

=====

## Framework

=====

- => Semi Developed software which provides common logics required for projects development
- => Frameworks will help the developers to implement more functionality in less time
- => When we use framework to develop the project, we can focus only on business logic.

=====

## Types of Frameworks

=====

- 1) Frontend Frameworks : To develop user interface in the project.

Ex: Angular

- 2) Web Frameworks : To develop web layer in the project

Ex: Struts (Outdated)

- 3) ORM Frameworks : To develop persistence layer in the project.

Ex: Hibernate

=> By using Struts we can develop only Web Layer in the Project ( Controllers )

=> By using Hibernate we can develop only Data Access Layer (Persistence Layer)

Note: To overcome the problems of Struts framework, Spring Framework came into market.

=> Spring Framework is called as Application Development Framework

=> By using Spring framework we can develop end to end application

=> Spring is free & open source framework

=> Spring Framework developed in Modular Fashion

Note: Spring framework means collection of modules

=====

## Spring Modules

=====

- 1) Spring Core
- 2) Spring Context
- 3) Spring JDBC
- 4) Spring ORM
- 5) Spring AOP
- 6) Spring Web MVC
- 7) Spring Security
- 8) Spring Social
- 9) Spring Batch
- 10) Spring Data JPA
- 11) Spring REST
- 12) Spring Cloud

Note: Spring is very flexible framework. It will not force to use all modules. Based on requirement we can pickup particular module and we can use it.

=> Spring is versatile framework ( Easily it can be integrated with other frameworks )

=> The current version of Spring framework is 6.0

=> Spring framework is under license of VM Ware Tanza...

URL : [www.spring.io](http://www.spring.io)

=====

1) Spring Core : It is base module in the spring framework.

=> Spring Core Module providing fundamental concepts of Spring Framework

    1) IOC Container ( Inversion Of Control )

    2) Dependency Injection

    3) Bean Life cycle

    4) Bean Scopes

    5) Autowiring etc...

2) Spring Context : It will deal with configurations required for our Spring Applications.

3) Spring AOP : Aspect Oriented Programming

=> AOP is used to separate business logics & Secondary logics in the project

    Ex: Security, Logging, Tx, Auditing, Exception Handling...

Note: If we combine business logics & secondary logics then we will face maintenance issues of our project.

4) Spring JDBC : Spring JDBC is used to simplify Database Communication logic

=> In java jdbc we need to write boiler plate code (repeated code) like below in several classes

    => Load driver  
    => Get connection  
    => Create Statement  
    => Execute Query  
    => Close connection

=> Using Spring JDBC we can directly execute query the remaining part Spring JDBC will take care

5) Spring Web MVC : It is used to develop both Web Applications & Distributed Applications

=> Web Applications ( C 2 B )

    Ex: Gmail.com, facebook.com etc...

=> Distributed Applications ( B 2 B ) / Web Services or RESTful Service

    Ex:

    IRCTC ---- MakeMyTrip

## Passport --- AADHAR app

### 6) Spring ORM (Object Relational Mapping)

=> Spring Framework having integration with ORM frameworks

Ex: Spring ORM , Spring Data JPA etc....

Note: JDBC will represent data in text format where as Hibernate ORM will represent data in Objects format.

### 7) Spring Security

=> Security is very crucial for every application

=> Using Spring Security We can implement Authentication & Authorization

=> Spring Security with OAuth2.0

=> Spring Security with JWT (JSON Web Tokens)

### 8) Spring Batch : Batch means bulk operation

=> Reading data from Excel and store it into database table

=> Sending Monthly statements to customers in email

=> Sending Reminders to customers as Bulk SMS

### 9) Spring Cloud : It provides some common tools to quickly build distributed systems.

=> It provides service registry to register all our microservices at one place

=> It provides API Gateway to have single entry point for all our apis

=> Load Balancer

=> Monitoring

=> Circuit Breaker ( Fault Tolerant Systems / Resillence )

=> Distributed Messaging

=> Routing

### 10) Spring Test : It provides Unit Test framework

=====

=====  
Spring Core : It is all about Managing dependencies among the classes with loosely coupling  
=====

=> In project we will develop several classes. All those classes we can categorize into 3 types

1) POJO

2) Java Bean

### 3) Component

=====
 What is Pojo ( Plain Old Java Object )
 =====

=> Any Java class which can be compiled by using only JDK software is called as POJO class.

