Spring Day1

Spring is an application framework software, to develop an Enterprise application.

The software community treats spring as a framework of frameworks because it gives the support of various other frameworks also like Hibernate, Struts, JSF, etc.

Spring is an open-source, lightweight application f that can be used in all the layers of a java based business application (i.e. Presentation Layer, Business Logic Layer/Service Layer, Data Access Layer)

Spring makes programming Java quicker, easier, and safer for everybody. Spring’s focus on speed, simplicity, and productivity has made it the world's most popular Java framework.

Spring framework allows to write the business logic of a business application in a POJO class, and its **Spring container** provides the other services with less processing overhead.

Spring framework is considered a lightweight application framework by taking the following things into the consideration:

- Size of the spring container. (it is a simple java class, which can be activated inside any java application)
- Processing overhead.
- POJO and POJI programming model.
- No need to install any server to develop and run our business logic.

The Architecture of Spring framework:

With the help of the Spring framework, we can develop all types of Java applications like:

Desktop(standalone) application, Remoting application, Web application, Enterprise application, etc.

Spring follows the modular architecture, which means spring has several modules to develop a Java Application, based on our requirement we can choose either all the modules or we can choose some specific modules to develop an application.

Some of the spring modules are:

- Spring core module (IOC container): this is the base module of the remaining other modules.
- AOP(Aspect-oriented programming) module: to apply the middleware services in a cross-cutting concern fashion.
- JDBC and DAO module: to develop a data access layer using the JDBC (abstraction on the plain JDBC)
- ORM module: to develop the data access layer using the ORM approach (abstraction over the ORM software).
- Web-MVC module: to develop the presentation layer in an easy manner using MVC architecture (Spring MVC).

- Test module: to conduct the unit test by using the mock objects.

Spring Core module:

Tight coupling and loose coupling between objects:

If one class calls another class functionality, then we can say that both classes are coupled with the other.

Example: If class A calls the functionality of class B then class A will be called a dependent class and class B will be called a dependency class.

```
class B{

    public void funB(){
      System.out.println("inside funB of B");
    }
}



class A {   // Dependent

  B b1 = new B();  //dependency

  public void funA(){
    System.out.println("inside funA of A");
    b1.funB();
  }

}
```

Here A class needs the service of B class then A will be dependent on B

If the dependent class wants to call the methods of the dependency class then, it has to create an object of its dependency class, and then the dependent class can call the functionality of its dependency class.

Now suppose, if any changes are made in the dependency class and if it is forced to do the changes inside the dependent class also then we can say that both classes are tightly coupled with each other.

Example:

```
//dependency class
public class Car {


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

```
//dependent on the car class
public class Travel {

  Car c=new Car();

  public void journey() {
    c.start();
    System.out.println("Jounrney started...");
  }

}
```

Here both the dependent and the dependency classes are tightly coupled with each other, because if change the method name start() to go() inside the dependency class then we need to change the same inside the dependent class also.

Tight coupling generates the problem in another way also, when the dependent class wants to change from the current dependency to another similar dependency ex:

```
//another dependency
public class Bike {


  public void ride() {
    System.out.println("ride started....");
  }
}
```

Here if we change the dependency from Car to Bike we need to modify the dependent class also.

Example:

```

public class Travel {

  //Car c=new Car();

  Bike b=new Bike();

  public void journey() {
    //c.start();

    b.ride();
    System.out.println("Jounrney started...");
  }

}
```

In order to get loose coupling from dependent to its dependencies we need to follow the following rules:

1. Design the dependencies classes by the following POJO by the POJI model.

2. Apply the Dependency Injection mechanism.

Example:

```
//Vehicle.java
 interface Vehicle {

  public void go();

}
```

```
//dependency class
 class Car implements Vehicle{


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

  @Override
  public void go() {
    start();

  }
}


 class Bike implements Vehicle{


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

  @Override
  public void go() {
    ride();

  }
}




//dependent on the car class
 class Travel {

  //it is the dependency
  Vehicle v; //here we can store one of its implemented class obj.


  //constructor injection point
  /*public Travel(Vehicle v) {
    this.v=v;
  } */

  //setter injection point
  public void setV(Vehicle v) {
    this.v=v;
  }

  public void journey() {
    v.go();
    System.out.println("Jounrney started...");
  }

}




 class Demo {

  public static void main(String[] args) {


    //Travel tr=new Travel(new Car()); //injecting the dependency obj to the dependent, by calling constrcutor injection.

