



spring

# INDEX

Spring Core -----	
1. Spring Core Module	<a href="#">05</a>
2. Dependency Injection	<a href="#">05</a>
a. Setter Injection	<a href="#">05</a>
b. Working with new Instance of java.lang.reflect.Constructor class	<a href="#">14</a>
c. Spring First application flow of execution	<a href="#">20</a>
d. Constructor Injection	<a href="#">24</a>
3. Design Pattern	<a href="#">36</a>
a. Factory Design Pattern	<a href="#">36</a>
b. Strategy Design Pattern	<a href="#">42</a>
4. Resolving/ Identifying params in Constructor Parameters	<a href="#">67</a>
5. Cyclic Dependency Injection or Circular Dependency Injection	<a href="#">71</a>
a. Using Setter Injection	<a href="#">72</a>
b. Using Constructor Injection	<a href="#">75</a>
c. Cyclic Dependency Injection having one side Setter injection and another side Constructor injection	<a href="#">79</a>
6. Difference between setter injection and constructor injection	<a href="#">82</a>
7. Layered Application (Mini Project)	<a href="#">85</a>
a. Story Board of Layered Application	<a href="#">95</a>
8. Collection Injection	<a href="#">116</a>
a. Null Injection	<a href="#">130</a>
9. Bean Inheritance	<a href="#">135</a>
10. Collection Merging	<a href="#">146</a>
a. Default Bean Ids	<a href="#">151</a>
b. Inner Bean	<a href="#">154</a>
11. Bean Alias	<a href="#">159</a>
12. Dependency Lookup	<a href="#">161</a>
13. Bean Wiring (Auto wiring)	<a href="#">166</a>
14. Factory Method Bean Instantiation	<a href="#">181</a>
a. Factory Method	<a href="#">181</a>
15. Singleton java class and Bean Scopes	<a href="#">186</a>
a. Singleton class/ singleton java class	<a href="#">186</a>
b. Bean Scopes	<a href="#">192</a>
16. ApplicationContext container (IoC container)	<a href="#">199</a>

a. Pre-instantiation of singleton scope beans	<a href="#">201</a>
b. Ability to work with place holders and Properties file	<a href="#">206</a>
c. Support for I18n (Internationalization)	<a href="#">211</a>
d. Support for Event handling & Publishing	<a href="#">219</a>
e. Difference between BeanFactory container and ApplicationContext container	<a href="#">222</a>
17.P-Namespace and C-Namespace	<a href="#">223</a>
18.Nested BeanFactory/ Nested ApplicationContext	<a href="#">227</a>
a. Mini Project 2 Discussion	<a href="#">229</a>
19.Annotation driven Spring Programming in Core module	<a href="#">252</a>
a. Stereo type annotations	<a href="#">264</a>
b. Java Config Annotations in Spring environment	<a href="#">281</a>
20.Spring Bean Life Cycle	<a href="#">286</a>
a. Using Declarative Approach	<a href="#">287</a>
b. Using Programmatic Approach	<a href="#">293</a>
c. Using Annotation Approach	<a href="#">296</a>
21.Aware Injection	<a href="#">301</a>
22.Lookup Method Injection (LMI)	<a href="#">310</a>
23.Method Injection or Method Replacer	<a href="#">324</a>
24.Factory Bean	<a href="#">334</a>
25.Service Locator (use case of Factory Bean)	<a href="#">339</a>
26.BeanPostProcessor (BPP)	<a href="#">355</a>
27.BeanFactoryPostProcessors	<a href="#">373</a>
28.PropertyEditor	<a href="#">375</a>
a. Custom PropertyEditor	<a href="#">379</a>
b. PropertyEditorRegistry	<a href="#">381</a>
29.100% code Driven Spring App development	<a href="#">388</a>
30.Spring Boot	<a href="#">401</a>

# Spring Core

## Spring Core Module

- It is base module for other modules.
- If we use this module alone in application development, we can develop only standalone applications.
- This module gives one spring containers/ IoC containers [Inverse of Control]. But we can implement the container in two ways BeanFactory and ApplicationContext to perform.
  - a. Spring bean lifecycle management  
(To manage Spring bean from birth [object creation] to death [object destruction])
  - b. Dependency Management
    - i. Dependency Lookup
    - ii. Dependency Injection

## Dependency Injection

Different modes of Dependency Injection that Spring Supports:

- a. Setter Injection
- b. Constructor Injection
- c. Aware Injection/ Interface Injection/ Contextual Dependency Lookup/ Injection
- d. Lookup method Injection
- e. Method Injection/ Method Replacer

### Setter Injection

- Here IoC container calls setter method of Target class to assign dependent class object to the target class object.
- new operator creates the object at runtime, but expects the presence of Java class from compile time onwards. i.e., we cannot use new operator to create objects for Java class whose class names comes to the application dynamically at runtime.

Test t = new Test();

- In Spring applications, we get Spring bean class names to Spring containers dynamically at runtime from Spring bean configuration file (XML file). So new operator cannot be used to creates objects Spring bean classes, so we need special Spring container that internally uses other concepts to create Spring class object.
- Spring containers simplifies/ automates the Spring Beans Life cycle management and Dependency Management based on the inputs

instructions collected from Spring bean configuration file.

### Sample code for Setter Injection [POC]:

WishMessageGenerator (Target class), java.util.Date (Dependent class)

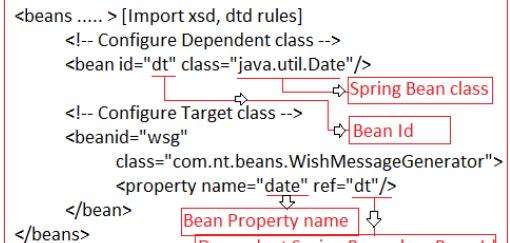
- It has to get system date by using Date class object, so it can use current hours of the day to generate wish message like "Good morning", "Good afternoon", and etc. So "Date" is called as Dependent class and "WishMessageGenerator" is called as target class.

WishMessageGenerator.java [Spring bean - Target class]

```
package com.nt.beans;  
  
import java.util.Date;  
  
public class WishMessageGenerator {  
  
    private Date date; //Spring bean property  
  
    //Setter method to support Setter Injection process  
    public void setDate(Date date) {  
        this.date = date();  
    }  
  
    //Business method using the injection Date class object in business logic  
    public String generateWishMessage (String user) {  
        int hour=0;  
        //Get current hour of the day  
        hour = date.getHour(); //gives in 24 hours format  
        //Business logic to generate wish message  
        if (hours<12)  
            return "Good Morning : "+user;  
        else if (hours<16)  
            return "Good Afternoon : "+user;  
        else if (hours<02)  
            return "Good Evening : "+user;  
        else  
            return "Good Night : "+user;  
    }  
}
```

applicationContext.xml [com.nt.cfgs] : Recommended

```
<beans .... > [Import xsd, dtd rules]  
    <!-- Configure Dependent class -->  
    <bean id="dt" class="java.util.Date"/>  
    <!-- Configure Target class -->  
    <beanid="wsg"  
        class="com.nt.beans.WishMessageGenerator">  
        <property name="date" ref="dt"/>  
    </bean>  
</beans>
```



<property> tag instructs to perform Setter Injection.

<bean> ---> for Spring bean class configuration  
<property> ---> To configure bean property for Setter Injection

- Servlet component is identified with its URL pattern (at servlet container).
- Java object is identified with its Hash code (at JVM).
- Spring bean is identified with its bean id (at Spring container)

### SetterInjectionTest.java (Client App):

```
package com.nt.test;
```

```
public class SetterInjectionTest {  
  
    public static void main (String [] args) {  
        Resource res=null;  
        BeanFactory factory=null;  
        WishMessageGenerator target = null;  
        //Hold the name and location Spring bean configuration file in  
        //Resource object
```

```

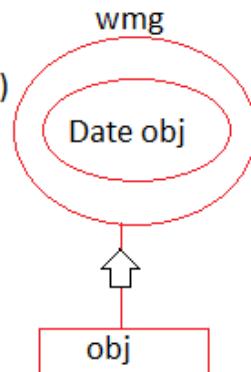
        res = new FileSystemResource
            ("<location>/applicationContext.xml");
        //Create IoC container/ Spring container (BeanFactory)
        factory = new XmlBeanFactory(res);
        //Get Target bean class object from BeanFactory container
        Object obj = factory.getBean("wsg");
        S.o.println(obj.generateWishMessage("raja")); X
        target = (WishMessageGenerator)obj;
        String result = target.generateWishMessage("raja");
        S.o.println("Result is :" + result);
    }

}

```

**public Object getBean(String beanId)**  
 if the java method return type `java.lang.Object`  
 then that method can return any java class object  
 because `java.lang.Object` is the top most in the  
 inheritance hierarchy any java class.

- Using `java.lang.Object` class reference variable we can  
 refer any java class object



### Eclipse:

- Type: IDE for Java
- Version: 2020-06 (compatible with JDK 1.8+)
- Vendor: Eclipse organization
- Open source
- To download software: [download as ZIP file] [\[Download\]](#)
- To install software: Extract the zip file
- Flavors:
  - Eclipse SDK [only for standalone application]
  - Eclipse JEE [for all types of applications development]

Procedure to develop the above Setter Injection example using Eclipse IDE:

**Step 1:** Keep the following software setup ready.

Spring latest version [zip extraction]

JDK 1.8+ version

Eclipse 2019/ 20 versions

**Step 2:** Launch eclipse IDE by choosing workspace folder

**Step 3:** Create java project in Eclipse IDE

**Step 4:** Add Spring core module library to the Project.

spring-beans-<version>.jar

spring-context-<version>.jar

spring-context-support-<version>.jar

common-logging-<version>.jar [collect from internet]

Link common-logging: [\[Download\]](#)

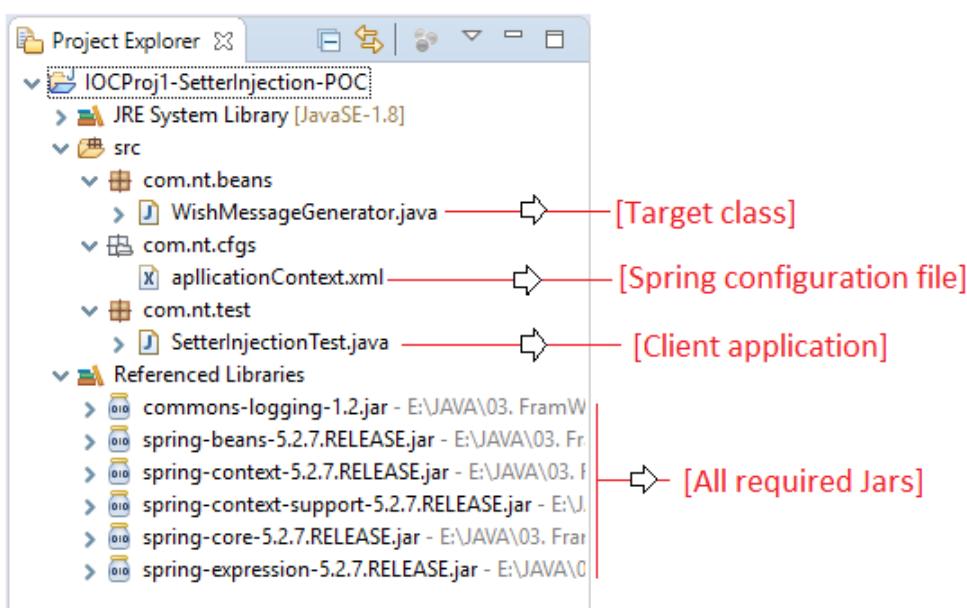
**Step 5:** make sure that following packages created to have resources for application development.

#### IoCProj1-SetterInjection-POC

```
|---> src
    |---> com.nt.beans
        |---> WishMessageGenerator.java [target]
    |---> com.nt.cfgs
        |---> applicationContext.xml [Spring configuration file]
    |---> com.nt.test
        |---> SetterInjectionTest.java [Client application]
|---> reference libraries
    |---> 5+1 jar files
```

**Step 6:** Run the Application

#### Directory Structure of IOCProj01-SetterInjection-POC:



- Develop the above directory structure and package, class, XML file and add the jar also, then use the following code with in their respective file.

### WishMessageGenerator.java

```

package com.nt.beans;

import java.util.Date;

public class WishMessageGenerator {

    // bean property
    private Date date;

    public void setDate(Date date) {
        this.date = date;
    }

    public String generateWishMessage(String user) {
        int hour = 0;
        // get current hour of the dat
        hour = date.getHours();
        // generate wish message (Business logic)
        if (hour < 12)
            return "Good Morning: " + user;
        else if (hour < 16)
            return "Good Afternoon: " + user;
        else if (hour < 20)
            return "Good Evening: " + user;
        else
            return "Good Night: " + user;
    }

}

```

### Collecting XML schema for applicationContext.xml file:

1. Go to spring installation folder [\spring-framework-5.2.7.RELEASE\]
2. Open docs folder

3. Go to spring-framework-reference folder
4. Open core.html file
5. Scroll down and collect the information

```
<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">
```

### 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">

    <!-- Dependent bean configuration -->
    <bean id="dt" class="java.util.Date"/>

    <!-- Target bean configuration -->
    <bean id="wmg" class="com.nt.beans.WishMessageGenerator">
        <property name="date" ref="dt"/>
    </bean>

</beans>
```

### SetterInjectionTest.java [Run the test class after finish]

```
package com.nt.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;

import com.nt.beans.WishMessageGenerator;

public class SetterInjectionTest {
```

```

public static void main(String[] args) {
    Resource res = null;
    BeanFactory factory = null;
    WishMessageGenerator generator = null;
    Object obj = null;
    String result = null;

    // hold name and location of spring bean configuration file
    res = new
    FileSystemResource("src/com/nt/cfgs/applicationContext.xml");
    // create BeanFactory/ IOC container
    factory = new XmlBeanFactory(res);
    // get Target bean class object
    obj = factory.getBean("wmg");
    // typecasting
    generator = (WishMessageGenerator) obj;
    // invoke the method
    result = generator.generateWishMessage("Nimu");
    System.out.println("Wish Message is : " + result);

}

}

```

#### DTD & XSD:

- DTD, XSD are building blocks for constructing xml document. Each DTD/ XSD document/ file contains bunch of rules having xml tag names, attributes names and their structure details to be used in the construction of XML file.
- We need to import DTD, XSD rules at the top of XML file in order to inform frameworks/ containers/ server that we have developed XML file according to certain DTD, XSD rules.

DTD: Document Type Definition (old)

XSD: Xml Schema Definition (latest)

#### In XSD/ xml schema: Xml schema Namespaces

- Namespace is a library that contains set of xml tags and their rules.
- Every Schema Namespace identified with its Namespace URI.

- Every Schema Namespace or bunch Namespace together will come as one XSD file.
- The Spring framework supplied multiple predefined XML schema Namespace. They must be important in spring bean configuration file by specifying their URLs/ URIs.

### Example of some Namespace:

Namespace	Namespace URI/ URL	XSD file
beans	<a href="http://www.springframework.org/schema/beans">http://www.springframework.org/schema/beans</a> <Website URL>/schema/beans	Spring-beans.xsd
c	<a href="http://www.springframework.org/schema/c">http://www.springframework.org/schema/c</a>	Spring-beans.xsd
p	<a href="http://www.springframework.org/schema/p">http://www.springframework.org/schema/p</a>	Spring-beans.xsd
context	<a href="http://www.springframework.org/schema/context">http://www.springframework.org/schema/context</a>	Spring-context.xsd

- ✓ +10 Namespaces are given in Spring frameworks.
- ✓ All these XSD file as are available Spring libraries (jar files of <Spring\_homes>\libs folder).

### Problem without XSD file:

For example, govt. to create hospital information in XML file, so every state creates their own tag and file the information and submitted to the Govt. And it is not easily for govt. to read all the information, because every state uses their own tag like below.

ts-corona.xml

```
<davakhanas>
  <davakhana>
    <name>OSG</name>
    <sankya>1000</sankya>
  </davakhana>
  <davakhana>
    <name>GH</name>
    <sankya>1000</sankya>
  </davakhana>
</davakhanas>
```

ap-corona.xml

```
<hospitals>
  <hospital>
    <name>RIMS</name>
    <size>500</size>
  </hospital>
  <hospital>
    <name>SIMS</name>
    <size>200</size>
  </hospital>
</hospitals>
```

To overcome from this problem govt., create an XSD file having some rule and guidelines, Namespace, and tags.

#### Solution with XSD:

Central govt. gives corona.xsd

```
|----> Namespace: corona  
|----> URI: http://www.nit.com/schema/corona  
|----> <hospitals>  
|----> <hospital>, <name>, <size>
```

#### Setter Injection Application internals:

- Importing DTD/ XSD rules in web.xml is optional but mandatory in Spring bean configuration file.
- The Bean Id given to Spring bean becomes object name (nothing ref variable name pointing to Spring bean class object) when Spring container instantiating given Spring Bean class object.

#### Limitations of " new" operator:

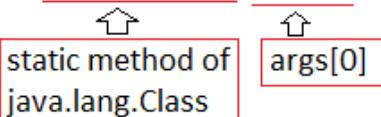
Test t - new Test();

- new operator creates the object at runtime but expects the presence of Java class from compile time onwards, i.e. it cannot be used to create the object of java class whose class name comes to application dynamically at runtime.
- To overcome this problem
  - Use newInstance() method of java.lang.Class or can use only 0-param constructor.
  - Use newInstance() method of java.lang.reflect.Constructor or can use only any constructor.
- Each object of java.lang.Class represents one class/ interface/ enum/ annotation in the application execution.
- String class object can hold string value, numeric data type variable can hold numeric values, similarly the object of java.lang.Class can hold class name/ interface name/ enum name/ annotation name coming to application dynamically at runtime.
- Each object of java.lang.reflect.Constructor class hold the details of one constructor of a given java class.

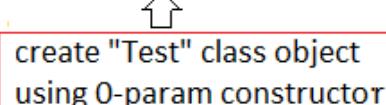
## Working with newInstance() of java.lang.Class:

### e.g.1

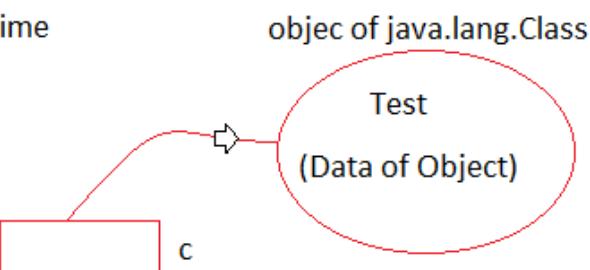
```
//Load the java class dynamically at runtime  
Class c = Class.forName("Test");
```



```
// Instantiate the class  
Object obj = c.newInstance();
```



```
//Type casting  
Test t = (Test)obj;
```

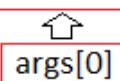


Annotation for the type casting step:

- A callout box contains the text: `Class.forName(-) method makes the JVM to load the given java class dynamically at run time and returns the object of java.lang.Class having the loaded name as the data of the object`.

### e.g.2

```
//Load the class  
Class c = Class.forName("java.util.Date");
```

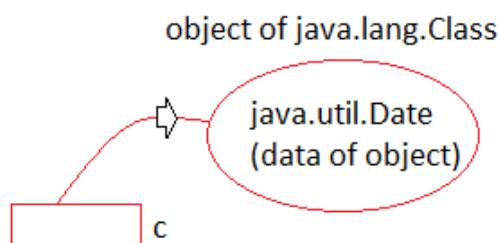


```
//instantiate the class (create the object of loaded class)  
Object obj = c.newInstance();
```

```
S.o.println(obj.toString()); or S.o.println(obj);
```

Annotation for the instantiation and printing steps:

- The code line `S.o.println(obj.toString()); or S.o.println(obj);` is shown.
- An annotation `System date and time will be printed` points to the line `S.o.println(obj);`



**Note:** Spring container and IoC container creates Spring bean class object by using `newInstance ()` of `java.lang.Class` or `java.lang.reflect.Constructor` dynamically at runtime because it is getting Spring class names dynamically at runtime from Spring bean configuration file i.e. Spring container is not using "new" operator to create any Spring bean class object.

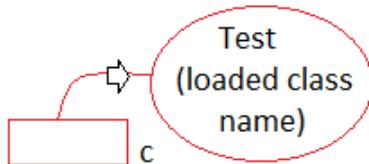
## Working with new Instance of `java.lang.reflect.Constructor` class

- Very useful if dynamic object creation should happen by parameterized constructor

Test.java

```
-----  
package com.nt.component;  
  
public class Test {  
    int a, b;  
    public Test(int a, int b) {  
        this.a=a;  
        this.b=b;  
    }  
    //toString  
    .....  
}
```

object of java.lang.Class



- 0 Constructor obj (Test())
- 1 Constructor obj (Test(10,20))

Main class

```
-----  
package com.nt.test;  
  
public class NewInstanceTest1 {  
    public static void main(String args[]) {  
        Class c1=null;  
        try{  
            //load the class  
            c1=Class.forName(args[0]);  
            //get access to all the constructor  
            //of the loaded class  
            Constructor cons[]=  
            c1.getDeclaredConstructors();  
            //instantiate the class using 2-  
            //param constructor  
            Object obj1  
            =cons[1].newInstance(10, 20);  
            So.pln(obj1)  
        } catch(Exception e ){.....}  
    }  
}
```

#### Directory Structure of IOCProj02-newInstanceMethod:

```
IOCProj02-newInstanceMethod  
  > JRE System Library [JavaSE-1.8]  
  > src  
    > com.nt.component  
      > Test.java  
    > com.nt.test  
      > NewInstanceTest1.java  
      > NewInstanceTest2.java
```

- Develop the above directory structure and package, class then use the following code with in their respective file.
- Here not required any XML file and Jar because we are developing a normal Java POC based application for using newInstance() method

## Test.java

```
package com.nt.component;

public class Test {

    private int a, b;

    static {
        System.out.println("Test.static block");
    }

    public Test() {
        System.out.println("Test.Test()");
    }

    public Test(int a, int b) {
        System.out.println("Test.Test(-,-)");
        this.a = a;
        this.b = b;
    }

    @Override
    public String toString() {
        return "Test [getClass()=" + getClass() + ", hashCode()=" +
               hashCode() + ", toString()=" + super.toString()
               + "]";
    }
}
```

## NewInstanceTest.java

```
package com.nt.test;

public class NewInstanceTest1 {

    public static void main(String[] args) {
        Class class1 = null, class2 = null;
        Object object1 = null, object2 = null;
```

```

try {
    // load the class
    class1 = Class.forName(args[0]);
    // instantiate the loaded class
    object1 = class1.newInstance();
    System.out.println(object1);
    System.out.println("-----");
    // load the class
    class2 = Class.forName(args[1]);
    // instantiate the loaded class
    object2 = class2.newInstance();
    System.out.println(object2);
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (InstantiationException e) {
    e.printStackTrace();
} catch (IllegalAccessException e) {
    e.printStackTrace();
}
}

// main

}

// class

```

### NewInstanceTest2.java

```

package com.nt.test;

public class NewInstanceTest1 {

    public static void main(String[] args) {
        Class class1 = null, class2 = null;
        Object object1 = null, object2 = null;
        Constructor cons[] = null;
        try {
            // load the class
            class1 = Class.forName(args[0]);
            // get all Constructor of the loaded class
            cons = class1.getDeclaredConstructors();
            // instantiate the loaded class
            object1 = cons[0].newInstance(10, 20);
        }
    }
}

```

```

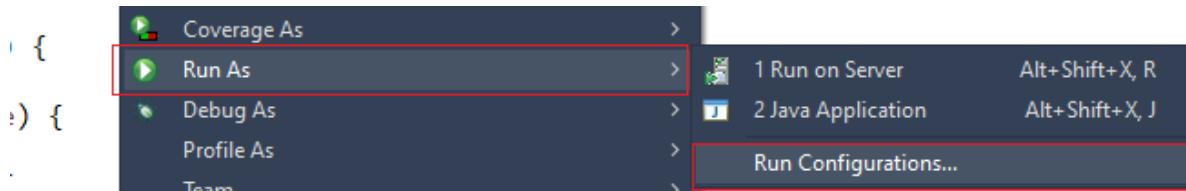
        System.out.println("-----");
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
}
// main

}// class

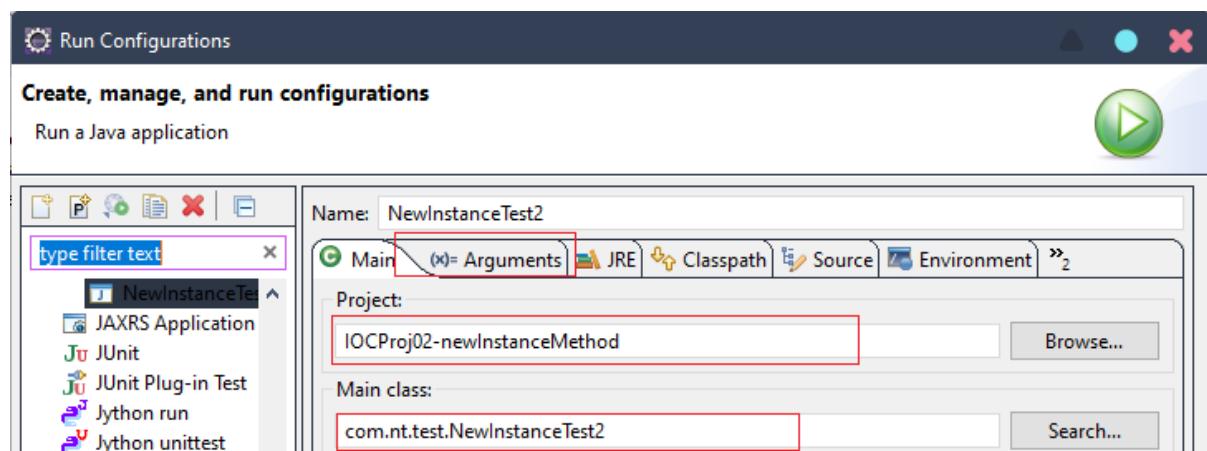
```

Here you have to pass the class name as command line argument for that follow the below steps,

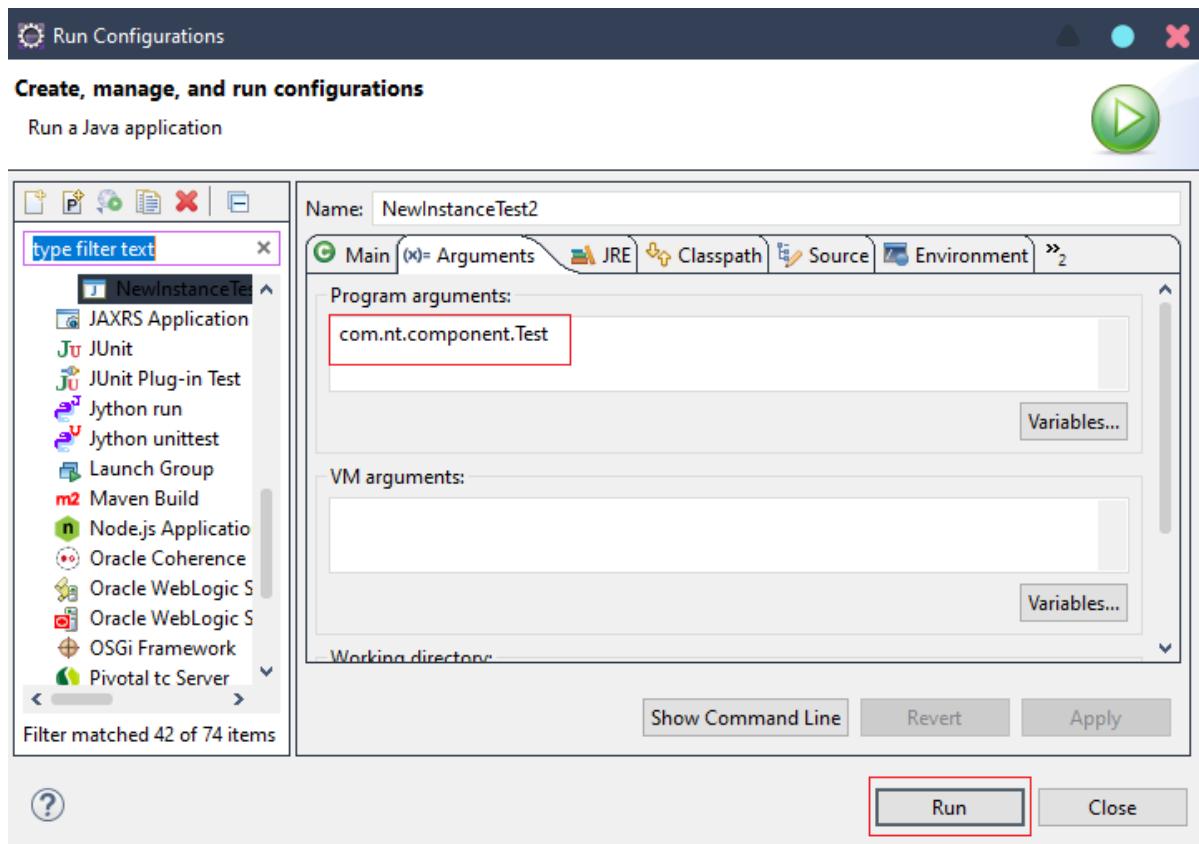
**Step #1:** Right click on your Main class/ Test class go to Run As then click on Run Configurations.



**Step #2:** Then a Run configuration wizard will open then click on the Arguments session before that check that you are pointing the right Project and right Main class.



**Step #3:** The give the class name or required argument then click on Run, then you will get the output.

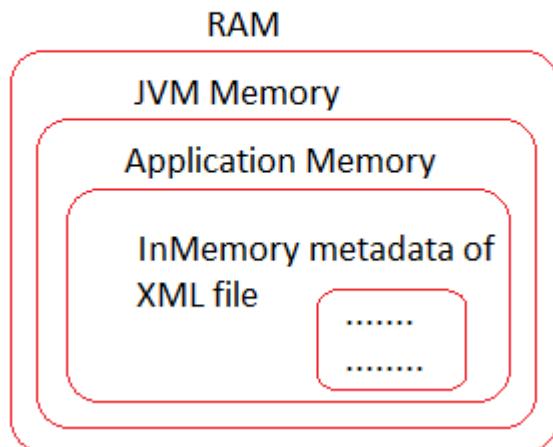


### XML parser:

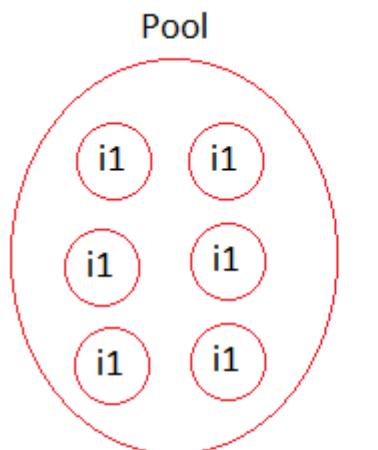
- If the XML document/ file is satisfying XML syntax rules then it is called well-formed XML document.
- If the XML document/ file is satisfying the important DTD/ XSD rules then it is called valid XML document.
- XML parser is a software program that can load given XML file, can check whether XML file is well-formed or not, valid or not. If well-formed and valid it can read and process the given XML file.  
E.g., SAX parser (Simple API XML parser), DOM parser (Document Object Model parser), DOM4J parser (DOM foe Java parser), and etc.
- The XML parser also create in memory metadata of the loaded XML file in the memory where the current application having XML parser is running that is nothing but JVM memory of the RAM, so that application can use XML file into for multiple times from in memory metadata of JVM memory itself.

**Note:** Spring/ IoC container are having built in XML parsers.

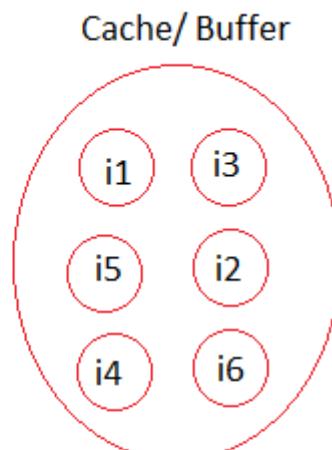
Data about data is called Metadata



### Pool vs Cache:



(Set of same items)  
[It is gives reusability of  
same items]  
[prefer List Collection]



(Set of different items)  
[It is gives reusability of  
different items]  
[prefer Map Collection to  
construct cache]

## Spring First application flow of execution

- Since our First application is standalone applications the flow of start with main (-) method and ends with main (-) method.

1. //hold name and location of spring bean configuration file

```
res = new  
FileSystemResource("src.com/nt/cfgs/applicationContext.xml");
```

2. //create BeanFactory IoC container

```
factory = new XmlBeanFactory(res);
```

3. //get Target bean class object  
obj = factory.getBean("wmg");
4. //type casting  
generator = (WishMessageGenerator)obj;
5. //invoke the method  
result=generator.generateWishMessage("Nimu");  
System.out.println("Wish Message is: "+result);
6. //End of main method

Step by step execution we are going to see.

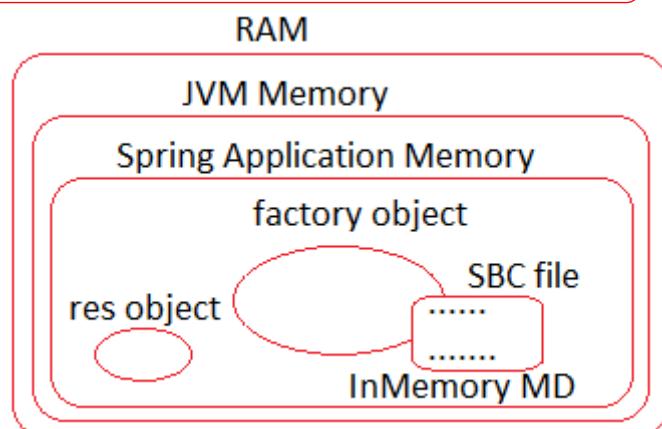
**Execution step 1 - Hold name and location of spring bean configuration file:**

```
Resource res = new
FileSystemResource("src/com/nt/cfgs/applicationContext.xml");
```

- Internally uses java.io.File class to hold the name and location of Spring bean configuration file. At this line it will not try to locate the given Spring bean configuration file.
- There are two popular implementation classes for Resources (I)
  - a. FileSystemResource (Make spring container to locate the given Spring bean file from the specified path of the File system).
  - b. ClassPathResource (Make spring container to locate the given Spring bean file from the directories and jars added to the CLASSPATH/ BUILD PATH).

**Execution step 2 - Create BeanFactory/ IoC container:**

```
BeanFactory factory = new XmlBeanFactory(res);
```



- The above line creates BeanFactory container in that process it will do
  - Takes the given Resource object (res) and locates Spring bean configuration file (applicationContext.xml) from the specified path of File system.
  - Loads the spring bean file from File system and checks whether it is well-formed or not, valid or not, if not application throws exception. If it is well-formed and valid then it creates InMemory metadata of Spring bean configuration file in the memory (JVM memory of the application) where our application is running.
  - Create XmlBeanFactory class object representing BeanFactory container and return that object back to application.

#### Execution step 3 - Get Target bean class object:

```
Object obj = factory.getBean("wmg");
```

- getBean(-) takes the given "wmg" bean id and goes to InMemory metadata of Spring bean configuration file and checks the availability of Spring bean configuration having bean id "wmg" and gets "com.nt.beans.WishMessageGenerator" as Spring bean class and also observes Setter Injection enabled on the Spring bean class (based <property> tag), so it loads and instantiates Spring bean class (target class) as shown below,

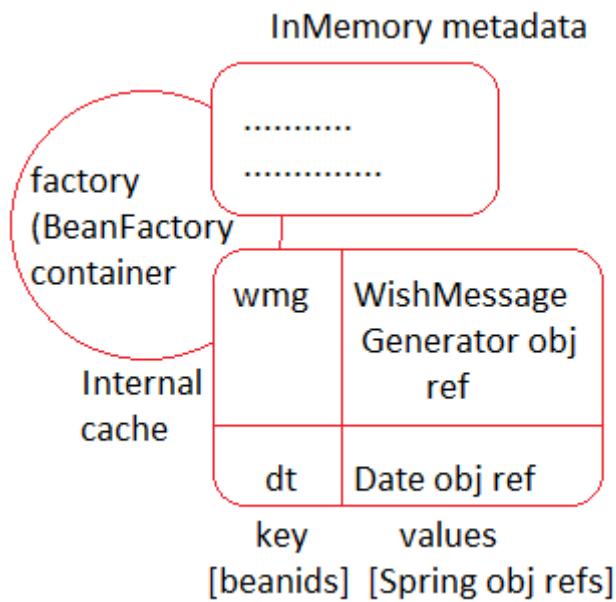
```
Class c1 = Class.forName("com.nt.beans.WishMessageGenerator");
WishMessageGenerator wmg = (WishMessageGenerator)c1.newInstance();
```

- Collects name="date", ref="dt" from <property>, checks the availability of setDate(-) method target class using Reflection API, and also checks Spring bean class configuration having bean id "dt" and gets java.util.Date as Spring bean class from the InMemory metadata of Spring bean configuration file. Since no injection is configured on java.util.Date class (dependent) the Spring container loads and creates the object as shown below,

```
Class c2 = Class.forName();
Date dt = (Date)c2.newInstance();
```

- Because of <property> the container performs Setter Injection by calling Setter method on target class object (wmg) having dependent class object (dt) as the argument.