Ex-1 : Below class is valid POJO

```
class Demo1 {  
    int id;  
    String name;  
}
```

Ex-2 : Below class is valid POJO

```
class Demo2 extends Thread {  
    int id;  
    String name;  
}
```

Ex-3 : Below class is valid POJO

```
class Demo3 implements Runnable {  
    // run method  
}
```

Ex-4 : Below class is not POJO because Servlet is part of JEE

```
class Demo4 implements Servlet {  
    // run method  
}
```

=====
 What is Java Bean ?
 =====

=> Any java class which follows bean specification rules is called as Java Bean.

- 1) Class should implement serializable interface
- 2) Class should have private data members (variables)
- 3) Every private variable should have public setter & public getter method
- 4) Class should have zero-param constructor

Note : Bean classes are used to write business logic and to store and retrieve data

=====
 What is Component ?
 =====

=> The java classes which contains business logic is called as Component classes

Ex: Controllers, Services , Dao classes

=> Controller classes will have logic to deal with Request & Response

=> Service classes will have business logic of our project

Ex: Generate OTP, Send OTP, Send Email, Encrypt & Decrypt PWD etc...

=> DAO classes will contain the logic to communicate with Database

---

=> In a project we will develop multiple classes and those classes will be dependent on other classes.

Ex:

=> Controller class will call service class methods

=> Service class will call Dao class methods

=> In Java one class can talk to another class in 2 ways

1) Inheritance ( IS - A )

2) Composition ( HAS - A )

---

Car & Engine Example

---

```
class Engine {  
    void start( ) {  
        // logic  
    }  
}
```

```
class Car {  
    void drive( ) {  
        // start the engine  
        // drive the car  
    }  
}
```

=> If we want to drive the car then we need to start the Engine that means Car class drive ( ) method should call Engine class start ( ) method.

Q) In how many ways Car class can call Engine class method ?

=> in 2 ways

1) Inheritance

2) Composition

---

===== IS-A approach =====

package in.ashokit;

```
public class Engine {
```

```

public int start ( ) {
    // logic
    return 1;
}
}

package in.ashokit;

public class Car extends Engine {

    public void drive() {

        // start the engine

        int start = super.start();

        if (start >= 1) {
            System.out.println("Journey Started");
        }

        // start the journey
    }
}
===== HAS - A Relation =====

```

```

public class Car {

    public void drive() {

        Engine eng = new Engine ( );
        int start = eng.start();

        if (start >= 1) {
            System.out.println("Journey Started");
        }

        // start the journey
    }
}

```

Note: Tomorrow, if Engine class constructor modified then our Car class will be effected.

Note: If we use any approach from above then Car class will become tightly coupled with Engine class. That is not recommended.

##### Always we need to develop our classes with Loosely Coupling #####

=> Loosely coupling means without creating Object and without Inheriting properties we should be able to access one class method in another class.

=> If we make any changes in Engine class then Car class shouldn't be effected then we can say our classes are loosely coupled.

##### To develop classes with loosely coupling we need to use Interfaces #####

```
package in.ashokit;
```

```

public class Car {

    private IEngine eng;

    public Car(IEngine eng) {
        this.eng = eng;
    }

    public void drive() {
        int start = eng.start();

        if (start >= 1) {
            System.out.println("Journey Started...");
        } else {
            System.out.println("Engine in trouble...");
        }
    }
}

-----
package in.ashokit;

public class Main {

    public static void main(String[] args) {

        Car car = new Car (new PetrolEngine());
        car.drive();
    }
}

```

=====
What is Dependency Injection ?  
=====

=> The process of injecting one class object into another class is called as 'Dependency Injection'.  
=> We can perform Dependency Injection in 3 ways

- 1) Setter Injection
- 2) Constructor Injection
- 3) Field Injection

----- Example to Understand Dependency Injection -----

```

public class Car {

    private IEngine eng ;

    public void setEng(IEngine eng) {
        this.eng = eng;
    }

    public void drive() {
        int start = eng.start();

        if (start >= 1) {
            System.out.println("Journey Started...");
        } else {

```

```

        System.out.println("Engine in trouble...");
    }
}
-----
```

=> In the above program 'Car' class is dependent on 'Engine' object that means 'Engine' class object should be injected into 'Car' class.