    Travel tr=new Travel();

    tr.setV(new Car()); // injecting the dependency obj to the dependent, by calling setter method.

    tr.journey();


  }
}
```

IOC (Inversion of control):

- In general, when a class depends on another class then the dependent class creates the object of its dependency class directly, and then uses(call) its methods, it is called the dependency object created in the main control.
- But if some external entity is taking care of creating the dependency object then we say that control is inverted to some external entity, this is called **Inversion of Control (IOC).**

The IOC is a design principle that is purely conceptual, in which an external entity provides the dependencies to the application components instead of hard coding them in the component class.

Dependency Injection:

Here external entity will push the dependency object to the dependent class (configuring the object inside some container. and the container will push the dependency object to the dependent class).

Note: In the spring framework, The **spring container** is the external entity that will push the dependency object to the dependent class.

Spring Bean:

Any java class, which will be registered with the "**Spring container software"** is known as Spring bean.
The Spring bean is basically a POJO class.

Spring container:

In spring application, A pure java class that controls all the spring beans is known as a spring container.

The Spring container is not like a J2EE container (EJB container, Web container), so no need to install any kind of server.

The Spring container is a lightweight container.it can be activated in any kind of Java Application.

Note: All the core functionalities provided by the spring framework are available through the spring container only.

Spring container provides enterprise services with less processing overhead and reduced complexities to the spring beans.

In addition to enterprise services, spring container provides the following basic services to the Spring Bean:

1. Life-cycle management of the spring bean without implementing any interface inside our spring bean class.
2. Dependency resolution: it will inject the dependency object to the dependent object based on the configuration done in a spring configuration file(XML file) / by using annotations also.

Spring framework provides 2 types of containers:

1. BeanFactory
2. ApplicationContext

```
BeanFactory(I)
           |
ApplicationContext(I)
```

ApplicationContext is the child interface of the BeanFactory interface.

BeanFactory is the basic container with some basic functionalities whereas ApplicationContext is the advanced container, with enhanced functionalities.

In the Spring latest version, the BeanFactory container is deprecated. we should use only ApplicationContext container for Spring Core module.

To start the ApplicationContext container we need to instantiate

1. ClassPathXmlApplicationContext

ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml"); // if the "applicationContext.xml"(spring configuration file is in same path)

Spring configuration file:

In the spring application, the spring container creates the object of spring beans, injects the dependencies to the dependent and manages the life cycle of the spring bean.

To tell the spring container about our application classes and their dependencies, we must configure our classes into an XML file which is called a spring configuration file.

We register any java class with the spring container, using this configuration file only. then that java class will become a spring bean.

We can give any name to this XML file but the recommended name is (**applicationContext.xml**)

In this XML file, the root element is **<beans>**

Inside this **<beans>** tag we need to configure our java classes using **<bean>** tag.

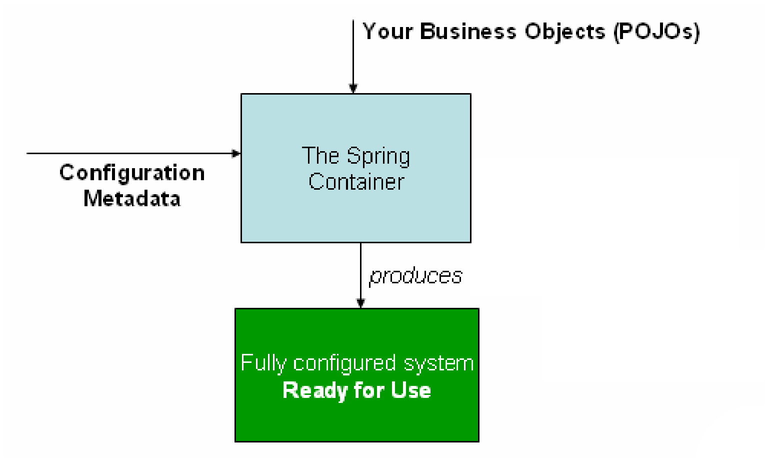
```
<bean id="id" class="fully qualified java class name"/>
```

Example:

```
<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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="d1" class="com.masai.Demo"/>

</beans>
```



Steps to create a Spring application:

Step 1: Create a maven application.

Step 2: Add the following dependency inside pom.xml file:

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>5.3.18</version>
</dependency>
```

Step 3: Develop a Java class :

```
package com.masai;

public class MyBusinessClass {

  public void fun1() {
    System.out.println("inside fun1 ");
  }
}
```

Step 4: Register the above java class with the Spring container and make it a Spring bean using the Spring configuration xml file

Develop the Spring configuration file **“applicationContext.xml”** inside **src/main/resources** folder .