```
wmg.setDate(dt);
```



- Keeps all the created Spring bean class objects in the internal cache of IOC container having bean id as keys and bean class object reference as values for the reusability the objects.
- getBean method returns WishMessageGenerator class object (target object back to client application and we are referring that object with Object class reference variable (obj)).

#### Execution step 4 - Typecasting:

```
WishMessageGenerator generator = (WishMessageGenerator) obj;
```

- Type casting/ down casting to call direct method (business methods) of WishMessageGenerator class.

#### Execution step 5 - Invoke the method:

```
String result = generator.generateWishMessage("Nimu");
System.out.println("Wish Message is: " + result);
```

- Invoking b.method on target class object and this b.method will use the injected dependent class object (Date object) to generate the wish message based on the current hour of the day.

#### Execution step 6 - End of the main (-) method:

- Since all objects are Local objects to main (-) method they will be vanished at the end of main (-) method, in that process IoC container

(factory) and its internal cache and InMemory metadata will also be vanished.

## Constructor Injection

- Here IoC container uses parameterized constructor to create targeted bean class object, in that process it assigns/ injects dependent class object to target class object.
- In Setter Injection first target class object will be created, next dependent class object will be created.
- In Constructor Injection first dependent class object will be created next target class object will be created using dependent class object as the parameterized constructor argument value.
- Use <property> tag for Setter Injection.
- Use <constructor-arg> for Constructor Injection, if you place <constructor-arg> tags for "n" times under <bean> tag then the IoC container uses n-param/ n-arg constructor to create Spring bean class object.

### Example of Constructor Injection:

WishMessageGenerator.java

```
package com.nt.beans;

public class WishMessageGenerator {
    private Date date;

    //For Constructor Injection
    public WishMessageGenerator (Date date) {
        this.date=date;
    }

    public String generateWishMessage (String user) {
        ...
        ...//same as previous
        ...
    }
}
```

applicationContext.xml

```
<beans .... > [Import xsd, dtd rules]
    <!-- Configure Dependent class -->
    <bean id="dt" class="java.util.Date"/>
    <!-- Configure Target class -->
    <bean id="wsg" class="com.nt.beans.WishMessageGenerator">
        <constructor-arg name="date" ref="dt"/>
    </bean>
</beans>
```

<constructor-arg> tag instructs to perform Constructor Injection.

ConstructorInjectTest.java

```
Same as previous
```

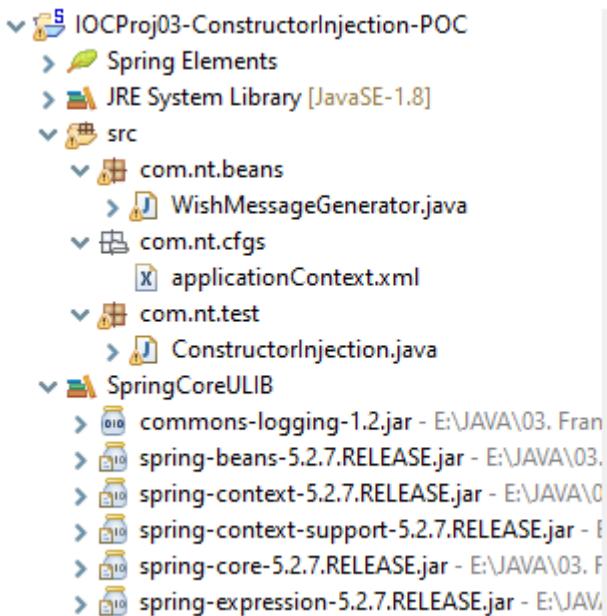
**Note:** While configure spring bean, if we enable only Setter Injection then IoC container uses 0-param constructor to create Spring bean class object, if enable Constructor Injection then IoC container uses parameterized constructor for spring bean class object creation.

**Q. What happens if we place wrong bean id in spring program?**

**Ans.** The IoC container throws Exception like below

[org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'xxx' available.](#)

## Directory Structure of IOCProj03-ConstructorInjection-POC:



- Develop the above directory structure and package, class, XML file and add the jar also, then use the following code with in their respective file.

### WishMessageGenerator.java

```
package com.nt.beans;

import java.util.Date;

public class WishMessageGenerator {

    //bean property
    private Date date;

    static {
        System.out.println("WishMessageGenerator : static block");
    }

    public WishMessageGenerator(Date date) {
        System.out.println("WishMessageGenerator : 1 param
constructor");
        this.date = date;
    }

    public void setDate(Date date) {
```

```

        this.date = date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public String generateWishMessage(String user) {
        System.out.println("WishMessageGenerator:
generateWishMessage() date : "+date);
        int hour=0;
        //get current hour of the dat
        hour=date.getHours();
        //generate wish message (Business logic)
        if (hour<12)
            return "Good Morning : "+user;
        else if (hour<16)
            return "Good Afternoon : "+user;
        else if (hour<20)
            return "Good Evening : "+user;
        else
            return "Good Night : "+user;
    }
}

```

### 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">

    <!-- Dependent bean configuration -->
    <bean id="dt" class="java.util.Date" />

    <!-- Dependent bean configuration with our date -->
    <bean id="dt1" class="java.util.Date">
        <property name="year" value="99" />
        <property name="month" value="04" />

```

```

        <property name="date" value="05" />
    </bean>

    <!-- Target bean configuration -->
    <bean id="wmg" class="com.nt.beans.WishMessageGenerator">
        <property name="date" ref="dt1" />
        <constructor-arg name="date" ref="dt" />
    </bean>

</beans>
```

### ConstructorInjectionTest.java

```

package com.nt.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;

import com.nt.beans.WishMessageGenerator;

public class ConstructorInjection {

    public static void main(String[] args) {
        Resource res = null;
        BeanFactory factory = null;
        WishMessageGenerator generator = null;
        Object obj = null;
        String result = null;

        // hold name and location of spring bean configuration file
        res = new
        FileSystemResource("src/com/nt/cfgs/applicationContext.xml");
        // create BeanFactory/ IOC container
        factory = new XmlBeanFactory(res);
        // get Target bean class object
        obj = factory.getBean("wmg");
        // typecasting
        generator = (WishMessageGenerator) obj;
        ...
    }
}
```

```

    // invoke the method
    result = generator.generateWishMessage("Nimu");
    System.out.println("Wish Message is: " + result);

}

}

```

In Constructor Injection enabled application:

```

WishMessageGenerator generator = (WishMessageGenerator)
                                factory.getBean("wmg");

```

- Takes the bean id "wmg", searches internal cache for bean class object (not available), then goes to InMemory metadata of Spring bean configuration file and search for bean class configuration whose bean id is "wsg" and gets "com.nt.beans.WishMessageGenerator" as Spring bean class but notices the Constructor Injection is enabled by seeing <constructor-arg> tag under <bean> tag.
- Goes to name="date", ref="dt" of <constructor-arg> tag, then search in internal cache for bean class object using bean id "dt" (not available), then goes to InMemory metadata of Spring bean configuration file gets "java.util.Date" class having bean id. Since no injection is enabled. IoC container loads and creates the object for java.util.Date having "dt" as object name (ref variable name).

```

Class cl = Class.forName("java.util.Date");
Date dt = (Date)cl.newInstance();

```

- IoC container get access to 1-param constructor of Target class and uses that constructor to create target class object (WishMessageGenerator) having dependent class object (Date) as the argument value of constructor.

```

Class c2=Class.forName("com.nt.beans.WishMessageGenerator");
Constructor cons[] = c2.getDeclaredConstructors();
WishMessageGenerator wmg = (WishMessageGenerator)
                           cons[0].newInstance();

```

- IOC container keeps both target and dependent class object in the internal cache.
- Returns WishMessageGenerator class object reference (target) back to client application.

#### **ref and value attribute:**

- <ref>, "ref" attribute to configure bean id-based spring bean class object injection to target bean class properties.
- <value>, "value" attribute to configure simple values injection to target bean class properties.  
e.g. java.util.Date
  - |---> year (int)
  - |---> month (int)
  - |---> date (int)
  - |---> hours (int)
  - |---> minutes(int)
  - |---> seconds (int)

**Q. What happens if we enable both Setter Injection and Constructor Injection on the bean property? Tell me whose values will be injected as final values?**

**Ans.** Since setter method always execute after constructor execution, we can say Setter Injection overrides the values injected by the Constructor Injection i.e., the values injected by Setter Injection will become final value.

#### **Note:**

- ✓ Bean id should be unique with in an IoC container.
- ✓ We can configure two beans having same class names but we should take different bean ids.

#### **Name Attribute:**

- In the property tag "name" value is not bean property name, it is xxx/Xxx word of setXxx(-) method.
- In the <constructor-arg> tag, "name" value is not bean property name, it is parameter name of parameterized constructor.

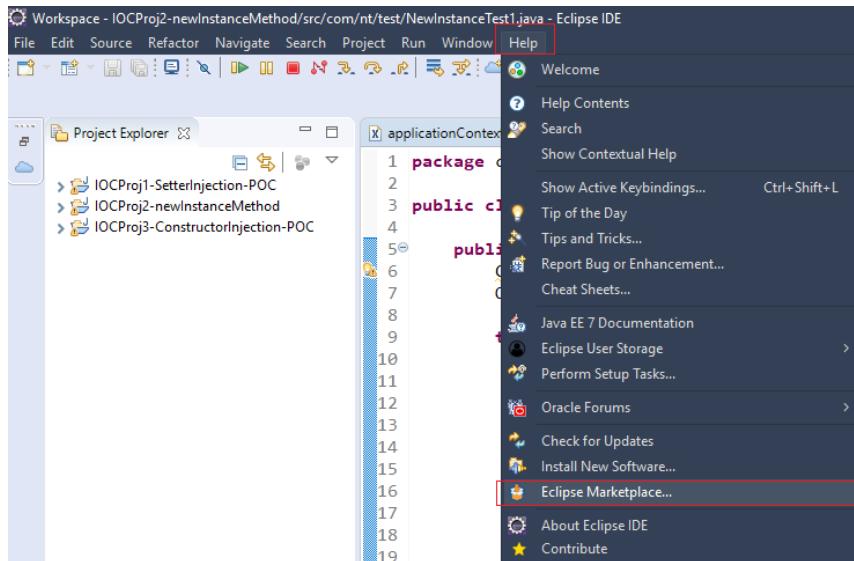
#### **Eclipse plugin - STS [Spring Tool Suite]:**

- Plugin is a patch software, that provides additional functionalities to existing software.
- STS plugin added to eclipse makes Spring application development very easy in Eclipse IDE.

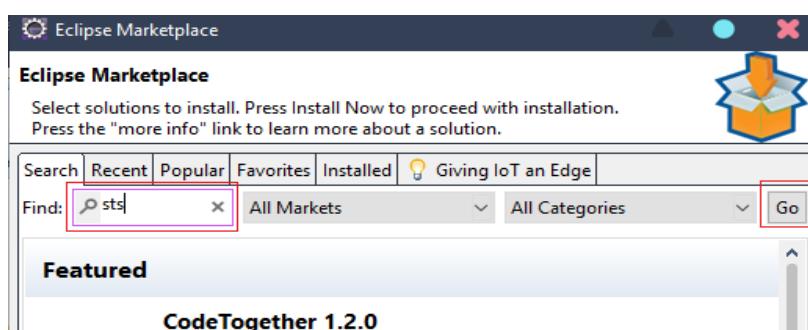
- To install third party supplied plugins use "Eclipse Market Place" of help menu.
- To install Eclipse supplied plugins use "Install new software" of help menu

### To install STS plugin:

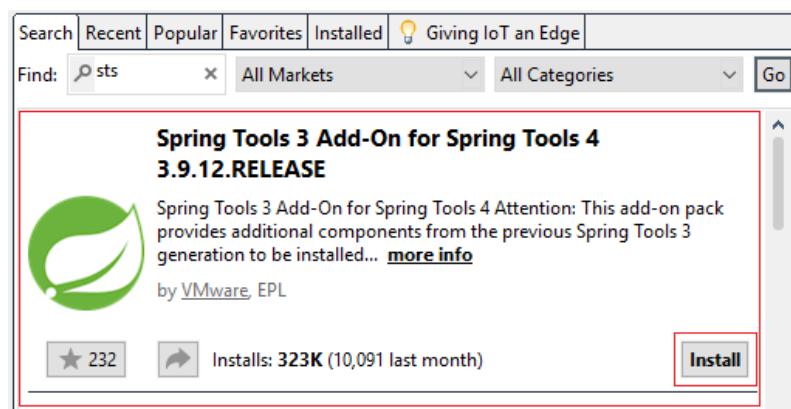
**Step #1:** Click on Help menu choose Eclipse Marketplace



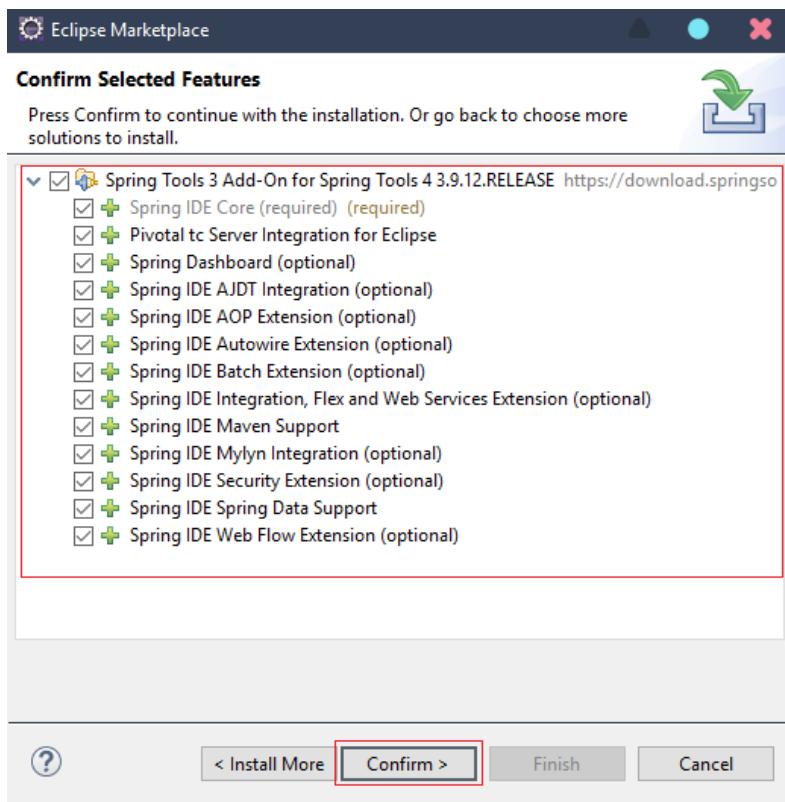
**Step #2:** Type STS, click on Go for Search



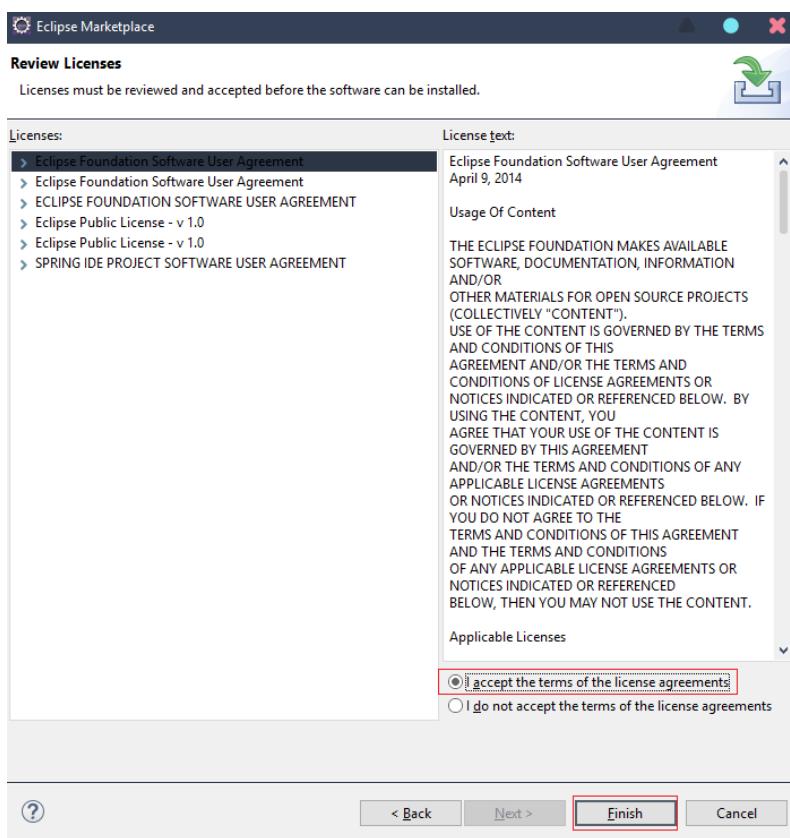
**Step #3:** Choose the First one, second one will be installed automatically, then click on Install



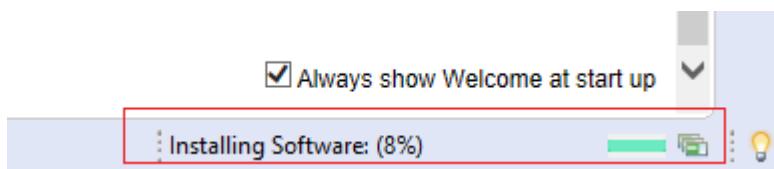
**Step #4:** Make sure all are chosen, then click on Confirm, wait few second



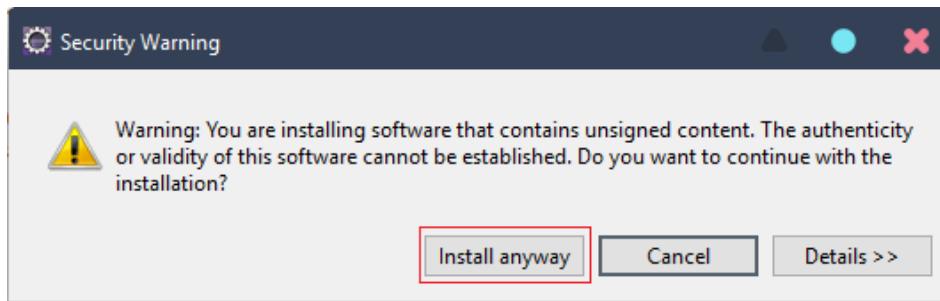
**Step #5:** Choose "I accept the terms of the license agreements", then click on Finish



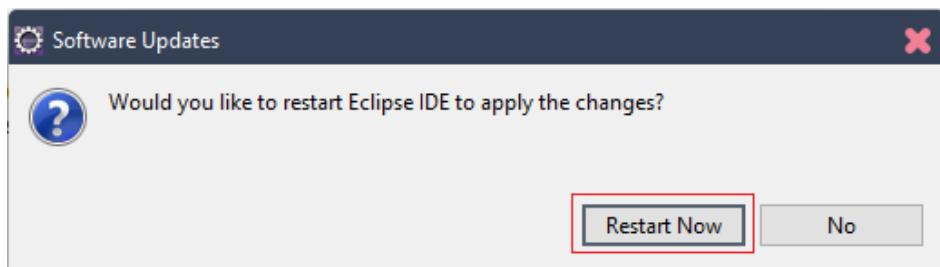
**Step #6:** installation process will start and wait for some time



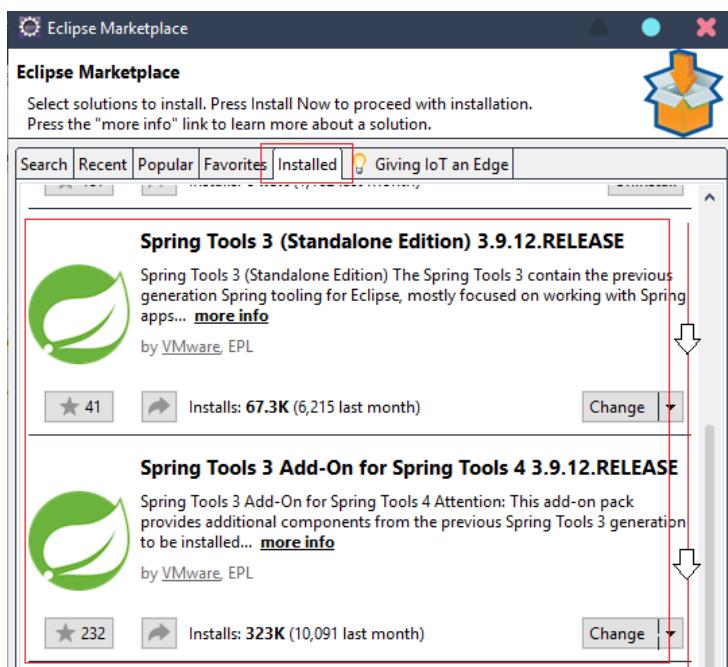
**Step #7:** In between installation you will get Security Warning click on "Install anyway"



**Step #8:** Click on "Restart Now"



**Step #9:** For verification go to Eclipse Marketplace, click installed scroll down you will see all the installed software



**Q. What is the difference between FileSystemResource and ClassPathResource**

**Ans. Note:** Both are the implementation classes of  
org.springframework.core.io.Resource;

- + FileSystemResource object given to BeanFactory container makes the BeanFactory container to locate the given Spring bean configuration file from specified of path File system, we can give either relative path (recommend) or absolute path of Spring bean configuration file.

```
Resource res = new  
FileSystemResource("src/com/nt/cfgs/applicationContext.xml");
```

- ClassPathResource class object given to BeanFactory container makes the BeanFactory container to search and locate Spring bean configuration file from the directory and jar file that are added to CLASSPATH/ BUILD PATH.

**Note:** In every eclipse project 'src' folder is in CLASSPATH/ BUILD PATH by default.

```
Resource res = new  
ClassPathResource("com/nt/cfgs/applicationContext.xml");
```

- we can add any location of the project to CLASSPATH/ BUILD PATH if we add "com/nt/cfgs" folder of project to BUILD PATH then we can write like this,

```
Resource res = new  
ClassPathResource("applicationContext.xml");
```

- To add "com/nt/cfgs" folder to buildpath ---> right click on project ---> buildpath ---> configure buildpath ---> libraries tab ---> add class folder --> select project and package/ folder.

**java.lang.Class:**

- + The object of java.lang.Class can hold class name/ interface name/ annotation name/ enum name in a running java application.
- + We can't create for java.lang.Class using "new" operator because it have only one 0- param private constructor.

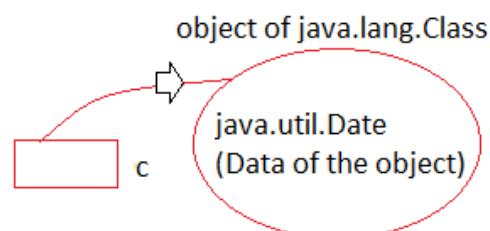
Class c = new Class (); X

## Different ways of creating object for java.lang.Class:

### Approach 1:

Using Class.forName(-) method

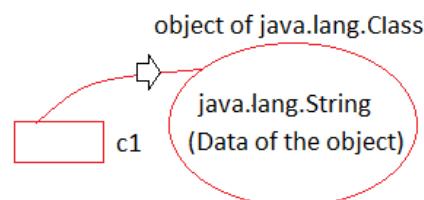
```
Class c = Class.forName("java.util.Date");
```



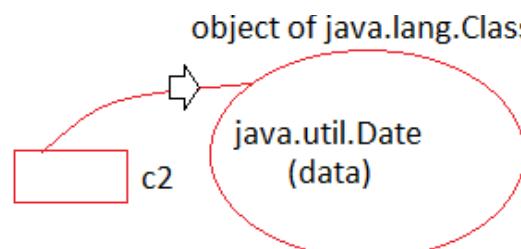
### Approach 2:

Using getClass() method of java.lang.Object class

```
String s = new String ("ok");
Class c1 = s.getClass();
Date d = new Date();
Class c2 = d.getClass();
...
```



String s1="java.util.Date";  
using s1 we can not instantiate "Date" class because "s1" holds "Date" class as String value.  
Using "c", "c2" we can get any info about "Date" class like methods, constructors, fields and etc. more over we can instatntiate Date class also.

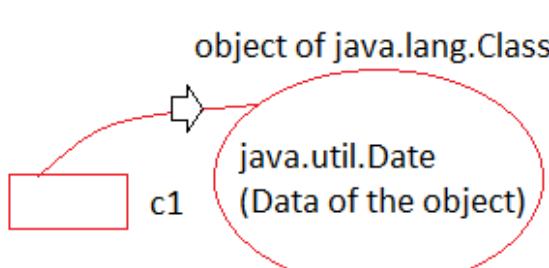
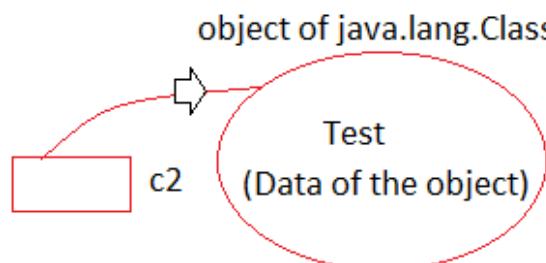


### Approach 3:

Using "class" property of java.

- Two built-in/ implicit properties of java language is "class", "length".
  - Two implicit reference variable in java are "this", "super".
  - Two implicit threads in java are "main", "gc".
- e.g. Class c1 = java.util.Date.class  
e.g. Class c2 = Test.class

"class" ---> built-in property  
"class" ---> keyword  
"java.lang.Class" ---> predefined class.  
"class" ---> OOPs terminology



**Compiler added/ generated code in our Java class:**

- java.lang.Object as super class if there is no super class for our class.
- public 0-param constructor, if there are no constructor in our class
- public static property "class" of type java.lang.Class and etc.

**Two overloaded signature of getBean(-) method in BeanFactory (I) :**

- a. public Object getBean(String beanId)

- Here Type casting is required, i.e. code is not type safe, possibility of getting java.lang.ClassCastException.

e.g.

```
WishMessageGenerator generator = (WishMessageGenerator)  
factory.getBean("wmg");
```

- b. public <T> T getBean(String beanId, Class<T> requiredType)

[recommend]

- Here type casting not required, i.e., our code becomes type safe code.

e.g.

1. Class c = WishMessageGenerator.class;  
WishMessageGenerator generator = factory.getBean("wmg",  
c);
2. WishMessageGenerator generator = factory.getBean("wmg",  
WishMessageGenerator.class);

**Generics:**

- Generics are given from java 1.5v.
- They avoid type casting in code, they make our code as type safe code.
- Syntax <> (template/ generic)

T: Type

K: Key

V: Value

N: Node

E: Element

**Q. While working with (b) signature of getBean (-,-) if we give wrong class name as the 2nd argument then what happens?**

**Ans.** We will get the following exception

[org.springframework.beans.factory.BeanNotOfRequiredTypeException](http://org.springframework.beans.factory.BeanNotOfRequiredTypeException): Bean named 'xxx' is expected to be of type 'AAA' but was actually of type 'BBB'.

## Design Pattern

**Def1:** Design patterns is set of rules that are given as best solutions for reoccurring problems of application development.

**Def2:** Design patterns are best practices to use software languages, technologies and frameworks effectively in application development.

### Two types of patterns in Java:

- a. GOF Design Patterns [Core patterns]
  - Can be implemented any OOP languages.  
E.g., Singleton, Factory, Abstract Factory, Strategy and etc. 23 patterns
- b. JEE Patterns [Given by Sun MS]
  - Can be implemented only in Java.  
E.g., DAO, FrontController View, Helper Composite View and etc.

## Factory Design Pattern

 It returns one of the several related class objects based on data we supply.

**Problem:** Different objects will be created using different techniques, making all developers knowing the creation process objects makes developer work bit heavy.

**Solution:** Create a factory and provide abstraction on object creation process and return the object based on the data that is supplied (nothing hides object creation process from client).

e.g.

```
Connection con = DriverManager.getConnection(,-,-,-);
```



It is returning one of the several implementation class objects of `java.sql.Connection(I)` based on the data (url, username, pwd) details we passed i.e. returns JDBC von object by creating necessary dependent objects like socket objects and uses then in the creation of JDBC con object.

**Note:** To keep multiple classes as related classes make them implementing common interface or extending from common super class.

```
Object obj = factory.getBean("beanId");
```

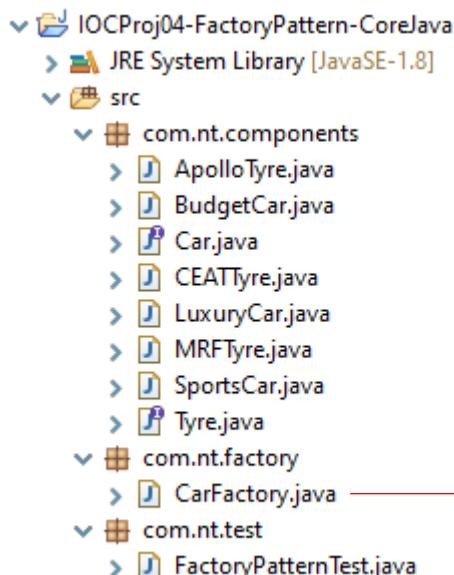


It will create and return Spring bean class objects using different techniques and also assigns depending objects based on the bean id that we pass as input value.

#### Important points on method return types:

- a. If java method return type is an interface, then it returns one of its implementation's classes object as the return value.
  - b. If java method return type is an abstract class, then it returns one of its sub classes objects as the return value.
  - c. If java method return type is a concrete class, then it returns either that concrete class object or one of its sub class objects as the return value.
- 
- ✓ BeanFactory container means it factory to create and manage spring bean class objects.
  - ✓ Spring/ IoC container is designed based on Flyweight Design Pattern which internally also uses Factory Pattern.

#### Directory Structure of IOCProj04-FactoryPattern-CoreJava:



Having one factory method,

This method that creates/ gathers and returns object is called factory method

- Develop the above directory structure and package, class, then use the following code with in their respective file.
- No jar is required here because it is a simple Java POC application.

### Tyre.java

```
package com.nt.components;

public interface Tyre {
    public String roadGrip();
}
```

### ApolloTyre.java

```
package com.nt.components;

public class ApolloTyre implements Tyre {

    public ApolloTyre() {
        System.out.println("ApolloTyre : ApolloTyre()");
    }
    @Override
    public String roadGrip() {
        return "Apollo Tyre : Standard Road Grip - Suitable for
Transportation";
    }

}
```

### CEATTyre.java

```
package com.nt.components;

public class CEATTyre implements Tyre {

    public CEATTyre() {
        System.out.println("CEATTyre : CEATTyre()");
    }
    @Override
    public String roadGrip() {
        return "CEAT Tyre : Smooth Road Grip - Suitable for Luxury and
comfort";
    }

}
```

### MRFTyre.java

```
package com.nt.components;

public class MRFTyre implements Tyre {

    public MRFTyre() {
        System.out.println("MRFTyre : MRFTyre()");
    }
    @Override
    public String roadGrip() {
        return "MRF Tyre : Super Road Grip - Suitable for Sports";
    }

}
```

### Car.java

```
package com.nt.components;

public interface Car {
    public void drive();
}
```

### BudgetCar.java

```
package com.nt.components;

public class BudgetCar implements Car {

    private Tyre tyre;
    public BudgetCar(Tyre tyre) {
        System.out.println("BudgetCar : BudgetCar(-)");
        this.tyre = tyre;
    }
    @Override
    public void drive() {
        System.out.println("Driving Budget car having
"+tyre.roadGrip());
    }
}
```

### LuxuryCar.java

```
package com.nt.components;

public class LuxuryCar implements Car {

    private Tyre tyre;
    public LuxuryCar(Tyre tyre) {
        System.out.println("LuxuryCar : LuxuryCar(-)");
        this.tyre = tyre;
    }

    @Override
    public void drive() {
        System.out.println("Driving Luxury car having
"+tyre.roadGrip());
    }

}
```

### SportsCar.java

```
package com.nt.components;

public class SportsCar implements Car {

    private Tyre tyre;

    public SportsCar(Tyre tyre) {
        System.out.println("SportsCar : SportsCar(-)");
        this.tyre = tyre;
    }

    @Override
    public void drive() {
        System.out.println("Driving Sports car having
"+tyre.roadGrip());
    }

}
```

## CarFactory.java

```
package com.nt.factory;

import com.nt.components.ApolloTyre;
import com.nt.components.BudgetCar;
import com.nt.components.CEATTyre;
import com.nt.components.Car;
import com.nt.components.LuxuryCar;
import com.nt.components.MRFTyre;
import com.nt.components.SportsCar;
import com.nt.components.Tyre;

public class CarFactory {

    // Factory method
    public static Car getInstance(String type) {
        Tyre tyre = null;
        Car car = null;
        if (type.equalsIgnoreCase("luxury")) {
            tyre = new CEATTyre();
            car = new LuxuryCar(tyre);
        } else if (type.equalsIgnoreCase("sports")) {
            tyre = new MRFTyre();
            car = new SportsCar(tyre);
        } else if (type.equalsIgnoreCase("budget")) {
            tyre = new ApolloTyre();
            car = new BudgetCar(tyre);
        } else {
            throw new IllegalArgumentException("Invalid car type
you are providing");
        }
        return car;
    } // method

} // class
```

### FactoryPatternTest.java

```
package com.nt.test;

import com.nt.components.Car;
import com.nt.factory.CarFactory;

public class FactoryPatternTest {

    public static void main(String[] args) {
        Car car = null;
        car = CarFactory.getInstance("luxury");
        car.drive();
        System.out.println("-----");
        car = CarFactory.getInstance("budget");
        car.drive();
    }

}
```

## Strategy Design Pattern

- ⊕ It is not Spring design pattern
- ⊕ It is GOF design pattern and can be implemented in any OOP language including java.
- ⊕ It has become popular because of Spring framework. It is given to provide set of guidelines while keeping classes in dependency. Since Spring also supports dependency management. It is recommended to design target and dependent classes according to this design pattern.
- ⊕ This design pattern says, I allow the class of Dependency Management (target and dependent classes) as loosely coupled interchangeable parts.

### Tight Coupled and Loose Coupled:

- If the degree of dependency is less between two components, then they are called loosely coupled.
- If the degree of dependency is more between two components, then they are called tightly coupled.

### To implement Strategy Design Pattern:

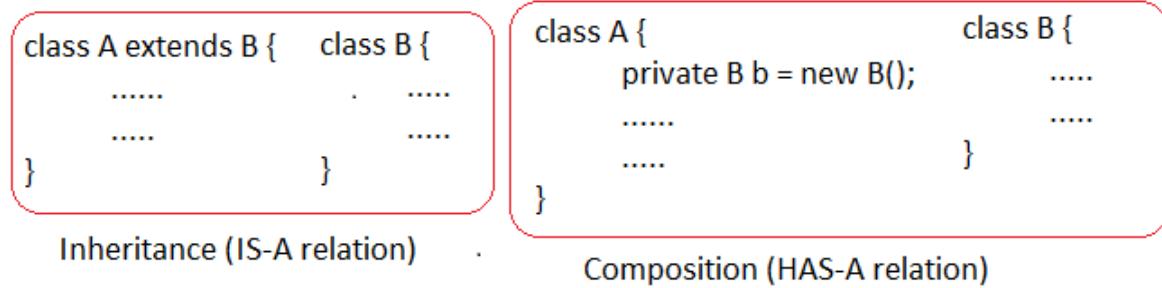
We need to implement 3 principles/ rules,

- a. Prefer composition over inheritance.

- b. Always code to interface or abstract classes i.e., never code to concrete classes/ implementation classes/ sub classes.
- c. Our code must be open for extension and should be close for modification.

### Strategy Design Pattern Principle - 1:

- Prefer composition over inheritance.



- If one class wants to use entire state and behaviour of another class then design those classes having inheritance.
- If one class wants to use limit/ some state and behaviour another class then design those classes having composition.

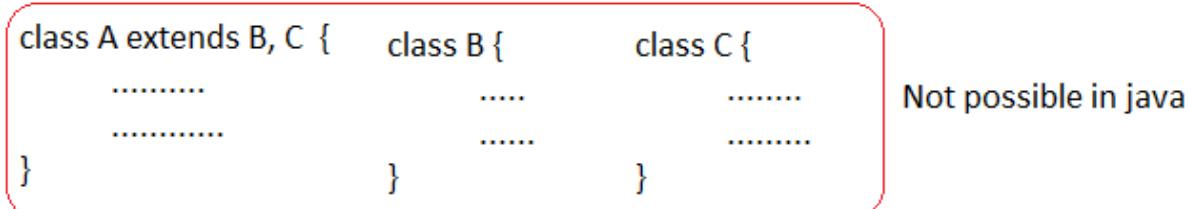
### Limitations of inheritance:

- Few OOP language like java does not support multiple inheritance (i.e., we cannot access the state and behaviour from multiple classes at a time).
- Code becomes fragile (easily breakable).
- Testing of code is bit complex.

### Inheritance limitation - 1: Problem and Solution:

- Few OOP language like java does not support multiple inheritance (i.e., we cannot access the state and behaviour from multiple classes at a time).

### Problem:



- Don't look at interfaces in java in the angle of inheritance because there is nothing to inherit from interfaces. Look at interfaces in the angle of

polymorphism and loose coupling.

