✯ POJO - Plain Old Java Objects

POJO: Plain Old Java Objects

It is a Java object that doesn't extend or implement some specialized classes and interfaces respectively.

All normal Java objects are POJO.

In POJO model, it is recommended that, interfaces should be explicitly implemented whenever you want to pick and choose the methods of the interface

```
Rules;

1)class must be public .

2)variable must be private.

3)must having public default constructor.

4) can have args constructor .

5)every property/field/variable have public getter and setter method
```

Advantages of POJO Classes

```
1)increase the readability and re-usability of a program

2)easy to write and understand .

getter/accessor---update a value of a variable
setter/mutator --read value of avariable
```

POJO simply *means* a class that does not implement any interface, does not extend any specific class, does not contain any pre-described annotation.

Employee.java:

```
// POJO class Exmaple
public class Employee
{
        private String name;
        private String id;
        private double sal;

        public String getName() {return name; }
        public void setName(String name) {this.name = name; }

        public String getId() {return id; }
        public void setId(String id) {this.id = id; }

        public double getSal() {return sal; }
        public void setSal(double sal) {this.sal = sal; }
}
```

MainClass.java:

```
//Using POJO class objects in MainClass Java program
public class MainClass {
        public static void main(String[] args) {
                // Create an Employee class object
                Employee obj= new Employee(); //POJO class object created.
                obj.setName("Alisha"); // Setting the values using the set() method
                obj.setId("A001");
                obj.setSal(200000);
                //Getting the values using the get() method
                System.out.println("Name: "+ obj.getName());
                System.out.println("Id: " + obj.getId());
                System.out.println("Salary: " +obj.getSal());
        }
}
```

Q5) Explain constructor injection with example. [illegible]

[illegible]

6.2 Dependency injection: [illegible]

- Dependency injection (DI) is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments and properties that are set on the object instance after it is constructed.
- The container injects these dependencies when it creates the object.
- Dependency injection exists in two major variants, Constructor-based dependency injection and Setter-based dependency injection.

6.2.1 Constructor-based dependency injection:

Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency.

The following example shows a class that can only be dependency-injected with constructor injection.

```
public class SimpleMovieLister
{
  // the SimpleMovieLister has a dependency on a MovieFinder
  private IMovieFinder movieFinder;
  // a constructor so that the Spring container can 'inject' a MovieFinder
  public MovieLister(IMovieFinder movieFinder)
   {
    this.movieFinder = movieFinder;
   }
  // business logic that actually 'uses' the injected IMovieFinder is omitted...
}
```

```
<beans>
  <bean id = "movieFinder1" class = "IMovieFinder">
         <property name = “movieName" value = “abcd"/>
  </bean>

  <bean id = "simpleMovieLister1" class = "SimpleMovieLister">
    <constructor-arg ref="movieFinder1"/>
  </bean>
</beans>
```

(3) Explain setter injection with example

6.2.2 Setter-based dependency injection : Setter-based DI is accomplished by the container invoking setter properties on your objects after invoking a no-argument constructor or no-argument static factory method to instantiate your object. The following eample shows a class that can only be dependency injected using pure setter injection.

```
public class SimpleMovieLister
  {
     private IMovieFinder movieFinder;
     public void setMovieFinder (IMovieFinder movieFinder)
     {
           this.movieFinder = movieFinder;
     }
  }
```

```
<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-3.0.xsd">
  <bean id = “add1" class = "Vesit.Address">
    <property name = “Citye" value = “Mumbaii"/>
    <property name = “state" value = “Maharashtra"/>
    <property name = “pin" value = “615702/>
  </bean>
<bean id = "emp1" class = "Vesit.Employee">
   <property name = "name" value = "Rishi"/>
   <property name = "id" value = "12"/>
   <property name = “address" ref= “add1"/>
  </bean>
</beans>
```

```
<beans>
     <bean id = "MovieFinder1" class = "IMovieFinder">
          <property name = "movieName" value = "abcd"/>
     </bean>

     <bean id = "SimpleMovieLister1" class = "SimpleMovieLister">
          <property name = "movieFinder"
                    ref = "MovieFinder1"/>
     </bean>
</beans>
```

* Circular Dependency → occurs during Constructor injection

6.3 Circular dependencies:

There are the issue caused during dependency injection when *spring-context* tries to load objects and one bean depends on another bean. Suppose when Object A & B depends on each other. i.e. A depends on B and vice-versa.
Spring throws UnsatisfiedDependencyException while creating objects of A and B because A object cannot be created until unless B is created and visa-versa.