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="mb" class="com.masai.MyBusinessClass" />


</beans>
```

Step 5: Activate the Spring Container and pull the object of Spring Bean from the container through its **id** and call the business method.

Example

Main.java

```
package com.masai;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

  public static void main(String[] args) {

    //activating the Spring container
    ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");

    MyBusinessClass mb = ctx.getBean(MyBusinessClass.class,"mb");

    mb.fun1();

  }

}
```

Configuring different types of dependencies in spring application:

In spring application, every instance variable defined inside a spring bean class can be considered as a dependency.

A spring bean has 3 types of dependencies:-

1. Simple value type dependency (primitive type or String type) :ex:- int, float, double, char, String
2. User-defined object type dependency (ex: Student, Employee, College)
3. Collection type dependencies: (ex: normal array, List, Set, Map)

Example:

```
class X{
   int i;  //simple type dependency
   String s; //simple type dependency

   Student student; // Obejct type dependency

   int[] arr; // collection Depenendency

   List<Student> theList; // collection dependency

 }
```

dthType of Injection (Injection point):

In spring, we have 2 types of injection (Injection point)

1. **Setter injection:** Here container will call the setter method to inject the dependency to the dependent.

   Here we use **<property>** tag inside the **<bean>** tag.

   The attribute of this **<property>** tag are:

   1. **name:** Name of the property (dependency)
   2. **value**
   3. **ref**

   If the dependency is the simple value then we use **"value"** whereas if the dependency is object type then we use **"ref"**.

   We can't use **value** and **ref** at the same time.

   In case of ref:- we need to pass other registered spring bean id.

   Example:

   ```
   package com.masai;
   public class Travel {

     //it is the dependency (object type)
     Vehicle v;


     //simple type
     int numberOFPerson;


     //setter injection point for numberOfPerson
     public void setNumberOFPerson(int numberOFPerson) {
       this.numberOFPerson = numberOFPerson;
     }


     //setter injection point for v
     public void setV(Vehicle v) {
       this.v=v;
     }


     public void journey() {
       v.go();
       System.out.println("Jounrney started...");
   ```

```
    System.out.println("number of person are :"+numberOFPerson);
  }

}
```

applicationContext.xml:

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="c" class="com.masai.Car" />


   <bean id="b" class="com.masai.Bike" />


   <bean id="t" class="com.masai.Travel" >

    <property name="v" ref="b"/>   //injecting Object dependency
    <property name="numberOFPerson" value="8"/>  //injecting simple value

   </bean>


</beans>
```

Main.java:

```
package com.masai;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

  public static void main(String[] args) {

    //activating the Spring container
    ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");

    Travel tr = ctx.getBean(Travel.class,"t");

      tr.journey();

  }

}
```

2. **Constructor injection:** Here container will call the parameterized constructor defined in our class to inject the dependency to the dependent.

For this we use **<constructor-arg>** tag inside the **<bean>** tag.

For each dependency, we need to take a separate **<constructor-arg>** tag.

Example: Travel.java using constructor injection point

```
package com.masai;


public class Travel {

  //it is the dependency (object type)
  Vehicle v;


  //simple type
  int numberOFPerson;


  //constructor injection point..
  public Travel(Vehicle v, int numberOFPerson) {
    this.v = v;
    this.numberOFPerson = numberOFPerson;
  }

  public void journey() {
    v.go();
    System.out.println("Jounrney started...");

    System.out.println("number of person are :"+numberOFPerson);
  }




}
```

applicationContext.xml:

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="c" class="com.masai.Car" />


   <bean id="b" class="com.masai.Bike" />



   <bean id="t" class="com.masai.Travel" >

   <constructor-arg ref="c" />
   <constructor-arg value="10" />

   </bean>


</beans>
```

Note:- if we define a parameterized constructor injection point to a spring bean class, then passing value through the <constructor-arg> tag is mandatory, otherwise it will raise an exception.

Circular Dependency:

Suppose we have 2 beans A and B, where A depends upon B and B depends upon A, and in both A and B parameterized constructors are defined for injecting the dependency, then it will cause a circular dependency and it will throw an exception:.

Example:

```
A.java:-
----------

package com.masai;

public class A {


  private B b1;

  public A(B b1) {
    this.b1=b1;
  }

  public void showA() {

    System.out.println("inside showA of A ");
    System.out.println(b1);


  }
}


B.java:-
---------

package com.masai;

public class B {

  private A a1;


  public B(A a1) {
    this.a1=a1;
  }


  public void showB() {
    System.out.println("inside showB of B");
    System.out.println(a1);

  }

}
```

applicationContext.xml:

```
<bean id="aid" class="com.masai.A" >
    <constructor-arg ref="bid"/>
  </bean>

  <bean id="bid" class="com.masai.B" >
    <constructor-arg ref="aid"/>
  </bean>
```

Demo.java:

```
package com.masai;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Demo {

  public static void main(String[] args) {


    //activate the spring container by locating the spring configuration file..

    ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml");

    A a =  ctx.getBean("aid",A.class);//pulling the A obj

    a.showA();

  }

}
```

Note:- to solve the circular dependency injection problem, at least one side we need to change the dependency injection type from the constructor to setter injection.