### Solution:

```
class A {  
    private B b = new B();  
    private C c = new C();  
    .....  
}  
  
class B {  
    .....  
}  
  
class C {  
    .....  
}
```

- Using composition, we can make one class using the state and behaviour of multiple classes.

### Inheritance limitation - 2: Problem and Solution:

- Code becomes fragile (easily breakable).

### Problem:

```
class X {  
    float  
    public int m1() {  
        .....  
        returns 100;  
    }  
}  
  
class Y extends X {  
    public int m1() {  
        .....  
        returns 1000;  
    }  
}  
  
class Z extends Y {  
    .....  
}  
  
class Z1 extends Y {  
    .....  
}
```

- If we change the signature of m1() method in X class then all the classes of the inheritance hierarchy will be disturbed.

### Solution:

```
class X {  
    float  
    public int m1() {  
        .....  
        returns 100;  
    }  
}  
  
class Y {  
    private X x = new X();  
    public int m1() {  
        .....  
        int val = x.m1();  
        Math.round(x.m1());  
        returns val+100;  
    }  
}
```

### Note:

- ✓ Here when m1() method signature is changed (int to float) then it disturbs only one line of Y class no other classes that ate inheriting or

- using Y class, that means this indicates code is not easily breakable.
- ✓ If we modify the signature of any method i.e. there in java.lang.Object class, that will disturb the entire classes of the inheritance hierarchy.

### Inheritance limitation - 3: Problem and Solution:

- Testing of code is bit complex.

#### Problems:

```

class X {
    public int m1() {
        ....
        returns ...;
    }
    public int m2() {
        ....
        return ...;
    }
}

class Y extends X {
    public int m3() {
        ....
        returns ...;
    }
}

class Z extends Y {
    public int m4() {
        ....
        return ...;
    }
}

```

Client app

---

```

Z z = new Z();
z.m4(); //valid
z.m1(); //valid
z.m2(); //valid
z.m3(); //valid

```

**Note:** while test Z class we need to test Z class direct method (m4) and inherited method (m1, m2, m3) this improves burden on the programmer.

#### Solution:

```

class X {
    public int m1() {
        ....
        returns ...;
    }
    public int m2() {
        ....
        return ...;
    }
}

class Y extends X {
    public int m3() {
        ....
        returns ...;
    }
}

class Z {
    private X x = new X();
    private Y y = new Y();
    public int m4() {
        int val1 = x.m1();
        int val2 = x.m2();
        int val3 = x.m3();
        ....
        return ...;
    }
}

```

Client App

---

```

Z z = new Z();
z.m4(); //valid
z.m1(); //invalid
z.m2(); //invalid
z.m3(); //invalid

```

- Here we can only m4() method on "Z" class object and we cannot call m1(), m2(), m3() methods on the same object, because composition. So, we need to unit testing only on "m4()" method. In that process unit testing of m1(), m2(), m3() methods also completed indirectly.

### Unit testing:

- Programmer's testing on his own piece of code is called Unit testing.
- Testing is all about matching expected results with actual results.
- If match Test results are positive.
- If not, matched Test results are negative.

### Strategy Design Pattern Principle - 2:

- Always code to interface or abstract classes i.e., never code to concrete classes/ implementation classes/ sub classes.

### Problem:

Target class :

```
-----  
class Flipkart {  
    private DTDC dtdc = new DTDC();  
    public String shopping (float[] items,  
                           float[] prices) {  
        ....  
        ....  
        dtdc.deliver();  
        return .....  
    }  
}
```

```
//Dependent class 1  
public class DTDC {  
    public void deliver () {  
        ....  
        ....  
    }  
  
//Dependent class 2  
public class BlueDart {  
    public void deliver () {  
        ....  
        ....  
    }  
}
```

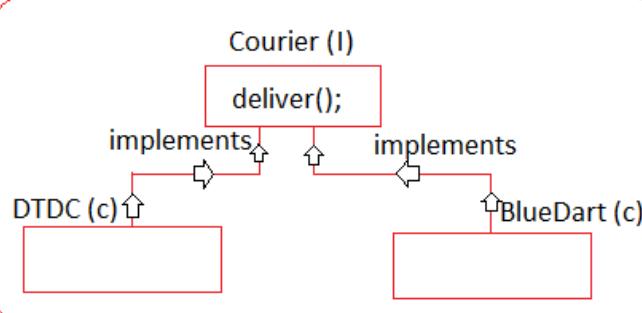
- Here Flipkart and DTDC/ BlueDart (target class, dependent class) are in tight coupling,
  - If Flipkart want to use BlueDart services instead of DTDC services then we need to modify the code of Flipkart class (tight coupling).
  - If the method names are DTDC/ BlueDart (dependent class) are changed (like from deliver () to supply ()) then we need to modify the code of Target class (tight coupling).

## Solution:

- Make all dependent classes implementing common interface having common methods declaration and use that common interface reference variable in target class while creating HAS-A relation.

Code.

```
-----  
interface Courier {  
    public String  
    delivery(int orderId);  
}
```



```
//Dependent class1 //Dependent class2  
public class DTDC implements Courier { public class BlueDart implements Courier {  
    public String delivery (int orderId) { public String delivery (int orderId) {  
        ...  
        ...  
        .....//delivery logic  
        ...  
        return status; }  
    } }  
    ...  
    .....//delivery logic  
    ...  
    return status;  
}
```

Target class

```
-----  
public class Flipkart {  
    private Courier courier = new DTDC(); [Bad practice]  
    private Courier courier;  
  
    public void setCourier(Courier courier) {  
        this.courier = courier;  
    }  
  
    public String shopping(float[] items, float[] prices) {  
        ....  
        .....//bill cal logic  
        ....  
        .....//logic to generate order id  
        String status = courier.deliver(orderId)  
        return billAmount+status  
    }  
}
```

Here DTDC, BlueDart classes can not change **deliver(-)** method name because they are getting it from the common interface called **Courier**.

Client App

```
-----  
Flipkart fpkt = new Flipkart();  
Courier cr1 = new DTDC();  
fpkt.setCourier(cr1);  
Courier cr2 = new BlueBart();  
fpkt.setCourier(cr2);
```

**Note:** Here we can change Dependent of the target without touching the source code of target class (Loose coupling).

### Strategy Design Pattern Principle - 3:

- Our code must be open for extension and should be close for modification.
- Principle 2/ rule 2 Implementation indirectly makes our code open for extension, because we can add more implementation classes for Courier (I) to keep more possible dependents ready.  
e.g., By adding more Dependent classes to Courier (I) like DTDC, FastFlight and etc. classes we can keep more Dependent class ready to work with target class (Flipkart).

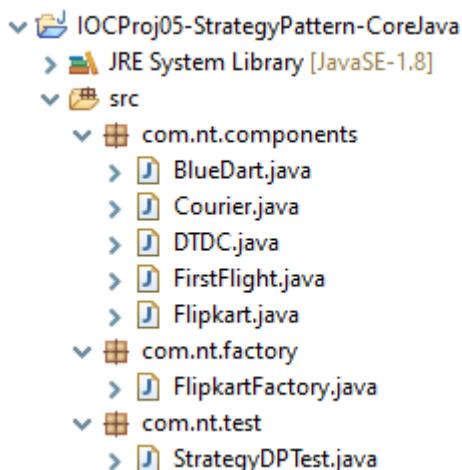
```
public final class DHL implements Courier {  
    public String deliver(int orderId) {  
        .....  
        .....  
        return status;  
    }  
}
```

//code is open for extension  
closed for modification

- By making both target and dependent classes as final classes or by taking their methods as final methods. We can make our code closed for modification, because we can't sub class the final class and we can't override the final methods in the sub classes.

**Note:** In the implementation of Strategy design pattern we also use Factory DP to provide abstraction on dependent, target class object creation and to assign dependent class object to target.

### Directory Structure of IOCProj05-StrategyPattern-CoreJava:



- Develop the above directory structure and package, class, XML file and add the jar also, then use the following code with in their respective file.

#### Courier.java

```
package com.nt.components;

public interface Courier {

    public String deliver(int orderId);

}
```

#### BlueDart.java

```
package com.nt.components;

public final class BlueDart implements Courier {

    public BlueDart() {
        System.out.println("BlueDart : BlueDart()");
    }
    @Override
    public String deliver(int orderId) {
        return "BlueDart courier will deliver Order Id : "+orderId+
order products";
    }

}
```

#### DTDC.java

```
package com.nt.components;

public final class DTDC implements Courier {

    public DTDC() {
        System.out.println("DTDC : DTDC()");
    }
    @Override
    public String deliver(int orderId) {
```

```
        return "DTDC courier will deliver Order Id : "+orderId+" order
products";
    }

}
```

### FirstFlight.java

```
package com.nt.components;

public final class FirstFlight implements Courier {

    public FirstFlight() {
        System.out.println("FirstFlight : BlueDart()");
    }
    @Override
    public String deliver(int orderId) {
        return "FirstFlight courier will deliver Order Id : "+orderId+
order products";
    }

}
```

### Flipkart.java

```
package com.nt.components;

import java.util.Arrays;
import java.util.Random;

public final class Flipkart {

    //property
    private Courier courier;

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

    public void setCourier(Courier courier) {
        System.out.println("Flipkart : setCourier()");
    }
}
```

```

        this.courier = courier;
    }

public String shopping(String[] items, float[] prices) {
    System.out.println("Flipkart : shopping()");
    float billAmount = 0.0f;
    int orderId = 0;
    String msg = null;
    //calculate bill amount
    for (float p : prices)
        //billAmount = bilAmount+p;
        billAmount+=p;

    //Generate order id dynamically as random number
    orderId = new Random().nextInt(10000);
    //use courier service for delivering the products
    msg = courier.deliver(orderId);

    return Arrays.toString(items)+" are purchased having prices
"+Arrays.toString(prices)+"\nwill bill amount "+billAmount+" .... \n"+msg;
}

}

```

### FlipkartFactory.java

```

package com.nt.factory;

import com.nt.components.BlueDart;
import com.nt.components.Courier;
import com.nt.components.DTDC;
import com.nt.components.FirstFlight;
import com.nt.components.Flipkart;

public class FlipkartFactory {

    public static Flipkart getInstane(String courierName) {
        Courier courier=null;
        Flipkart fpkt=null;
        if (courierName.equalsIgnoreCase("dtdc"))

```

```

        courier = new DTDC();
    else if (courierName.equalsIgnoreCase("blueDart"))
        courier = new BlueDart();
    else if (courierName.equalsIgnoreCase("firstFlight"))
        courier = new FirstFlight();
    else
        throw new IllegalArgumentException("Invalid courier
name");
    //create target class object
    fpkt=new Flipkart();
    //assign dependent class object to target class object
    fpkt.setCourier(courier);
    return fpkt;
}

}

```

### StrategyDPTest.java

```

package com.nt.test;

import com.nt.components.Flipkart;
import com.nt.factory.FlipkartFactory;

public class StrategyDPTest {

    public static void main(String[] args) {
        Flipkart fpkt = null;
        fpkt=FlipkartFactory.getInstance("dtdc");
        System.out.println(fpkt.shopping(new String[] {"rain coat",
"umbrella", "Flu tablets"},
                new float[] {5000, 2000, 500})
        );
    }
}

```

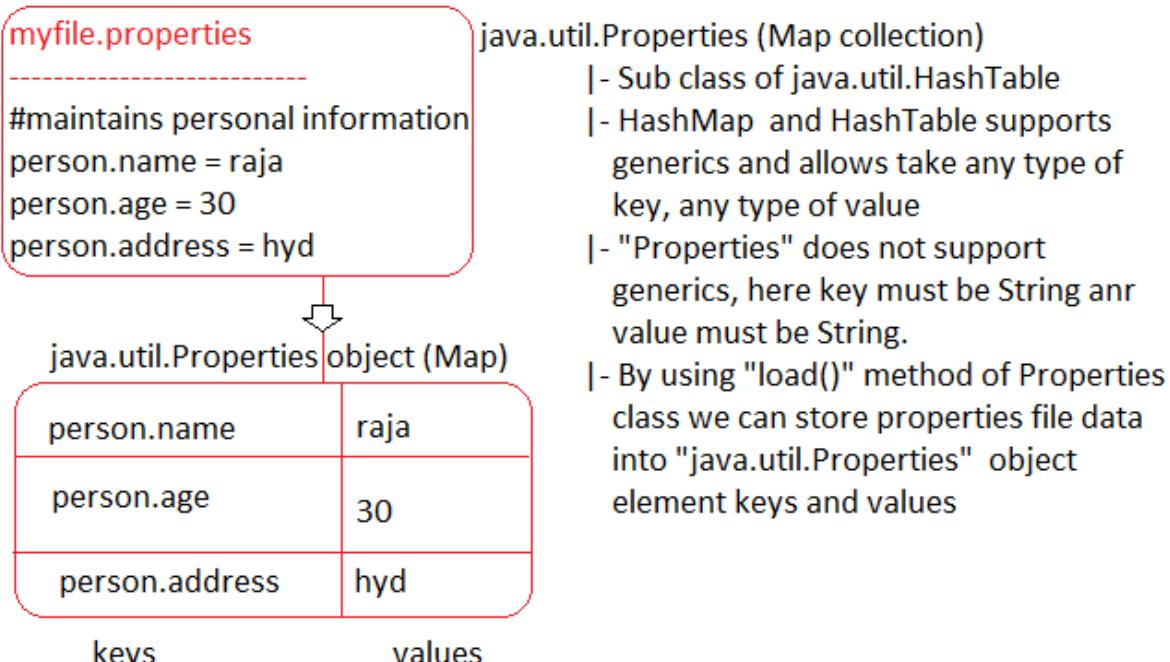
### IOCProj5-StrategyDP-CoreJava Explanation:

- It is implementing Strategy Design Pattern with support of Factory Pattern.

- This project is having following limitations,
  - This project is not 100% loosely coupled, the target and dependent classes are designed effectively, but if new dependent class is added or to change to another dependent class from current dependent class, we need to modify Factory, client applications code respectively.
  - But taking the support of properties file we can solve this problem up to some extent.
  - Don't hard code target, dependent class name in factory class, get them dynamically from properties file load them dynamically instantiate them and inject dependent to target class object.

### Properties file:

- It is a text file that maintain the entries in the form of key=value/ name=value pairs.
- We can take any extension, but recommended to take ".properties".
- '#' symbol can be used to comment the line.
- We can take "\_", ".", "- "symbols in the key naming.
- Duplicate keys are not allowed, but duplicate values can be taken.
- We can read properties file information into Map collection like java.util.Properties.



Sample code for read properties file data:

```
//Locate the file using stream  
InputStream is = new FileInputStream("<location of file>/myfile.properties");  
//Read data from properties files and write into java.util.Properties class object  
Properties props = new Properties();  
props.load(is);  
//To get specific value of specific key  
S.o.println("person.age keys value is :" + props.getProperty("person.age")); //30  
//To print all keys and values  
S.o.println("Data : " + props) //gives entire data of the object by calling toString()  
method internally
```

IO Streams:

- ✓ Byte Streams like InputStream and OutputStream can read, write both text data, binary data (images, audio files, video file and etc.).
- ✓ Character Streams like Reader, Writer streams can read, write only text data.
- ✓ InputStream, Reader Streams are useful for reading data.
- ✓ OutputStream, Writer streams are useful for writing data.

Improved Factory class Strategy Design Pattern to achieve more loose coupling:

[info.properties \(com.nt.commons\)](#)

sdp.dependent - com.nt.comp.DTDC

[FlipkartFactory.java](#)

```
public class FlipkartFactory {  
    private static Properties props;  
    static {  
        try {  
            //Locate properties file  
            InputStream is = new FileInputStream  
                ("src/com/nt/commons/info.properties");  
            //Load properties file info to java.util.Properties  
            //object  
            props = new Properties();  
            props.load(is);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

static block is class level one time executing block whereas constructor is object level 1 time executing block

```

    }
    //Factory method
    public static Flipkart getInstance() {
        Courier courier = null;
        Flipkart fpkt=null;
        try {
            //Get dependent class name from Properties collection
            //(props), load that class and instantiate that class
            courier = (Courier) Class.forName(props.getProperty
                ("sdp.dependent").newInstance());
            //Create target class object
            fpkt = new Flipkart ();
            //Assign dependent class object to target class object
            fpkt.setCourier(courier);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return fpkt;
    }

```

props object

sdp.dependent	com.nt.comp.DTDC
Key	Value

c1 → obj of java.lang.Class<  
com.nt.comp.DTDC  
(data of the  
object)>

### Client App

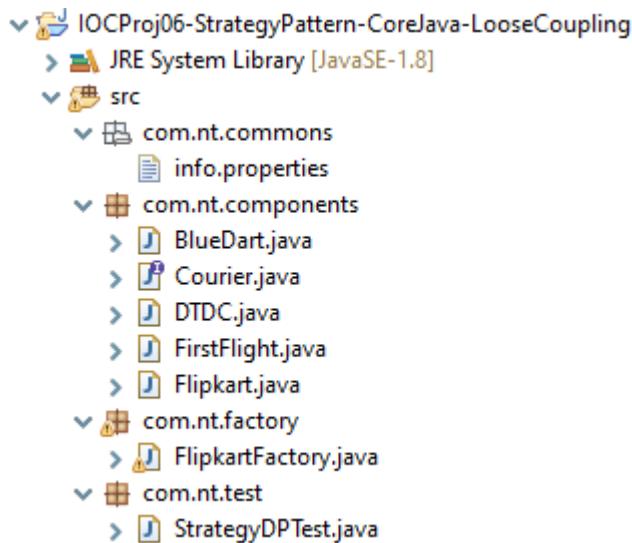
```

public class StrategyDPTest {
    public static void main(String[] args) {
        Flipkart fpkt = null;
        //Get Flipkart object from Factory
        fpkt = FlipkartFactory.getInstance();
        //invoke b.method
        System.out.println(fpkt.shopping(new String[] {"new 666", "777"}));
    }
}

```

### Directory Structure of IOCProj06-StrategyPattern-CoreJava-LooseCoupling:

- Copy paste the IOCProj05-StrategyPatter-CoreJava, add the new packages and file in their respective location and use also previous code for rest class.
- Develop the below directory structure and package, class, then use the following code with in their respective file and rest are collected from the previous project.



### info.properties

```
#Dependent class name configuration for achieve Loose coupling  
sdp.dependent = com.nt.components.DTDC
```

### FlipkartFactory.java

```
package com.nt.factory;  
  
import java.io.FileInputStream;  
import java.io.InputStream;  
import java.util.Properties;  
  
import com.nt.components.Courier;  
import com.nt.components.Flipkart;  
  
public class FlipkartFactory {  
  
    private static Properties props;  
  
    static {  
        InputStream is = null;  
        try {  
            //load Properties file using InputStreams  
            is = new  
FileInputStream("src/com/nt/commons/info.properties");  
            //Load properties file into java.util.Properties object
```

```

        //Load properties file into java.util.Properties object
        props = new Properties();
        props.load(is);
    } catch (Exception e) {
        e.printStackTrace();
    }

}

public static Flipkart getInstane() throws Exception {
    Courier courier=null;
    Flipkart fpkt=null;
    //get Dependent class object and create      class object
    courier = (Courier)
    Class.forName(props.getProperty("sdp.dependent")).newInstance();
    //create target class object
    fpkt=new Flipkart();
    //assign dependent class object to target class object
    fpkt.setCourier(courier);
    return fpkt;
}

}

```

### StrategyDPTest.java

```

package com.nt.test;

import com.nt.components.Flipkart;
import com.nt.factory.FlipkartFactory;

public class StrategyDPTest {

    public static void main(String[] args) {
        Flipkart fpkt = null;
        try {
            fpkt=FlipkartFactory.getInstane();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```

        System.out.println(fpkt.shopping(new String[] {"rain coat",
"umbrella", "Flu tablets"},
                new float[] {5000, 2000, 500})
);
}

}

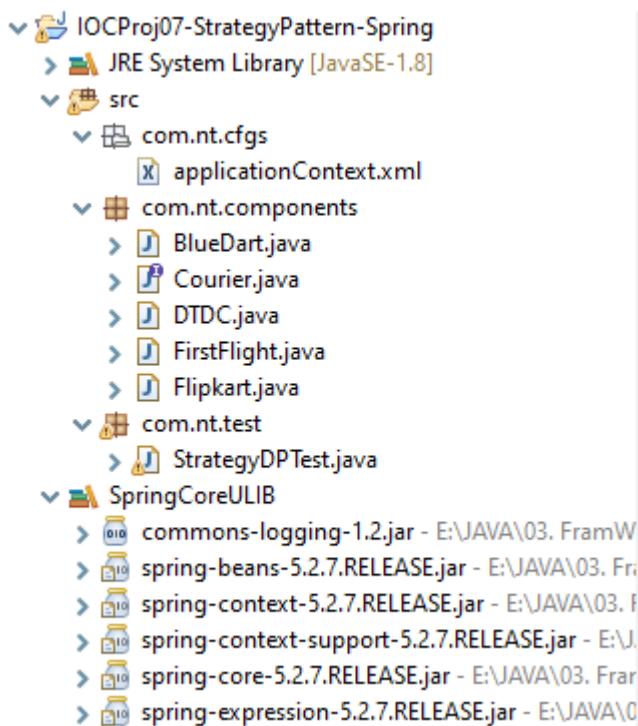
```

Limitations of developing Strategy Design Pattern using core java to achieve more loose coupling:

- Programmer should take care of instantiations (object creation).
- Programmer should take care of dependency management (assignment dependent class object to target class object).
- To achieve looser coupling we need write additional code in factory class by taking support of properties file (bit complex).
- No IoC container support.
- No internal cache supports.
- and etc.

To overcome these problems, implement Strategy Design Pattern in Spring environment.

Directory Structure of IOCProj07-StrategyPattern-Spring:



- Develop the above directory structure and package, class, XML file and add the jar also, then use the following code with in their respective file.
- Copy pastes the target and dependent class from IOCProj05-StrategyPatter-CoreJava, project.

### 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">

    <!-- Dependent bean configuration -->
    <bean id="dtdc" class="com.nt.components.DTDC" />
    <bean id="fFlight" class="com.nt.components.FirstFlight" />
    <bean id="bDart" class="com.nt.components.BlueDart" />

    <!-- Target bean configuration -->
    <bean id="fpkt" class="com.nt.components.Flipkart">
        <!-- <constructor-arg name="courier" ref="dtdc"/> -->
        <property name="courier" ref="fFlight" />
    </bean>

</beans>
```

### StrategyDPTest.java

```
package com.nt.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.CoroutinesUtils;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

import com.nt.components.Courier;
import com.nt.components.Flipkart;

public class StrategyDPTest {
```

```

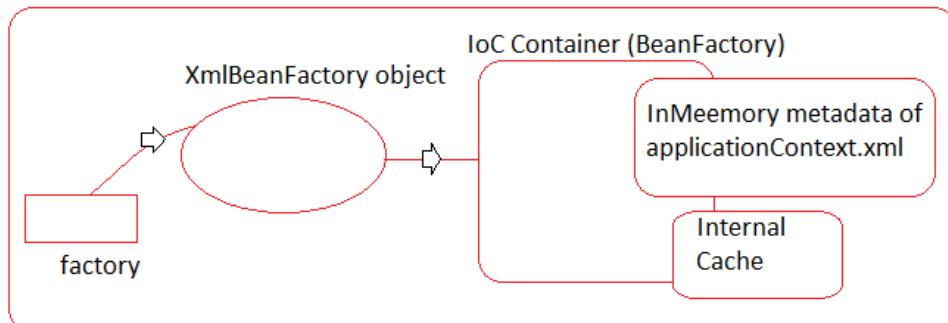
public static void main(String[] args) {
    Resource res = null;
    BeanFactory factory = null;
    Flipkart fpkt = null;
    // hold the location of the spring configuration file
    res = new
    ClassPathResource("com/nt/cfgs/applicationContext.xml");
        // create Bean Factory container [IoC container]
        factory = new XmlBeanFactory(res);
        // generate target class object
        // fpkt = (Flipkart) factory.getBean("fpkt");
        fpkt = factory.getBean("fpkt", Flipkart.class);
        // invoke the method
        System.out.println(fpkt.shopping(new String[] { "PPT", "Mask",
    "Senitizer" }, new float[] { 500, 200, 350 }));
    }

}

```

### Advantages of developing Strategy Design Pattern using Spring:

- The IOC container of spring gives the following benefits,
  - Takes care of instantiation Spring bean (object creation).
  - Takes care of dependency management either Setter Injection or Constructor Injection in our example (Assigning dependent class object to target class object).
  - No need of developing Factory class separately because IoC container itself acts as Factory.
  - The Spring bean configuration file gives loose coupling support to change from one dependent to another dependent.
  - IoC container provides internal cache support for the reusability Spring bean class object. and etc.



**Note:** IoC Container is create based on Factory, Flyweight Design pattern. While designing target and dependent classes we need Strategy design pattern principles to design them best.

### Thumb rule of Setter & Constructor Injection:

- ⊕ In Setter Injection then thumb rule is the IoC container first creates Target class object and later creates Dependent class object, calls setter method on target class object having dependent class object as the argument.
- ⊕ In Constructor Injection then thumb rule is the IoC container first creates Dependent class object and later creates Target class object using parameterized constructor having dependent class object as the argument value.

### Q. When to use Setter Injection and when to use Constructor Injection?

**Ans.** If all properties of Spring bean class are mandatory to participate in injection, then go for Constructor Injection otherwise go for Setter Injection.

e.g.

```
class Student {  
    private int sno;  
    private String sname;  
    private String sadd;  
    private float avg;  
  
    //4 param constructor  
    public Student (int sno, String sname, String sadd, float avg) {  
        this.sno = sno;  
        this.sname = sname;  
        this.sadd = sadd;  
        this.av = avg;  
    }  
}  
in applicationContext.xml  
-----  
<beans ....>  
    <bean id="st" class="<pkg>. Student"></b>  
        <constructor-arg name="sno" value="101"/>  
        <constructor-arg name="sname" value="raja"/>  
        <constructor-arg name="sadd" value="hyd"/>  
        <constructor-arg name="avg" value="67.55"/>  
    </bean>  
</beans>
```

```

class Student {
    private int sno;
    private String sname;
    private String sadd;
    private float avg;
    //4 setter method
    .....
    .....
}

in applicationContext.xml
-----
<beans ....>
    <bean id="st" class="<pkg>. Student"

```

**Note:**

- If you are using certain parameterized constructor for Constructor Injection, then we must involve all parameters of constructor in injection process by writing param count number of **<constructor-arg>** tags under **<bean>** otherwise exception will be raised, but no such restrictions for Setter Injection, we can involve our choice number of setter methods for setter injection.

**For example,**

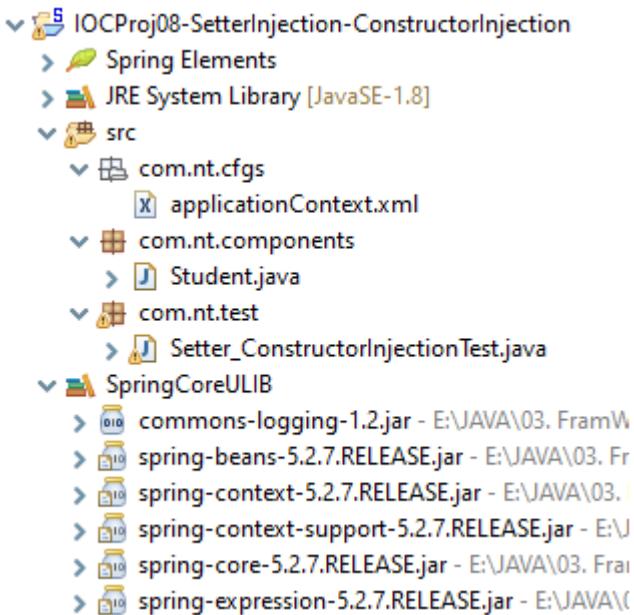
**Case 1:** If we have 10 bean properties i want involve my choice number of properties in the injection then we need just 10 setter methods to perform injection in all permutation and combination if we do same thing using Constructor Injection, we need  $10!$  (10 factorial - means some lakhs) number of constructors. Which quite complex so, prefer Setter injection here.

**Case 2:** If we have 10 bean properties, i want involve all properties in the injection as mandatory then we need one 10 param constructor to perform injection in all bean properties, if we do same thing using Setter injection, we need 10 setter methods but it doesn't keep the restriction of involving all the properties of the injection, so prefer constructor injection here.

**Note:** Every 0-param constructor is not default constructor, only the java compiler generated 0-param constructor is called default constructor. Java compiler generates 0-param constructor as default constructor when class not having any another user-defined constructor.

#### Directory Structure of IOCProj08-SetterInjection-ConstructorInjection:

- Develop the below directory structure and package, class, XML file then use the following code with in their respective file.



### Student.java

```
package com.nt.components;

public class Student {

    private int sno;
    private String sname;
    private String sadd;
    private float avg;

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

    public Student(int sno, String sname, String sadd, float avg) {
        System.out.println("Student : Student(-,-,-,-)");
        this.sno = sno;
        this.sname = sname;
        this.sadd = sadd;
        this.avg = avg;
    }

    public void setSno(int sno) {
        System.out.println("Student : setSno()");
        this.sno = sno;
    }
}
```

```

}

public void setSname(String sname) {
    System.out.println("Student : setSname()");
    this.sname = sname;
}

public void setSadd(String sadd) {
    System.out.println("Student : setSadd()");
    this.sadd = sadd;
}

public void setAvg(float avg) {
    System.out.println("Student : setAvg()");
    this.avg = avg;
}

@Override
public String toString() {
    return "Student [sno=" + sno + ", sname=" + sname + ", sadd="
+ sadd + ", avg=" + avg + "]";
}

}

```

### 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-4.3.xsd">

    <!-- Spring bean configuration -->
    <bean id="stud" class="com.nt.components.Student">
        <!-- Constructor Injection -->
        <!-- <constructor-arg value="101"/>
        <constructor-arg value="Nimu"/>
        <constructor-arg value="Hyd"/>
        <constructor-arg value="76.8"/> -->
    
```

```

<!-- Setter Injection -->
<property name="sno" value="101"/>
<property name="sname" value="Nimu"/>
<property name="sadd" value="Hyd"/>
</bean>

</beans>

```

### Setter\_ConstructorInjectionTest.java

```

package com.nt.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.nt.components.Student;

public class Setter_ConstructorInjectionTest {

    public static void main(String[] args) {
        BeanFactory factory = null;
        Student student = null;
        //Create BeanFactory [IoC] Container
        factory = new XmlBeanFactory(new
ClassPathResource("com/nt/cfgs/applicationContext.xml"));
        //get Student class Object
        student = factory.getBean("stud", Student.class);
        System.out.println(student);
    }
}

```

#### Note:

- ✓ Spring IoC container uses 0-param constructor for Spring bean instantiate in the following situations,
  - When Spring bean is configured without any injections.
  - When Spring bean is configured only having Setter Injection.
- ✓ Spring IoC container uses parameterized constructor for Spring bean

instantiate in the following situations,

- When at least one property is configured for Constructor Injection.

Object class to `toString()`:

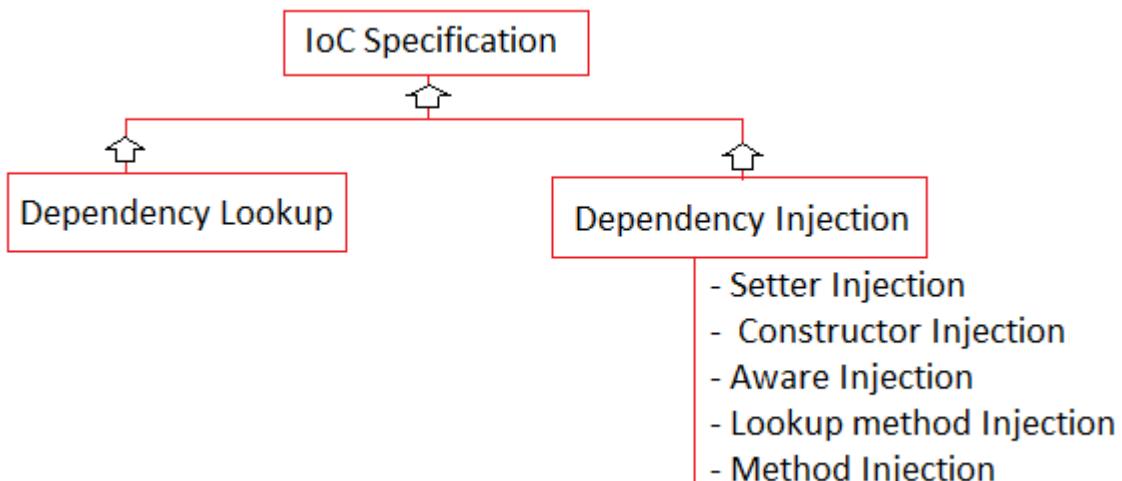
```
public String toString() {  
    return getClass().getName() + "@" + Integer.toHexString(hashCode());  
}
```

The `toString()` method of `java.lang.Object` class displays <Fully qualified current class name>@<hexa decimal String of hashCode> by using the above code.

**Q. What is the difference between IoC and Dependent Injection?**

**Ans.**

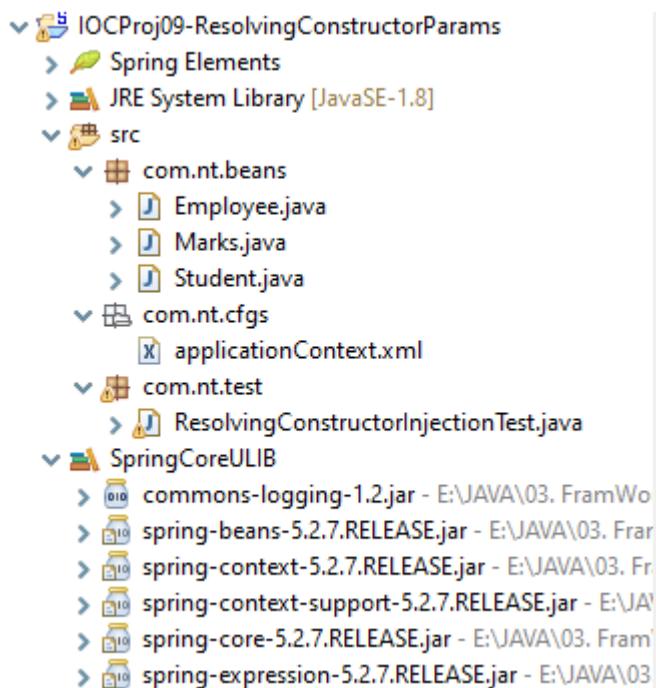
- IoC: Inversion of Control nothing but inverting control i.e., taking control from programmer and giving to some else like container/ framework and etc. to create and manage objects. Here that someone is called IoC container. Servlet container, JSP container, EJB container, Spring container can be called as IoC container.
- IoC is specification or plan that is having set of rules and guidelines asking other than programmer like IoC containers to create, manage objects and to keep object/ classes in dependency (as target, dependent classes/ objects).
- Dependency Lookup and Dependency Injection are implementation techniques of IoC, Spring IoC containers support both techniques.



**Resolving/ Identifying params in Constructor Parameters**

- Generally, we do Constructor injections configurations, in Spring bean configuration file in the order the parameters are placed in constructor. If we mismatch the order of parameters then we can identify/ resolve them.
  - a. Using index
  - b. Using type
  - c. Using name (from Spring 3.x Best)

### Directory Structure of IOCProj09-ResolvingConstructorParams:



- Develop the above directory structure and package, class, XML file and add the jar also, then use the following code with in their respective file.

#### Employee.java

```
package com.nt.beans;

public class Employee {

    private int eno;
    private String ename;
    private float salary;

    public Employee(int eno, String ename, float salary) {
```

```

        System.out.println("Employee : Employee(-,-,-)");
        this.eno = eno;
        this.ename = ename;
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Employee [eno=" + eno + ", ename=" + ename + ",
salary=" + salary + "]";
    }

}

```

### Marks.java

```

package com.nt.beans;

public class Marks {

    private float m1, m2, m3;

    public Marks(float m1, float m2, float m3) {
        System.out.println("Marks : Marks(-,-,-)");
        this.m1 = m1;
        this.m2 = m2;
        this.m3 = m3;
    }

    @Override
    public String toString() {
        return "Marks [m1=" + m1 + ", m2=" + m2 + ", m3=" + m3 + "]";
    }

}

```

### Student.java

```

package com.nt.beans;

public class Student {

    private int sno;
    private String sname;
    private String sadd;
    private float total;

    public Student(int sno, String sname, String sadd, float total) {
        System.out.println("Student : Student(-,-,-)");
        this.sno = sno;
        this.sname = sname;
        this.sadd = sadd;
        this.total = total;
    }

    @Override
    public String toString() {
        return "Student [sno=" + sno + ", sname=" + sname + ", sadd=" +
+ sadd + ", total=" + total + "]";
    }

}

```

### 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-3.1.xsd">

    <!-- Spring bean Configuration -->
    <!-- Using Index -->
    <bean id="mark" class="com.nt.beans.Marks">
        <constructor-arg index="1" value="70.4"/>
        <constructor-arg index="0" value="75.2"/>
        <constructor-arg index="2" value="67.3"/>
    
```

```

</bean>

<!-- Using Type -->
<bean id="emp" class="com.nt.beans.Employee">
    <constructor-arg value="2700.0" type="float"/>
    <constructor-arg value="Raja" type="java.lang.String"/>
    <constructor-arg value="3456" type="int"/>
</bean>

<!-- Using name -->
<bean id="stud" class="com.nt.beans.Student">
    <constructor-arg name="sname" value="Nirmala"/>
    <constructor-arg name="sno" value="103"/>
    <constructor-arg name="total" value="486"/>
    <constructor-arg name="sadd" value="hyd"/>
</bean>
</beans>
```

### ResolvingConstructorInjectionTest.java

```

package com.nt.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.nt.beans.Employee;
import com.nt.beans.Marks;
import com.nt.beans.Student;

public class ResolvingConstructorInjectionTest {

    public static void main(String[] args) {
        BeanFactory factory = null;
        Marks marks = null;
        Employee employee = null;
        Student student = null;
        //Create BeanFactory [IoC] container
        factory = new XmlBeanFactory(new
ClassPathResource("com/nt/cfgs/applicationContext.xml"));
```

```

//get Marks class object
marks = factory.getBean("mark", Marks.class);
System.out.println(marks);
System.out.println("-----");
employee = factory.getBean("emp", Employee.class);
System.out.println(employee);
System.out.println("-----");
student = factory.getBean("stud", Student.class);
System.out.println(student);
}

}

```

**Note:**

- ✓ Giving duplicate index, out of range index will throw exceptions.
- ✓ If we ignore to pass indexes for few params, then they will be assigned with left over indexes.
- ✓ If multiple params of a constructor are having same data type then go for resolving parameter either index or name (recommended).