Note: Car is dependent on Engine

```
=====
Setter Injection (SI)
=====
```

=> Setter Injection means, Injecting dependent object into target object using target class setter method.

```

public class Car {

    private IEngine eng;

    public void setEng(IEngine eng) {
        this.eng = eng;
    }

    public void drive() {
        int start = eng.start();
        // logic
    }
}

public class Main {

    public static void main(String[] args) {

        // creating target obj
        Car car = new Car();

        // injecting dependent obj into target thru setter method (Setter Injection - SI)
        car.setEng(new PetrolEngine());

        car.drive();
    }
}
```

```
=====
Constructor Injection ( CI )
=====
```

=> Constructor Injection means, Injecting dependent object into target object using target class constructor.

```

public class Car {

    private IEngine eng;

    public Car (IEngine eng) {
        this.eng = eng;
    }
}
```

```

        public void drive() {
            int start = eng.start();
            // logic
        }
    }

public class Main {

    public static void main(String[] args) {

        // creating target obj ( Constructor Injection )
        Car car = new Car(new DieselEngine());

        car.drive();
    }
}
=====
```

Q) Can we perform both SI & CI for single variable ?

Yes, but Setter Injection will override Constructor Injection value.

```

public class Main {

    public static void main(String[] args) {

        // creating target obj ( Constructor Injection - CI )
        Car car = new Car(new DieselEngine());

        // Setter Injection - SI
        car.setEng(new PetrolEngine());

        car.drive();
    }
}
```

=====  
**Field Injection - FI**  
=====

=> Field Injection means, injecting depending object into target class using target class variable is called as Field Injection.

```

public class Car {

    private IEngine eng;

    public void drive() {
        int start = eng.start();

        if (start >= 1) {
            System.out.println("Journey Started...");
        } else {
            System.out.println("Engine in trouble...");
        }
    }
}
```

##### Note: We can access private variables outside of the class using Reflection API like below

```
#####

```

```
public class Main {

    public static void main(String[] args) throws Exception {
        Class<?> clz = Class.forName("in.ashokit.Car");

        Object object = clz.newInstance();
        Car carObj = (Car) object;

        Field engField = clz.getDeclaredField("eng");
        engField.setAccessible(true);

        // Injecting value to variable
        engField.set(carObj, new PetrolEngine());

        carObj.drive();
    }
}
```

```
=====
IOC Container
=====
```

- > IoC stands for Inversion Of Control.
- > IoC is responsible for Dependency Injection in Spring Applications.
- > Dependency Injection means creating and injecting dependent bean objects into target bean classes.

Note: IoC container will manage life cycle of Spring Beans.

Note : We need to provide " Java classes + Bean Configuration " as input for IOC then IOC will perform DI and provides Spring Beans which are ready to use.

```
=====
What is Spring Bean ?
=====
```

-> Any Java class whose lifecycle (creation to destruction) is managed by IOC is called as Spring Bean.

-> We can represent Java class as Spring Bean in 2 ways

1) XML Approach

Ex: <bean id ="id1" class = "pkg.ClassName" />

2) Annotation Approach (Recommended)

Ex: @Component, @Service, @Repository etc....

Note: In Spring we can use both XML & Annotation approaches. SpringBoot will support only Annotations (no xmls)

```
=====
How to Start IoC in Spring ?
=====
```

1) BeanFactory (Outdated)

2) ApplicationContext (recommended)

```
Ex:     ApplicationContext context = new ClassPathXmlApplicationContext(String configFile);
```

Note: Bean Configuration file contains Bean Definitions

(target class, dependent class, Dependency Injection type)

Note: When IoC container started it will read bean definitions from Bean Configuration File and it will perform Dependency Injection.