```
A.java:-
---------

package com.masai;

public class A {


  private B b1;

//setter injection point
  public void setB1(B b1) {
    this.b1 = b1;
  }

  public void showA() {

    System.out.println("inside showA of A ");
    System.out.println(b1);


  }
}
```

```
B.java:-
---------

package com.masai;

public class B {

  private A a1;

  //constructor injecton point
  public B(A a1) {
    this.a1=a1;
  }


  public void showB() {
    System.out.println("inside showB of B");
    System.out.println(a1);

  }

}



applicationContext.xml:-
----------------------------

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


  <bean id="aid" class="com.masai.A" >
    <property name="b1" ref="bid"/>
  </bean>

  <bean id="bid" class="com.masai.B" >
    <constructor-arg ref="aid"/>
  </bean>

Demo.java:-
---------------


public class Demo {

  public static void main(String[] args) {


    //activate the spring container by locating the spring configuration file..

    ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml");

    A a =  ctx.getBean("aid",A.class);//pulling the A obj

    a.showA();

  }

}
```

Here container will create A class object by using default constructor, and B class object by using parameterized constructor supplying A class object which is already created.

Difference between setter and constructor injection:

Constructor	**Setter**
It may cause circular dependency	It resolve the circular dependency
If the dependency is mandatory	If the dependency is optional
If the dependency is "final" variable then it must be injected through the constructor injection only	We can not inject the final variable through the setter
constructor injected values will be accessible to setter method	setter injected value can not be accessible through the constructor.
constructor injected value can not override the setter injected value.	It can override the constructor injected value.

We Problem:

Dummy Layered Application:

```
DAOBean.java:-
--------------------


package com.masai.daoApp;

public class DAOBean {

  public void findAccount() {

    //taking jdbc or ORM approcah to get the account info from the DB

    System.out.println("Account details feteched and given by DAO Bean of DAL");
  }

}


ServiceBean.java:-
---------------------

package com.masai.daoApp;

public class ServiceBean {


  private DAOBean dao;

  public void setDao(DAOBean dao) {
    this.dao = dao;
  }


  public void calculateInterest() {
    dao.findAccount();

    System.out.println("Interest calculated in Service Layer..");
  }


}
```

```
PresentationBean.java:-
----------------------------

package com.masai.daoApp;

public class PresentationBean {

  private ServiceBean service;

  public void setService(ServiceBean service) {
    this.service = service;
  }


  public void present() {
    service.calculateInterest();

    System.out.println("Pesenting the calculated interesest in PL");
  }


}


applicationContext.xml:-
--------------------------------



  <bean id="db" class="com.masai.daoApp.DAOBean" />


  <bean id="sb" class="com.masai.daoApp.ServiceBean" >
    <property name="dao" ref="db"/>
  </bean>


  <bean id="pb" class="com.masai.daoApp.PresentationBean" >
    <property name="service" ref="sb"/>
  </bean>



Demo.java:-
-------------


public class Demo {

  public static void main(String[] args) {


    //activate the spring container by locating the spring configuration file..

    ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml");

  PresentationBean pbean =  ctx.getBean("pb",PresentationBean.class);

    pbean.present();

  }

}
```

Bean Auto wiring:

The process of creating association between/among application components is known as "wiring".(variable is wired with appropriate object)

We have 2 kind of wiring in spring application:

1. Explicit wiring
2. Auto-wiring (implicit wiring)

If a spring developer specifies the associations for the dependency bean by <property> tag or <constructor-arg> tag, it is known as explicit wiring.

Whereas if spring container on its own detects the dependencies implicitly and injecting them into the dependent bean is known as "auto-wiring".

To instruct the spring container to perform auto-wiring for a particular bean, we make use of the **"autowire"** attribute inside the <bean> tag with any one of following values:

1. no (default)
2. byName
3. byType
4. constructor

In bean auto-wiring, spring container automatically injects a bean dependencies by either calling a setter method or calling parameterized constructor without writing explicitly inside the
spring configuration file.

Limitation of bean auto-wiring:

1. It can be used only to inject the objects but not the simple value dependencies.
2. If container have multiple dependencies of same type to inject , then ambiguity problem may raise.

Note: <bean> tag has "autowire" attribute and its default value is "no" . it means by default auto-wiring is disabled in xml based auto-wiring.

Whereas in annotation based auto-wiring the default type is **"byType".**

1. **byName:**

   In this strategy , if a bean **id** attribute value in applicationContext.xml file matches with the variable/property name of a dependent bean, then spring container implicitly performs the setter injection into the dependent bean , it is known as auto-wire byName

   If the dependency is unmatched , container does not inject that dependency . its means property remains with null value,

   Example:

With respect to previous dummy DAO application:

```
<bean id="dao" class="com.masai.daoApp.DAOBean"/>

<bean id="service" class="com.masai.daoApp.ServiceBean" autowire="byName" />

<bean id="pb" class="com.masai.daoApp.PresentationBean" autowire="byName" />
```

byType:

In this strategy spring container will search for a bean class in spring configuration file, that matched with the property type.

If matched then spring container injects that dependency obj by calling the setter method of dependent class.

If unmatched then that property will remain with null value.

Note:- if more than one bean of same type is encountered in spring configuration file ,then container will throw "UnsatifyDependencyException"

ex:-

```
<bean id="db" class="com.masai.daoApp.DAOBean"/>

<bean id="sb" class="com.masai.daoApp.ServiceBean" autowire="byType" />

<bean id="pb" class="com.masai.daoApp.PresentationBean" autowire="byType" />
```

Here id could be anything.

example :- ambiguity error:

```
<bean id="db" class="com.masai.daoApp.DAOBean"/>

<bean id="db2" class="com.masai.daoApp.DAOBean"/>


<bean id="sb" class="com.masai.daoApp.ServiceBean" autowire="byType" />


<bean id="pb" class="com.masai.daoApp.PresentationBean" autowire="byType" />
```

constructor auto-wiring:

In this strategy , spring container uses constructor injection instead of setter injection.

dependency resolution is done by using max number of argument constructor, if constructors are overloaded.

Note:- it first uses byName and if not found then it uses byType , and in case byType is matched with more that 1 then ambiguity exception will occur.

And if not matched ,here variable will not remain null, it will throw "UnsaticfiedDependencyException"

This kind of auto-wiring is least preferred because of constructor ambiguity. and circular dependency

example:

```
A.java:
--------

package com.masai;
public class A {

public void funA() {
  System.out.println("inside funA of A");
  }
}

 B.java:-
----------
package com.masai;

public class B {
public void funB() {
  System.out.println("inside funB of B");

  }
}

Demo.java:
------------

package com.masai;

public class Demo {

private A a1;
private A a2;
private B b1;

public Demo() {
  System.out.println("inside zero argument constructor..");
}

public Demo(B b1, A a1) {

  this.b1=b1;
  this.a1=a1;
  System.out.println("inside 2 argument constructor..");
}

public Demo(B b1, A a1,A a2) {

  this.b1=b1;
  this.a1=a1;
  this.a2=a2;
  System.out.println("inside 3 argument constructor..");
}

public void showDetails() {

  System.out.println("inside showDetails....");
  System.out.println("b1 is :"+b1);
  System.out.println("a1 is :"+a1);
  System.out.println("a2 is :"+a2);

  }
}
```

applicationContext.xml:-

```
<bean id="d1" class="com.masai.Demo" autowire="constructor" />

  <bean id="a5" class="com.masai.A"/>

  <bean id="a6" class="com.masai.A" />

  <bean id="b5" class="com.masai.B" />
```

```
Main.java:
----------
package com.masai;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {


public static void main(String[] args) {

  //activate the spring container by locating the spring configuration file..

  ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml");

  Demo d=  ctx.getBean("d1",Demo.class);

  d.showDetails();

}


}
```

- -here container will create the object of Demo class by executing zero argument constructor, because 3 argument and 2 argument constructor does not qualify for the dependencies (they generate ambiguity exception A a1)
- -if we change the id of any A class inside the spring configuration file as "a1" then 2nd argument constructor will gets the priority (here 3rd argument constructor will raise the ambiguity exception at A a2).
- -if we change the id of one A class as a1 and another A class as a2 then 3rd argument constructor will gets the priority,
  here B will follow byType and A a1 and A a2 will follow byName strategy.

Initializing and Disposing a Bean:

Before a bean object goes to the client , if we want to perform some initialization logic on that bean obj, then spring f/w provides a way , to write that initialization logic inside a user-defined method and configure that user-defined method name using "**init-method**" attribute of the <bean> tag.

Note:- that method must be zero argument with void return type.

- This "init-method" will be called by the spring container automatically after setting the properties i.e. after performing Dependency Injection.
- So inside this this method we can assess the injected values also.

Example:

```
A.java:-
--------

package com.masai;

public class A {

  private String message;


  public void setMessage(String message) {
    this.message = message;
  }



  public void mySetup() {

    System.out.println("inside mySetup method to write any initialization logic...");
    System.out.println("message is :"+message);
  }



  public void funA() {
    System.out.println("inside funA of A");
  }

}



applicationContext.xml:-
-----------------------------


  <bean id="a1" class="com.masai.A" init-method="mySetup">
    <property name="message" value="Welcome to Spring.."/>

  </bean>

Demo.java:-
--------------

package com.masai;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Demo {

  public static void main(String[] args) {


    //activate the spring container by locating the spring configuration file..

    ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml");

    A obj =  ctx.getBean("a1",A.class);

    obj.funA();
  }
}
```

Disposing a bean:

Before a bean object goes out from the spring container, it is mandatory to release the resources which are allocated by this bean, otherwise memory leak problem will occur.

- -here we can define a user-defined destory() method(any name) and configure it inside <bean> tag by using "**destroy-method**" attribute and write the resource releasing logic in that method.

Note:- destroy() method will be called by the container only when we close/shutdown the spring container explicitly, then all the beans associated by that container will be released.

- To shut down the container we need to downcast the ApplicationContext reference to the its implementation class:
  because close() method is not available inside the ApplicationContext , it is available inside its implementation class only

  ```
  ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml");

  ClassPathXmlApplicationContext ctx2= (ClassPathXmlApplicationContext)ctx;

  ctx2.close();
  ```

Exmaple:

```
A.java:-
---------

package com.masai;

import java.sql.Connection;

public class A {

  private String message;



  public void destroy() {

    System.out.println("inside destroy method..here we can write resource releasing logic..");
  }


  public void setMessage(String message) {
    this.message = message;
  }


  public void mySetup() {

    System.out.println("inside mySetup method to write any initialization logic...");
    System.out.println("message is :"+message);
  }
```

```
  public void funA() {
    System.out.println("inside funA of A");
  }

}


applicationContext.xml:-
------------------------------


  <bean id="a1" class="com.masai.A" init-method="mySetup" destroy-method="destroy">
    <property name="message" value="Welcome to Spring.."/>

  </bean>


Demo.java:-
---------------


public class Demo {

  public static void main(String[] args) {


    //activate the spring container by locating the spring configuration file..

    ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml");

    A obj =  ctx.getBean("a1",A.class);

    obj.funA();

    ClassPathXmlApplicationContext ctx2= (ClassPathXmlApplicationContext)ctx;

    ctx2.close();

  }

}
```

Lazy and eager initialization of a Bean:

ApplicationContext container by default performs early initialization, it means container creates the object of Spring bean when spring configuration xml will be loaded into the memory.

example:

ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml");

- At this line only all the bean class registered inside this xml file will be instantiated,
  i.e. their object will be created, their dependencies will be resolved and if any init-method is configured those method will be called.
- If we want to tell ApplicationContext container that lazy initialize a bean (create the bean class object and resolve its dependencies and call the init-method at time of pulling/ accessing the bean object) then we need to use "**lazy-init**" attribute to the <bean> tag with value **"true".**

```
<bean id="a1" class="com.masai.A" init-method="mySetup" lazy-init="true">
    <property name="message" value="Welcome to Spring.."/>

</bean>
```

Bean scope:

A scope indicates the life span of an object of Spring bean.

Spring f/w defines 4 scopes for a bean:

1. singleton scope (it is a default scope of spring bean)
2. prototype scope
3. request scope
4. session scope

Note: request and session scopes are used only in the web-app (spring -mvc module)

singleton and prototype scope can be used in both web and non-web(standalone) application.

Singleton scope : it returns same bean class object for all the getBean() method call with the same bean id. i.e whenever we are pulling an object with the same id multiple time ,container will return same object.

Example:

```
    A obj1 =  ctx.getBean("a1",A.class);


    A obj2 =  ctx.getBean("a1",A.class);

    System.out.println(obj1 == obj2); //true
```

Note: If the same bean class is configured with different **id** then container creates one more object for that bean, it means container makes a spring bean object as a singleton with respect to the id.

Prototype scope:

It returns a separate bean class object for every getBean() method call even with the same id also.

In order to tell the container to put a bean object in a prototype scope we need to mention by using "**scope**" attribute of the <bean> tag.

Example:

```
<bean id="a1" class="com.masai.A" scope="prototype">
    <property name="message" value="Welcome to Spring.."/>

</bean>
```

```
ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml");


    A obj1 =  ctx.getBean("a1",A.class);


    A obj2 =  ctx.getBean("a1",A.class);