**Q. What happens if we specify name, type, index attributes in one <constructor-arg> at a time?**

**Ans.** If all these are pointing to same param of constructor, then the given value will inject to that param otherwise exception will be raised.

[org.springframework.beans.factory.UnsatisfiedDependencyException](#): Error creating bean with name 'xxx' defined in class path resource [com/nt/cfgs/applicationContext.xml]: Unsatisfied dependency expressed through constructor parameter 2: Ambiguous argument values for parameter of type [yyy] - did you specify the correct bean references as arguments?

**Note:** Do not mismatch constructor parameters order in dependency Injection configurations and do not resolve them using different techniques.

## Cyclic Dependency Injection or Circular Dependency Injection

- It is all about making two classes dependent on each other like A on B and B on A.
- It is not at all industry practice. (No real time uses).

- Setter injection support Cyclic dependency injection and Constructor injection does not support.
- One side Settee injection and another side Construction injection also support Cyclic dependency injection.

## Using Setter Injection

```
class A {
    public B b;
    public void setB(B b) {
        this.b=b;
    }
    //toString()
}

class B {
    public A a;
    public void setA(A a) {
        this.a=a;
    }
    //toString()
}
```

applicationContext.xml

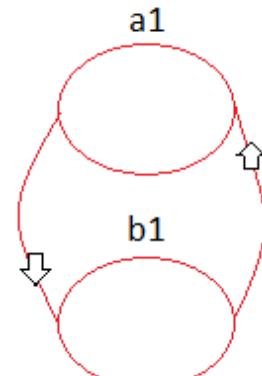
```
<beans ....>
    <bean id="a1" class="pkg.A">
        <property name="b" ref="a1"/>
    </bean>
    <bean id="b1" class="pkg.B">
        <property name="a" ref="b1"/>
    </bean>
</beans>
```

Client app

```
A a1 = factory.getBean("a1",
                      A.class);
B b1 = factory.getBean("b1",
                      B.class);
```

### Directory Structure of IOCProj10-CyclicDI-SetterInjection:

```
IOCProj10-CyclicDI-SetterInjection
    > Spring Elements
    > JRE System Library [JavaSE-1.8]
    > src
        > com.nt.beans
            > A.java
            > B.java
        > com.nt.cfgs
            > applicationContext.xml
        > com.nt.test
            > CyclicDITest.java
    > SpringCoreULIB
        > commons-logging-1.2.jar - E:\JAVA\03, Frameworks\Spring\SpringCoreULIB\commons-logging-1.2.jar
        > spring-beans-5.2.7.RELEASE.jar - E:\JAVA\03, Frameworks\Spring\SpringCoreULIB\spring-beans-5.2.7.RELEASE.jar
        > spring-context-5.2.7.RELEASE.jar - E:\JAVA\03, Frameworks\Spring\SpringCoreULIB\spring-context-5.2.7.RELEASE.jar
        > spring-context-support-5.2.7.RELEASE.jar - E:\JAVA\03, Frameworks\Spring\SpringCoreULIB\spring-context-support-5.2.7.RELEASE.jar
        > spring-core-5.2.7.RELEASE.jar - E:\JAVA\03, Frameworks\Spring\SpringCoreULIB\spring-core-5.2.7.RELEASE.jar
        > spring-expression-5.2.7.RELEASE.jar - E:\JAVA\03, Frameworks\Spring\SpringCoreULIB\spring-expression-5.2.7.RELEASE.jar
```



- Develop the above directory structure and package, class, XML file and add the jar also, then use the following code with in their respect file.

### A.java

```
package com.nt.beans;

public class A {

    private B b;

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

    public void setB(B b) {
        System.out.println("A : setB()");
        this.b = b;
    }

    @Override
    public String toString() {
        return "A [b]";
    }
}
```

### B.java

```
package com.nt.beans;

public class B {

    private A a;

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

    public void setA(A a) {
        System.out.println("B : setA()");
    }
}
```

```

        this.a = a;
    }

@Override
public String toString() {
    return "B [a]";
}

}

```

### 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-3.1.xsd">

    <bean id="a1" class="com.nt.beans.A">
        <property name="b" ref="b1"/>
    </bean>

    <bean id="b1" class="com.nt.beans.B">
        <property name="a" ref="a1"/>
    </bean>

</beans>

```

### CyclicDITest.java

```

package com.nt.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.nt.beans.A;

public class CyclicDITest {

```

```

public static void main(String[] args) {
    BeanFactory factory = null;
    A a1 = null;
    //Create BeanFactory [IoC] container
    factory = new XmlBeanFactory(new
ClassPathResource("com/ht/cfgs/applicationContext.xml"));
    //get Beans class object
    a1 = factory.getBean("a1", A.class);
    System.out.println(a1);
}

}

```

We will get the following Exception: when you use `toString()` method in both the classes

**Exception in thread “main” java.lang.StackOverflowError** : So in `toString()` method we just display the message.

## Using Constructor Injection

```

class A {
    public B b;
    public A(B b) {
        this.b=b;
    }
    //toString()
}

```

applicationContext.xml

```

<beans ....>
    <bean id="a1" class="pkg.A">
        <constructor-arg ref="b1"/>
    </bean>
    <bean id="b1" class="pkg.B">
        <constructor-arg ref="a1"/>
    </bean>
</beans>

```

```

class B {
    public A a;
    public B(A a) {
        this.a=a;
    }
    //toString()
}

```

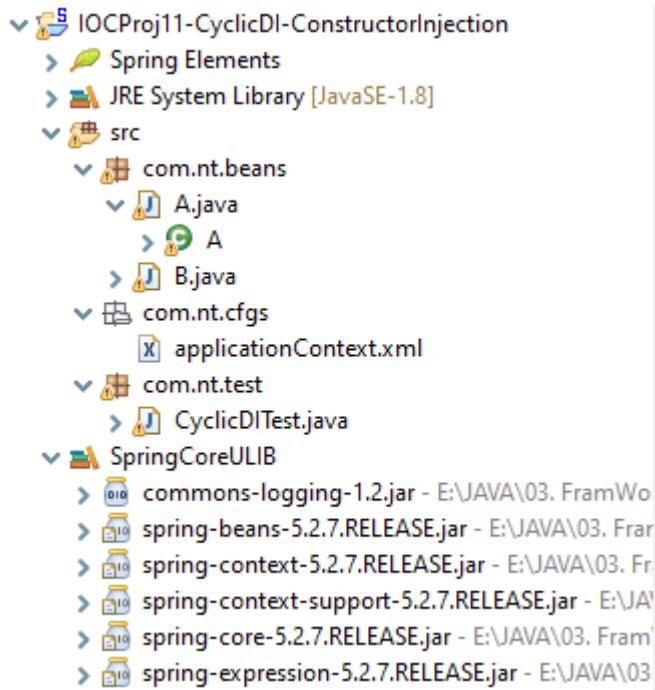
Client app

```

A a1 = factory.getBean("a1",
A.class);

```

## Directory Structure of IOCProj11-CyclicDI-ConstructorInjection:



- Develop the above directory structure and package, class, XML file and add the jar also, then use the following code with in their respective file.

### A.java

```
package com.nt.beans;

public class A {

    private B b;

    public A(B b) {
        System.out.println("A : A(-)");
        this.b = b;
    }

    @Override
    public String toString() {
        return "A [b]";
    }
}
```

## B.java

```
package com.nt.beans;

public class B {

    private A a;

    public B(A a) {
        System.out.println("B : B(-)");
        this.a = a;
    }

    @Override
    public String toString() {
        return "B [a]";
    }
}
```

## 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-3.1.xsd">

    <bean id="a1" class="com.nt.beans.A">
        <constructor-arg ref="b1"/>
    </bean>

    <bean id="b1" class="com.nt.beans.B">
        <constructor-arg ref="a1"/>
    </bean>

</beans>
```

## CyclicDITest.java

```
package com.nt.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.nt.beans.A;

public class CyclicDITest {

    public static void main(String[] args) {
        BeanFactory factory = null;
        A a1 = null;
        //Create BeanFactory [IoC] container
        factory = new XmlBeanFactory(new
ClassPathResource("com/nt/cfgs/applicationContext.xml"));
        //get Beans class object
        a1 = factory.getBean("a1", A.class);
        System.out.println(a1);
    }
}
```

**Note:** We will get the following Exception in this case and we can't solve that like previous,

Exception in thread "main"

[org.springframework.beans.factory.BeanCreationException](#): Error creating bean with name 'a1' defined in class path resource [com/nt/cfgs/applicationContext.xml]: Cannot resolve reference to bean 'b1' while setting constructor argument; nested exception is  
[org.springframework.beans.factory.BeanCreationException](#): Error creating bean with name 'b1' defined in class path resource [com/nt/cfgs/applicationContext.xml]: Cannot resolve reference to bean 'a1' while setting constructor argument; nested exception is  
[org.springframework.beans.factory.BeanCurrentlyInCreationException](#): Error creating bean with name 'a1': Requested bean is currently in creation: Is there an unresolvable circular reference?

## Cyclic Dependency Injection having one side Setter injection and another side Constructor injection

**Note:** We must call factory.getBean(--) having that spring bean class bean id in which Setter injection support is there

```
class A {                                class B {  
    public B b;  
    public void setB(B b) {  
        this.b=b;  
    }  
    //toString()  
}  
                                         }  
                                         public A a;  
                                         public B(A a) {  
                                         this.a=a;  
                                         }  
                                         //toString()
```

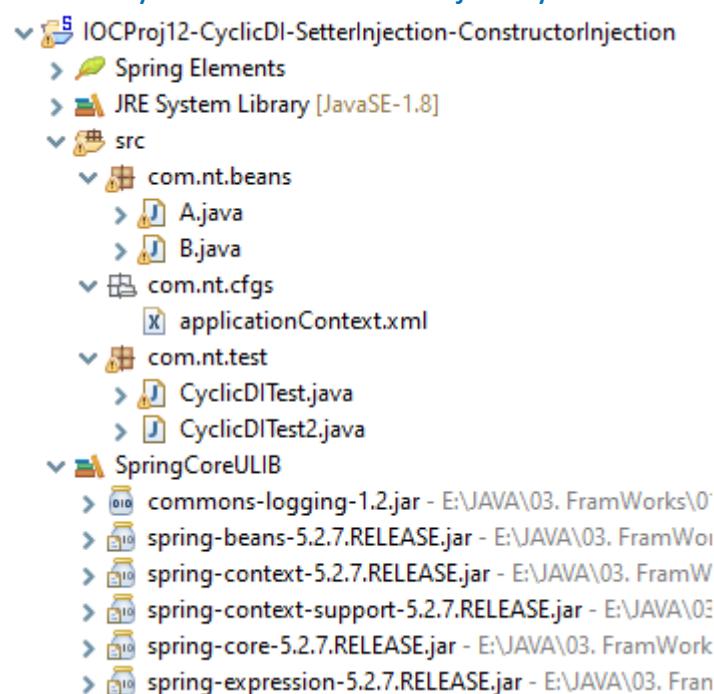
### applicationContext.xml

```
<beans ....>  
    <bean id="a1" class="pkg.A">  
        <property name="b" ref="b1"/>  
    </bean>  
    <bean id="b1" class="pkg.B">  
        <constructor-arg ref="a1"/>  
    </bean>  
</beans>
```

### Client app

```
A a1 =  
factory.getBean("a1",  
A.class); //success  
B b1 =  
factory.getBean("b1",  
B.class); //error
```

### Directory Structure of IOCProj12-CyclicDI-SetterInjection-ConstructorInjection:



- Develop the above directory structure and package, class, XML file and add the jar also, then use the following code with in their respective file.

### A.java

```
package com.nt.beans;

public class A {

    private B b;

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

    public void setB(B b) {
        System.out.println("A : setB()");
        this.b = b;
    }

    @Override
    public String toString() {
        return "A [b]";
    }
}
```

### B.java

```
package com.nt.beans;

public class B {

    private A a;

    public B(A a) {
        System.out.println("B : B()");
        this.a = a;
    }
}
```

```

@Override
public String toString() {
    return "B [a]";
}

}

```

### 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-3.1.xsd">

    <bean id="a1" class="com.nt.beans.A">
        <property name="b" ref="b1"/>
    </bean>

    <bean id="b1" class="com.nt.beans.B">
        <constructor-arg ref="a1"/>
    </bean>

</beans>

```

### CyclicDITest.java

```

package com.nt.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.nt.beans.A;
import com.nt.beans.B;

public class CyclicDITest {

    public static void main(String[] args) {
        BeanFactory factory = null;

```

```

A a1 = null;
B b1 = null;
//Create BeanFactory [IoC] container
factory = new XmlBeanFactory(new
ClassPathResource("com/nt/cfgs/applicationContext.xml"));
//get Beans class object
a1 = factory.getBean("a1", A.class);
System.out.println(a1);
b1 = factory.getBean("b1", B.class);
System.out.println(b1);
}

}

```

**Note:** If we first called B b1 = factory.getBean("b1", B.class); then you will get the below error

Exception in thread "main"

[org.springframework.beans.factory.BeanCreationException](#): Error creating bean with name 'b1' defined in class path resource [com/nt/cfgs/applicationContext.xml]: Cannot resolve reference to bean 'a1' while setting constructor argument; nested exception is  
[org.springframework.beans.factory.BeanCreationException](#): Error creating bean with name 'a1' defined in class path resource [com/nt/cfgs/applicationContext.xml]: Cannot resolve reference to bean 'b1' while setting bean property 'b'; nested exception is  
[org.springframework.beans.factory.BeanCurrentlyInCreationException](#): Error creating bean with name 'b1': Requested bean is currently in creation: Is there an unresolvable circular reference?

## Difference between setter injection and constructor injection

Setter Injection	Constructor Injection
a. Uses setter method to inject dependent value/ objects to target class object.	a. Uses param constructor to instantiate target class and also to inject dependent values/ objects to target class object.
b. We need to use <property> tag for this.	b. We need to use <constructor-arg> tag for this.

c. We can resolve/ identity setter method only passing its name like by passing xxx/Xxx word of setter method in "name" of <property>.	c. We can resolve/ identity constructor params using name, index, type.
d. Resolving setter method by using "name" of <property> tag is mandatory	d. Resolving constructor param using name/ index/ type is optional if you write <constructor-arg> tags in the same order of constructor params.
e. Supports Cyclic dependency injection.	e. Doesn't support.
f. Bit slow because injection happens after creating target bean class object.	f. Bit faster because injection takes place while instantiating target bean class object.
g. In this mode, IoC container first creates target class object, then creates dependent class object.	g. In this mode of injection, IoC container first creates dependent class object then creates target class object.
h. Suitable when you want involve your choice number of properties in dependency injection.	h. Suitable when you want involve all bean properties in dependency injection on mandatory basis.
i. To involve our choice number of bean properties to Setter injection we just need n setter methods for n properties.	i. To involve our choice number of bean properties to Constructor injection we just need n! overloaded constructor for n properties (very complex).
j. If Spring bean is configured with no injection or only with Setter injection then IoC container creates Spring bean class object using 0-param constructor.	j. Once Constructor Injection is enabled IoC container creates Spring bean class object using parameterized constructor.
k. We can use "p" namespace tags/ attributes as alternate to <property> to configure Setter injection.	k. We can use "c" namespace tags/ attributes as alternate to <constructor-arg> to configure Constructor injection.

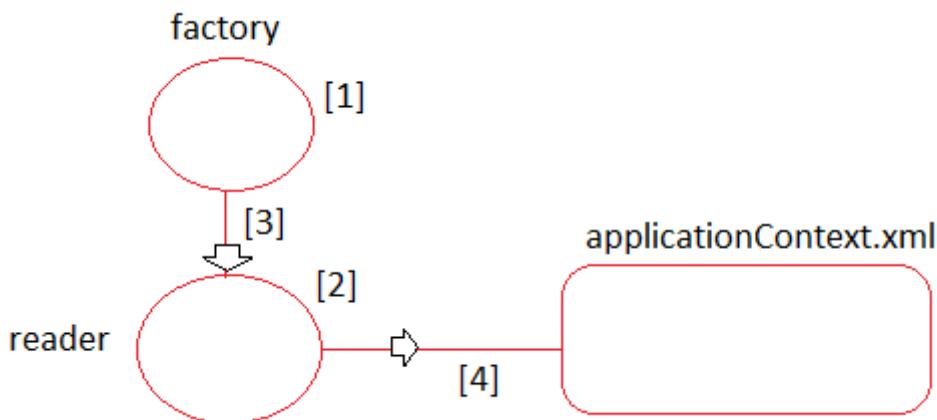
**Note:** XmlBeanFactory class is deprecated from Spring 3.1 and it is recommended to use DefaultListableBeanFactory as alternate to create BeanFactory container.

### Limitations of XmlBeanFactory:

- a. Doesn't allow to take multiple XML files at a time as Spring bean configuration file.
- b. Its internally uses DefaultListableBeanFactory for registering every Spring bean.
- c. The XML parser used by XmlBeanFactory to parse and process XML files is not so good towards performance.
- d. Needs to use Resource object to hold the name and location of Spring bean configuration file.

### Creating BeanFactory container using DefaultListableBeanFactory:

```
DefaultListableBeanFactory factory = null;
XmlBeanDefinationReader reader = null;
//Create IoC container
factory = new DefaultListableBeanFactory();
//Create reader object
reader = new XmlBeanDefinationReader(factory);
//Specify the name of location Spring bean configuration
reader.loadBeanDefinations("com/nt/cfgs/applicationContext.xml");
```



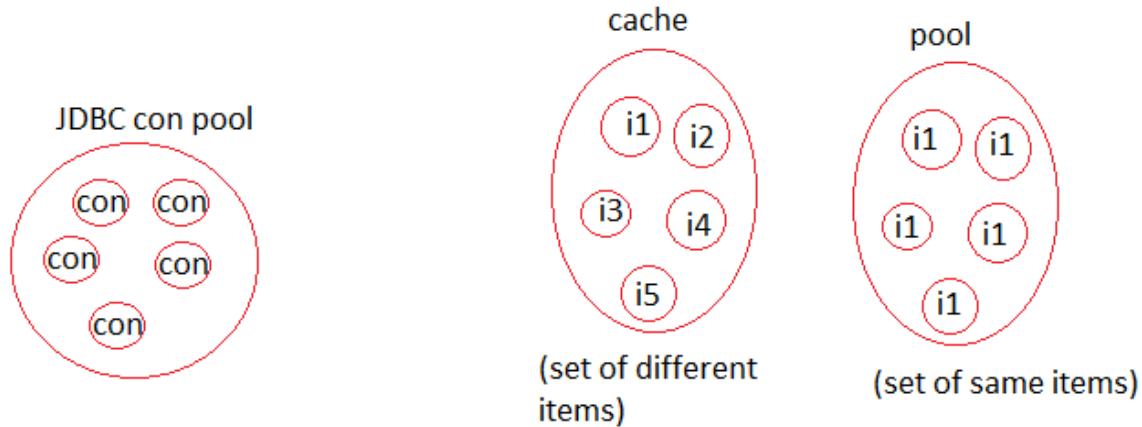
### Advantages of DefaultListableBeanFactory to create BeanFactory container:

- a. It directly recognizes the Spring Beans without passing this work to another classes.
- b. Allow to pass multiple XML files as Spring bean configuration files because loadBeanDefination() is designed taking var args.
- c. No need of taking Resource object to hold the name and location of Spring bean configuration file. We can pass the same info directly as String.
- d. The performance towards reading and processing XML file is good.

## Layered Application (Mini Project)

Background preparation for developing real-time Dependency Injection, Real-time Strategy Design Pattern based layered application (Mini Project Discussions)

JDBC connection pool:



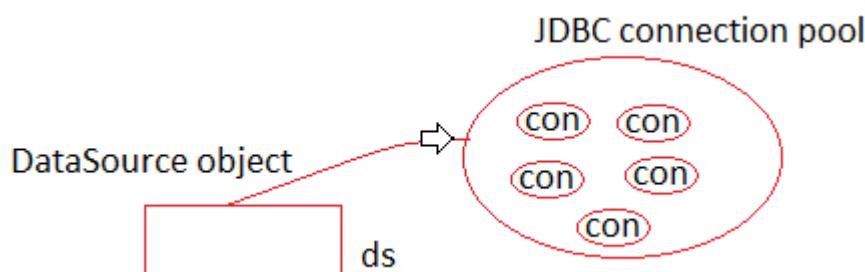
- JDBC connection pool is a factory that contains set of readily available set of JDBC connection objects actually being used.

**Advantages:**

- a. Gives reusability of JDBC connection objects.
- b. With minimum number of JDBC connection objects we can make maximum requests/ applications talking to DB software.
- c. Creating JDBC connection object, managing JDBC connection object and close JDBC connection will be taken care by JDBC connection pool itself.

**DataSource object:**

- DataSource object represents JDBC connection pool and it acts entry point for JDBC connection pool i.e. each JDBC connection object from JDBC connection pool should be accessed through DataSource object.
- DataSource object means it is the object of java class that implements javax.sql.DataSource (I).



```
Connection con = ds.getConnection();
gets one JDBC connection object from JDBC connection pool.
```

#### Explain different types of JDBC connection objects:

##### a. Direct JDBC connection object:

- Create by the programmer manually.

```
Class.forName(".....");
```

```
Connection con = DriverManager.getConnection(-,-,-);
```

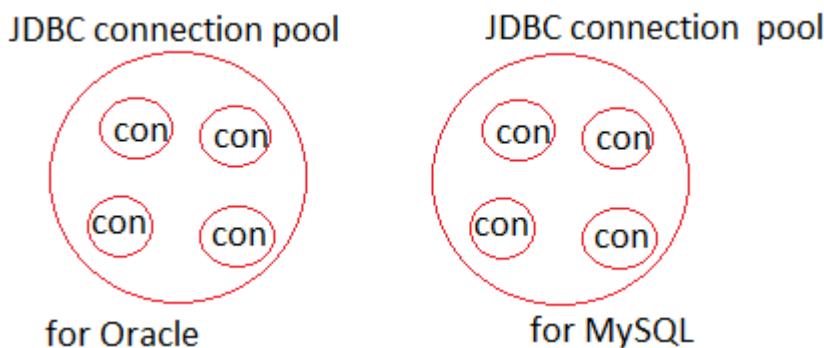
##### b. Pooled Connection object (recommended):

- Collected from the JDBC connection pool through DataSource object.

```
Connection con = ds.getConnection();
```

#### Note:

- ✓ All JDBC connection objects of a JDBC connection pool represents connectivity with same DB s/w. For example, JDBC connection pool for Oracle means all JDBC connection objects in the JDBC connection pool represents connectivity with same Oracle DB s/w.
- ✓ We can create different JDBC connection pool for different DB softwares.



#### Two types of connection pool:

##### a. Standalone JDBC connection pool:

- Useful in standalone applications.
- Runs outside of the servers like Tomcat, WebLogic, etc.
- These are independent JDBC connection pool.  
e.g. apache DBCP, Prxool, Hikari CP (best), C3PO and etc.

##### b. Server managed JDBC connection pool:

- Created and managed in servers like Tomcat, WebLogic and etc.

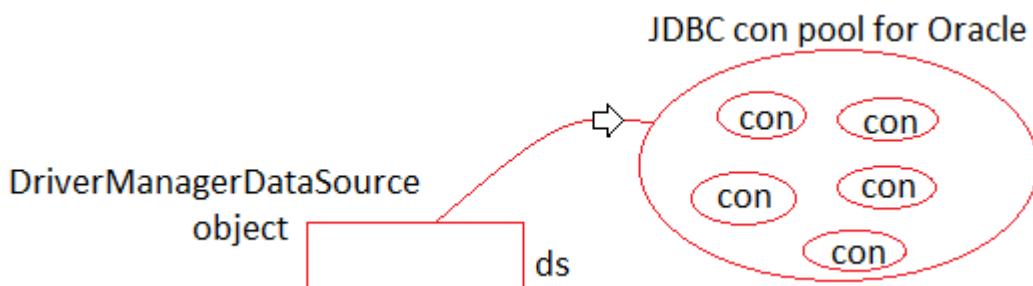
- Useful in those applications that are deployed and executable in servers like web applications.  
e.g. Tomcat managed JDBC connection pool, WebLogic managed JDBC connection pool and etc.

**Note:** Spring framework gives built-in JDBC connection pool support. i.e. it is giving one predefined DataSource class like DriverManagerDataSource implementing javax.sql.DataSource (I). If we configure this class to Spring bean, then Spring container/ IoC container creates DataSource object representing JDBC connection pool.

To get Connection pool in applicationContext.xml file:

```
<beans ....>
    <bean id="drds"
          class="org.springframework.jdbc.datasource.
                           DriverManagerDataSource">
        <property name="driverClassName"
                  value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url"
                  value="jdbc:oracle:thin:@localhost:1521:xe"/>
        <property name="username" value="system"/>
        <property name="password" value="manager"/>
    </bean>
</beans>
```

**Note:** We can get Bean property name of predefined class by using API focus or source code of that class.



3 types of Java beans (based on the kind of data they hold):

- VO class (Value object class)
- DTO class (Data transfer object class)
- BO class (Business object class)

### VO class:

- The java beans whose object holds either input or output is called VO class. Generally, in most of the appropriate the inputs or outputs will be there as "text" values. So, this class generally maintains all its properties as String properties.

#### StudentVO.java

```
public class StudentVO {  
    //Bean properties  
    private String sno;  
    private String sname;  
    private String sadd;  
    private String m1, m2, m3;  
    //Setters and getters  
    public void setSno(String sno) {  
        this.sno = sno;  
    }  
    public String getSno() {  
        return sno;  
    }  
    .....  
    .....  
}
```

### DTO class:

- It is also called as TO class (Transfer object).
- It is java bean whose object holds shippable (transferable) data from one class to another class with in a project or from one project to another project.
- This class implements java.io.Serializable (I) and contains different types properties.

**Note:** We can send only Serializable objects data over the Network, to make object as Serializable object the class of the object must implement.

java.io.Serializable (I) (Marker interface - Empty Interface).

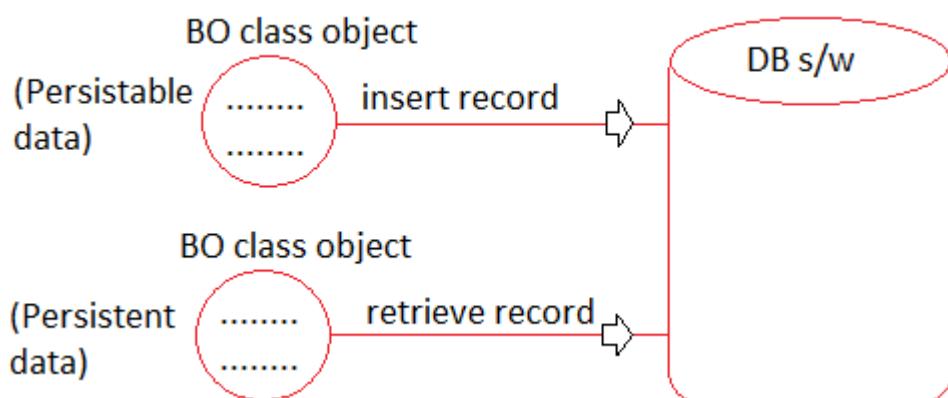
Serialization is the process of converting object data into bits and bytes or (Streams of bytes). These bits can be sent over the Network or can be written to the file as needed.

### CardDetailsDTO.java

```
public class CardDetailsDTO implements java.io.Serializable
{
    //Bean properties
    private long cardNo;
    private String holderName;
    private String bankName;
    private String paymentGateway;
    private Date expiryDate;
    private int cvv;
    //Setters and getters
    .....
    .....
}
```

### BO class:

- It is the java bean whose holds persistable data (to inserted) or persistent data (already saved).
- Generally, this class will be taken on 1 per DB tables basis. This properties type, count must be compatible with DB tables columns type and count.



- BO class is also called as Entity class or Model class or Domain class or Persistence class.
- BO holds persistable/ persistent data.
- DAO hold Persistence logic (JDBC code, hibernate code) .

Oracle DB table :-

Custom

|---> cno (n) (pk)  
|---> cname (vc2)  
|---> cadd (vc2)  
|---> billAmt (float)

BO class/ Entity class

Customer.java

```
public class CustomerBO {  
    private int cno;  
    private String cname;  
    private String cadd;  
    private float billAmt;  
    //Setters and getters  
    .....  
    .....
```

Writing multiple logics in single java class is a bad practice the reasons are:

- Since we mix-up multiple logics in a single class the code looks very clumsy and the clean separation between logics will not be possible.
- The modifications done in one kind of logics it may affects other kinds of logics.
- The maintenance and enhancement of the project becomes difficult.
- Parallel development is not possible, so the productivity is very poor.
- It is not industry standard.

Different logics:

 **Presentation Logic:**

The logic that gives user interface either supply input or the display output.

e.g. HTML, CSS, JSP logics  
scanner, sysout, awt, swing logics

 **Business Logic/ service Logic:**

The main logic of the application that deals with calculations, analysis and etc.

e.g. calculate interest amount, calc emp gross, net salaries, calc student total, avg, rank

 **Persistence Logic:**

The logic that interacts with persistence store like DB s/w and manipulate its data by performing CURD operations.

e.g. JDBC code, 10 stream code, hibernate code, Spring JDBC/ ORM/ Data code etc.

### Why Layered Application:

- ⊕ To solve the above problem, develop your applications as layered applications.
- ⊕ A layer is logical partition in the application representing certain logics having 1 or more class/ files.
- ⊕ Presentation layer contains Presentation logic.
- ⊕ Service layer contains Business logic/ Service logic.
- ⊕ Persistence layer contains Persistence logic.
- ⊕ The different layers of layered application will interact with each other.

### Typical Standalone Layered in Java:

VO	DTO	BO
Client App	Controller class	Service class
(Presentation logic)	(Monitoring logics)	(Business logic)
		(Persistence logic)
		(Service logic)

### DAO class:

- ⊕ DAO stands for Data Access Object Class.
- ⊕ The java class that contains only persistence logic and makes that logic reusable logics and flexible logics to modify is called DAO class.
- ⊕ It is called DAO class because it is having code that is developed by using Data Access Technologies/ frameworks like JDBC, hibernate, and etc.
- ⊕ If number of DB tables in DB s/w are < 100 then take 1 DAO per 1 DB table.
- ⊕ If number of DB tables in DB s/w are > 100 then take 1 DAO per 4 or 5 related DB tables.
- ⊕ Every DAO class contains,
  - a. **Query part (SQL queries used for persistence logic):**
    - Declare them at top of DAO class String constant values having uppercase letters.
    - It is recommended to avoid symbol in the SQL query and place column name always.

### In DAO class

```
private static final String GET_STUDENT_BY_SNO = "SELECT  
SNO, SNAME, SADD FROM STUDENT WHERE SNO=?";
```

### b. Code part:

- Contain multiple methods to perform multiple persistence operations (CURD) operation on 1 method per each persistence operation. These methods get inputs from service class (<= 3 values as normal params, >3 values as BO class objects).

#### In DAO class

```
public StudentBO getStudentBySno(int sno) {  
    .....  
    .....  
}  
public int insert (StudentBO bo) {  
    .....  
    .....  
}
```

#### Note:

- ✓ This DAO class can use either Direct connection or pooled connection to interact with DB s/w.
- ✓ DB s/w like Oracle, MySQL, PostgreSQL and etc. falls under Data storage softwares.
- ✓ JDBC, hibernate, Spring JDBC/ ORM/ Data falls under Data Access Technologies/ frameworks because we develop Persistence logic to manipulate DB s/w data.

#### Service class:

- ⊕ It is java class that contains either business logic or service logic.
  - ⊕ Business logic means that main logics of the application which fulfill the main requirement of application having calculations, analyzation with Transaction Management support.
- Transaction Management:** Executing logics by applying do everything or nothing principle.
- ⊕ Calculating total, avg, and generating results.
  - ⊕ Calculating simple, compound interest amount.
  - ⊕ We take this class on 1 per module basis having multiple business methods.
  - ⊕ One service class can use 1 or more DAO classes internally.
  - ⊕ This class gets more than 3 inputs as DTO object from controller and gives more than 1 output value to controller as another DTO object.

- o Set of instructions is called 1 program.
- o Set of programs is called 1 application.
- o Set of applications is called 1 module.
- o Set of modules is called 1 project.

#### In Service class

```
public String registerStudent(StudentDTO dto) {
    .....
    .....
    //B.logic
    return ....;
}
public String registerEmployee(EmployeeDTO dto) {
    .....
    .....
    //B.logic
    return ....;
}
```

#### Controller class:

- + This java class that contains monitoring logics or controlling logics like passing appropriate client application inputs to appropriate service class and passing service class outputs to appropriate client application.
- + Gets inputs as VO from client application and passing to service class as DTO.
- + This will be taken on 1 per project basis.

#### In Controller class

```
public String processStudent(StudentVO vo) {
    .....
    .....
    return ....;
}
public String processEmployee(EmployeeVO vo) {
    .....
    .....
    return ....;
}
```

### Client Application:

- ⊕ Contains user interface logic providing environment to read inputs from end-user and display outputs for end-user.
- ⊕ There can be multiple client applications to gather different types of inputs to display their outputs (count will be very best on the requirement).

### How classes are Dependent:

VO                  DTO                  BO  
Client App ----> Controller class ----> Service class ----> DAO class ----> DB s/w

- While dealing with pooled connections on DAO class it needs DataSource object as dependent, so We can inject DataSource to DAO.
- Similarly, we can inject DAO object to service class and service class object to controller. But client applications get controller class objects by using factory.getBean(-) because we create IoC container in the client application itself.
- Controller is just act as Target class and not dependent to another class, so controller class need not to implement any interface whereas Service class, DAO class, DataSource class should implement respective interfaces to achieve loose coupling because they acting as dependent class to other classes.

### Exception propagation:

```
public class EmployeeDAO {  
    public int insert(EmployeeBO bo) {  
        try {  
            .....  
            ....// JDBC code  
            .....  
        }  
        catch(Exception e) {  
            .....  
        }  
    }  
}
```

```
public class EmployeeDAO {  
    public int insert(EmployeeBO bo) {  
        .....  
        .....  
        .....  
    }  
} //Best practice
```

- When exception is raised in the insert (-) method the control automatically goes to the catch (-) block and the exception will be eaten/ suppressed there but we need to propagate/ pass the exception to client

application because end-user expects error/ success output from client application only, to solve this problem do not place try/ catch block in DAO, service, controller classes just make them to declaring the exception to thrown using "throws" clause and make client application catching the handling the exception.

- Here the exception raised in DAO propagates Client application through service, controller classes because they are also not catching and handling exception and they are also propagating the exception using "throws".

## Story Board of Layered Application

```
Client App --- VO ---> MainController --- DTO ---> CustomerMgmtServiceImpl --  

(Presentation logic)      (Monitoring logic)          (Business logic)  

- BO ---> CustomerDAOImpl ----> DB s/w  

(Persistence logic)
```

**Note :** All DataSource classes implementing classes of javax.sql.DataSource().  
eg:: DriverManagerDataSource  
HikariDataSource --> hikari cp  
BasicDataSource --> Apache DBCP

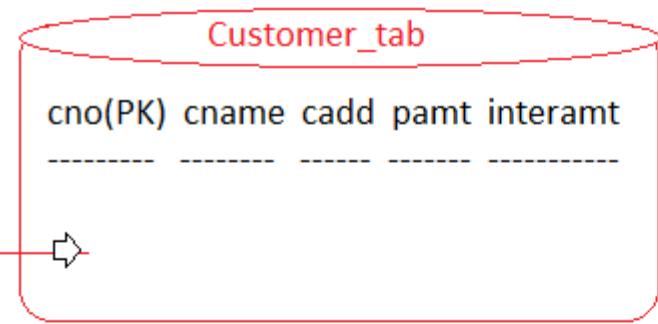
### applicationcontext.xml

