specifier of the matching method. It is not mandatory.
  + **<return-type>** determines the return type of the method in order for a join point to be matched.
    - It is mandatory. If the return type doesn't matter wildcard \* is used.
  + **<fully qualified class name>** specifies the fully qualified name of the class which has methods on the execution of which advice gets executed.
    - It is optional. You can also use \* wildcard as name or part of a name.
  + **<method-name>** specifies the name of the method on the execution of which advice gets executed. It is mandatory. You can also use \* wildcard as name or part of a name.
    - parameters are used for matching parameters. To skip parameter filtering, use two dots .. as parameters.
  + When using @AfterReturning – use the following syntax to access method’s return value:
    - @AfterReturning(value = "execution(\* com.infy.service.\*Impl.\*(..)),returning = "returnvalue"")
    - public void afterReturning(String returnvalue){ }
* To use Spring AOP and AspectJ include this in pom.xml:
  + <dependency>
    - <groupId>org.springframework.boot</groupId>
    - <artifactId>spring-boot-starter-aop</artifactId>
  + </dependency>
* Example seen in @Aspect class:

@Before("execution(\* com.infy.service.\*Impl.\*(..))")

public void beforeAdvice() throws Exception

System.out.println("Before advice called.");

# Test cases with JUnit

spring-boot-starter-test starter has to be added in pom.xml as follows:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

This starter provides most commonly used libraries for writing tests such as **JUnit 4, Mockito, Hamcrest, AssertJ, Spring Test** and **Spring Boot Test** etc.

Example Junit usage in testing:

* When a thrown message is expected:
  + expectedException.expect(Exception.class);
  + expectedException.expectMessage("Service.WRONG\_CREDENTIALS");
* Asserts:
  + Assert.assertEquals("SUCCESS", actual);

**@SpringBootTest** - This annotation loads ApplicationContext using SpringApplication so that all the Spring Boot features will be available to the tests.

**@RunWith(SpringRunner.class)** – This annotations tells JUnit to invoke SpringRunner class to run the tests instead of the runner built into it. If you are using JUnit 5, there is no need to use this annotation as @SpringBootTest is already annotated with its equivalent annotation.

Example Mockito usage with Junit:

* Creating a mock dependency for testing purposes. Example: Service needs a consistent DAO to test with.
* Create Mock object:

@Mock

private CustomerDAO customerDao;

* Inject the Mock:

@InjectMocks

private CustomerService customerService = new CustomerServiceImpl();

Configuring Behavior of Mock Object

* **when**().
* It is a static method defined in org.mockito.Mockito class.
* It specifies the value returned when a method of mock object is called.
* For example, the following code snippet tells Mockito that when **getCustomerByCustomerId()** method of the mock CustomerDAO is called with 1002 as parameter then it should return null:
  + **when(customerDao.getCustomerByCustomerId(1002).thenReturn(null);**
* But sometimes you need to define behavior for range of values or beforehand unknown values.
* For this, you can configure mocked methods using matcher methods as follows:
  + when(customerDao.getCustomerByCustomerId( **anyInt()** ).thenReturn(null);
    - Whenever any integer value is passed to getCustomerByCustomerId() method of the mock CustomerDAO it will return null value.
* Some other matcher methods are as follows:
  + **anyBoolean**() -It returns any boolean or Boolean value.
  + **anyInt**() - It returns any integer or Integer value.
  + **anyFloat**() - It returns any float or Float value.
  + **anyDouble**() - It returns any double or Double value.
  + **anyLong**() - It returns any long or Long value.
  + **any**() - It returns any Object.
* You must take care of following points while using Mockito:
  + The private and static methods cannot be mocked.
  + Dont use matcher methods as values returned by a function. For return value use exact values.
  + If a method has multiple arguments then you must provide all arguments either by matchers or by exact values.
  + The matcher methods cannot be used outside of verification or stubbing.

# Spring Caching

Spring provides provides method-level caching mechanism in which multiple executions of a method can be avoided to overcome the performance drawbacks.

If a method that has already executed for the supplied arguments is called again with same arguments then cached results from previous execution are returned without executing the method again.

To enable:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-cache</artifactId>

</dependency>