    System.out.println(obj1 == obj2); //false
```

Bean life-cycle:

For ordinary java class constructor and finalize() methods can be considered as life cycle methods.   These methods will be called automatically by the JVM,

- Spring container will control the life-cycle of a **spring bean**, i.e. from instantiation to destruction.
- Spring bean class is POJO class, it need not implements any spring f/w API specific interface or extend any class to facilitate the spring container to control the life cycle of a spring bean. i.e. for spring beans, component contract is not required.

Life-cycle of a spring bean has 5 stages:

1. Does not exist
2. Instantiated
3. Initialized
4. Service
5. Destroy

1. **Does not exist phase:**

- Initially a spring bean object does not exist, if the bean object is in **singleton** scope then bean is instantiated by container at the time of loading the xml file.
- If the scope is **prototype** then it is instantiated at the time of pulling the object/utilizing that object. (i.e. at the time of calling the getBean() method).

2. **Instantiation phase:**

- Spring container loads the spring bean class into the memory and creates the bean class object.
  after that then bean class object is created, spring container uses DI to populate the bean fields (properties/variables)
- All the dependencies will be resolved for a spring bean before its instantiation is finished.

3. I**nitialization phase:**

- If inside the spring bean class any user-defined **init-method** is configured , spring container will call that method.

Note:- unless initialization phase is over , spring container does not gives bean reference to the caller.

4. S**ervice phase / Ready to use phase:**

- Here spring container provides the bean object reference to the caller, then caller can call its business method on that bean object.

5. **Destruction phase:**

- Upon the container shutdown (when we call close() method on the container class) spring container will call **user-defined destroy()** method if it is configured, just before bean instance is garbage collected.

Note:- when we shutdown the container, all the associated bean obj will be eligible for the garbage collection.

Injecting Collection type Dependencies:

Spring framework provides configuration support for the following 5 types of collections:

1. java.util.List
2. java.util.Set
3. java.util.Map
4. java.util.Properties
5. normal arrays

1. If the dependency is the **java.util.List** or **normal arrays** then in spring configuration file we need to use of the **<list>** tag.

- We use this <list> tag under <property> or under <constructor-arg> tag.
- Inside the <list> tag we can use either <value> tag for simple value type dependency or <ref> tag for object type dependency.

Example:

```
A.java:-
----------

package com.masai;

import java.util.List;

public class A {

  //collection type dependency
  private List<String> names;


  //setter injection point
  public void setNames(List<String> names) {
    this.names = names;
  }
  public void show() {

    System.out.println("inside showA of A ");
    System.out.println(names);

  }
}
```

applicationContext.xml file:

```
<bean id="aid" class="com.masai.A">

    <property name="names">

      <list>

        <value>Delhi</value>
        <value>Chennai</value>
        <value>Kolkata</value>
        <value>Mumbai</value>

      </list>

    </property>

  </bean>
```

Demo.java

```
public class Demo {

  public static void main(String[] args) {


    //activate the spring container by locating the spring configuration file..

    ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml");

    A a =  ctx.getBean("aid",A.class);//pulling the A obj

    a.show();

  }

}
```

Example2 :

```
Student.java:-
----------------

package com.masai;

public class Student {
  private int roll;
  private String name;
  private int marks;


  //constructor injection point
  public Student(int roll, String name, int marks) {
    super();
    this.roll = roll;
    this.name = name;
    this.marks = marks;
  }


  public void displayDetails() {
    System.out.println("Roll is :"+roll);
    System.out.println("Name is :"+name);
    System.out.println("Marks is :"+marks);
  }

}


Collage.java:-
------------------

package com.masai;

import java.util.List;

public class Collage {

  private String collageName;

  private List<Student> students;


  public String getCollageName() {
    return collageName;
  }

  //constructor injection point for collageName
  public Collage(String collageName) {
    this.collageName = collageName;
  }

  //setter injection point for List
  public void setStudents(List<Student> students) {
    this.students = students;
  }

  public List<Student> getStudents() {
    return students;
  }
}
```

applicationContext.properties

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="s1" class="com.masai.Student">

    <constructor-arg name="roll" value="100" />
    <constructor-arg name="name" value="Ram" />
    <constructor-arg name="marks" value="780" />

  </bean>

  <bean id="s2" class="com.masai.Student">

    <constructor-arg name="roll" value="101" />
    <constructor-arg name="name" value="Ramesh" />
    <constructor-arg name="marks" value="680" />

  </bean>

  <bean id="s3" class="com.masai.Student">

    <constructor-arg name="roll" value="102" />
    <constructor-arg name="name" value="Sunil" />
    <constructor-arg name="marks" value="880" />

  </bean>


  <bean id="cl1" class="com.masai.Collage">

  <constructor-arg name="collageName" value="NIT"/>

    <property name="students">

      <list>

        <ref bean="s1" />
        <ref bean="s2" />
        <ref bean="s3" />

      </list>

    </property>


  </bean>

</beans>
```

Demo.java:

```
package com.masai;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Demo {

  public static void main(String[] args) {


    //activate the spring container by locating the spring configuration file..
    ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml");



    Collage collage =  ctx.getBean("cl1",Collage.class);

    List<Student> students= collage.getStudents();

    System.out.println("Collage name is "+collage.getCollageName());

    System.out.println("-----------------------------");

    students.forEach(student -> {

      student.displayDetails();
      System.out.println("=================");
    });


  }

}
```

Normal Array:

```
A.java:-
----------

package com.masai;

import java.util.Arrays;

public class A {

  //normal array type dependency
  private String[] names;

  public void setNames(String[] names) {
    this.names = names;
  }


  public void show() {

    System.out.println("inside showA of A ");
    System.out.println(Arrays.toString(names));


  }
}
```

applicationContext.xml:

```
<bean id="aid" class="com.masai.A">

    <property name="names">

    <list>
      <value>Delhi</value>
      <value>Chennai</value>
      <value>Kolkata</value>
      <value>Mumbai</value>

    </list>

    </property>


  </bean>
```

java.util.Set:

Here we need to use <set> tag.