```
<beans ....>
<bean id="drds" class="org.sf.jdbc.datasource.DriverManagerDataSource">
.....
//inject values jdbc properties(driverClassname,url,username,password )
.....
</bean> (11)

<bean id="custDAO" class="com.nt.dao.CustomerDAOImpl">
<constructor-arg ref="drds"/> (12)
</bean> (10)

<bean id="custService" class="com.nt.service.CustomerServiceImpl"> (13)
<costructor-arg ref=" custDAO"/>
</bean> (7)

<bean id="controller" class="com.nt.controller.MainController"> (14)
<constructor-arg ref="custService"/>
</bean> (4)
</beans>
```



### CustomerDAO.java

```

public interface CustomerDAO{
    public int insert(CustomerBO bo)throws Exception;
}

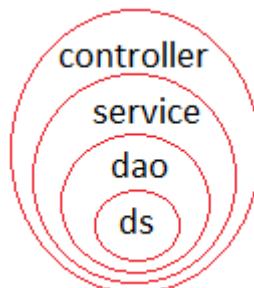
```

### CustomerDAOImpl.java

```

public final class CustomerDAOImpl implements CustomerDAO{
    private DataSoruce ds;
    public CustomerDAOImpl(DataSource ds) {
        this.ds=ds;
    }
    (h)
    public int insert(CustomerBO bo)throws Exception{
        .....
        .....
        //jdbc code to insert record
        .....
        return 0/1; (i)
    }
}

```



(15)  
Keeping  
beans  
in the  
cache

### Internal cache of IoC Container (2?)

drds	DriverMangerDataSource class ref
custDAO	CustomerDAOImpl obj ref (5?)
custService	CustomerMgmtServiceImpl obj ref (8?)
controller	MainController obj ref (10?)

keys  
(beanIds)      values  
(bean class objects)

#### CustomerMgmtService.java

```
-----  
public interface CustomerMgmtService{  
    public String calculateSimpleIntrestAmount(CustomerDTO dto)throws Exception;  
}
```

#### CustomerMgmtServiceImpl .java

```
-----  
final  
public class CustomerMgmtServiceImpl implements CustomerMgmtService{  
    private CustomerDAO dao;  
    public CustomerMgmtServiceImpl(CustomerDAO dao){  
        this.dao=dao;  
    }  
    (f)  
    public String calculateSimpleIntrestAmount(CustomerDTO dto)throws Exception {  
        //calculate simple intrest amount from DTO  
        .....  
        .....  
        //prepare CustomerBO having persistable Data like cno,cname,cadd,pamt, intrestamt  
        .....  
        .....  
        //use DAO  
        (g)  
        (i) int count=dao.insert( bo );  
        if(count==0)  
            retun "Customer registration failed intrest amount is ::"+iamt;  
        else  
            return "Customer registration succeded intrest amount is ::"+iamt;  
    }  
}
```

### MainController.java

```
-----  
public final class MainController {  
    private CustomerMgmtService service;  
    public MainController(CustomerMgmtService service){  
        this.service=service;  
    }  
    public String processCustomer(customervo vo) throws Exception{  
        //convert CustomerVO to CustomerDTO  
        .....  
        .....  
        //use service  
        (l) String result = serive.calculateSimpleIntrestAmount(dto);  
        return result; (m)  
    }  
}
```

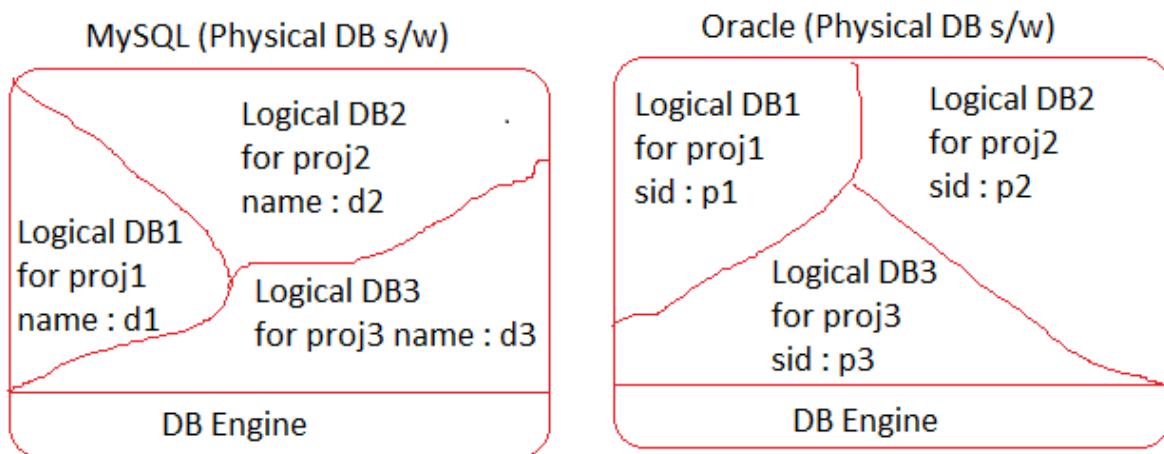
### RealtimeDITest.java (ClientApp)

```
-----  
public class RealtimeDI{  
    public static void main(String[] args){  
        //read inputs from enduser using scanner  
        .....  
        //store inputs into VO class Object  
        .....  
        //create BeanFactory Container  
        .....  
        //get Controller class object (b) (1)  
        (16) MainController controller = factory.getBean("Controller",  
                                            MainController.class);  
        //invoke methods  
        try {  
            (n) String result = controller.processCustomer(vo);  
            S.o.p(result); (o)  
        catch(Exception e) {  
            S.o.P("Internal Problem:"+e.getMessage());  
            e.printStackTrace();  
        }  
    } //main  
}
```

## MySQL:

- type: DB s/w
- version: 8.x
- vendor: devx/ Sun Ms/oracle corp
- default: 3306
- admin username: root
- password: root (choose during installation)
- To download s/w: <https://www.mysql.com.download/> [Download]
- Gives MySQL workbench as GUI DB (built-in tool)

## Physical DB s/w vs Logical DB:



- Apartment: Physical DB s/w, Flats: Logical DB.
- Oracle JDBC driver s/w (ojdbc6/ 7/ 8.jar).
- To interact with MySQL DB s/w, we need MySQL connector/j JDBC driver and it comes in from of jar file (mysql-connector-java-<ver>.jar)
- DBA will create these logical Databases.

## Connector/j JDBC driver details:

Driver class name: com.mysql.cj.jdbc.Driver (or) com.mysql.jdbc.Driver (or)  
org.gjt.mm.mysql.Driver Implements java.sql.Driver ()

JDBC URL:

jdbc:mysql://<Logical DB> (for local MySQL DB s/w)  
e.g. jdbc:mysql:///nssp3721db  
jdbc:mysql://<host>:<port>/<LogicalDB> (for Remote MySQL DB s/w)  
e.g. jdbc:mysql://123:55:342:3306/nssp3022db

Jar file: mysql-connctor-java-<ver>.jar

(Separate download from internet or use gradle/ maven dependency management)

Procedure to develop the above layered application using Gradle:

**Step 1:** make sure that gradle build ship plugin is available in Eclipse.

**Step 2:** create Gradle Project.

File --> Project --> gradle --> gradle project --> next --> fill the details

Project name: IOCProj13-RealtimeDI

choose gradle installation folder

select Auto synchronization project

next --> finish.

**Step 3:** add the following dependencies info (jars info) in dependencies {}

enclosure by collecting them from mvnrepository.com [\[Go\]](#)

spring-context-support-<ver>.jar ojdbc6.jar spring-jdbc-<ver>.jar	build.gradle
---	--------------

**Step 4:** Develop resources as below by creating packages in src/main/java folder.

IOCProj13-LayeredApp

```
|---> src/main/java
|   |---> com.nt.vo
|   |   |---> CustomerVO.java
|   |---> com.nt.dto
|   |   |---> CustomerDTO.java
|   |---> com.nt.bo
|   |   |---> CustomerBO.java
|   |---> com.nt.cfgs
|   |   |---> applicationContext.xml
|   |---> com.nt.dao
|   |   |---> CustomerDAO.java
|   |   |---> CustomerDAOImpl.java
|   |---> com.nt.service
|   |   |---> CustomerMgmtService.java
|   |   |---> CustomerMgmtServiceImpl.java
|   |---> com.nt.controller
|   |   |---> MainController.java
|   |---> com.nt.test
|   |   |---> RealTimeDITest.java
|---> build.gradle
```

**Step 5:** Keep DB table ready in Oracle DB s/w.

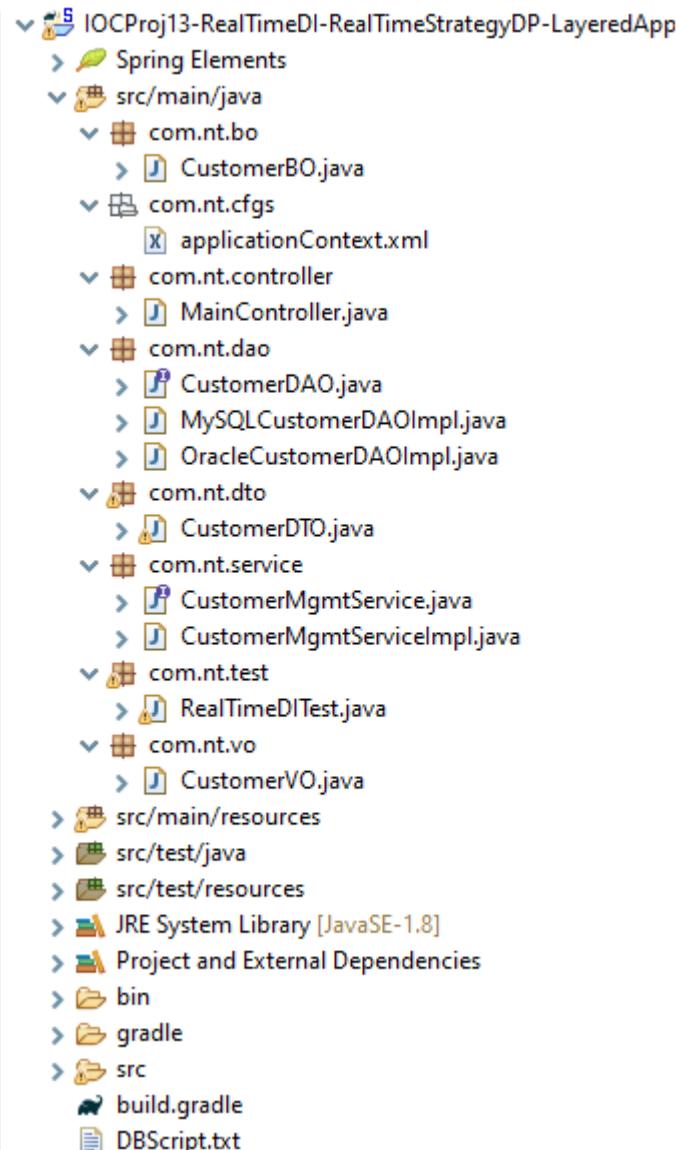
i. create connection in SQL developer

ii. create DB table

iii. create sequence to generate "CNO" column dynamically

**Step6:** Develop the code and run the Client App either directly or using gradle.

### Directory Structure of IOCProj13-RealTimeDI-RealTimeStrategyDP-LayeredApp:



- Develop the above directory structure and package, class, XML file and add the jar dependencies in build.gradle file and ready the DB table, sequence using DBScript.txt file then use the following code with in their respective file.
- Here we used to DB software Oracle and MySQL so that you have to ready the table in Oracle as well as in MySQL, create sequence in Oracle but in MySQL there is no Sequence concept so enable Auto Increment [AI].
- Now onwards we will develop each and every project using Gradle.

## DBScript.txt

-----  
Oracle  
-----

Table Detail  
-----

```
CREATE TABLE "SYSTEM"."CUSTOMER"
 ("CNO" NUMBER(*,0) NOT NULL ENABLE,
  "CNAME" VARCHAR2(20 BYTE),
  "CADD" VARCHAR2(20 BYTE),
  "PAMT" FLOAT(126),
  "INTERAMT" FLOAT(126),
  CONSTRAINT "CUSTOMER_PK" PRIMARY KEY ("CNO"));
```

Sequence Detail  
-----

```
CREATE SEQUENCE "SYSTEM"."CNO_SEQ"
  MINVALUE 101
  MAXVALUE 1000
  INCREMENT BY 1
  START WITH 101
  CACHE 20 ORDER NOCYCLE ;
```

-----  
MySQL  
-----

Table Detail  
-----

```
CREATE TABLE `nssp713db`.`customer` (
  `cno` INT NOT NULL AUTO_INCREMENT,
  `cname` VARCHAR(45) NULL,
  `cadd` VARCHAR(45) NULL,
  `pamt` FLOAT NULL,
  `interamt` FLOAT NULL,
  PRIMARY KEY (`cno`));
```

## build.gradle

```
plugins {
    // Apply the java-library plugin to add support for Java Library
    id 'application'
}

mainClassName = 'com.nt.test.RealTimeDITest'

run {
    standardInput = System.in
}

repositories {
    mavenCentral()
}

dependencies {
    // https://mvnrepository.com/artifact/org.springframework/spring-
    context-support
    implementation group: 'org.springframework', name: 'spring-context-
    support', version: '5.2.8.RELEASE'

    // https://mvnrepository.com/artifact/org.springframework/spring-
    jdbc
    implementation group: 'org.springframework', name: 'spring-jdbc',
    version: '5.2.8.RELEASE'

    //
    https://mvnrepository.com/artifact/com.oracle.database.jdbc/ojdbc6
    implementation group: 'com.oracle.database.jdbc', name: 'ojdbc6',
    version: '11.2.0.4'

    // https://mvnrepository.com/artifact/mysql/mysql-connector-java
    implementation group: 'mysql', name: 'mysql-connector-java',
    version: '8.0.21'
}
```

## CustomerVO.java

```
package com.nt.vo;

public class CustomerVO {

    // All bean properties
    private String cname;
    private String cadd;
    private String pAmt;
    private String time;
    private String rate;

    // Setters and getters
    public String getCname() {
        return cname;
    }
    public void setCname(String cname) {
        this.cname = cname;
    }
    public String getCadd() {
        return cadd;
    }
    public void setCadd(String cadd) {
        this.cadd = cadd;
    }
    public String getpAmt() {
        return pAmt;
    }
    public void setpAmt(String pAmt) {
        this.pAmt = pAmt;
    }
    public String getTime() {
        return time;
    }
    public void setTime(String time) {
        this.time = time;
    }
    public String getRate() {
        return rate;
    }
}
```

```
    public void setRate(String rate) {
        this.rate = rate;
    }

}
```

### CustomerDTO.java

```
package com.nt.dto;

import java.io.Serializable;

public class CustomerDTO implements Serializable {

    //All bean properties
    private String cname;
    private String cadd;
    private float pAmt;
    private float rate;
    private float time;

    //Setters and getters
    public String getCname() {
        return cname;
    }
    public void setCname(String cname) {
        this.cname = cname;
    }
    public String getcAdd() {
        return cadd;
    }
    public void setcAdd(String cadd) {
        this.cadd = cadd;
    }
    public float getpAmt() {
        return pAmt;
    }
    public void setpAmt(float pAmt) {
        this.pAmt = pAmt;
    }
}
```

```
public float getRate() {
    return rate;
}
public void setRate(float rate) {
    this.rate = rate;
}
public float getTime() {
    return time;
}
public void setTime(float time) {
    this.time = time;
}

}
```

### CustomerBO.java

```
package com.nt.bo;

public class CustomerBO {

    //all bean property
    private String cname;
    private String cadd;
    private float pAmnt;
    private float interAmt;

    //setters and getters
    public String getCname() {
        return cname;
    }
    public void setCname(String cname) {
        this.cname = cname;
    }
    public String getCadd() {
        return cadd;
    }
    public void setCadd(String cadd) {
        this.cadd = cadd;
    }
}
```

```

public float getpAmnt() {
    return pAmnt;
}
public void setpAmnt(float pAmnt) {
    this.pAmnt = pAmnt;
}
public float getInterAmt() {
    return interAmt;
}
public void setInterAmt(float interAmt) {
    this.interAmt = interAmt;
}

}

```

#### CustomerDAO.java

```

package com.nt.dao;

import com.nt.bo.CustomerBO;

public interface CustomerDAO {
    public int insert(CustomerBO bo) throws Exception;
}

```

#### OracleCustomerDAOImpl.java

```

package com.nt.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;

import javax.sql.DataSource;

import com.nt.bo.CustomerBO;

public final class OracleCustomerDAOImpl implements CustomerDAO {

    private static final String CUSTOMER_INSERT_QUERY = "INSERT INTO
    CUSTOMER VALUES(CNO_SEQ.NEXTVAL,?, ?, ?, ?)";
}

```

```

private DataSource ds;

public OracleCustomerDAOImpl(DataSource ds) {
    this.ds = ds;
}

@Override
public int insert(CustomerBO bo) throws Exception {
    Connection con = null;
    PreparedStatement ps = null;
    int count = 0;
    //get JDBC connection pool object
    con = ds.getConnection();
    //Create prepareStatement object
    ps = con.prepareStatement(CUSTOMER_INSERT_QUERY);
    //set the value to query param
    ps.setString(1, bo.getcname());
    ps.setString(2, bo.getcadd());
    ps.setFloat(3, bo.getpAmnt());
    ps.setFloat(4, bo.getInterAmt());
    //execute the query
    count = ps.executeUpdate();
    //close JDBC objects
    ps.close();
    con.close();
    return count;
}

}

```

#### MySQLCustomerDAOImpl.java

```

package com.nt.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;

import javax.sql.DataSource;

```

```

import com.nt.bo.CustomerBO;

public final class OracleCustomerDAOImpl implements CustomerDAO {

    private static final String CUSTOMER_INSERT_QUERY = "INSERT INTO
CUSTOMER VALUES(CNO_SEQ.NEXTVAL,?, ?, ?, ?)";

    private DataSource ds;

    public OracleCustomerDAOImpl(DataSource ds) {
        this.ds = ds;
    }

    @Override
    public int insert(CustomerBO bo) throws Exception {
        Connection con = null;
        PreparedStatement ps = null;
        int count = 0;
        //get JDBC connection pool object
        con = ds.getConnection();
        //Create prepareStatement object
        ps = con.prepareStatement(CUSTOMER_INSERT_QUERY);
        //set the value to query param
        ps.setString(1, bo.getcname());
        ps.setString(2, bo.getcadd());
        ps.setFloat(3, bo.getpAmnt());
        ps.setFloat(4, bo.getInterAmt());
        //execute the query
        count = ps.executeUpdate();
        //close JDBC objects
        ps.close();
        con.close();
        return count;
    }

}

```

### CustomerMgmtService.java

```
package com.nt.service;

import com.nt.dto.CustomerDTO;

public interface CustomerMgmtService {

    public String calculateSimpleInterestAmount(CustomerDTO dto)
throws Exception;

}
```

### CustomerMgmtServiceImpl.java

```
package com.nt.service;

import com.nt.bo.CustomerBO;
import com.nt.dao.CustomerDAO;
import com.nt.dto.CustomerDTO;

public final class CustomerMgmtServiceImpl implements
CustomerMgmtService {

    private CustomerDAO dao;

    public CustomerMgmtServiceImpl(CustomerDAO dao) {
        super();
        this.dao = dao;
    }

    @Override
    public String calculateSimpleInterestAmount(CustomerDTO dto)
throws Exception {
        float interAmt = 0.f;
        CustomerBO bo = null;
        int count = 0;
        //Calculate the simple interest amount from DTO
        interAmt = (dto.getpAmt()*dto.getTime()*dto.getRate())/100;
        //Prepare CustomerBO having persist able Data
        bo = new CustomerBO();
    }
}
```

```

        bo.setCname(dto.getCname());
        bo.setCadd(dto.getCadd());
        bo.setpAmnt(dto.getpAmt());
        bo.setInterAmt(interAmt);
        //use the dao
        count = dao.insert(bo);
        if (count==0)
            return "Customer registration failed - Insert amount is :
"+interAmt;
        else
            return "Customer registration succeded - Insert amount
is : "+interAmt;
    }

}

```

### ManiController.java

```

package com.nt.service;

import com.nt.bo.CustomerBO;
import com.nt.dao.CustomerDAO;
import com.nt.dto.CustomerDTO;

public final class CustomerMgmtServiceImpl implements
CustomerMgmtService {

    private CustomerDAO dao;

    public CustomerMgmtServiceImpl(CustomerDAO dao) {
        super();
        this.dao = dao;
    }

    @Override
    public String calculateSimpleInterestAmount(CustomerDTO dto)
throws Exception {
        float interAmt = 0.f;
        CustomerBO bo = null;

```

```

int count = 0;
//Calculate the simple interest amount from DTO
interAmt = (dto.getpAmt()*dto.getTime()*dto.getRate())/100;
//Prepare CustomerBO having persist able Data
bo = new CustomerBO();
bo.setCname(dto.getCname());
bo.setCadd(dto.getcadd());
bo.setpAmnt(dto.getpAmt());
bo.setInterAmt(interAmt);
//use the dao
count = dao.insert(bo);
if (count==0)
    return "Customer registration failed - Insert amount is :
"+interAmt;
else
    return "Customer registration succeded - Insert amount
is : "+interAmt;
}

}

```

### 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-4.3.xsd">

    <!-- Configure the DataSource for Oracle-->
    <bean id="oracleDmnds"
          class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url"
value="jdbc:oracle:thin:@localhost:1521:xe"/>
        <property name="username" value="system"/>
        <property name="password" value="manager"/>
    </bean>

```

```

<!-- Configure the DataSource for MySQL-->
<bean id="mysqlDmnds"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
value="com.mysql.cj.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql:///nssp713db"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
</bean>

<!-- Configure Oracle DataSoruce object to Oracle DAO class -->
<bean id="oracleCustDAO"
class="com.nt.dao.OracleCustomerDAOImpl">
    <constructor-arg ref="oracleDmnds"/>
</bean>

<!-- Configure MySQL DataSoruce object to MySQL DAO class -->
<bean id="mysqlCustDAO"
class="com.nt.dao.MySQLCustomerDAOImpl">
    <constructor-arg ref="mysqlDmnds"/>
</bean>

<!-- Configure DAO object to Service class -->
<bean id="custService"
class="com.nt.service.CustomerMgmtServiceImpl">
    <!-- <constructor-arg ref="oracleCustDAO"/> -->
    <constructor-arg ref="mysqlCustDAO"/>
</bean>

<!-- Configure Service object to Controller class -->
<bean id="controller" class="com.nt.controller.MainController">
    <constructor-arg ref="custService"/>
</bean>

</beans>

```

## RealTimeDITest.java

```
package com.nt.test;

import java.util.Scanner;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.support.BeanDefinitionRegistry;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;

import com.nt.controller.MainController;
import com.nt.vo.CustomerVO;

public class RealTimeDITest {

    public static void main(String[] args) {
        Scanner sc = null;
        String name=null, address=null, Amount=null, time=null,
rate=null;
        CustomerVO vo = null;
        DefaultListableBeanFactory factory = null;
        XmlBeanDefinitionReader reader = null;
        MainController controller = null;
        String result = null;
        //Read inputs from end-user using scanner
        sc = new Scanner(System.in);
        System.out.println("Enter the Details for registration: ");
        System.out.print("Enter Customer Name : ");
        name = sc.next();
        System.out.print("Enter Customer Address : ");
        address = sc.next();
        System.out.print("Enter Customer Principle Amount : ");
        Amount = sc.next();
        System.out.print("Enter Customer Time : ");
        time = sc.next();
        System.out.print("Enter Customer Rate of Interest: ");
        rate = sc.next();
        //Store into VO class object
        vo = new CustomerVO();
```

```

        vo.setCname(name);
        vo.setCadd(address);
        vo.setpAmt(Amount);
        vo.setTime(time);
        vo.setRate(rate);
        //Create BeanFactory [IoC] container
        factory = new DefaultListableBeanFactory();
        reader = new XmlBeanDefinitionReader(factory);

        reader.loadBeanDefinitions("com/nt/cfgs/applicationContext.xml");
        //get controller class object
        controller = factory.getBean("controller",
MainController.class);
        //invoke methods
        try {
            result = controller.processCustomer(vo);
            System.out.println(result);
        } catch (Exception e) {
            System.out.println("Internal problem :
"+e.getMessage());
            e.printStackTrace();
        }
    } //main

} //class

```

After development run the project as it will run smoothly, but for run as gradle project we have to follow the below process.

#### To run Layered Application in Gradle environment:

- place the following additional lines of code in build.gradle

```

mainClassName="com.nt.RealTimeDITest"
run {
    standardInput = System.in
}

```

- Place com.nt.cfgs packages that is applicationContext.xml file in src/main/resources folder

- Run the Application

Go Gradle tasks ---> Project (our) ---> application run ---> Run Gradle task go to console and give inputs then it will give the output.

**Q. Where did you use Dependency injection or Strategy DP in your real project development?**

**Ans.** Every project is layered application having DataSource, DAO, Service, Controller and etc. classes.

Standalone layered application

Client ---> Controller class ---> Service class ---> DAO class ---> DB s/w

Web based layered application

HTML/JSP(UI) -> Controller class -> Service class -> DAO class -> DB s/w

 If these applications are developed in spring environment, we use dependency injection (mostly constructor injection) based on strategy Design pattern for the following injections.

- DataSource will be injected to DAO.
- DAO will be injected to Service.
- Service will be injected to Controller.

## Collection Injection

- It is all about injecting values to array, collection type bean properties through Dependency Injection, for different tags in Spring bean configuration file.

Property type	tag or attribute
simple/ primitive (including String)	value attribute or <value> tag
Object type/ reference type	ref attribute or <ref> tag
array	<array> or <list>
java.util.List	<list>
java.util.Set	<set>
java.util.Map	<map>
java.util.Properties	<props>

### Spring Bean properties:

- A Spring Bean can have 4 types bean Properties
  - a. Simple bean properties  
(Primitive data type/ wrapper data type/ Spring bean properties)
  - b. Object type/ reference type bean properties
  - c. Array type bean properties
  - d. Collection type bean properties

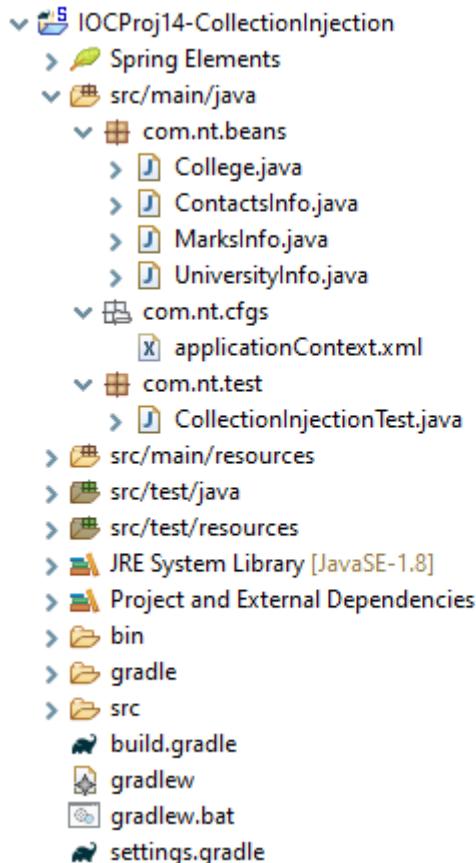
## Array properties:

- Arrays are homogeneous.
- Arrays elements need memory continuously.
- Arrays are having fixed size.
- we can access array elements through index (0 based index).
- When we removed element values from array, its size will not be decreased.
- Arrays are objects in java having one built-in property called length.
- Preserves the insertion order of the array elements

## Injecting values to array type bean properties:

- If the array is simple/ string data type then use <array> with <value> tag to perform Injection to elements.
- If the array is reference/ object data type then use <array> with <ref> tag with bean attribute to perform injection to elements.

## Directory Structure of IOCProj14-CollectionInjection



- Develop the above directory structure and package, class, XML file and add the dependencies to build.gradle, then use the following code with in their respective file.
- ⊕ For injecting array value, we have to use the following codes in their file.

## build.gradle

```
plugins {
    // Apply the java-library plugin to add support for Java Library
    id 'application'
}

mainClassName = 'com.nt.test.CollectionInjectionTest'

repositories {
    mavenCentral()
}

dependencies {
    // https://mvnrepository.com/artifact/org.springframework/spring-
    context-support
    implementation group: 'org.springframework', name: 'spring-context-
    support', version: '5.2.8.RELEASE'
}
```

## MarksInfo.java

```
package com.nt.beans;

import java.util.Arrays;
import java.util.Date;

public class MarksInfo {

    //beans properties
    private int marks[];
    private Date dates[];

    public void setDates(Date[] dates) {
        this.dates = dates;
    }

    public void setMarks(int[] marks) {
        this.marks = marks;
    }
}
```

```

@Override
public String toString() {
    return "MarksInfo [marks=" + Arrays.toString(marks) +
"\ndates=" + Arrays.toString(dates) + "]";
}

}

```

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

    <!-- Some Date beans -->
    <bean id="sysDate" class="java.util.Date"/>
    <bean id="dobDate" class="java.util.Date">
        <property name="year" value="90"/>
        <property name="month" value="11"/>
        <property name="date" value="24"/>
    </bean>

    <!-- Inject Array to the Bean Class -->
    <bean id="marksBean" class="com.nt.beans.MarksInfo">
        <!-- Inject Simple type -->
        <property name="marks">
            <array value-type="java.lang.Integer">
                <value>10</value>
                <value>20</value>
                <value>30</value>
            </array>
        </property>

        <!-- Inject Object/ reference type -->
        <property name="dates">
            <array value-type="java.util.Date">
                <ref bean="sysDate"/>

```

```
        <ref bean="dobDate"/>
    </array>
</property>
</bean>

</beans>
```

### CollectionInjection.java

```
package com.nt.test;

import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;

import com.nt.beans.College;
import com.nt.beans.ContactsInfo;
import com.nt.beans.MarksInfo;
import com.nt.beans.UniversityInfo;

public class CollectionInjectionTest {

    public static void main(String[] args) {
        DefaultListableBeanFactory factory = null;
        XmlBeanDefinitionReader reader = null;
        MarksInfo marks = null;
        College college = null;
        ContactsInfo contact = null;
        UniversityInfo university = null;
        //Create IoC container
        factory = new DefaultListableBeanFactory();
        reader = new XmlBeanDefinitionReader(factory);
        reader.loadBeanDefinitions("com/nt/cfgs/applicationContext.xml");
        //get MarksInfo class object
        marks = factory.getBean("marksBean", MarksInfo.class);
        System.out.println(marks);
        System.out.println("-----");
    }
}
```

### Internal code:

```
//Create Bean class object  
MarksInfo mi = (MarksInfo)  
    Class.forName("com.nt.beans.MarksInfo").newInstance();  
int[] marks = new int[]{10,20,30};  
mi.setMarks(marks);
```

### Why should we go for Collections?

- Collections are heterogeneous.
- Collections are dynamically growable.
- Collection elements does not allocate memory continuously.
- Provide lots of built-in methods to access and manipulate elements values.
- Internally uses multiple Data Structure algorithms to manage data.
- Collections with generics makes our code as type safe code (No type casting issues).
- If we insert simple values, they will be converted into wrapper objects (autoboxing).
- and etc.

### List Collection properties:

- Insertion order preserved.
- Duplicate are allowed.
- Can access elements through index (0- based).
- Null insertion possible.
- List Collection are ArrayList, Vector, Stack, LinkedList.

### Injecting value to java.util.List Collection:

- If the List is simple/ string data type then use <list> with <value> tag to perform Injection to elements.
  - If the List is reference/ object data type then use <list> with <ref> tag with bean attribute to perform injection to elements.
- ⊕ For injecting List collection value, we have to use the following codes in their file.
- ⊕ We have to add the code with the previous codes in some file like applicationContext.xml, CollectionInjectionTest.java.
- ⊕ Don't remove these files code just and with the existing code.

## College.java

```
package com.nt.beans;

import java.util.Date;
import java.util.List;

public class College {

    //bean properties
    private List<String> studName;
    private List<Date> joinDate;

    public void setStudName(List<String> studName) {
        System.out.println(studName.getClass());
        this.studName = studName;
    }

    public void setJoinDate(List<Date> joinDate) {
        System.out.println(joinDate.getClass());
        this.joinDate = joinDate;
    }

    @Override
    public String toString() {
        return "College [studName=" + studName + "\njoinDate=" +
joinDate + "]";
    }
}
```

## applicationContext.xml

```
<!-- Inject List Collection -->
<bean id="collegeBean" class="com.nt.beans.College">
    <!-- Inject Simple type -->
    <property name="studName">
        <list value-type="java.lang.String">
            <value>ram</value>
            <value>hari</value>
            <value>Lab</value>
        </list>
    
```

```

</property>

<!-- Inject Object/ reference type -->
<property name="joinDate">
    <list value-type="java.util.Date">
        <ref bean="sysDate"/>
        <ref bean="dobDate"/>
    </list>
</property>
</bean>

```

### CollectionInjectionTest.java

```

//get College class obejct
college = factory.getBean("collegeBean", College.class);
System.out.println(college);
System.out.println("-----");

```

#### Note:

- ✓ If the bean property type is java.util.List then the IoC container internally uses java.util.ArrayList tpye List Collection and we can't change it.
- ✓ If want to work with other List Collection like Vector, Stack and etc. then take concrete class type bean properties as shown below

```

private Vector<String> studName;
private Stack<String> studName;

```

#### Internal code for List<Date> dateList property:

```

//Create Bean class object
College Clg = (College)
Class.forName("com.nt.beans.College").newInstance ();
//Dependent object creation
Date sysDate = new Date ();
Date dobDate = new Date();
dobDate.setYear(1900+90);
dobDate.setMonth(11);
dobDate.setDate(22);

```

```

List<Date> datesList = new ArrayList ();
datesList.add(sysDate);
datesList.add(dobDate);
//Setter injection
cfg.setDatesList(datesList);

```

### Set Collection Properties:

- Doesn't allow duplicate.
- Doesn't maintain indexes for elements.
- Insertion order not preserved, to preserved use LinkedHashSet.
- Allows only one null.

### Injecting values into java.util.Set type Collection:

- If the Set is simple/ string data type then use <set> with <value> tag to perform Injection to elements.
  - If the Set is reference/ object data type then use <set> with <ref> tag with bean attribute to perform injection to elements.
-  For injecting Set collection value, we have to use the following codes in their file.
-  We have to add the code with the previous codes in some file like applicationContext.xml, CollectionInjectionTest.java.
-  Don't remove these files code just and with the existing code.

### ContactsInfo.java

```

package com.nt.beans;

import java.util.Date;
import java.util.Set;

public class ContactsInfo {
    //bean properties
    private Set<Long> phoneNumber;
    private Set<Date> dates;

    public ContactsInfo(Set<Long> phoneNumber, Set<Date> dates) {
        System.out.println(phoneNumber.getClass()+"\n"+dates.getClass());
        this.phoneNumber = phoneNumber;
    }
}

```

```

        this.dates = dates;
    }

@Override
public String toString() {
    return "ContactsInfo [phoneNumber=" + phoneNumber +
"\ndates=" + dates + "]";
}

}

```

### applicationContext.xml

```

<!-- Inject Set Collection -->
<bean id="contactBean" class="com.nt.beans.ContactsInfo">
    <!-- Inject Simple type -->
    <constructor-arg name="phoneNumber">
        <set value-type="java.lang.Long">
            <value>9999999999</value>
            <value>8888888888</value>
            <value>7777777777</value>
            <value>9999999999</value>
        </set>
    </constructor-arg>
    <!-- Inject Object/ reference type -->
    <constructor-arg name="dates">
        <set value-type="java.util.Date">
            <ref bean="sysDate"/>
            <ref bean="dobDate"/>
            <ref bean="dobDate"/>
        </set>
    </constructor-arg>
</bean>

```

### CollectionInjectionTest.java

```

//get ContactsInfo class object
contact = factory.getBean("contactBean", ContactsInfo.class);
System.out.println(contact);
System.out.println("-----");

```

### Internal code of Set injecting using construction injection:

```
//Creates class objects
Date sysDate = (Date)Class.forName("java.util.Date").newInstance ();
Date dobDate = (Date)Class.forName("java.util.Date").newInstance ();
dobDate.setYear(90);
dobDate.setMonth(11);
dobDate.setDate(22);
//Create set Collection having given data
Set<long> phoneNumbers = new LinkedHashSet();
phoneNumbers.add(9999999999);
phoneNumbers.add(8888888888);
phoneNumbers.add(7777777777);
Set<Date> dates = new LinkedHashSet();
dates.add(sysDate);
dates.add(dobDate);
//Create target class object
Class c = Class.forName("com.nt.beans.ContactsInfo");
Constructor cons[] = c.getDeclaredConstructors();
ContactsInfo cinfo = (ContactsInfo)
    cons[0].newInstance(phoneNumbers, dates);
```

### Map Collection properties:

- Can maintain elements having key-values Paris.
- Keys cannot be duplicated, but values can be duplicated.
- Insertion order not preserved by default to preserve use LinkedHashMap.
- Keys cannot be null, but values can be null.
- Implementations are HashMap, HashTable, LinkedHashMap and etc.
- Key can be any object and value can be any object.

### Injecting values into java.util.Map type Collection:

- Use <map> tags with <entry> tags for injecting values.
- In <entry> tag
  - for simple keys use either <key> with <value> tags or key attribute.
  - for simple values use <value> tag or "value" attribute.
  - for object type keys use either <key> with <ref> tags or key-ref attribute.
  - for object type values use either <ref> tag or value-ref attribute

### Properties Collection properties:

- It is a Map Collection.
- Sub class of HashTable.
- Here elements allow only String as keys and String as values.
- No generics is required here.
- Can load element values (keys, values) from external properties file (text file).
- Keys cannot be duplicated, but values can be.
- Insertion order is not Preserved.

### Injecting values into java.util.Properties type Collection:

- Use <props> tags with <pro> tags for injecting values.
  - For injecting Map and Properties collection value, we have to use the following codes in their file.
  - We have to add the code with the previous codes in some file like applicationContext.xml, CollectionInjectionTest.java.
  - Don't remove these files code just and with the existing code.

#### UniversityInfo.java

```
package com.nt.beans;

import java.util.Date;
import java.util.Map;
import java.util.Properties;

public class UniversityInfo {

    private Map<Long, String> facultyDetails;
    private Map<String, Date> dates;
    private Properties fruits;

    public void setFacultyDetails(Map<Long, String> facultyDetails) {
        System.out.println(facultyDetails.getClass());
        this.facultyDetails = facultyDetails;
    }

    public void setDates(Map<String, Date> dates) {
        System.out.println(dates.getClass());
        this.dates = dates;
    }
}
```

```

public void setFruits(Properties fruits) {
    System.out.println(fruits.getClass());
    this.fruits = fruits;
}

@Override
public String toString() {
    return "UniversityInfo [facultyDetails=" + facultyDetails +
"\ndates=" + dates + "\nfruits=" + fruits + "]";
}

}

```

### applicationContext.xml

```

<!-- Inject Map Collection -->
<bean id="universityBean" class="com.nt.beans.UniversityInfo">
    <!-- Inject <Simple, Simple> type -->
    <property name="facultyDetails">
        <map key-type="java.lang.Long" value-
type="java.lang.String">
            <entry>
                <key><value>10001</value></key>
                <value>Ramesh</value>
            </entry>
            <entry key="10002" value="Hari"/>
            <entry key="10003" value="Hari"/>
        </map>
    </property>

    <!-- Inject <Simple, Object> type -->
    <property name="dates">
        <map key-type="java.lang.String" value-
type="java.util.Date">
            <entry>
                <key><value>Today</value></key>
                <ref bean="sysDate"/>
            </entry>
            <entry key="BirthDay" value-ref="dobDate"/>
        </map>
    </property>

```

```

</property>

<!-- Inject Properties collection -->
<property name="fruits">
    <props>
        <prop key="banana">yellow</prop>
        <prop key="graps">green</prop>

        <prop key="apple">red</prop>
        <prop key="mango">yellow</prop>
    </props>
</property>
</bean>

```

#### CollectionInjectionTest.java

```

//get ContactsInfo class object
university = factory.getBean("universityBean",
UniversityInfo.class);
System.out.println(university);

```

#### Collection Injection in Real-time DI Application:

DriverManagerDataSource (Spring API supplied Pre-defined class)

---> driverClassName	String type bean properties (simple bean properties)
---> url	
---> username *	
---> password	
---> ConnectionProperties   java.util.Properties (Collection ** type)	
---> allows to pass DB username, DB password having fixed key "user", "password"	

- ⊕ If \*\* is used to supply DB username, password there is no need of using \* properties.
- ⊕ If both are used \* properties values will override \*\* bean properties values.

## Directory Structure of IOCProj15-RealTimeDI-CollectionInjection:

- Copy paste the Layered application and change rootProject.name to IOCProj15-RealTimeDI-CollectionInjection in settings.gradle file.
- Need not to add any new file same structure as IOCProj13-RealTimeDI-RealTimeStrategyDP-LayeredApp.
- Add the following code in applicationContext.xml as part of the replacement for the particular section.

### applicationContext.xml

```
<!-- Configure the DataSource for Oracle-->
<bean id="oracleDmnds"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url"
value="jdbc:oracle:thin:@localhost:1521:xe"/>
    <!-- <property name="username" value="system"/>
        <property name="password" value="manager"/> -->
    <property name="connectionProperties">
        <props>
            <prop key="user">system</prop>
            <prop key="password">manager</prop>
        </props>
    </property>
</bean>
```

## Null Injection

- In Constructor Injection, all parameters must participate in injection process otherwise exception will be raised.
- If constructor param type is object/ reference type and we are not ready with value then we can go for Null injection i.e. injecting null value constructor param to satisfy the process.
- This is very handy (useful), while working with pre-defined classes as Spring bean that are having limited no. of overloaded constructors and no setter injection is available.