=====
First Application Development using Spring Core Module
=====

Pre-Requisites : JDK 1.8v, STS IDE

1) Create Maven Project in IDE

2) Add 'Spring Context' Dependency in pom.xml file (search here : [www.mvnrepository.com](http://www.mvnrepository.com))

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.22.RELEASE</version>
    </dependency>
</dependencies>
```

3) Create Required Java classes (Ex: IEngine, PetrolEngine, DieselEngine and Car )

4) Create Bean Configuration File and configure Bean Definitions

```
<?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="petrolEng" class="in.ashokit.beans.PetrolEngine" />

    <bean id="car" class="in.ashokit.beans.Car">
        <property name="eng" ref="petrolEng" />
    </bean>

</beans>
```

Note: Here <property/> tag represents setter injection.

5) Create Main class and start IOC Container to test the application.

```
public class App {

    public static void main(String[] args) {
        // starting iOC container
        ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

        // getting bean object from IOC
```

```

        Car car = context.getBean(Car.class);
        car.drive();
    }
}
=====
```

## Difference Between BeanFactory & ApplicationContext

=> BeanFactory interface having - XmlBeanFactory as implementation class

```
BeanFactory factory = new XmlBeanFactory(new ClassPathResource("Beans.xml"));
```

=> ApplicationContext interface having - ClassPathXmlApplicationContext as implementation class

```
ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
```

=> BeanFactory will follow Lazy Loading concept that means when we request then only it will create Bean object.

=> ApplicationContext will follow Eager Loading for Singleton Beans. For Prototype beans it will also follow Lazy Loading

-----Note: Spring Bean default scope is Singleton-----

Eager Loading means creating objects for Spring Bean when IoC starts

Lazy Loading means creating objects for Spring Bean when we call getBean ( ) method.

Note: XmlBeanFactory is deprecated that means it may not available in future versions of Spring.

Note: It is recommended to create IoC using Application Context.

```

public class Main {

    public static void main(String[] args) {
        System.out.println("===== BeanFactory Started =====");
        BeanFactory factory = new XmlBeanFactory(new ClassPathResource("Beans.xml"));

        System.out.println("===== ApplicationContex Started =====");
        ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
        Car bean = context.getBean(Car.class);
        bean.drive();
    }
}
```

## How to differentiate Setter Injection & Construction Injection in Bean Config File ?

=> <property /> tag represents setter injection

=> <constructor-arg /> tag represents constructor injection

```
<bean id="petrolEng" class="in.ashokit.beans.PetrolEngine" />
<bean id="dieselEng" class="in.ashokit.beans.DieselEngine" />
<bean id="car" class="in.ashokit.beans.Car">
    <property name="eng" ref="dieselEng" />
    <constructor-arg name="eng" ref="petrolEng" />
</bean>
```

Note: When we perform both SI & CI then SI will override CI value.

=====

=====
Bean Scopes
=====

=> Scope represents how many objects should be created for a Spring Bean

=> In Spring framework we have below scopes

- 1) singleton ( default scope )
- 2) prototype
- 3) request
- 4) session

=> To represent bean scope we will use "scope" attribute

```
<bean id="id" class="pkg.classname" scope = "singleton | prototype | request | session" />
```

-> Singleton scope means only one object will be created for the class in IOC Container. This is default scope of spring bean.

-> Prototype scope means every time new object will be created.

Note: request & session scopes are related to Spring Web MVC Module.

=====
Why Spring Bean is by default Singleton ?
=====

=> To save memory of JVM spring team made singleton as default scope for the spring beans.

Ex: Rest Controllers, Controllers, Services and DAOs will be considered as Singleton in the project

Ex: TicketGenerator class is used to generate new Ticket for every customer

TicketGenerator ----> Singleton bean

Ticket -----> Prototype Bean

```
=====
Autowiring
=====
```

=> We can inject dependent bean into target in 2 ways

- 1) Manual Wiring

- 2) Autowiring

-> Manual wiring means programmer will inject dependent object into target object using <property/> tag or <constructor-arg> tag using 'ref' attribute.