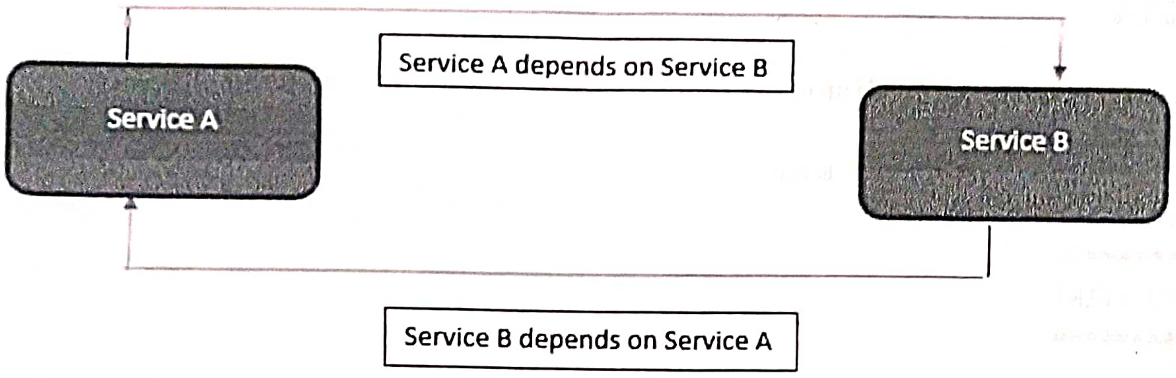

Service A depends on Service B
Service A
Service B
Service B depends on Service A

Fig. **Circular dependencies**

Let' s understand it using the real code example.
Create two services ServiceA and ServiceB and try to inject ServiceA into ServiceB and visa-versa as shown in the above picture.

ServiceA.java

```
import org.springframework.stereotype.Service;
@Service
public class ServiceA {
      private ServiceB serviceB;
      public ServiceA(ServiceB serviceB) {
            System.out.println("Calling Service A");
            this.serviceB = serviceB;
      }
}
```

ServiceB.java

```
import org.springframework.stereotype.Service;
@Service
public class ServiceB {
      private ServiceA serviceA;
      public ServiceB(ServiceA serviceA) {
            System.out.println("Calling Service B");
            this.serviceA = serviceA;
      }
}
```

To simulate the circular dependency issue, run the below class, and see the console log.

CircularDependenciesTestApp.java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
```

```
    public static void main(String[] args) {
        SpringApplication.run(CircularDependenciesTestApp.class, args);
    }
}
```

When we execute CircularDependenciesTestApp class it won' t be able to inject the dependencies due to circular dependencies on each other and will throw a checked exception as shown below:

console log

```
Error starting ApplicationContext. To display the conditions report re-run your
application with 'debug' enabled.
2020-05-27 21:22:46.368 ERROR 4480 --- [ main]
o.s.b.d.LoggingFailureAnalysisReporter :
***************************
APPLICATION FAILED TO START
***************************
```

Description:

```
The dependencies of some of the beans in the application context form a cycle:
┌─────┐
| serviceA defined in file [F:¥sts4-workspace¥circular-dependenciesspring¥
target¥classes¥org¥websparrow¥service¥ServiceA.class]
↑ ↓
| serviceB defined in file [F:¥sts4-workspace¥circular-dependenciesspring¥
target¥classes¥org¥websparrow¥service¥ServiceB.class]
└─────┘
```

6.3.1 How to resolve this issue?

To solve the circular dependency issue, you have two options: Using @Lazy with constructor injection and ~~Using @Autowired along with @Lazy annotation.~~
using setter injection

6.3.1.1 Using @Lazy with constructor injection

We can lazily initialize ServiceB bean during constructor injection in order to delay constructing ServiceB bean. Here are the code changes in ServiceA for more clarity:

ServiceA.java

```
import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Service;
@Service
public class ServiceA {
    private ServiceB serviceB;
    public ServiceA(@Lazy ServiceB serviceB) {
        System.out.println("Calling Service A");
        this.serviceB = serviceB;
    }
}
```

} * . annotation. *

If you run the CircularDependenciesTestApp class again, you' ll find the circular dependency issue is solved.

console log:

```
Calling Service A
Calling Service B
```

Q14) What do you understand by lightweight Containers.

A lightweight container is one that has few (or fewer) additional requirements such as required 3rd party libraries, servers, or other services.
Spring calls itself 'lightweight' because you don't need all of Spring to use part of it. For example, you can use *Spring JDBC* without *Spring MVC*.
Spring provides various modules for different purposes; you can just inject dependencies according to your required module. That is, you don't need to download or inject all dependencies or all JARs to use a particular module.

Fig. Spring architecture

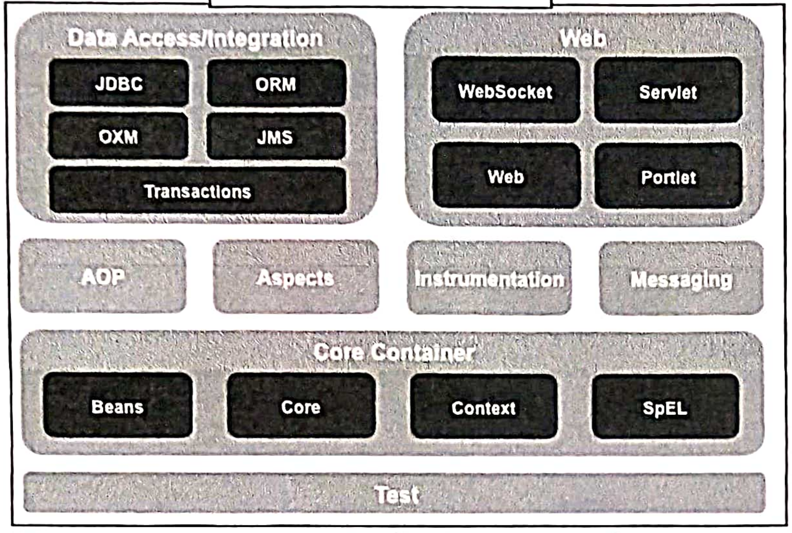
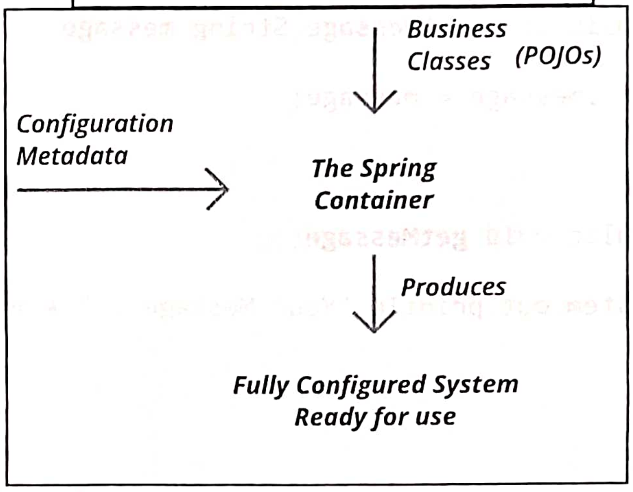


Fig. The Spring IoC container



If you want to run a *Java EE* application, you can't just create a small application that will run on its own. You will need a *Java EE* application server to run your application, such as *Glassfish*, *WebLogic* or *WebSphere*. Most application servers are big and complex pieces of software, that are not trivial to install or configure.
You don't need such a thing with Spring. You can use Spring dependency injection, for example, in any small, standalone program.

Q5) Explain types of IoC containers.

IoC containers

Spring IoC Container is the core of Spring Framework. It creates the objects, configures and assembles their dependencies, manages their entire life cycle. The Container uses Dependency Injection(DI) to manage the components that make up the application. It gets the information about the objects from a configuration file(XML) or Java Code or Java Annotations and Java POJO class. These objects are called Beans. Since the Controlling of Java objects and their lifecycle is not done by the developers, hence the name **Inversion Of Control**.
Spring IoC generally directly refers to a core container that uses the DI/DC pattern to implicitly provide an object reference in a class during runtime. The IoC container contains assembler code that handles the configuration management of application objects.

types of IoC Containers in Spring:

a. **BeanFactory**: BeanFactory is like a factory class that contains a collection of beans. It instantiates the bean whenever asked for by clients.
b. **ApplicationContext**: The ApplicationContext interface is built on top of the BeanFactory interface. It provides some extra functionality on top BeanFactory.

let's see one by one:

a. Spring BeanFactory Container

Spring BeanFactory Container is the simplest container which provides basic support for DI.

It is defined by org.springframework.beans.factory.BeanFactory interface.

The code of HelloWorld.java is as shown.

```
package com.example;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}
```

The following is the code of MainApp.java.

```
package com.example;

import org.springframework.beans.factory.InitializingBean;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;

public class MainApp {

public static void main(String[] args) {

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

HelloWorld obj = (HelloWorld) factory.getBean("helloWorld");

obj.getMessage();

}

}
```

There are some points which should be taken about the main program:

- Write a factory object where you have used APIXmlBeanFactory() to load bean config file in CLASSPATH.
- Use getBean() which uses bean ID to return a generic object to get the required bean.

Following is the XML code for Beans.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

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
<beans>
<bean id = "helloWorld" class = "com.example.HelloWorld">

<property name = "message" value = "Hello World!"/>

</bean>

</beans>
```

After you run the application you will see the following message as output.
Your Message: Hello World!

b. Spring ApplicationContext Container

The ApplicationContext container is Spring's advanced container. It is defined by org.springframework.context.ApplicationContext interface. The ApplicationContext container has all the functionalities of BeanFactory. It is generally recommended over BeanFactory.

The code for HelloWorld.java file:

```
package com.example;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}
```

The code for MainApp.java:

```
package com.example;
```

```
import org.springframework.context.ApplicationContext;

import org.springframework.context.support.FileSystemXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new FileSystemXmlApplicationContext

("C:/Users/ADMIN/workspace/HelloSpring/src/Beans.xml");

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");

obj.getMessage();

}

}
```

ii. Some Points

Some points should note about the main program:

- Using framework API FileSystemXmlApplicationContext create a factory object. This API takes care of creating and initializing all objects.
- Use getBean() which uses bean ID to return a generic object to get the required bean.

The code for Beans.xml is as given:

```
~~<?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-3.0.xsd">~~
<beans>
<bean id = "helloWorld" class = "com.example.HelloWorld">

<property name = "message" value = "Hello World!"/>

</bean>

</beans>
```

After creating source and bean config files, run the application. You will see the following as your output.

Your Message: Hello World!