```
<constructor-arg><null/><constructor-arg/>
```

### Note:

- ✓ This null injection is not possible if param type is primitive data type.

- ✓ Do not look at Spring's Dependency injection to inject end user supplied non- technical inputs (either Scanner or using HTML from) to Spring bean properties. It is there to inject only programmer supplied technical input values (like JDBC properties) either from XML file or properties file and also inject on Spring bean class object another spring bean class object (like DS to DAO, DAO to Service and etc.).

### Directory Structure of IOCProj16-NullInjection:

```

└── IOCProj16-NullInjection
    ├── src/main/java
    │   └── com.nt.beans
    │       └── PersonInfo.java
    ├── src/main/resources
    │   └── applicationContext.xml
    ├── src/test/java
    └── src/test/resources
        └── JRE System Library [JavaSE-1.8]
        └── Project and External Dependencies
    ├── bin
    ├── gradle
    └── src
        └── build.gradle

```

- Develop the above directory structure and package, class, XML file and add the jar dependencies in build.gradle file then use the following code with in their respective file.

#### build.gradle

```

plugins {
    // Apply the java-library plugin to add support for Java Library
    id 'application'
}

repositories {
    // Use jcenter for resolving dependencies.
    // You can declare any Maven/Ivy/file repository here.
    jcenter()
}

dependencies {

```

```
// https://mvnrepository.com/artifact/org.springframework/spring-
context-support
implementation group: 'org.springframework', name: 'spring-context-
support', version: '5.2.8.RELEASE'
}
```

### PersonInfo.java

```
package com.nt.beans;

import java.util.Date;

public class PersonInfo {

    //Bean Properties
    private long aadharNumber;
    private String name;
    private Date dob;
    private Date doj;
    private Date dom;

    public PersonInfo(long aadharNumber, String name, Date dob, Date
doj, Date dom) {
        System.out.println("PersonInfo : PersonInfo(-,-,-,-)");
        this.aadharNumber = aadharNumber;
        this.name = name;
        this.dob = dob;
        this.doj = doj;
        this.dom = dom;
    }

    @Override
    public String toString() {
        return "PersonInfo [aadharNumber=" + aadharNumber + ", "
name=" + name + ", dob=" + dob + ", doj=" + doj
                + ", dom=" + dom + "]";
    }
}
```

## 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-4.3.xsd">

    <!-- Some Date Bean -->
    <bean id="dob" class="java.util.Date">
        <property name="year" value="99"/>
        <property name="month" value="11"/>
        <property name="date" value="17"/>
    </bean>
    <bean id="doj" class="java.util.Date">
        <property name="year" value="114"/>
        <property name="month" value="5"/>
        <property name="date" value="27"/>
    </bean>
    <bean id="dom" class="java.util.Date">
        <property name="year" value="117"/>
        <property name="month" value="5"/>
        <property name="date" value="6"/>
    </bean>

    <bean id="ramInfo" class="com.nt.beans.PersonInfo">
        <constructor-arg value="3456789022"/>
        <constructor-arg value="Ram"/>
        <constructor-arg ref="dob"/>
        <constructor-arg ref="doj"/>
        <constructor-arg ref="dom"/>
    </bean>

    <bean id="hariInfo" class="com.nt.beans.PersonInfo">
        <constructor-arg value="3456789022"/>
        <constructor-arg value="Ram"/>
        <constructor-arg ref="dob"/>
        <constructor-arg><null/></constructor-arg>
        <constructor-arg><null/></constructor-arg>
    </bean>
</beans>
```

## NullInjectionTest.java

```
package com.nt.test;

import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;

import com.nt.beans.PersonInfo;

public class NullInjectionTest {

    public static void main(String[] args) {
        DefaultListableBeanFactory factory = null;
        XmlBeanDefinitionReader reader = null;
        PersonInfo ram=null, hari=null;
        //Create Bean Factory IoC Container
        factory = new DefaultListableBeanFactory();
        reader = new XmlBeanDefinitionReader(factory);
        reader.loadBeanDefinitions("com/nt/cfgs/applicationContext.xml");
        //create PersonInfo class object
        ram = factory.getBean("ramInfo", PersonInfo.class);
        System.out.println(ram);
        System.out.println("-----");
        hari = factory.getBean("hariInfo", PersonInfo.class);
        System.out.println(hari);
    }
}
```

- ⊕ To make gradle Spring application collecting sources code and XML file from the packages of src/main/java folder location add the following code in build.gradle

```
sourceSets {
    main {
        resources {
            srcDirs = ["src/main/java"]
            includes = ["**/*.xml"]
        }
    }
}
```

**Q. Can we do Null injection while working with Setter injection?**

**Ans.** Yes, but not required because while working with Setter injection there is no mandatory to inject the value/ object to bean property. We can involve our choice properties in Setter injection, more over the reference type or object type bean properties by default holds null values.

```
<property name="dob"><null/></property>
```

## Bean Inheritance

- This is no way related to java classes level inheritance.
- It is XML file level inheritance across the Spring bean configuration to reuse bean properties related values injection.

**Note:** Both Setter injection and Constructor Injection supports bean Inheritance

**Problem:**

```
public class Car {  
    private String regNo;  
    private String engineNo;  
    private String model;  
    private String company;  
    private String type;  
    private int engineCC;  
    private String color;  
    private String owner;  
    private String fuelType;  
    //Setter and getter methods  
    .....  
    //toString()  
    .....  
}
```

### applicationContext.xml

```
<beans >  
    <bean id="rajaCar1" class="pkg.Car">  
        <property name="regNo" value="TS07EN8909"/>  
        <property name="enginNo" value="5461728829"/>  
        <property name="model" value="swift"/> *  
        <property name="company" value="Suzuki"/> *  
        <property name="type" value="hatachback"/> *
```

```

<property name="enginCC" value="1200"/> *
<property name="color" value=red"/>
<property name="owner" value="raja"/> *
<property name="fuelType" value="diesel"/> *

</bean>
<bean id="rajaCar2" class="pkg.Car">
    <property name="regNo" value="TS07EN8652"/>
    <property name="enginNo" value="5461343529"/>
    <property name="model" value="swift"/>
    <property name="company" value="Suzuki"/>
    <property name="type" value="hatachback"/>
    <property name="enginCC" value=" 1200"/>
    <property name="color" value="blue"/>
    <property name="owner" value="raja"/>
    <property name="fuelType" value="diesel"/>

</bean>
</beans>

```

In both Spring beans configurations “\*” bean property values are same but we are not able to use them across the multiple spring bean configurations of XML file.

### Solution 1: (Bean Inheritance):

```

<beans >
    <bean id="rajaCar1" class="pkg.Car">
        <property name="regNo" value="TS07EN8909"/>
        <property name="enginNo" value="5461728829"/>
        <property name="model" value="swift"/> *
        <property name="company" value="Suzuki"/> *
        <property name="type" value="hatachback"/> *
        <property name="enginCC" value="1200"/> *
        <property name="color" value=red"/>
        <property name="owner" value="raja"/> *
        <property name="fuelType" value="diesel"/> *

    </bean>
    <bean id="rajaCar2" class="pkg.Car" parent="rajaCar1">
        <property name="regNo" value="TS07EN8652"/>
        <property name="enginNo" value="5461343529"/>
        <property name="color" value=blue"/>
    </bean>

```

```
</beans>
```

### Solution 2: (improved solution 1):

```
<beans>
    <bean id="baseCar" class="pkg.Car" abstract="true">
        <property name="model" value="swift"/>
        <property name="company" value="Suzuki"/>
        <property name="type" value="hatchback"/>
        <property name="enginCC" value="1200"/>
        <property name="owner" value="raja"/>
        <property name="fuelType" value="diesel"/>
    </bean>
    <bean id="rajaCar1" class="pkg.Car" parent="baseCar">
        <property name="regNo" value="TS07EN8909"/>
        <property name="enginNo" value="5461728829"/>
        <property name="color" value="red"/>
    </bean>
    <bean id="rajaCar2" class="pkg.Car" parent="baseCar">
        <property name="regNo" value="TS07EN8652"/>
        <property name="enginNo" value="5461343529"/>
        <property name="color" value="blue"/>
    </bean>
</beans>
```

### client App

```
Car car1 = factory.getBean("baseCar", Car.class); //gives error
Car car2 = factory.getBean("rajaCar1", Car.class); //success
S.o.println(car2) //9 properties (6 inherited from "baseCar", 3 direct)
Car car3 = factory.getBean("rajaCar2", Car.class); //success
S.o.println(car3) //9 properties (6 inherited from "baseCar", 3 direct)
```

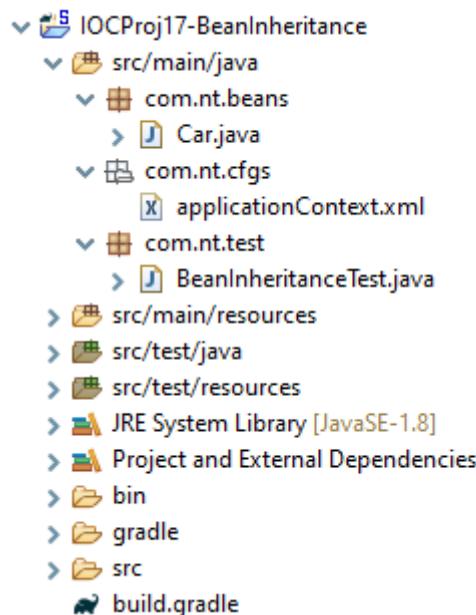
### Important points:

- a. This is not class level inheritance. It is Spring bean configuration file level bean properties inheritance across the multiple Spring bean configurations.
- b. <bean> tag abstract="true" does not make java class as abstract but makes Spring bean configuration as abstract i.e. we cannot call factory.getBean(-) having that bean id. We cannot inject this bean id-based spring bean class object to other Spring beans. Gives

Exception in thread "main"  
[org.springframework.beans.factory.BeanIsAbstractException](#): Error  
creating bean with name 'baseCar': Bean definition is abstract.

- c. One spring bean can inheritance and reuse bean properties only from 1 Spring bean.
- d. The class names in base bean configuration and child bean configuration can be the same or can be the different either participating in the inheritance or not participating in the inheritance.

### Directory Structure of IOCProj17-BeanInheritance



- Develop the above directory structure and package, class, XML file and add the jar dependencies in build.gradle file then use the following code with in their respective file.

### build.gradle

```
plugins {
    // Apply the java-library plugin to add support for Java Library
    id 'application'
}

repositories {
    mavenCentral()
}

dependencies {
```

```
// https://mvnrepository.com/artifact/org.springframework/spring-
context-support
implementation group: 'org.springframework', name: 'spring-context-
support', version: '5.2.8.RELEASE'
}
```

### Car.java

```
package com.nt.beans;

public class Car {

    //bean properties
    private String regNo;
    private String engineNo;
    private String model;
    private String company;
    private String type;
    private int engineCC;
    private String color;
    private String owner;
    private String fuelType;

    public void setRegNo(String regNo) {
        this.regNo = regNo;
    }
    public void setEngineNo(String engineNo) {
        this.engineNo = engineNo;
    }
    public void setModel(String model) {
        this.model = model;
    }
    public void setCompany(String company) {
        this.company = company;
    }
    public void setType(String type) {
        this.type = type;
    }
    public void setEngineCC(int engineCC) {
        this.engineCC = engineCC;
    }
}
```

```

    }
    public void setColor(String color) {
        this.color = color;
    }
    public void setOwner(String owner) {
        this.owner = owner;
    }
    public void setFuelType(String fuelType) {
        this.fuelType = fuelType;
    }

    //toString()
    @Override
    public String toString() {
        return "Car [regNo=" + regNo + ", engineNo=" + engineNo + ",
model=" + model + ", company=" + company
                + ", type=" + type + ", engineCC=" + engineCC + ",
color=" + color + ", owner=" + owner + ", fuelType="
                + fuelType + "]";
    }
}

```

### BeanInheritanceTest.java

```

package com.nt.test;

import
org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;

import com.nt.beans.Car;

public class BeanInheritanceTest {

    public static void main(String[] args) {
        DefaultListableBeanFactory factory = null;
        XmlBeanDefinitionReader reader = null;
        Car car1=null, car2=null, car3=null;
        //Create Bean Factory IoC Container
    }
}

```

```

factory = new DefaultListableBeanFactory();
reader = new XmlBeanDefinitionReader(factory);

reader.loadBeanDefinitions("com/nt/cfgs/applicationContext.xml");
//create PersonInfo class object
car1 = factory.getBean("rajaCar1", Car.class);
System.out.println(car1);
System.out.println("-----");
car2 = factory.getBean("rajaCar2", Car.class);
System.out.println(car2);
System.out.println("-----");
car3 = factory.getBean("baseCar", Car.class);
System.out.println(car3);
}

}

```

### Bean Inheritance can even through Constructor Injection:

```

<beans ....>
    <bean id="baseCar" class="com.nt.beans.Car" abstract="true">
        <constructor-arg name="model" value="swift"/>
        <constructor-arg name="company" value="Suzuki"/>
        <constructor-arg name="type" value="hatachback"/>
        <constructor-arg name="engineCC" value="1200"/>
        <constructor-arg name="owner" value="raja"/>
        <constructor-arg name="fuelType" value="diesel"/>
        <constructor-arg name="fuelType"><null/></constructor-arg>
    </bean>
    <bean id="rajaCar1" class="com.nt.beans.Car" parent="baseCar">
        <constructor-arg name="regNo" value="TS07EN8909"/>
        <constructor-arg name="engineNo" value="5461728829"/>
        <constructor-arg name="color" value="red"/>
    </bean>
    <bean id="rajaCar2" class="com.nt.beans.Car" parent="baseCar">
        <constructor-arg name="regNo" value="TS07EN83439"/>
        <constructor-arg name="engineNo" value="5434343829"/>
        <constructor-arg name="color" value="blue" />
        <constructor-arg name="owner" value="rabi"/>
    </bean>

```

```
</beans>
```

#### Directory Structure of IOCProj18-BeanInheritance-UsingConstructorInjection

- + Copy paste the IOCProj17-BeanInheritance and change rootProject.name to IOCProj18-BeanInheritance-UsingConstructorInjection in settings.gradle file.
- + Need not to add any new file, same structure as IOCProj18-BeanInheritance.
- + Change the following files code remain same as the previous.

#### Car.java

```
package com.nt.beans;

public class Car {

    //bean properties
    private String regNo;
    private String engineNo;
    private String model;
    private String company;
    private String type;
    private int engineCC;
    private String color;
    private String owner;
    private String fuelType;

    public Car(String regNo, String engineNo, String model, String
company, String type, int engineCC, String color,
String owner, String fuelType) {
        System.out.println("Car : Car(-,-,-,-,-,-,-)");
        this.regNo = regNo;
        this.engineNo = engineNo;
        this.model = model;
        this.company = company;
        this.type = type;
        this.engineCC = engineCC;
        this.color = color;
        this.owner = owner;
        this.fuelType = fuelType;
    }
}
```

```

//toString()
@Override
public String toString() {
    return "Car [regNo=" + regNo + ", engineNo=" + engineNo + ",
model=" + model + ", company=" + company
        + ", type=" + type + ", engineCC=" + engineCC + ",
color=" + color + ", owner=" + owner + ", fuelType="
        + fuelType + "]";
}
}

```

### applicationContext.xml

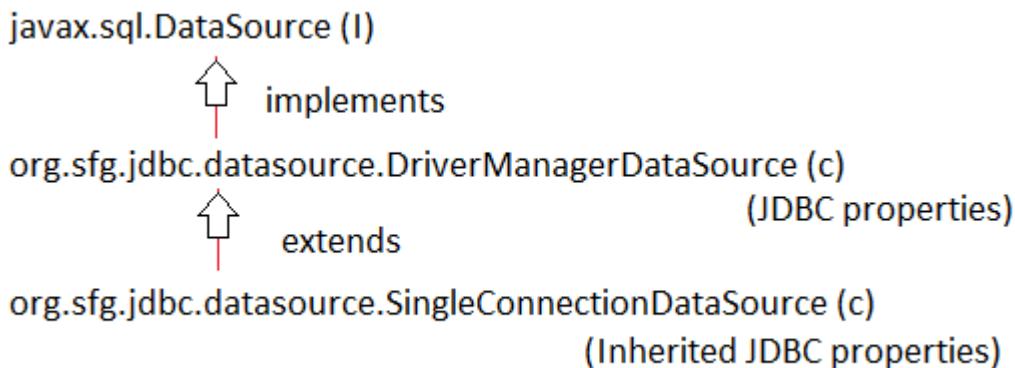
```

<?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-4.3.xsd">

    <bean id="baseCar" class="com.nt.beans.Car" abstract="true">
        <constructor-arg name="model" value="swift" />
        <constructor-arg name="company" value="Suzuki" />
        <constructor-arg name="type" value="hatchback" />
        <constructor-arg name="engineCC" value="1200" />
        <constructor-arg name="owner" value="raja" />
        <!-- <constructor-arg name="fuelType" value="diesel" /> -->
        <constructor-arg name="fuelType"><null/></constructor-arg>
    </bean>
    <bean id="rajaCar1" class="com.nt.beans.Car" parent="baseCar">
        <constructor-arg name="regNo" value="TS07EN8909" />
        <constructor-arg name="engineNo" value="5461728829" />
        <constructor-arg name="color" value="red" />
    </bean>
    <bean id="rajaCar2" class="com.nt.beans.Car" parent="baseCar">
        <constructor-arg name="regNo" value="TS07EN83439" />
        <constructor-arg name="engineNo" value="5434343829" />
        <constructor-arg name="color" value="blue" />
        <constructor-arg name="owner" value="rabi" />
    </bean>
</beans>

```

## Realtime Example of Bean inheritance:



Q. In which situation we can pass less than available number of bean property values for Constructor Injection?

Ans.

- While working with Null injection `<null/>`.
- While working with constructor-based bean inheritance.

## Directory Structure of IOCProj19-RealTimeDI-BeanInheritance:

- + Copy paste the Layered application and change `rootProject.name` to `IOCProj19-RealTimeDI-BeanInheritance` in `settings.gradle` file.
- + Need not to add any new file same structure as `IOCProj19-RealTimeDI-RealTimeStrategyDP-LayeredApp`.
- + Add the following code in `applicationContext.xml` as part of the replacement for the particular section.

### applicationContext.xml

```
<!-- Configure the SingleConnectionDataSource -->
<bean id="oracleScds"
      class="org.springframework.jdbc.datasource.SingleConnectionDataSource"
      parent="oracleDmnds"/>

<!-- Configure Oracle DataSoruce object to Oracle DAO class -->
<bean id="oracleCustDAO"
      class="com.nt.dao.OracleCustomerDAOImpl">
    <!-- <constructor-arg ref="oracleDmnds"/> -->
    <constructor-arg ref="oracleScds"/>
</bean>
```

- To find which class inject write the following line in implementation DAO class constructors, like below

### OracleCustomerDAOImpl.java

```
public OracleCustomerDAOImpl(DataSource ds) {  
    System.out.println(ds.getClass().getName());  
    this.ds = ds;  
}
```

Q. What is the JDBC connection pool or DataSource that you used in your spring project?

Ans.

- Do not use DriverManagerDataSource because this class is not an actual connection pool, it does not actually pool connections. It just servers as simple replacement of full-blown connection pool, implementing the same standard interface but creating new connections on every call (ds.getConnection()).
- Don't use SingleConnectionDataSource because it just creates only JDBC connection object and reuses that Connection object, reusing single JDBC connection object across the multiple requests coming to a web application raises Data inconsistency problems like if any requests call con.rollback() then the simultaneous requests related persistence operations will also be rollback. (This is not suitable in multithreaded standalone and web application environment).

Note:

- ✓ Both the above DataSource are not providing Connection pool parameters like initialPoolSize, maxPoolSize and etc.
- ✓ If your project is standalone project then use third party supplied standalone JDBC connection pool software like Apache DBCP, C3PO, Proxool, Vibur, Hikari CP (best) and etc.
- ✓ If your spring project is web application/ website then use underlying server managed JDBC connection pool, like WebLogic managed JDBC connection pool, Tomcat managed JDBC connection pool and etc.

HikariCP details:

- DataSource class name: com.zaxxer.hikari.HikariDataSource
- Jar file: HikariCP-<ver>.jar
- Gradle dependencies:

```
// https://mvnrepository.com/artifact/com.zaxxer/HikariCP  
implementation group: 'com.zaxxer', name: 'HikariCP', version: '3.4.5'
```

**Note:** In Real-time developers directly configure HikariDataSource with all its properties because that is the best DataSource in standalone environment.

- In the previous example add the following code with their respective file for HikariCP use.

### build.gradle

```
// https://mvnrepository.com/artifact/com.zaxxer/HikariCP  
implementation group: 'com.zaxxer', name: 'HikariCP', version: '3.4.5'
```

### applicationContext.xml

```
<!-- Configure HikariDataSource -->  
    <bean id="oracleHkds" class="com.zaxxer.hikari.HikariDataSource"  
parent="oracleDmds">  
        <property name="jdbcUrl"  
value="jdbc:oracle:thin:@localhost:1521:xe"/>  
        <property name="minimumIdle" value="10"/>  
        <property name="maximumPoolSize" value="20"/>  
        <property name="connectionTimeout" value="20000"/>  
    </bean>  
  
    <!-- Configure Oracle DataSoruce object to Oracle DAO class -->  
    <bean id="oracleCustDAO"  
class="com.nt.dao.OracleCustomerDAOImpl">  
        <!-- <constructor-arg ref="oracleDmds"/> -->  
        <!-- <constructor-arg ref="oracleScds"/> -->  
        <constructor-arg ref="oracleHkds"/>  
    </bean>
```

**Note:** In Realtime developers directly configures HikarDataSource with all its properties because that is the best DataSource in standalone environment.

## Collection Merging

- It should be used along with bean inheritance.
- It can be applied only on collection/ array type bean properties.
- It is all about adding more values Collection/ array type property in the child bean configuration by inheriting it from parent bean configuration.

```

public class EnggCourse {
    private Set<String> subjects;
    public void setSubjects(Set<String> subjects) {
        this.subjects=subjects;
    }
    //toString()
}

```

### applicationContext.xml

```

<beans ....>
    <bean id="base1stYear" class="com.nt.EnggCourse" abstract="true">
        <property name="subjects">
            <set>
                <value>C </value>
                <value>M1 </value>
                <value>English </value>
            </set>
        </property>
    </bean>
    <bean id="ec1stYear" class="pkg.EnggCourse">
        <property name="subjects">
            <set merge="true">
                <value> EDC </value>
                <value>NT</value>
            </set>
        </property>
    </bean>
</beans>

```

### Client App

```

EnggCourse engg=factory.getBean(" ec1stYear" , EnggCourse. class);
S.o.p(engg); //gives 4+2 = 6 subjects

```

### Collection Merging important point:

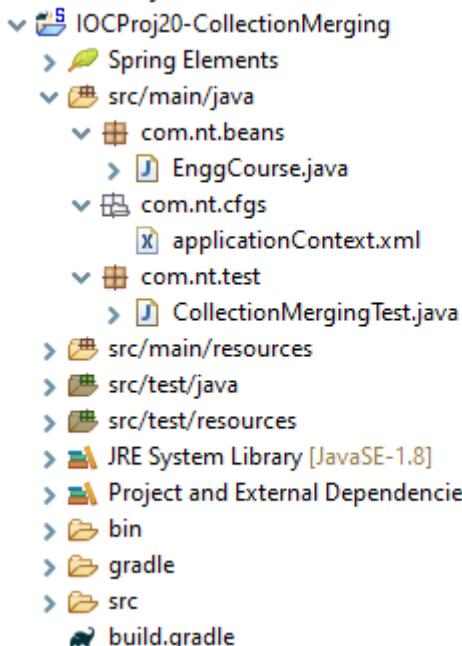
- Base/ parent bean Collection/ array property name and type must match with child bean Collection/ array property name and type.
- Collection Merging is possible only with Collection, array bean properties not on simple, object type bean properties.
- "merge" attribute is available only in <list>, <set>, <map>, <props>, <array> tags.

- The possible values for "merge" attitude are
  - a. False
  - b. true
  - c. default (default)
- Both setter, constructor injection supports the "collection merging".
  - merge="true" enables collection merging.
  - merge="false" disables collection merging.
  - merge="default" (default) fallbacks to "default-merge" attribute value of <beans> tag.

The possible values for default-merge attribute are:

- a. <beans default-merge="true">: enables collection merging on all collection properties of all spring beans belonging to current spring bean configuration file.
- b. <beans default-merge="false">: disables collection merging on all collection properties of all spring beans belonging to current spring bean configuration file.
- c. <beans default-merge="default">: The default is "default", indicating inheritance from outer 'bean' sections in case of nesting, otherwise falling back to "false".

#### Directory Structure of IOCProj20-CollectionMerging:



- Develop the above directory structure and package, class, XML file and add the jar dependencies in build.gradle file then use the following code with in their respect file.

## build.gradle

```
plugins {
    // Apply the java-library plugin to add support for Java Library
    id 'application'
}

repositories {
    // Use jcenter for resolving dependencies.
    // You can declare any Maven/Ivy/file repository here.
    jcenter()
}

dependencies {
    // https://mvnrepository.com/artifact/org.springframework/spring-
    context-support
    implementation group: 'org.springframework', name: 'spring-context-
    support', version: '5.2.8.RELEASE'
}
```

## EnggCourse.java

```
package com.nt.beans;

import java.util.Set;

public class EnggCourse {
    private Set<String> subjects;

    public void setSubjects(Set<String> subjects) {
        this.subjects = subjects;
    }

    @Override
    public String toString() {
        return "EnggCourse [subjects=" + subjects + "]";
    }
}
```

## applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans default-merge="true"
    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-4.3.xsd">

    <!-- Bean Configuration -->
    <bean id="base1stYear" class="com.nt.beans.EnggCourse"
abstract="true">
        <property name="subjects">
            <set>
                <value>C</value>
                <value>M1</value>
                <value>Engilsh</value>
                <value>Drawing</value>
            </set>
        </property>
    </bean>

    <!-- New Beans -->
    <bean id="mechanical1stYear" class="com.nt.beans.EnggCourse"
parent="base1stYear">
        <property name="subjects">
            <!-- <set merge="default"> -->
            <!-- <set merge="false"> -->
            <set merge="true">
                <value>FD</value>
                <value>Mechanics</value>
                <value>TD</value>
            </set>
        </property>
    </bean>
</beans>
```

**Q. Can we perform collection merging without taking merge: "true"?**

**Ans.** Possible but we need write default-merge: "true" in <beans> tag (at top of spring bean configuration file)

## CollectionMergingTest.java

```
package com.nt.test;

import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;

import com.nt.beans.EnggCourse;

public class CollectionMergingTest {

    public static void main(String[] args) {
        DefaultListableBeanFactory factory = null;
        XmlBeanDefinitionReader reader = null;
        EnggCourse course = null;
        //Create Bean Factory IoC Container
        factory = new DefaultListableBeanFactory();
        reader = new XmlBeanDefinitionReader(factory);

        reader.loadBeanDefinitions("com/nt/cfgs/applicationContext.xml");
        //get bean
        course = factory.getBean("mechanical1stYear",
        EnggCourse.class);
        System.out.println(course);
    }

}
```

## Default Bean Ids

- The IoC container generates default bean id for spring bean class if we do not provide any bean id to it.  
Generally, that is <fully qualified class name>

```
<bean class="com.nt.beans.EnggCourse
.....
.....
</bean>
```

The default bean id: "com.nt.beans.EnggCourse" (or)  
"com.nt.beans.EnggCourse#0"

## Directory Structure of IOCProj21-DefaultBeanIds:

- ⊕ Copy paste the IOCProj20-CollectionMerging application and change rootProject.name to IOCProj21-DefaultBeanIds in settings.gradle file.
- ⊕ Need not to add any new file same structure as IOCProj20-CollectionMerging.
- ⊕ Modify the following file with below line of code.

### applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans default-merge="true"
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-4.3.xsd">

    <!-- Bean Configuration -->
    <bean id="base1stYear" class="com.nt.beans.EnggCourse"
abstract="true">
        <property name="subjects">
            <set>
                <value>C</value>
                <value>M1</value>
                <value>Engilsh</value>
                <value>Drawing</value>
            </set>
        </property>
    </bean>

    <!-- New Beans -->
    <bean class="com.nt.beans.EnggCourse" parent="base1stYear">
        <property name="subjects">
            <set>
                <value>FD</value>
                <value>Mechanics</value>
                <value>TD</value>
            </set>
        </property>
    </bean>

    <bean class="com.nt.beans.EnggCourse" parent="base1stYear">
```

```

<bean class="com.nt.beans.EnggCourse" parent="base1stYear">
    <property name="subjects">
        <set>
            <value>EC</value>
            <value>DE</value>
            <value>AE</value>
        </set>
    </property>
</bean>

<bean class="com.nt.beans.EnggCourse" parent="base1stYear">
    <property name="subjects">
        <set>
            <value>RCC</value>
            <value>Surveying</value>
            <value>MOS</value>
        </set>
    </property>
</bean>

</beans>

```

### CollectionMergingTest.java

```

//get beans
course = factory.getBean("com.nt.beans.EnggCourse#0",
EnggCourse.class);
System.out.println(course);
System.out.println("-----");
course = factory.getBean("com.nt.beans.EnggCourse#1",
EnggCourse.class);
System.out.println(course);
System.out.println("-----");
course = factory.getBean("com.nt.beans.EnggCourse#2",
EnggCourse.class);
System.out.println(course);

```

### applicationContext.xml

```
<bean class="com.nt.beans.EnggCourse">
    .....
        default bean id: "com.nt.beans.EnggCourse"
    .....
        (or)
    .....
        "com.nt.beans.EnggCourse#0"
</bean>

<bean class="com.nt.beans.EnggCourse">
    .....
        default bean id: "com.nt.beans.EnggCourse#1"
</bean>

<bean class="com.nt.beans.EnggCourse">
    .....
        default bean id: "com.nt.beans.EnggCourse#2"
</bean>
```

Default bean id syntax is: <pkg>.<classname>#<n>, n is "0" based index.

**Q. Can we have same bean id for two different spring beans configure with in an IoC container?**

**Ans.** No, the bean ids must be unique with in the IoC container.

**Q. Can we have same class names for diff spring bean configurations?**

**Ans.** Yes, we must configure them with diff unique bean ids.

### Inner Bean

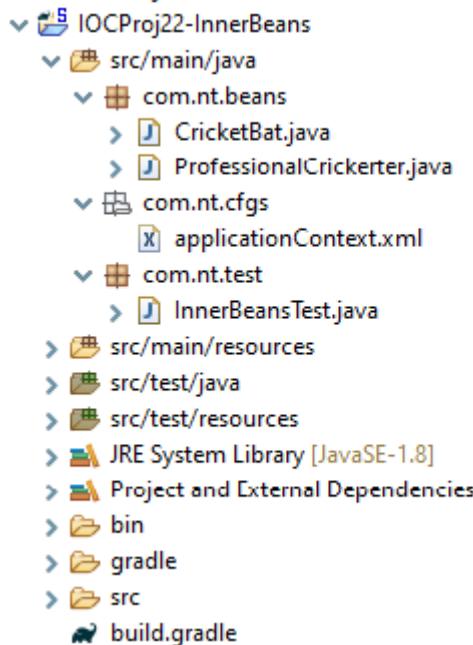
- It is no way related to inner classes of java.
- It is all about configuring one Spring bean inside another Spring bean configuration.
- For this we need to place <bean> tag as the sub tag of <property> or <constructor- arg> tags of another <bean> (directly writing <bean> tag under another <bean> tag)
- If we want to configure bean as the dependent bean only for one target bean configure it as inner bean to that target bean configuration.
- If we want to configure bean as the dependent bean for multiple target bean configurations either having same class names or different class name then configure it as normal bean to inject to multiple target beans.

**e.g.**

- Cricket bat to professional cricketer should be configured as inner bean.
- Cricket bat to Normal cricketer should be configured as normal bean.

- Cricket ball to cricketer should be configured as Normal bean.
- Car key to car should be configured as inner bean.
- Bank account details to customer should be configured as inner bean.
- DataSource to DAO should be configured as normal bean.
- DAO to Service should be configured as normal bean.
- Service to controller can be configured as inner bean because every project contains only one controller.

### Directory Structure of IOCProj22-InnerBeans:



- Develop the above directory structure and package, class, XML file and add the jar also, then use the following code with in their respective file.
- Copy the build.gradle dependencies from previous project because we will have used same dependencies.