```
<bean id="dieselEng" class="in.ashokit.beans.DieselEngine" />

<bean id="car" class="in.ashokit.beans.Car">
    <property name="eng" ref="dieselEng" />
</bean>
```

-> Autowiring means IoC container will identify dependent bean and it will inject into target bean (we no need to use any ref attribute in bean configuration file)

-> Autowiring will work based on below modes

- 1) byName
- 2) byType
- 3) constructor
- 4) no

Note: Autowiring will not work bydefault, We have to enable autowiring on target bean like below.

```
<bean id="id" class="pkg.Classname" auto-wire="byName | byType | constructor | no "
/>
```

```
=====
byName
=====
```

=> byName means IoC will identify dependent bean object based on bean id or bean name.

```
public class Car {

    private IEngine eng;

    public void setEng(IEngine eng) {
        System.out.println("setEng ( ) method called....");
        this.eng = eng;
    }

    public void drive() {
        int status = eng.start();

        if (status >= 1) {
            System.out.println("Journey Started..");
        } else {
            System.out.println("Engine Trouble");
        }
    }
}
```

```
}
<?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="eng1" class="in.ashokit.beans.PetrolEngine" />
    <bean id="eng" class="in.ashokit.beans.DieselEngine" />
    <bean id="car" class="in.ashokit.beans.Car" autowire="byName"/>
</beans>
```

Note: In the above example Car class variable name is matched with 'DieselEngine' bean id hence DieselEngine obj will be injected into Car.

```
=====
byType
=====
```

=> byType means IoC will identify dependent bean object based on data type of the variable in Target class.

```
private IEngine eng;      =====> data type of eng is IEngine which is an
interface
```

=> If one interface having 2 implementations then there is a chance of getting Ambiguity problem. To overcome that we need to use 'autowire-candidate' attribute.

```
autowire-candidate="false"      =====> Not Eligible for Autowiring
autowire-candidate="true"       ====> Eligible for Autowiring
```

Note: As an alternate for "autowire-candidate=true" we can use "primary=true" to consider bean for Autowiring.

```
<?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="eng2" class="in.ashokit.beans.PetrolEngine" autowire-candidate="false"/>
    <bean id="eng1" class="in.ashokit.beans.DieselEngine" autowire-candidate="true"/>
    <bean id="car" class="in.ashokit.beans.Car" autowire="byType" />
</beans>
```

(or)

```
<?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="eng2" class="in.ashokit.beans.PetrolEngine" primary="true" />
```

```

<bean id="eng1" class="in.ashokit.beans.DieselEngine" />
<bean id="car" class="in.ashokit.beans.Car" autowire="constructor" />
</beans>

```

=====  
constructor  
=====

=> It is used to perform Autowiring by calling target class constructor

```
package in.ashokit.beans;
```

```
public class Car {
```

```
    private IEngine eng;
```

```
    public Car(IEngine eng) {
        this.eng = eng;
    }
```

```
    public void drive() {
        int status = eng.start();

        if (status >= 1) {
            System.out.println("Journey Started..");
        } else {
            System.out.println("Engine Trouble");
        }
    }
}
```

```
<?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="eng2" class="in.ashokit.beans.PetrolEngine" autowire-candidate="false"/>
    <bean id="eng1" class="in.ashokit.beans.DieselEngine" autowire-candidate="true"/>
    <bean id="car" class="in.ashokit.beans.Car" autowire="constructor" />

```

```
</beans>
```

=====

Note: Autowiring is applicable for Reference Type variable (not applicable for primitive types)

=====  
Spring Bean Life Cycle  
=====

=> Life cycle means starting to ending or birth to death

- > Thread Life Cycle

- > Servlet Life Cycle

- > JSP Life cycle

-> Spring Bean Life Cycle

=> Spring Bean object creation and object destruction will be taken care by IOC container.

=> Spring Bean Life Cycle will be managed by Ioc Container.

=> We can perform some operations using Bean Life Cycle Methods