Example:

```
A.java:-
----------

package com.masai;

import java.util.Set;

public class A {


  private Set<String> theSet;

  public void setTheSet(Set<String> theSet) {
    this.theSet = theSet;
  }


  public void show() {

    System.out.println("inside showA of A ");
    System.out.println(theSet);

  }


}
```

application.properties:

```
<bean id="aid" class="com.masai.A">

    <property name="theSet">

    <set>

      <value>Red</value>
      <value>Blue</value>
      <value>White</value>
      <value>Green</value>


    </set>

    </property>
```

```
</bean>
```

java.util.Map:

Spring f/w has provided **<map>** tag to configure the Map type of dependency:

inside <map> tag we need to use **<entry>** tag to inject each entry inside a map.

We can use <entry> tag with the following combination:

1. key , value //(if both key and value is a simple type) //Map<String,String>
2. key-ref, value //(if key is a object and value is simple type) //Map<Student,String>
3. key, value-ref //(if key is simple type and value is an object) //Map<String,Student>
4. key-ref,value-ref (if both key and value is the object type) //Map<Collage,Student>

Note:- whenever we take an user-defined object inside the Map as a key , it is always recommended to override equals() and hashCode() method inside the user-defined object class.

Example:

```
Student.java:-
----------------

package com.masai;

public class Student {

  private int roll;
  private String name;
  private int marks;


  //constructor injection point
  public Student(int roll, String name, int marks) {
    super();
    this.roll = roll;
    this.name = name;
    this.marks = marks;
  }


  public void displayDetails() {
    System.out.println("Roll is :"+roll);
    System.out.println("Name is :"+name);
    System.out.println("Marks is :"+marks);
  }


  @Override
  public boolean equals(Object obj) {

    Student s1= this;
    Student s2=(Student)obj;

    if(s1.roll == s2.roll && s1.name.equals(s2.name) && s1.marks == s2.marks)
      return true;
    else
      return false;
  }

  @Override
  public int hashCode() {

    return this.roll;
  }

}


A.java:-
--------

package com.masai;

import java.util.Map;

public class A {


  private Map<Student, String> theMap;


  public void setTheMap(Map<Student,String> theMap) {
    this.theMap = theMap;
  }



  public void show() {

    System.out.println("inside showA of A ");

    System.out.println(theMap);

  }


}
```

applicationContext.properties:

```
<bean id="s1" class="com.masai.Student">
    <constructor-arg name="roll" value="100"/>
    <constructor-arg name="name" value="Raj"/>
    <constructor-arg name="marks" value="700"/>
  </bean>

  <bean id="s2" class="com.masai.Student">
    <constructor-arg name="roll" value="102"/>
    <constructor-arg name="name" value="Simran"/>
    <constructor-arg name="marks" value="720"/>
  </bean>

  <bean id="s3" class="com.masai.Student">
    <constructor-arg name="roll" value="104"/>
    <constructor-arg name="name" value="Rajesh"/>
    <constructor-arg name="marks" value="750"/>
  </bean>
  <bean id="aid" class="com.masai.A">

    <property name="theMap">

      <map>
          <entry key-ref="s1" value="NIT"/>
          <entry key-ref="s2" value="GIET"/>
          <entry key-ref="s3" value="AMITY"/>



      </map>

    </property>


  </bean>
```

Demo.java:

```


public class Demo {

  public static void main(String[] args) {


    //activate the spring container by locating the spring configuration file..

    ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml");




    A a1 =  ctx.getBean("aid",A.class);

    a1.show();


  }

}
```

java.util.Properties:

Properties is a subclass of Hashtable class , and it generally stores data (key-value) in the form of String.

- Spring f/w provides <props> tag to configure the Properties types of dependency.
- Under this <props> tag we configure each key and value pair with <prop> tag.

Example:

```
A.java:-
---------

package com.masai;

import java.util.Properties;
```

```
public class A {


  private Properties theProperties;


  public void setTheProperties(Properties theProperties) {
    this.theProperties = theProperties;
  }


  public void show() {

    System.out.println("inside showA of A ");

    System.out.println(theProperties);

  }
}
```

applicationContext.properties:

```
<bean id="aid" class="com.masai.A">

    <property name="theProperties">

      <props>

        <prop key="Raj">Delhi</prop>
        <prop key="Sunil">mumbai</prop>
        <prop key="Bikash">Chennai</prop>
        <prop key="Anil">Kolkata</prop>


      </props>

    </property>


  </bean>
```