#### CricketBat.java

```

package com.nt.beans;

import java.util.Random;

public class CricketBat {
    public int scoreRuns() {
        return new Random().nextInt(200);
    }
}
  
```

### ProfessionalCrickerter.java

```
package com.nt.beans;

public class ProfessionalCrickerter {

    private String name;
    private CricketBat bat;

    public ProfessionalCrickerter(String name, CricketBat bat) {
        System.out.println("ProfessionalCrickerter
:ProfessionalCrickerter()");
        this.name = name;
        this.bat = bat;
    }

    public String batting() {
        int runs = 0;
        runs = bat.scoreRuns();
        return "Professional Crickter Mr. "+name+" has scored
"+runs+" run.";
    }

}
```

### 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-4.3.xsd">

    <!-- Configure outer bean -->
    <bean id="kohil" class="com.nt.beans.ProfessionalCrickerter">
        <constructor-arg value="Virat Kohil"/>
        <constructor-arg>
            <bean class="com.nt.beans.CricketBat"/>
        </constructor-arg>
    </bean>
```

```

<bean id="dhoni" class="com.nt.beans.ProfessionalCrickerter">
    <constructor-arg value="M S Dhoni"/>
    <constructor-arg>
        <bean class="com.nt.beans.CricketBat"/>
    </constructor-arg>
</bean>

</beans>

```

### InnerBeanTest.java

```

package com.nt.test;

import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;

import com.nt.beans.ProfessionalCrickerter;

public class InnerBeansTest {

    public static void main(String[] args) {
        DefaultListableBeanFactory factory = null;
        XmlBeanDefinitionReader reader = null;
        ProfessionalCrickerter cricketer1 = null, cricketer2 = null;
        //Create Bean Factory IoC container
        factory = new DefaultListableBeanFactory();
        reader = new XmlBeanDefinitionReader(factory);

        reader.loadBeanDefinitions("com/nt/cfgs/applicationContext.xml");
        //get bean
        cricketer1 = factory.getBean("kohil",
ProfessionalCrickerter.class);
        System.out.println(cricketer1);
        System.out.println("-----");
        cricketer2 = factory.getBean("dhoni",
ProfessionalCrickerter.class);
        System.out.println(cricketer2);
    }
}

```

### Important points:

- Giving bean id to inner bean configuration is optional (not required) because inner bean cannot be injected to other than current outer bean and cannot be accessed by calling factory.getBean(-) method from the client application.
- In target class we can take both normal beans, inner beans based on dependency injection.
- One inner bean can have other inner beans injection.
- We cannot access inner bean directly; it should be accessed only through outer bean.

Q. Can we inject both normal bean and inner bean to a property of target bean class?

Ans. No, because <property>, <constructor-arg> tags can have only ref attribute or sub tag (not multiple at a time). This is XSD/ DTD level restriction.

### Directory Structure of IOCProj23-RealTimeDI-InnerBeans:

- + Copy paste the Layered application and change rootProject.name to IOCProj15-RealTimeDI-CollectionInjection in settings.gradle file.
- + Need not to add any new file same structure as IOCPro13-RealTimeDI-RealTimeStrategyDP-LayeredApp.
- + Add the following code in applicationContext.xml as part of the replace for the particular section.

### applicationContext.xml

```
<!-- Configure Service object to Controller class as Inner Bean -->
<bean id="controller" class="com.nt.controller.MainController">
    <constructor-arg>
        <!-- Configure DAO object to Service class -->
        <bean id="custService"
            class="com.nt.service.CustomerMgmtServiceImpl">
            <constructor-arg ref="oracleCustDAO"/>
            <!-- <constructor-arg ref="mysqlCustDAO"/> -->
        </bean>
    </constructor-arg>
</bean>
```

Q. If we configure same class as multiple spring beans with different bean ids or no bean ids then can tell me how many objects will be created for that class,

**when start using all the bean configurations?**

Ans. Multiple objects will be created on 1 per bean id/ default bean id basis.

```
<bean id="engg1" class="pkg.EnggCourse">  
.....  
</bean>  
<bean id="engg2" class="pkg.EnggCourse">  
.....  
</bean>
```

If we call both factory.getBean(-,-) having both bean ids, then IOC Container creates two objects for EnggCourse class.

internal cache of IOC container

engg1	EnggCourse object
engg2	EnggCourse object

```
EnggCourse e1=factory.getBean("engg1", EnggCourse.class);  
EnggCourse e2=factory.getBean("engg1", EnggCourse.class);  
here e1, e2 refers to same object of EnggCourse class
```

```
EnggCourse e1 = factory.getBean("engg1", EnggCourse.class);  
EnggCourse e1 = factory.getBean("engg2", EnggCourse.class);  
EnggCourse e1 = factory.getBean("engg2", EnggCourse.class);  
Creates 2 objects for EnggCourse class
```

**Note:** IoC container creates and reuse bean object by creating it based on per class and per Id i.e. IoC container creates 1 object per 1 Spring bean class and per bean id.

## Bean Alias

- Provide nick names to bean id.
- Useful when bean id is very lengthy to refer in multiple places.
- Generally, there is a practice of taking class name as the bean id, which is quite lengthy to give short names or nick names to that bean id we should go for Alias names.

```
<bean id="wishMessageGenerator" name="wmg, wmg1"  
class="com.nt.beans.WishMessageGenerator"/>
```

```
<alias name="wishMessageGenerator" alias="msg1"/>
<alias name="msg1" alias="msg2"/>
<alias name="wmg1" alias="msg3"/>
```

- Up to Spring 2.x we can use only "name" attribute, from Spring 3.x we can use both "name" attribute and <alias> tag to provide alias names/ nick names.
- Using one "name" we can provide multiple alias names at a time.
- Using one tag we can provide only one alias name/ nick name at a time.
- We can also provide alias names to default bean ids.
- We can provide alias name to alias name itself.

#### Directory Structure of IOCProj24-BeanAlias:

- ⊕ Copy paste the IOCProj01-SetterInjection-POC.
- ⊕ Need not to add any new file same structure as IOCProj01-SetterInjection-POC.
- ⊕ Add the following code in applicationContext.xml and SetterInjection.java as part of the replace for the particular section.

#### applicationContext.xml

```
<!-- Dependent bean configuration -->
<bean id="date" name="dt" class="java.util.Date"/>

<!-- Target bean configuration -->
<bean id="wishMessageGenerator" name="wmg1, wmg2"
class="com.nt.beans.WishMessageGenerator">
    <property name="date" ref="dt1"/>
</bean>

<alias name="wishMessageGenerator" alias="msg"/>
<alias name="msg" alias="msg1"/>
<alias name="wmg1" alias="msg2"/>
<alias name="dt" alias="dt1"/>
```

#### SetterInjectionTest.java

```
//get Target bean class object & Type casting
generator =
(WishMessageGenerator)factory.getBean("wmg1");
```

```

//invoke the method & result
System.out.println("Wish Message is:
"+generator.generateWishMessage("Nimu"));
System.out.println("-----");
generator = factory.getBean("msg2",
WishMessageGenerator.class);
//invoke the method & result
System.out.println("Wish Message is:
"+generator.generateWishMessage("Niru"));

```

## Dependency Lookup

- Here Target class writes logics to search and get Dependent class object by spending some time.

**Q. When should we go for dependency lookup and when should we go dependency injection?**

**Ans.**

- If the dependent class obj is required only in the one method of target class then go for dependency lookup.
- If the dependent class object is required in multiple methods of target class then go for Dependency injection (setter injection and constructor injection).

Cricketer (target) needs cricketBall, Bat, keeping Gloves (all are dependent)

```

|---> fielding ()
|---> batting ()
|---> blowing ()
|---> keeping ()

```

cricketBall: Since Cricket ball is required in all the 4 methods go for dependency injection.

Bat: Since Bat is required only in batting () method go for dependency lookup.

keeping gloves: Since keeping gloves are required only in keeping () go for dependency lookup.

**Note:** To perform dependency lookup create extra IoC container in the target class method and call factory.getBean(-) method having bean id of dependent bean to get Dependent Bean class object.

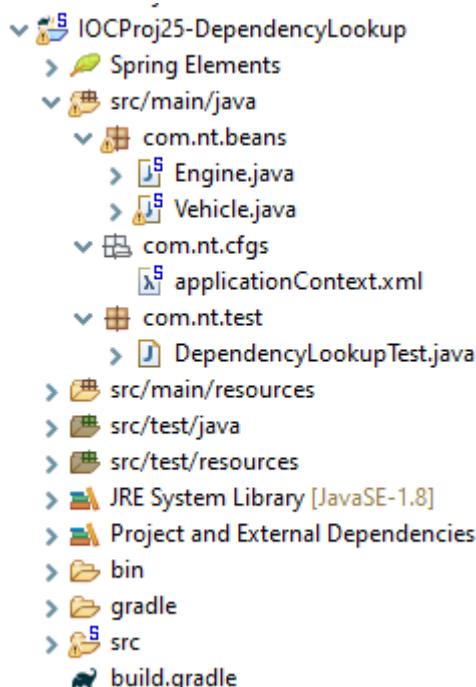
Vehicle (target class) needs Engine (dependent class)

|---> journey (srcPlace, destPlace)  
|---> entertainment ()  
|---> soundHorn ()

Engine: Here Engine is required only in one method of Target class (journey()). So, go for Dependency lookup.

**Note:** In target class we can place code supporting both Dependency lookup and injection.

### Directory Structure of IOCProj25-DependencyLookup:



- Develop the above directory structure and package, class, XML file then use the following code with in their respective file.
- Copy paste build.gradle from any other gradle project because we are using same spring-context-support dependency only.

#### Engine.java

```
package com.nt.beans;

public class Engine {

    public Engine() {
        System.out.println("Engine : Engine()");
    }
}
```

```

public void start() {
    System.out.println("Engine : start() - Engine Started");
}

public void stop() {
    System.out.println("Engine : stop() - Engine Stopped");
}

}

```

### Vehicle.java

```

package com.nt.beans;

import
org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;

public class Vehicle {

    private String beanId;

    public Vehicle(String beanId) {
        System.out.println("Vehicle : Vehicle(-)");
        this.beanId = beanId;
    }

    public void entertainment() {
        System.out.println("Vechicle is equipped with DVD player for
entertainment");
    }
    public void soundHorn() {
        System.out.println("Vechicle is equipped with skoda horn");
    }

    public void fillFuel() {
        System.out.println("Vechicle is having fuel tank of 60 liters");
    }

    public void journey(String sourcePlace, String destPlace) {
}

```

```

DefaultListableBeanFactory factory = null;
XmlBeanDefinitionReader reader = null;
Engine engg = null;
//Create BeanFactory or IoC container
factory = new DefaultListableBeanFactory();
reader = new XmlBeanDefinitionReader(factory);

reader.loadBeanDefinitions("com/nt/cfgs/applicationContext.xml");
//get dependent bean class object
engg = factory.getBean(beanId, Engine.class);
engg.start();
System.out.println("Journey started from: "+sourcePlace);
System.out.println("Journey was going on from
"+sourcePlace+" to "+destPlace);
System.out.println("Journey stoped at : "+destPlace);
}

}

```

#### 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-4.3.xsd">

    <!-- Dependent Bean Id -->
    <bean id="engg" class="com.nt.beans.Engine"/>

    <!-- Target bean id -->
    <bean id="vechicle" class="com.nt.beans.Vehicle">
        <!-- <constructor-arg value="engg2"/> -->
        <constructor-arg>
            <idref bean="engg"/>
        </constructor-arg>
    </bean>

</beans>

```

## DependencyLookupTest.java

```
package com.nt.test;

import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;

import com.nt.beans.Vehicle;

public class DependencyLookupTest {

    public static void main(String[] args) {
        DefaultListableBeanFactory factory = null;
        XmlBeanDefinitionReader reader = null;
        Vehicle vechicle=null;
        //Create BeanFactory or IoC container
        factory = new DefaultListableBeanFactory();
        reader = new XmlBeanDefinitionReader(factory);
        reader.loadBeanDefinitions("com/nt/cfgs/applicationContext.xml");
        //get dependent bean class object
        vechicle = factory.getBean("vechicle", Vehicle.class);
        //invoke methods
        vechicle.journey("Odisha", "Hyd");
        vechicle.entertainment();
        vechicle.soundHorn();
        vechicle.fillFuel();
    }

}
```

Q. What is the difference b/w injecting bean id using <value> or "value" attribute and using <idref> tag?

Ans. <value> or "value" attribute injects the bean id as an ordinary string value without checking the having that bean id spring bean is configure or not So exception comes lately when that wrongly injected spring id is used in the target class (not recommended).

<property value="engg"/> - dependent class bean id  
(or)

<constructor name="beanId" value="engg"/>

we can use <idref> tag to inject the bean id, it will check having bean id is there any spring bean configuration before injecting bean id to target class. If not there it will throw exception i.e. problem is detected very early.

```
<property name="beanId">
    <idref bean="engg"/>
</property>
(or)
.
<constructor-arg name="beanId">
    <idref bean="engg"/>
</constructor-arg>
```

**Conclusion:** To Inject bean id to spring bean prefer using <idref> tag.

#### Disadvantages of Traditional Dependency Lookup:

- a. Taking extra IOC container in the specific method of Target class is bad practice.
- b. The injected bean id of dependent class to the target class is visible in all the methods of target class though it is required in only one method.

**Note:** The Solution for above problems is Aware Injection or Lookup method Injection (Both these are extension to Traditional dependency lookup).

## Bean Wiring (Auto wiring)

- Assigning dependent class object to target class object is called Dependency Injection or bean wiring.
  - a. Explicit wiring/ Manual Injection
  - b. Auto wiring/ Auto Injection

#### Explicit wiring/ Manual Injection:

- Here we use <property> or <constructor-arg> to inject dependent value/objects to target class (So far, we are working on this).

#### Auto wiring/ Auto Injection:

- Here that IoC container automatically detects and injects Dependent spring bean class objects to target spring bean class object.
- Use "autowire" attribute of <bean> for this.
- No need of writing <property>, <constructor-arg> tags here.

#### Limitations of auto wiring:

- a. Auto wiring is possible only on object type/ reference type bean properties not possible on simple, array, collection type bean properties.

- b. There is a possibility of getting ambiguity Problem (It will confuse to choose one of the multiple dependents).
- c. Kills the readability of spring bean configuration file.

**Note:** Auto wiring is very useful, for Rapid application development (RAD) Faster development.

#### Different modes of Auto wiring:

- a. byName
- b. byType
- c. constructor
- d. autodetect (removed from spring 3.x)

#### autowire="byName":

- performs setter Injection.
- Detects/finds dependent spring bean class object based its bean id that is matching with target class property name (variable name).
- There is no possibility of getting Ambiguity problem because the bean ids in IOC container are unique ids.

#### e.g.

```
<!-- beans cfgs -->
<bean id="courier" class="pkg.DTDC"/>
<bean id="courier1" class="pkg.BlueDart"/>

<bean id="fpkt" class="pkg.Flipkart" autowire="byName"/>
```

Here injects DTDC class object to the courier property of Flipkart class because the DTDC class bean id(courier)is matching with "courier" property name of Flipkart class.

#### FlipKart (target class)

```
|---> private Courier courier;
|---> p v setCourier (Couier courer) {this.courier=courier;}
```

DTDC implements Courier

BlueDart implements Courier

#### Directory Structure of IOCProj26-StratergyPattern-Spring-AutoWiring:

⊕ Copy pastes the IOCProj07-StrategyPattern-Spring.

- ⊕ Need not to add any new file same structure as IOCProj07-StrategyPattern-Spring.
- ⊕ Add the following code in applicationContext.xml as part of the replace for the particular section.

#### applicationContext.xml

```

<!-- Dependent bean configuration -->
<bean id="courier" class="com.nt.components.DTDC"/>
<bean id="fFlight" class="com.nt.components.FirstFlight"
primary="true"/>
<bean id="bDart" class="com.nt.components.BlueDart"/>

<!-- Target bean configuration -->
<bean id="fpkt" class="com.nt.components.Flipkart"
autowire="byName"/>
```

#### autowire="byType":

- Performs setter injection or type.
- Target Bean class property [variable type and Dependent spring bean type (Dependent class name) must match].
- There is a possibility of getting ambiguity problem and we can solve it by using in one of the dependent spring beans configuration, throws **Exception in thread "main"** [org.springframework.beans.factory.UnsatisfiedDependencyException:](#) ambiguity problem.

#### applicationContext.xml

```

<!-- Dependent bean configuration -->
<bean id="dtdc" class="com.nt.components.DTDC"/>
<bean id="fFlight" class="com.nt.components.FirstFlight"
primary="true"/>
<bean id="bDart" class="com.nt.components.BlueDart"/>

<!-- Target bean configuration -->
<bean id="fpkt" class="com.nt.components.Flipkart"
autowire="byType"/>
```

`autowire="constructor":`

- Performs constructor injection by using parameterized constructor.
- Here Constructor param type and Dependent bean class type must match in order to use that parameter constructor for auto wiring.
- There is possibility of getting ambiguity problem and it can be solved in two ways
  - a. By matching dependent spring bean class id with constructor param name.
  - b. By keeping primary="true" in one of multiple possible dependent spring bean classes configuration.

**Note:** If both are placed in two different possible dependent spring beans then primary="true" gets high priority.

#### applicationContext.xml

```
<!-- Dependent bean configuration -->
<bean id="courier" class="com.nt.components.DTDC"/>
<bean id="fFlight" class="com.nt.components.FirstFlight"
primary="true"/>
<bean id="bDart" class="com.nt.components.BlueDart"/>

<!-- Target bean configuration -->
<bean id="fpkt" class="com.nt.components.Flipkart"
autowire="constructor"/>
```

`autowire="autodetect":`

- Removed from spring 3.x, because it killing more and more readability.
- If spring bean is having 0-param constructor directly or indirectly then it goes for "byType" mode of autowiring , if not there then it goes for "constructor" mode autowiring.
- To work with this mode auto wiring change spring version to 4.x/3.x and XSD version to 2.5/2.0.
- There is possibility of getting ambiguity problem and we resolve it by using the same solutions of "byType", "constructor" mode auto wirings all dependent classes as spring beans

**Note:** You have to change the jar version if you are developed in gradle, it is easy to change. Otherwise we can change the normal project to gradle project.

## applicationContext.xml

```
<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-2.5.xsd">

    <!-- Dependent bean configuration -->
    <bean id="courier" class="com.nt.components.DTDC"/>
    <bean id="fFlight" class="com.nt.components.FirstFlight"
primary="true"/>
    <bean id="bDart" class="com.nt.components.BlueDart"/>

    <!-- Target bean configuration -->
    <bean id="fpkt" class="com.nt.components.Flipkart"
autowire="constructor"/>

</beans>
```

## How to convert Eclipse standalone to Gradle standalone Project:

**Step 1:** Right click on your project, go to Configure then click on Add Gradle Nature.

**Step 2:** Now you can see the Gradle project has come to your project.

**Step 3:** Now go to Gradle tasks window then choose your project and click on Refresh.

**Step 4:** Expand your Project, expand the build setup folder, right click on init task then click on Run Gradle Tasks.

**Step 5:** right click on your project the click on New then choose File.

**Step 6:** Make sure you pointing the current project, then give the file name "build.gradle" then click on Finish, and add the required jar dependencies.

**Step 7:** Now time for Refresh Gradle Project, Right click on your project then go to Gradle click on Refresh Gradle Project.

**Step 8:** Now you can see all the Gradle related folder and file, and the Reference Libraries has gone and Project and External Dependencies has come with the jars.

**Step 9:** Right click on your project, go to new then click on Source Folder.

**Step 10:** Then give Folder Name like "src/main/java" then click on Finish.

**Step 11:** Now you can see the source folder is come then copy your package and source code then paste in the java folder.

**Step 12:** Now delete the com folder the all the folder and code will be deleted, now you can see the package and java file come to src/main/java folder nicely.

**Step 13:** Run your test class by right click on that file then go Run As then click on Java Application.

**Step 14:** Now you can see the output in console window.

"autowire" attribute possible values:

- byType
- byName
- constructor
- autodetect (removed from spring 3.x)
- default (default)
- no

**autowire="default":**

- Fallbacks to default autowire mode that is specified in "default-autowire" attribute of <beans> tags.

<beans default-autowire="constructor" ....>

Enables constructor mode auto wiring on all the spring beans of current spring bean configuration file Disables auto wiring on certain spring bean configuration, even though default autowiring is enabled.

**autowire="no":**

- Disables auto wiring on certain spring bean configuration, even though default autowiring is enabled.

<beans default-autowire="byType" ....>

<bean id "fpkt" class="com.nt.comp.Flipkart" autowire="no"/>  
<bean id "fpktl" class="com.nt.comp.Flipkart" autowire="by Name"/>

**The possible values of "default-autowire" attribute of <beans> tag is:**

- byName
- byType
- constructor
- default (default)

## applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans default-autowire="byName"
      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>

    <!-- Dependent bean configuration -->
    <bean id="courier" class="com.nt.components.DTDC"/>
    <bean id="fFlight" class="com.nt.components.FirstFlight"
          primary="true"/>
    <bean id="bDart" class="com.nt.components.BlueDart"/>

    <!-- Target bean configuration -->
    <!-- <bean id="fpkt" class="com.nt.components.Flipkart"
        autowire="default"/> -->
    <bean id="fpkt" class="com.nt.components.Flipkart" autowire="no"/>

</beans>
```

### **<beans default-autowire="no" ....>:**

- Disables autowiring on all spring beans that are there in current bean configuration file.

### **<beans default-autowire="default" ...>:**

- In case of nested beans/inner beans the outer bean autowire mode will be applied where as in case of normal beans it is equal to default-autowire="no" value out.

**Q. Can we enable autowiring on spring bean with using autowire attribute?**

**Ans.** Yes, by specifying autowire mode in "default-autowire" attribute of <beans> tag. (other than no, default values should specify).

**Q. Can we override default autowire mode?**

**Ans.** Yes, use "autowire" attribute in every <bean> keeping other than "default" value.

## Making Spring bean auto wire candidate:

- In order to make certain dependent spring beans not participating in autowiring, we can disable them as auto wire candidates by using this is one solution to solve ambiguity problem that comes with byType, constructor mode of autowiring without using primary="true".

### applicationContext.xml

```
<!-- Dependent bean configuration -->
<bean id="dtdc" class="com.nt.components.DTDC"/>
<bean id="fFlight" class="com.nt.components.FirstFlight" autowire-
candidate="false"/>
<bean id="bDart" class="com.nt.components.BlueDart" autowire-
candidate="false"/>

<!-- Target bean configuration -->
<bean id="fpkt" class="com.nt.components.Flipkart" autowire="no"/>

</beans>
```

#### Note:

- ✓ Still primary="true" is the best solution to solve ambiguity problem.
- ✓ <bean id="dtdc" class="com.nt.comp.DTDC" autowire-candidate="false" primary="true"/>  
Wrong combo because after making spring bean as non autowire-candidate there is no meaning of keeping primary="true".

The possible values for "autowire-candidate" attribute are:

- a. true
- b. false
- c. default (default)

**autowire-candidate="true":** makes spring bean to participate in autowiring.

**autowire-candidate="false":** makes spring bean not to participate in autowiring.

**autowire-candidate="default":** fallbacks to default-autowire-candidates attribute of beans tag, i.e. if the current bean id is there in the list of bean ids that are specified in "default-autowire-candidates" attribute of <beans> tag then it will act as autowire candidate otherwise it will not participate in autowiring. <beans default-autowire-candidates="dtdc, bDart" ...>

No default values for this default-autowire-candidates attribute, if we do not specify this attribute in <beans> then all spring beans are autowire candidates.

### applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans default-autowire="byName" default-autowire-candidate="dtdc,
bDart" 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>

    <!-- Dependent bean configuration -->
    <bean id="courier" class="com.nt.components.DTDC"/>
    <bean id="fFlight" class="com.nt.components.FirstFlight"/>
    <bean id="bDart" class="com.nt.components.BlueDart" autowire-
candidate="false"/>

    <!-- Target bean configuration -->
    <bean id="fpkt" class="com.nt.components.Flipkart"
autowire="byType"/>

</beans>
```

**Q. What is the diff b/w autowire=" no" and autowire-candidate="false"?**

**Ans.**

- autowire="no" make IoC container not perform any autowiring based injections to current spring bean by considering it as target spring bean.
- autowire-candidate="false" makes IoC container not to consider current spring bean as dependent to inject to other beans.

**Use case:** if want suspend certain bean as dependent bean to participate in autowiring for temporary period then go for this autowire-candidate="false" option.

**Note:** The dependent bean which is disabled as autowire-candidate, can be used as dependent through explicit wiring manual wiring concept.

**Q. What is happen, if we enable both autowiring and explicit wiring on the bean property of target class?**

Ans.

- If both explicit wiring and autowiring are performing same setter injection or same constructor injection the explicit wiring takes place.
- If one wiring performs setter injection and another wiring performs constructor injections then setter injection values will be taken as final values.

### applicationContext.xml

```
<!-- Dependent bean configuration -->
<bean id="dtdc" class="com.nt.components.DTDC"/>
<bean id="fFlight" class="com.nt.components.FirstFlight"
primary="true"/>
<bean id="bDart" class="com.nt.components.BlueDart"/>

<!-- Target bean configuration -->
<bean id="fpkt" class="com.nt.components.Flipkart"
autowire="byName">
    <constructor-arg rer="bDart"></constructor-arg>
</bean>
```

**Note:** we can enable both autowiring and explicit wiring on different bean properties of spring bean class i.e. one few property we can enable explicit wiring, on few other properties we can enable autowiring.

### Directory Structure of IOCProj27-RealTimeDI-AutoWiring:

- ⊕ Copy paste the Layered application and change rootProject.name to IOCProj27-RealTimeDI-AutoWiring in settings.gradle file.
- ⊕ Need not to add any new file same structure as IOCPro13-RealTimeDI-RealTimeStrategyDP-LayeredApp.
- ⊕ Add the following code in applicationContext.xml as part of the replace for the particular section.

### applicationContext.xml

```
<!-- Configure the DataSource for Oracle-->
<bean id="oracleDmds"
class="org.springframework.jdbc.datasource.DriverManagerDataSource"
primary="true">
    <property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver"/>
```

```

        <property name="url"
value="jdbc:oracle:thin:@localhost:1521:xe"/>
            <property name="username" value="system"/>
            <property name="password" value="manager"/>
</bean>

        <!-- Configure the DataSource for MySQL-->
<bean id="mysqlDmnds"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
            <property name="driverClassName"
value="com.mysql.cj.jdbc.Driver"/>
            <property name="url" value="jdbc:mysql:///nssp713db"/>
            <property name="username" value="root"/>
            <property name="password" value="root"/>
</bean>

        <!-- Configure Oracle DataSoruce object to Oracle DAO class -->
<bean id="oracleCustDAO"
class="com.nt.dao.OracleCustomerDAOImpl" autowire="constructor"
primary="true"/>

        <!-- Configure MySQL DataSoruce object to MySQL DAO class -->
<bean id="mysqlCustDAO"
class="com.nt.dao.MySQLCustomerDAOImpl">
            <constructor-arg ref="mysqlDmnds"/>
</bean>

        <!-- Configure DAO object to Service class -->
<bean id="custService"
class="com.nt.service.CustomerMgmtServiceImpl" autowire="constructor"/>

        <!-- Configure Service object to Controller class -->
<bean id="controller" class="com.nt.controller.MainController"
autowire="constructor"/>
```

**Q. What happens if we place only private constructor in spring bean class?**

**Ans.** Still IoC container creates Spring Bean class object by accessing private

constructor internally calls setAccessible(true) method of spring bean class using Reflection API on java.lang.reflect.Constructor class object that represents private constructor.

### Q. can we take spring bean class as private class?

**Ans.** In java itself we cannot take outer classes as private classes, So In spring also normal spring bean classes cannot be taken as private class.

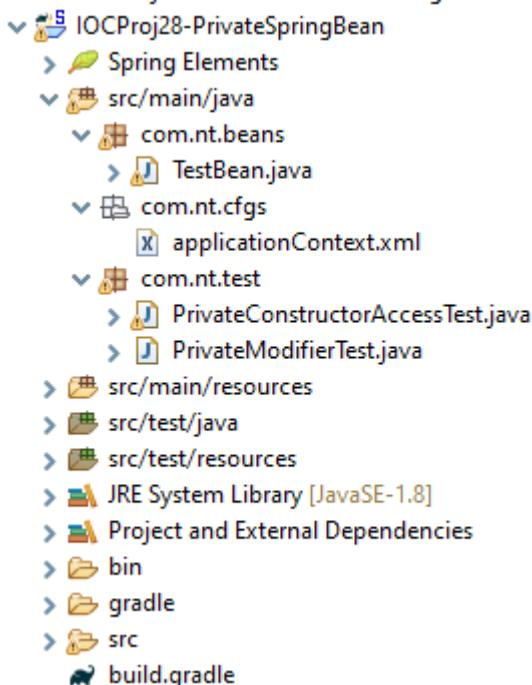
```
private class Test {//wrong
    <bean id="t" class="pkg.Test"/> (error)
}
```

In java we can take inner class as private class. So, we configure that inner class as spring bean specifying outer class name then the spring bean can be private class.

```
public class Test {
    private static class ABC {
        <bean class="pkg.Test.ABC"/> (valid)
    } (valid)
}
```

IoC container does load classes from .java files. It Load the classes from .class files.

### Directory Structure of IOCProj28-PrivateSpringBean:



- Develop the above directory structure and package, class, XML file then use the following code with in their respective file.
- Copy paste build.gradle from any other gradle project because we are using same spring-context-support dependency only.

### TestBean.java

```

package com.nt.beans;

public class TestBean {

    private int a, b;

    private TestBean() {
        System.out.println("TestBean : TestBean()");
    }

    private TestBean(int a, int b) {
        System.out.println("TestBean : TestBean(-,-)");
        this.a = a;
        this.b = b;
    }

    @Override
    public String toString() {
        return "TestBean [a=" + a + ", b=" + b + "]";
    }

    private static class InnerBean {
        private int c;
        public void setC(int c) {
            this.c = c;
        }

        @Override
        public String toString() {
            return "InnerBean [c=" + c + "]";
        }
    }
}

```

### 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-4.3.xsd">

    <bean id="beans" class="com.nt.beans.TestBean">
        <constructor-arg value="10"/>
        <constructor-arg value="20"/>
    </bean>

    <bean id="innerBeans" class="com.nt.beans.TestBean.InnerBean">
        <property name="c" value="30"/>
    </bean>

</beans>
```

### PrivateModifierTest.java

```
package com.nt.test;

import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;

import com.nt.beans.TestBean;

public class PrivateModifierTest {

    public static void main(String[] args) {
        DefaultListableBeanFactory factory = null;
        XmlBeanDefinitionReader reader = null;
        TestBean bean = null;
        Object obj = null;
        // Create Bean Factory IoC container
        factory = new DefaultListableBeanFactory();
        reader = new XmlBeanDefinitionReader(factory);
        reader.loadBeanDefinitions("com/nt/cfgs/applicationContext.xml");
        //get bean class object
```

```

        bean = factory.getBean("beans", TestBean.class);
        System.out.println(bean);
        System.out.println("-----");
        obj = factory.getBean("innerBeans", Object.class);
        System.out.println(obj);
    }

}

```

### PrivateConstructorAccessTest.java

```

package com.nt.test;

import java.lang.reflect.Constructor;

import com.nt.beans.TestBean;

public class PrivateConstructorAccessTest {

    public static void main(String[] args) {
        Class c = null;
        Constructor cons[] = null;
        TestBean b1=null, b2=null;
        Object obj = null;
        try {
            //load the class
            c = Class.forName("com.nt.beans.TestBean");
            //get all declared constructors
            cons = c.getDeclaredConstructors();
            //create object for that class
            cons[0].setAccessible(true);
            b1 = (TestBean)cons[0].newInstance();
            System.out.println(b1);
            System.out.println("-----");
            cons[1].setAccessible(true);
            b2 = (TestBean)cons[1].newInstance(10,30);
            System.out.println(b2);
            System.out.println("-----");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```
    }  
}  
}
```

## Factory Method Bean Instantiation

### Factory Method

- The method that is capable of creating and returning object is called Factory method.
- This method can return either its own class object or related class object/ unrelated class object.

Two types of Factory methods

- a. Static factory method
- b. Instance factory method (non-static factory method)

#### Static factory method:

e.g.

1. Thread t = Thread.currentThread(); // static factory method returning its own class object.
2. Class c = Class.forName("java.util.Date"); // static factory method returning its own class object.
3. Calendar cal = Calendar.getInstance(); //static factory method returning the sub class obj of Calendar class (AC) i.e. GregorianCalendar class object.
4. Connection con = DriverManager.getConnection(-, -, -); //static factory method returning un related class object, because there is no relation b/w JDBC con object class name and java.sql.DriverManager class. and etc.

#### Instance factory methods:

e.g.

1. String s1 = new String("hello");  
String s2 = s1.concat (" 123"); //gives hello10123 (Instance factory method returning its own class object)
2. StringBuffer sb=new StringBuffer ("hello how are u?");  
String s1 = sb.substring(0,5); //gives hello (instance factory method returning other class object (unrelated class object))

3. Statement st = con.createStatement(); (instance factory method returning other class obj (unrelated class obj))
4. HttpSession res=req.getSession();
5. SessionFActory factory=cfg.buildSessionFActory();
6. Object obj=factory.getBean(--)  
and etc...

**IOC container can create spring bean class object:**

- a. Using 0-param constructor  
[If spring bean class is configuration with no injection or only with setter injection]
- b. Using parameterized constructor  
[If spring bean class is configuration by enabling constructor injection]
- c. Using static factory method  
[If we configure "factory-method" attribute in <bean> tag]
- d. Using instance factory method  
[If we configure "factory-bean", "factory-method" attributes in <bean> tag]

**Note:** If we enable factory method bean instantiation, we can specify the factory method names using "factory-method" attribute of <bean> tag, and we can pass argument values to factory methods using <constructor-arg> tags.

#### applicationContext.xml

```
<bean class="java.lang.Class" factory-method="forName">
    <constructor-arg value="java.util.Date"/>    Class c1 =
</bean>                                              Class.forName("ja
                                                       va.util.Date");
```

#### **internal code**

here <constructor-arg> is used to pass argument value to forName(-) method call.

Class cl=new Class("java.util.Date"); **X**

(Wrong guess of internal code because java.lang.Class is having only one private 0- param constructor. So, it cannot be instantiated using new operator outside of its class)

#### Client App

```
Class c=factory.getBean("c1", Class.class);
```

## Directory Structure of IOCProj29-FactoryMethodBeanInstantiation-POC:

```
IOCProj29-FactoryMethodBeanInstantiation-POC
  +-- Spring Elements
  +-- src/main/java
    +-- com.nt.cfgs
      -- applicationContext.xml
    +-- com.nt.test
      +-- FactoryMethodBeanInstantiationTest.java
  +-- src/main/resources
  +-- src/test/java
  +-- src/test/resources
  +-- JRE System Library [JavaSE-1.8]
  +-- Project and External Dependencies
  +-- bin
  +-- gradle
  +-- src
  -- build.gradle
```

- Develop the above directory structure and package, class, XML file then use the following code with in their respective file.
- Copy paste build.gradle from any other gradle project because we are using same spring-context-support dependency only.

### 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-4.3.xsd">

    <!-- Static Factory Method Bean instantiation giving its own class
object -->
    <bean id="cls" class="java.lang.Class" factory-method="forName">
        <constructor-arg value="java.util.Date"/>
    </bean>

    <!-- Static Factory Method Bean instantiation giving its related class
object -->
    <bean id="cal" class="java.util.Calendar" factory-
method="getInstance"/>

    <!-- Static Factory Method Bean instantiation giving its unrelated class
object -->
```

```

<bean id="conn" class="java.sql.DriverManager" factory-
method="getConnection">
    <constructor-arg
value="jdbc:oracle:thin:@localhost:1521:xe"/>
        <constructor-arg value="system"/>
        <constructor-arg value="manager"/>
</bean>

    <!-- Static Factory Method Bean instantiation giving its unrelated class
object -->
    <bean id="sys" class="java.lang.System" factory-
method="getProperties"/>

    <!-- Instance Factory Method Bean instantiation giving its same class
object -->
    <bean id="s1" class="java.lang.String">
        <constructor-arg value="hello"/>
    </bean>
    <bean id="s2" factory-bean="s1" factory-method="concat">
        <constructor-arg value=" how are you"/>
    </bean>

    <!-- Instance Factory Method Bean instantiation giving its unrelated
class object -->
    <bean id="sb" class="java.lang.StringBuffer">
        <constructor-arg value="hello how are you"/>
    </bean>
    <bean id="s3" factory-bean="sb" factory-method="substring">
        <constructor-arg value="0"/>
        <constructor-arg value="5"/>
    </bean>
</beans>
```

### FactoryMethodBeanInstantiationTest.java

```

package com.nt.test;

import java.sql.Connection;
import java.util.Calendar;
import java.util.Properties;
```

```

import
org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;

public class FactoryMethodBeanInstantiationTest {

    public static void main(String[] args) {
        DefaultListableBeanFactory factory = null;
        XmlBeanDefinitionReader reader = null;
        Class clss = null;
        Calendar calender = null;
        Connection conn = null;
        Properties props = null;
        String str1 = null, str2 = null;
        //Create Bean Factory IoC Container
        factory = new DefaultListableBeanFactory();
        reader = new XmlBeanDefinitionReader(factory);
        reader.loadBeanDefinitions("com/nt/cfgs/applicationContext.xml");
        //get bean class object
        clss = factory.getBean("cls", Class.class);
        System.out.println("Object class name :
"+clss.getName()+"\nGet object data : "+clss.toString());
        System.out.println("-----");
        calender = factory.getBean("cal", Calendar.class);
        System.out.println("Object class name :
"+calender.getClass().getName()+"\nGet object data :
"+calender.toString());
        System.out.println("-----");
        conn = factory.getBean("conn", Connection.class);
        System.out.println("Object class name :
"+conn.getClass().getName()+"\nGet object data : "+conn.toString());
        System.out.println("-----");
        props = factory.getBean("sys", Properties.class);
        System.out.println("Object class name :
"+props.getClass().getName()+"\nGet object data : "+props.toString());
        System.out.println("-----");
        str1 = factory.getBean("s2", String.class);
        System.out.println("Object class name :
"+str1.getClass().getName()+"\nGet object data : "+str1.toString());
        System.out.println("-----");
    }
}

```

```

        System.out.println("-----");
        str2 = factory.getBean("s3", String.class);
        System.out.println("Object class name :
"+str2.getClass().getName()+"\nGet object data : "+str2.toString());
    }

}

```

**Q. Can u configure abstract class /interface as the spring bean in the spring bean configuration file?**

**Ans.** Possible, but we must enable static factory-method bean Instantiation. In this situation IoC container does not create object for the configured abstract class or interface, it will create object for the sub class/ Implementation class.  
e.g.