```
init ( )      -----> initialization logic
destroy ( )   -----> destruction logic
```

=> Spring Bean Life Cycle methods we can execute in 3 ways

- 1) XML Approach (Declarative)
- 2) Programmatic approach
- 3) Annotations

===== Bean Life Cycle using XML Approach =====

```
<bean id="motor" class="in.ashokit.beans.Motor"
      init-method="start"
      destroy-method="stop"/>
```

init-method ==> It represents the method which should be called after bean obj created

destroy-method ==> It represents the method which should be called when bean obj removing from IoC

```
package in.ashokit.beans;
```

```
public class Motor {

    public Motor() {
        System.out.println("Motor :: Constructor");
    }

    public void start() {
        System.out.println("Motor started....");
    }

    public void doWork() {
        System.out.println("Motor Pulling Water...");
    }

    public void stop() {
        System.out.println("Motor stopped.....");
    }
}
```

```
<?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="motor" class="in.ashokit.beans.Motor"
```

```
    init-method="start" destroy-method="stop" />
</beans>
```

===== Bean Life Cycle using Programmatic Approach =====

=> We need to implement predefined interfaces provided by Spring Framework

- 1) InitializingBean ---> afterPropertiesSet ( )
- 2) DisposableBean ---> destroy ( )

```
public class Motor implements InitializingBean, DisposableBean {

    public Motor() {
        System.out.println("Motor :: Constructor");
    }

    public void afterPropertiesSet() throws Exception {
        System.out.println("motor started.....");
    }

    public void doWork() {
        System.out.println("Motor Pulling Water...");
    }

    public void destroy() throws Exception {
        System.out.println("motor stopped.....");
    }
}
```

===== Bean Life Cycle using Annotation Approach =====

@PostConstruct ----> It represents init method

@PreDestroy ----> It represents destroy method

```
public class Motor {

    public Motor() {
        System.out.println("Motor :: Constructor");
    }

    @PostConstruct
    public void m1() throws Exception {
        System.out.println("motor started.....");
    }

    public void doWork() {
        System.out.println("Motor Pulling Water...");
    }

    @PreDestroy
    public void m2() throws Exception {
        System.out.println("motor stopped.....");
    }
}
```

=====

- 1) What is Spring Bean
- 2) How to represent Java class as Spring Bean
- 3) What is IoC Container
- 4) What is Bean Configuration File
- 5) How to start IOC Container
- 6) First Application Development using Spring Core Module
- 7) What is Dependency Injection (SI , CI & FI )
- 8) BeanFactory vs ApplicationContext
- 9) Lazy loading vs Eager Loading
- 10) Bean Scopes (Singleton & Prototype)
- 11) Autowiring & Modes
- 12) Bean Life Cycle Methods (XML, Programmatic & Annotations)

=====  
Spring Annotations  
=====

Annotation ==> Represent Metadata

=> Annotations Introduced in Java 1.5v

=> Annotations are alternate for xml configurations

=> In Spring Framework we have several annotations

- 1) @Configuration : To represent java class as Configuration class
  - 2) @ComponentScan : To identify Spring Bean classes available in the project based on "basePackageNames"
  - 3) @Component : To represent java class as Spring Bean.
  - 4) @Service: To represent java class as Spring Bean (Business layer classes)
  - 5) @Repository: To represent java class as Spring Bean (Persistence layer classes)
  - 6) @Scope : To represent bean scope
  - 7) @Autowired : To perform dependency injection
  - 8) @Qualifier : To perform autowiring based on byName
  - 9) @Primary : To represent primary bean for Autowiring
  - 10) @Bean : To call the method which returns bean object
- =====

```
basePackage = in.ashokit

    in.ashokit.beans -----> will be scanned
    in.ashokit.utils -----> will be scanned
    in.ashokit.dao -----> will be scanned
    com.ashokit.service ---> will not be scanned
```