<!--Static factory method bean instantiation giving related class object-->  
<bean class="java.util.Calendar" factory-method="getInstance"/>

Internal Cache of IoC container	
Key	value
cal1	GregorrialCalendar class object ref

**Note:** In Instance factory method bean instantiation specifying "class" attribute having spring bean class name is optional.

## Singleton java class and Bean Scopes

### Singleton class/ singleton java class

- The java class that allows us to create only one object, in the entire execution of the App is called singleton java class.
- It is GOF Pattern.
- Pre-defined singleton classes java.awt.Desktop, java.lang.Runtime

**Problem:** creating multiple objects for java class when class is not having state or class having read only state(final) or sharable state is waste of memory and CPU time.

**Solution:** In the above situations create only one object for java class and use it for multiple times i.e. we need restrict java class allows the programmers to

create only one object by making it as singleton java class.

**Implementation:** While developing singleton java class close all the doors of creating object for java class from outside of the class but open only one door (static factory method) where u can check for object is available or not before creating and returning object.

- a. Take class as public class to make it visible inside and outside of current package.
- b. Add private only constructors in the java class to stop outsiders to create object by using new operator.
- c. Take private static reference variable as the member variable of the class to hold/ refer the single object that will be created. So, we can we can use this ref variable in static factory method as criteria to create and return new object ref or to return existing object reference.
- d. public Static factory method, that checks the object availability. If available returns that object ref otherwise creates new object and returns that object reference.

#### Singleton class with minimum standards:

```
public class Printer {  
    private static Printer INSTANCE;  
  
    private Printer () {  
        //no body  
    }  
    //static factory method  
    public static Printer getInstance () {  
        if (INSTANCE==null)  
            INSTANCE=new Printer ();  
        return INSTANCE;  
    }  
    //b. method  
    public void printData (String msg) {  
        S.o.println(msg);  
    }  
}  
  
Printer p1=Printer.getInstance();  
Printer p2=Printer.getInstance();  
Printer p3=Printer.getInstance();
```

We can break above singleton class using:

- a. Multi-threaded environment
- b. Reflection API
- c. Using deserialization
- d. Using cloning
- e. Using Custom ClassLoaders

we can solve all these problems using different techniques.

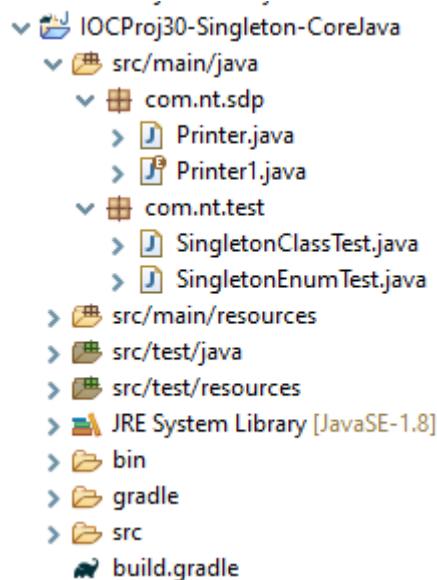
#### Enum based singleton:

```
public enum Printer {  
    //constant of current Enum type  
    INSTANCE;  
    //business logic  
    public void printData (String msg) {  
        S.o.p(msg);  
    }  
}
```

```
Printer p1=Printer.INSTANCE;  
Printer p2=Printer.INSTANCE;  
Printer p3=Printer.INSTANCE;
```

**Note:** By Developing singleton class as Enum. We can solve max of the above problems implicitly. Enums are internally classes.

#### Directory Structure of IOCProj30-Singleton-CoreJava:



- Develop the above directory structure and package, class then use the following code with in their respective file.
- Here need not to any dependencies in build.gradle because it is a simple java program.

### Printer.java

```
package com.nt.sdp;

public class Printer {

    private static Printer INSTANCE;

    //private constructor
    private Printer() {
        System.out.println("Printer : Printer()");
    }

    //static factory method
    public static Printer getInstance() {
        if (INSTANCE==null)
            INSTANCE = new Printer();
        return INSTANCE;
    }

    //business logic
    public void printData(String str) {
        System.out.println(str);
    }
}
```

### Printer1.java

```
package com.nt.sdp;

public enum Printer1 {

    INSTANCE;

    //business logic
    public void printData(String str) {
        System.out.println(str);
    }
}
```

### SingletonClassTest.java

```
package com.nt.test;

import com.nt.sdp.Printer;

public class SingletonClassTest {
    public static void main(String[] args) {
        Printer printer1 = null, printer2 = null;
        printer1 = Printer.getInstance();
        printer2 = Printer.getInstance();
        System.out.println(printer1.hashCode()+"..."+printer2.hashCode());
        System.out.println("is printer1 = printer2 ?
"+(printer1==printer2));
        System.out.println("-----");
        printer1.printData("Hello!");
        printer2.printData("how are you?");
    }
}
```

### SingletonEnumTest.java

```
package com.nt.test;

import com.nt.sdp.Printer1;

public class SingletonEnumTest {
    public static void main(String[] args) {
        Printer1 printer1 = null, printer2 = null;
        printer1 = Printer1.INSTANCE;
        printer2 = Printer1.INSTANCE;
        System.out.println(printer1.hashCode()+"..."+printer2.hashCode());
        System.out.println("is printer1 = printer2?
"+(printer1==printer2));
        System.out.println("-----");
        printer1.printData("Hello!");
        printer2.printData("how are you?");
        Printer1.INSTANCE.printData("Where are you?");
    }
}
```

## Q. Why should we take class as Singleton class?

Ans.

1. If class is not having any state (No member variable declaration).

e.g.

```
public class Arithmetic{  
    public int sum(int x,int y){  
        return x+y;  
    }
```

(bad) Creating multiple objs for a class with no state is bad practice

```
public enum Arithmetic{  
    INSTANCE;  
    //b.method  
    public int sum(int x,int y){  
        return x+y;  
    }  
}
```

(good)

```
public class Arithmetic{  
    private static Arithmetic INSTANCE;  
    private Arithmetic(){ }  
    //factory method  
    public static Arithmetic getInstance(){  
        if(INSTANCE==null)  
            INSTANCE=new Arithmetic();  
        return INSTANCE;  
    }  
    //b.method  
    public int sum(int x,int y){  
        return x+y;  
    }  
}
```

(good)

2. When class is read only state (final variables).

e.g.

```
public class Circle{  
    private final float PI=3.14f;  
    public float calcArea(float radius){  
        return PI*radius*radius;  
    }  
}
```

(Bad) creating multiple objects of a class having fixed state is bad practice.

```
public enum Circle{  
    INSTANCE;  
    private final float PI=3.14f;  
    //b.method  
    public float calcArea(float radius){  
        return PI*radius*radius;  
    }  
}
```

(Good)

```
public class Circle{  
    private final float PI=3.14f;  
    private static Circle INSTANCE=new Circle();  
    private Circle(){ } //eager instantiation  
    //factory method  
    public static Circle getInstance(){  
        return INSTANCE;  
    }  
    // b.method  
    public float calcArea(float radius){  
        return PI*radius*radius;  
    }  
}
```

(Good ) (not recommended)

3. When class is having sharable state (non-final) across the multiple classes other class of the App/Project in Synchronized environment (To avoid multi-threading issues) (e.g. Cache/buffer).

**Sample code:** we will discuss in AOP classes.

## Bean Scopes

- IoC Container can create and manage the objects of spring bean class having different scopes. They are

**spring 1.x**  
singleton (default)  
prototype

**spring 2.x/ 3.x/ 4.x**  
singleton (default)  
prototype  
request  
session  
globalSession  
(removed from 3.x)

**spring 5.x**  
singleton (default)  
prototype  
request  
session  
application  
websocket

(only in web applications)

(only in web applications)

**Note:** use "scope" attribute of <bean> tag to specify spring bean scope.

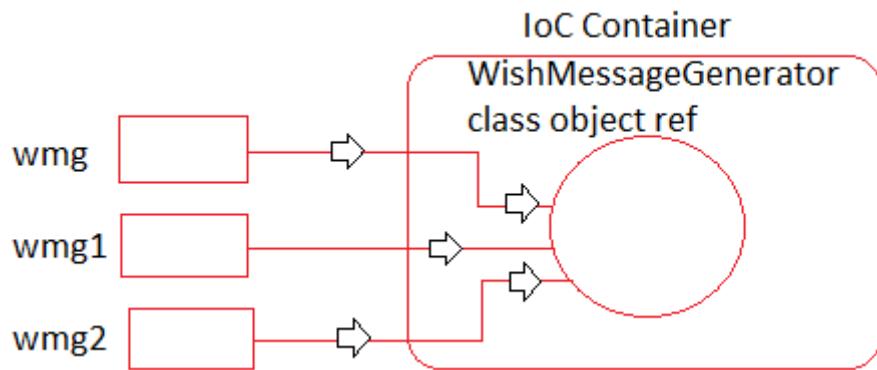
**scope="singleton":**

- It is default scope, if no scope is specified.
- IoC container creates only one object for spring bean class and reuses that object.
- IoC container does not make spring bean class as singleton java class. But it creates only one object, keeps that object in Internal Cache of IoC container and uses that object for multiple times i.e. returns that one object ref by collecting from Internal cache for multiple factory.getBean(-) method calls.

```
<bean id="wmg" class="pkg.WishMessageGenerator"  
      scope="singleton"/>
```

### Client App

```
WishMessageGenerator wmg =factory.getBean("wmg" ,  
                                         WishMessageGenerator.class);  
WishMessageGenerator wmg1 =factory.getBean("wmg" ,  
                                         WishMessageGenerator.class);  
WishMessageGenerator wmg2 =factory.getBean("wmg" ,  
                                         WishMessageGenerator.class);
```



**Note:** IoC container creates only object for each singleton scope spring bean class configuration.

#### applicationContext.xml

```
<bean id="wmg" class="com.nt.beans.WishMessageGenerator"
      scope="singleton"/>
<bean id="wmg1" class="com.nt.beans.WishMessageGenerator"
      scope="singleton"/>
```

#### Internal cache of IoC container

wmg	WishMessageGenerator class object reference
wmg1	WishMessageGenerator class object reference

(wmg1 proves that the "singleton" scope does not make spring bean class as singleton class)

**Note:** With in an IoC container, the bean ids must be unique, but same class we can configure as multiple spring beans with diff bean ids.

#### Directory Structure of IOCProj31-SpringBeanScope:

- ⊕ Copy paste the IOCProj01-SetterInjection-POC application.
- ⊕ Need not to add any new file same structure as IOCProj01-Setter Injection-POC, and change the test class name to BeanScopeTest.java
- ⊕ Add the following code in their respective files

#### 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">
```

```

<!-- Dependent bean configuration -->
<bean id="dt" class="java.util.Date" />

<!-- Target bean configuration -->
<bean id="wmg" class="com.nt.beans.WishMessageGenerator"
scope="singleton">
    <property name="date" ref="dt" />
</bean>

<bean id="wmg1" class="com.nt.beans.WishMessageGenerator"
scope="singleton">
    <property name="date" ref="dt" />
</bean>

</beans>

```

### BeanScopeTest.java

```

package com.nt.test;

import
org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;

import com.nt.beans.WishMessageGenerator;
import com.nt.sdp.Printer;

public class BeanScopeTest {

    public static void main(String[] args) {
        DefaultListableBeanFactory factory = null;
        XmlBeanDefinitionReader reader = null;
        WishMessageGenerator generator1 = null, generator2 = null,
generator3 = null;
        WishMessageGenerator gen1 = null, gen2 = null, gen3 = null;
        Printer printer1 = null, printer2=null;
        // Create Bean Factory IoC Container
        factory = new DefaultListableBeanFactory();
        reader = new XmlBeanDefinitionReader(factory);
        reader.loadBeanDefinitions("com/nt/cfgs/applicationContext.xml");
        // create object
    }
}

```

```

        generator1 = factory.getBean("wmg",
WishMessageGenerator.class);
        generator2 = factory.getBean("wmg",
WishMessageGenerator.class);
        generator3 = factory.getBean("wmg",
WishMessageGenerator.class);
        System.out.println(generator1.hashCode() + "..." +
generator2.hashCode() + "..." + generator3.hashCode());
        System.out.println("generator1==generator2?" + (generator1
== generator2));
        System.out.println("generator2==generator3?" + (generator2
== generator3));
        System.out.println("generator1==generator3?" + (generator1
== generator3));
        System.out.println("-----");
        gen1 = factory.getBean("wmg1",
WishMessageGenerator.class);
        gen2 = factory.getBean("wmg1",
WishMessageGenerator.class);
        gen3 = factory.getBean("wmg1",
WishMessageGenerator.class);
        System.out.println(gen1.hashCode() + "..." + gen2.hashCode() +
"..." + gen3.hashCode());
        System.out.println("gen1==gen2?" + (gen1 == gen2));
        System.out.println("gen2==gen3?" + (gen2 == gen3));
        System.out.println("gen1==gen3?" + (gen1 == gen3));
        System.out.println("-----");
    }

}

```

**scope="prototype":**

- IoC container creates new Object for Spring bean class for every factory.getBean(-) method call.
- IoC container does not keep this scope spring bean class objects in the "Internal Cache of IoC container".

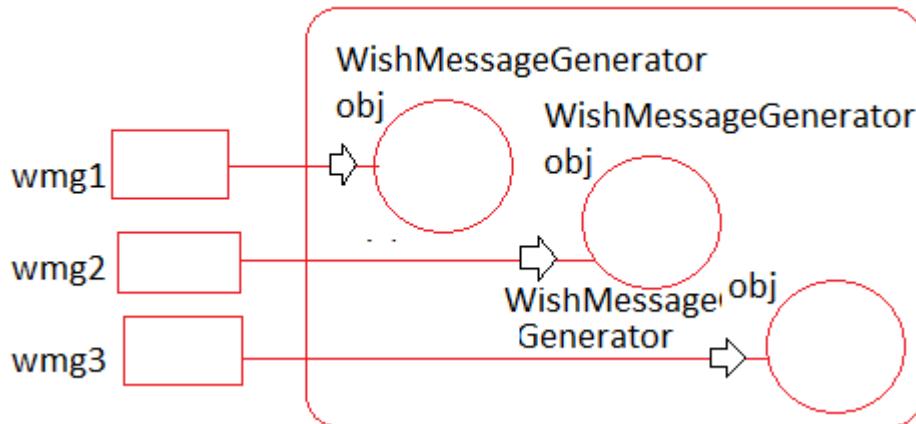
```

<bean id="wmg" class="pkg.WishMessageGenerator"
      scope="prototype"/>

```

### Client App

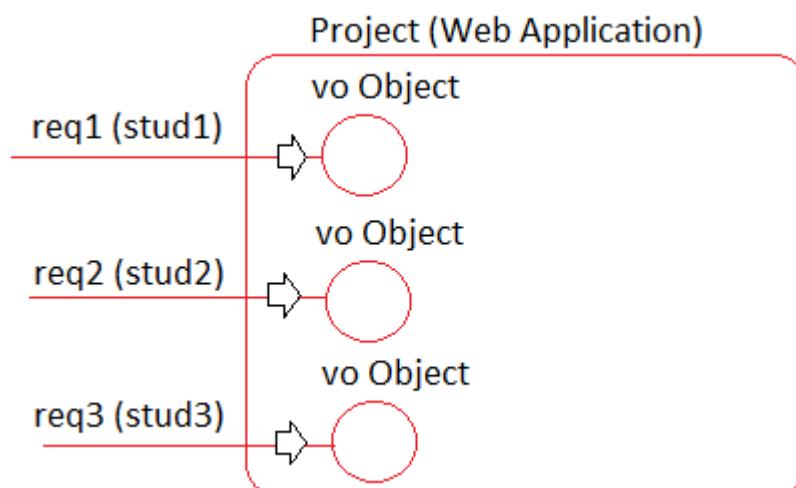
```
WishMessageGenerator gen1 =factory.getBean("wmg" ,  
                                         WishMessageGenerator.class);  
WishMessageGenerator gen2 =factory.getBean("wmg" ,  
                                         WishMessageGenerator.class);  
WishMessageGenerator gen3 =factory.getBean("wmg" ,  
                                         WishMessageGenerator.class);
```



### applicationContext.xml

```
<bean id="wmg1" class="com.nt.beans.WishMessageGenerator"  
scope="prototype">  
    <property name="date" ref="dt" />  
</bean>
```

**Note:** If spring bean class not having state or having only read-only state or sharable state across the other classes then go for configure that java class as singleton scope.



**Q. When should we use singleton scope and prototype scope in real time projects?**

**Ans.** In Realtime layered Applications, we configure DataSource class, DAO classes, Service classes and Controller class as singleton scope spring beans.

Reasons are,

- The State kept DataSource object (JDBC properties, con pool properties) are sharable across the multiple DAO classes. More every multiple DAOs want to use same DataSource object (indirectly same JDBC con pool) to interact with a DB s/w. So, we need to take DataSource class as singleton scope spring bean.
- DAO class contains SQL queries at the top of class as final String variables values (read only) and also DataSource object (having fixed state) So we can take DAO as singleton scope spring bean.
- Service classes contain DAO class objects as state (indirectly fixed state) So, we can take service classes as singleton scope spring beans.
- Controller class is 1 per Project contains service class objects as the state (indirectly fixed state) So, we can take controller class as singleton scope spring bean.
- We can take VO, BO, DTO classes (java beans) as prototype scope spring beans, if we want to configure them as spring beans. Because they need to hold multiple sets of inputs simultaneously (specially in web applications).

**Q. What happens if we configure real singleton java class as spring bean having prototype scope?**

**Ans.**

1. If we do not enable static factory method instantiation, the IoC container creates multiple objects by accessing private 0-param constructor for multiple factory.getBean(-) method calls.

```
<bean class="com.nt.sdp.Printer" scope="prototype"/>
```

2. If we enable static factory method Bean Instantiation, then the IoC container creates only one object for multiple factory.getBean(-) method calls Singleton java class as prototype scope spring bean.

```
<bean id="p1" class="com.nt.sdp.Printer" scope="prototype" factory-method="getInstance"/>
```

- + Add the singleton class along with the package from IOCProj30-Singleton-CoreJava.
- + Then add the following code in their respective files.

#### applicationContext.xml

```
<!-- Factory Spring bean configuration of Singleton class -->
    <!-- <bean id="printerBean" class="com.nt.sdp.Printer"
scope="prototype"/> -->
    <bean id="printerBean" class="com.nt.sdp.Printer" scope="prototype"
factory-method="getInstnnce"/>
```

#### BeanScopeTest.java

```
printer1 = factory.getBean("printerBean", Printer.class);
printer1.printData("Hello");
printer2 = factory.getBean("printerBean", Printer.class);
printer2.printData("Hill");
System.out.println(printer1.hashCode()+"..."+printer2.hashCode());
```

#### **scope=" request" (for http protocol):**

- + IOC container keeps Spring bean class object as request attribute of web application environment i.e. Spring bean class object will be created on 1 per request

**Note:** Here spring bean class object will be maintained as request attribute

#### **scope=" session " (http protocol):**

- + IoC container keeps Spring bean class object as session attribute of web application environment i.e. spring bean class object will be created on 1 per each browser s/w of a client machine.

#### **scope= "application" (http protocol):**

- + IoC container keeps spring bean class object as ServletContext attribute i.e. 1 spring bean class object for entire web application.

#### **scope= "websocket" (websocket protocol):**

- + IoC container keeps spring bean class object as required for websocket Programming.

**Note:** Singleton, prototype scope can be used both in standalone, web application environment.

## ApplicationContext container (IoC Container)

- It is extension of BeanFactory Container.
- To create container, create object for java class that implements org.springframework.context.ApplicationContext(I) (which is a sub interface of BeanFactory(I)).

The popular implementation classes for " ApplicationContext" (I) are:

### 1. FileSystemXmlApplicationContext (In standalone Apps):

- Creates "AC" container by locating spring bean configuration file from the specified path of file system.

e.g.

```
ApplicationContext ctx = new  
    FileSystemXmlApplicationContext(  
        "src/main/java/com/nt/cfgs/applicationContext.xml");
```

### 2. ClassPathXmlApplicationContext (in standalone Apps):

- Creates "AC" container by locating spring bean configuration file from jars and directories added to CLASSPATH or build path.
- Eclipse "src" default folder in CLASSPATH.
- Eclipse src/main/java -> default folder in CLASSPSATH.

e.g.

```
ApplicationContext ctx = new  
    ClassPathXmlApplicationContext(  
        "com/nt/cfgs/applicationContext.xml");
```

### 3. XmlWebApplicationContext (In Spring MVC web applications):

- Creates "AC" container in web application environment by taking <servlet logical name>- servlet.xml of WEB-INF folder as spring bean configuration file. (useful in spring MVC web applications).

### 4. AnnotationConfigApplicationContext (In standalone Apps):

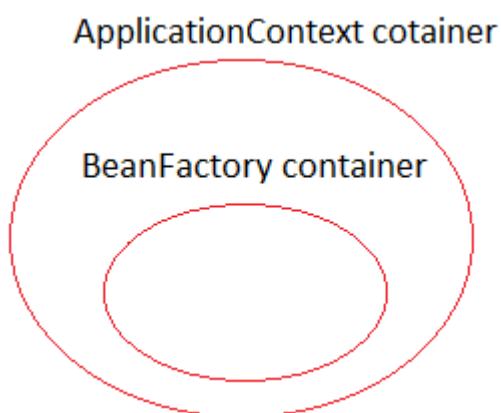
- In 100% code driven, Spring boot driven Spring application development, we replace spring bean configuration file (xml file) with Configuration class (java class).
- This class is useful to create IOC container by taking given Configuration class.

e.g.

```
ApplicationContext ctx = new  
    AnnotationConfigApplicationContext(AppConfig.class);  
    Configuration class
```

## 5. AnnotationConfigWebApplicationContext (In Spring MVC web applications):

- Given to create "ACM container by taking given Configuration class as input class in web application environment.  
and etc.



### Directory Structure of IOCProj33-ApplicationContextContainer:

- Copy paste the IOCProj32-SpringBeanScope application.
- Need not to add any new file same structure as IOCProj32-SpringBeanScope.
- Add the following code in their respective file.

#### BeanScopeTest.java

```
package com.nt.test;  
  
import org.springframework.context.ApplicationContext;  
import  
org.springframework.context.support.FileSystemXmlApplicationContext;  
  
import com.nt.beans.WishMessageGenerator;  
import com.nt.sdp.Printer;  
  
public class BeanScopeTest {  
  
    public static void main(String[] args) {
```

```

ApplicationContext ctx = null;
//create Application Context IoC container
ctx = new
FileSystemXmlApplicationContext("src/com/nt/cfgs/applicationContext.xml
");
WishMessageGenerator generator = null;
Printer printer = null;
// create object
generator = ctx.getBean("wmg", WishMessageGenerator.class);
System.out.println(generator.generateWishMessage("Raja"));
System.out.println("-----");
printer = ctx.getBean("printerBean", Printer.class);
printer.printData("Hello");

//close container
((FileSystemXmlApplicationContext)ctx).close();
}

}

```

**Additional features of ApplicationContext container with respect to BeanFactory Container:**

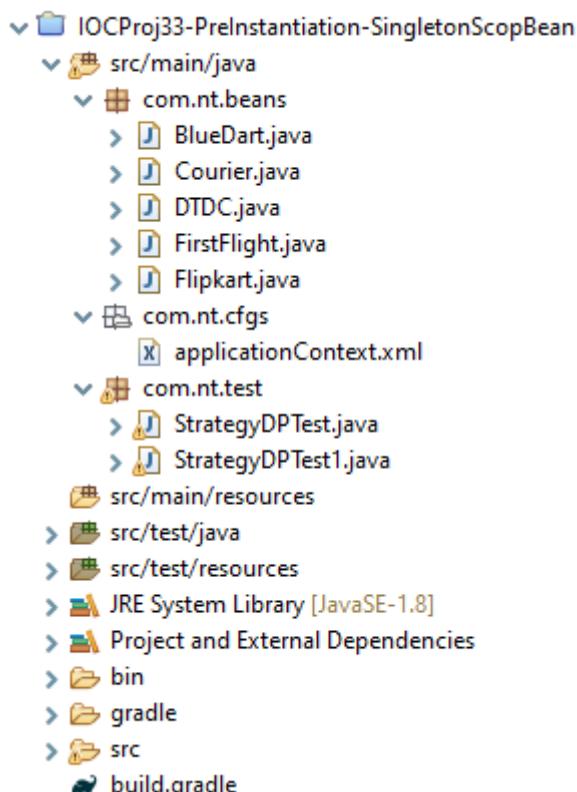
1. Pre-instantiation of singleton scope beans.
2. Ability to work with place holders and Properties file (automatic reorganization).
3. Support for I18n (Internationalization).
4. Support for Event handling & Publishing
5. Ability to stop/ close Container.
6. Automatic registration of BeanPostProcessors.
7. Automatic registration of BeanFactoryPostProcessors.
8. Support for Annotation driven, 100%Code driven and Boot driven Spring Programming  
and etc...

### Pre-instantiation of singleton scope beans

- BeanFactory Container creates spring bean class object only when factory.getBean(-) methods are called not when IoC container is created.  
So, we can say BeanFactory container is performing lazy instantiations (object creations) and Injections.

- The moment ApplicationContext container is created, all singleton scopes will be instantiated and injections takes place on those beans irrespective ctx.getBean(-) methods will be called or not this is called pre-instantiation/ eager instantiation of singleton scope beans.
- If prototype scope bean is dependent to singleton scope bean then that Prototype scope bean will also be pre-instantiated to support pre-instantiation and injection of singleton scope target bean. But it does not the scope of Dependent bean.

### Directory Structure of IOCProj33-PreInstantiation-SingletonScopBean:



- Develop the above directory structure.
- Copy paste build.gradle from any other gradle project because we are using same spring-context-support dependency only.
- Copy all the package and class from IOCProj07-StrategyPattern-Spring and modify accordingly.

#### StrategyDPTTest1.java

```
package com.nt.test;

import org.springframework.beans.factory.support.DefaultListableBeanFactory;
```

```

import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.ClassPathBeanDefinitionScanner;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

import com.nt.beans.Flipkart;

public class StrategyDPTest1 {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        Flipkart fpkt = null;
        // create ApplicationContext container [IoC container]
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        // generate target class object
        fpkt = ctx.getBean("fpkt", Flipkart.class);
        // invoke the method
        //System.out.println(fpkt.shopping(new String[] { "PPT",
        "Mask", "Senitizer" }, new float[] { 500, 200, 350 }));
    }

}

```

**Q. Why Prototype scope beans are not participating in pre-instantiation process directly?**

**Ans.** The object created prototype scope bean as part of pre-instantiation process will always be wasted because for every factory.getBean(-)/  
ctx.getBean(-) method call the IoC container should create separate new object for prototype scope bean.

**Note:**

- ✓ request, session, application, websocket scope beans also will not participate in pre-instantiation process.
- ✓ ServletContainer performs lazy instantiation on Servlet comps by default. To enable Pre-instantiation or eager instantiation on servlet comps enable <load-on-startup> on servlet comps.

**Q. What is the advantage of pre-instantiation of spring beans with respect to Real time Project?**

**Ans.**

html/JSP ---> ControllerServlet ---> service class ---> DAO classes ---> DB s/w  
(<load-on-startup>) (singleton)                      DataSource (Singleton)  
    (singleton)

<load-on-startup> - ApplicationContext container is created in the init () of Controller Servlet comp).

**Having the above setup, if we deploy web application:**

- ControllerServlet class obj is pre-instantiated during the deployment because of <load-on-startup>.
- Init (-) controller Servlet executes and creates ApplicationContext container.
- ApplicationContext container completes pre-instantiation of all singleton scope DataSource, DAO classes, Service classes with necessary injections

**Note:** Now the first request given to Servlet comp directly participates in request processing using already created object i.e. does not waste time for creating objects and doing injections.

**lazy-init:**

- If we want to disable pre-instantiation on singleton scope spring beans by Instructing IoC container then we can use lazy-init="true" of <bean> tag.

```
<bean id="dtdc" class="com.nt.beans.DTDC" scope="singleton" lazy-  
Init=" true"/>
```

**The possible values for "lazy-init" attribute are:**

- a. true
- b. false
- c. default (default)

**lazy-init="true":** Enables lazy instantiation on singleton scope spring bean

**lazy-init="false":** Enables pre-instantiation on singleton scope spring bean

**lazy-init="default":** Fallbacks to "default-lazy-init" attribute of <beans>

The possible values for "default-lazy-init" attribute of <beans> tag is:

**default-lazy-init="true"**: Enables lazy instantiation all singleton scope spring beans current spring bean configuration file

**default-lazy-init="false"**: Enables pre-instantiation on all singleton scope spring beans current spring bean configuration file

**default-lazy-init="default"**: If the singleton scope bean is inner bean/ nested bean then it carries lazy instantiation behaviour from its outer bean configurations otherwise acts default-lazy-init="false".

### applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans default-lazy-init="true"
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">

    <!-- Dependent bean configuration -->
    <!-- <bean id="dtdc" class="com.nt.beans.DTDC" /> -->
    <!-- <bean id="dtdc" class="com.nt.beans.DTDC" lazy-init="true"/> -->
    <bean id="dtdc" class="com.nt.beans.DTDC" lazy-init="default"/>
    <bean id="fFlight" class="com.nt.beans.FirstFlight"/>
    <bean id="bDart" class="com.nt.beans.BlueDart"/>
    <!-- <bean id="bDart" class="com.nt.beans.BlueDart"
scope="prototype"/> -->

    <!-- Target bean configuration -->
    <bean id="fpkt" class="com.nt.beans.Flipkart" lazy-init="true">
        <constructor-arg name="courier" ref="fFlight"/>
    </bean>

</beans>
```

Q. We have 10 spring beans but i want see pre-instantiation only on 6 spring beans, what should we do?

Ans.

- Configure 6 spring beans as singleton scope beans and 4 spring beans as prototype scope beans.

- Do not make prototype scope beans as dependent beans to singleton scope beans.
- Do not enable lazy instantiation directly or indirectly on singleton scope spring beans.

## Ability to work with place holders and Properties file (automatic reorganization)

properties file:

- It is the text file that maintains the entries in the form of key=value pairs. It is very useful to pass info to Application from outside of the Application (soft coding)

info.properties

```
#personal info
nit.name=raja
nit.age=30
nit.addrs=hyd
```

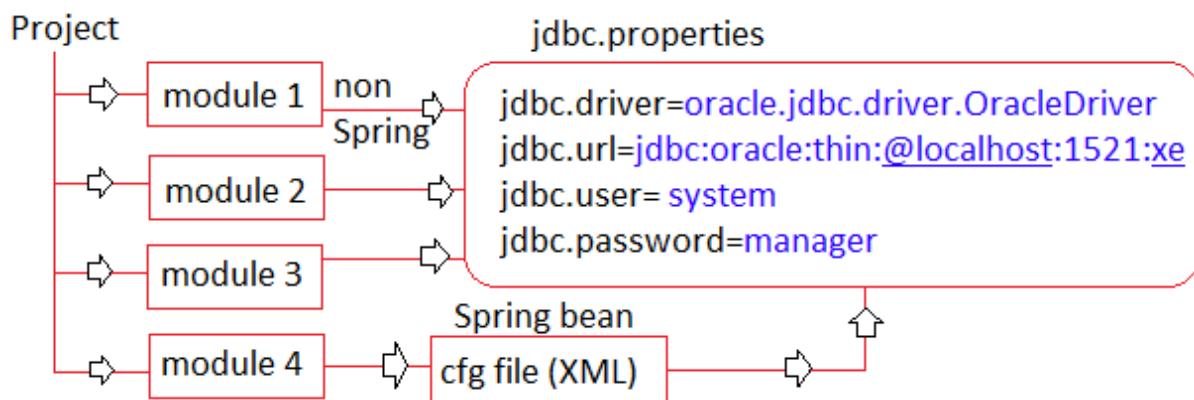
JDBC properties (driver class name, URL, DB username, password and etc.) will not be hardcoded inside the application they will be soft code with the support of properties file.

```
int a=10; // hardcoding
int a=sc.nextInt(); // soft coding
collecting from end-user using command line args, sys properties, streams, properties file, xml file and etc.
```

Q. Why should we encourage properties file in Spring Applications when it already supporting soft coding through xml file (spring bean configuration file)?

Ans.

1. Editing properties file is quite easy when compare to editing xml file as developer/ end-user.
2. If multiple non-spring modules of project are getting JDBC properties from common properties file, then the newly added module which is developed in spring should also get JDBC properties from same properties file to support the centralization of properties file.



3. While working with 100%code driven, spring boot driven Spring application development. We must avoid xml file then we should gather inputs with the support of properties file.

**Example on Linking spring bean configuration file (xml file with properties file):**

**Step 1:** Keep Miniprojector App ready.

**Step 2:** Add Properties file having JDBC properties.

**Step 3:** Configure Properties file with spring bean file by using "PropertyPlaceHolderConfigurer" specifying the name and location of properties file.

```
<bean id="pphc"
      class="org.springframework.beans.factory.config.PropertyPlaceholder
Configurer">
    <property name="location"
              value=.com/nt/commons/jdbc.properties"/>
</bean>
```

Makes the underlying AC container to recognize the place in applicationContext.xml holders to collect their values from the given properties file

**Step 4:** Keep place holders \${<key>} in spring bean configuration file specifying keys of the properties file to collect values from properties file.

**Step 5:** Develop the Client App using ApplicationContext container.

**Directory Structure of IOCProj34-RealTimeDI-PropertiesFile:**

- + Copy paste the Layered application and change rootProject.name to IOCProj34-RealTimeDI-PropertiesFile in settings.gradle file.
- + Add a package com.nt.commons and place a properties file jdbc.properties.
- + Add the following code in their respective files.

#### jdbc.properties

```
#Oracle details
jdbc.driver=oracle.jdbc.driver.OracleDriver
jdbc.url=jdbc:oracle:thin:@localhost:1521:xe
jdbc.user= system
jdbc.password=manager
```

## applicationContext.xml

```
<!-- Configure the DataSource for Oracle-->
<bean id="dmbs"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${jdbc.driver}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.user}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
<!-- Configure Oracle DataSoruce object to Oracle DAO class -->
<bean id="oracleCustDAO"
class="com.nt.dao.OracleCustomerDAOImpl">
    <constructor-arg ref="dmbs"/>
</bean>
<!-- Configure MySQL DataSoruce object to MySQL DAO class -->
<bean id="mysqlCustDAO"
class="com.nt.dao.MySQLCustomerDAOImpl">
    <constructor-arg ref="dmbs"/>
</bean>
<!-- Configure DAO object to Service class -->
<bean id="custService"
class="com.nt.service.CustomerMgmtServiceImpl">
    <constructor-arg ref="oracleCustDAO"/>
    <!-- <constructor-arg ref="mysqlCustDAO"/> -->
</bean>

<!-- Configure Service object to Controller class -->
<bean id="controller" class="com.nt.controller.MainController">
    <constructor-arg ref="custService"/>
</bean>
<!-- Configuration of PlaceHolder -->
<bean id="pphc"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location"
value="com/nt/commons/jdbc.properties"/>
</bean>

</beans>
```

## RealTimeDITest.java

```
package com.nt.test;

import java.util.Scanner;

import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.nt.controller.MainController;
import com.nt.vo.CustomerVO;

public class RealTimeDITest {

    public static void main(String[] args) {
        Scanner sc = null;
        String name=null, address=null, Amount=null, time=null,
rate=null;
        CustomerVO vo = null;
        ApplicationContext ctx = null;
        MainController controller = null;
        String result = null;
        //Read inputs from end-user using scanner
        sc = new Scanner(System.in);
        System.out.println("Enter following Details for registration:");
        System.out.print("Enter Customer Name :");
        name = sc.next();
        System.out.print("Enter Customer Address :");
        address = sc.next();
        System.out.print("Enter Customer Principle Amount :");
        Amount = sc.next();
        System.out.print("Enter Customer Time :");
        time = sc.next();
        System.out.print("Enter Customer Rate of Interest: ");
        rate = sc.next();
        //Store into VO class object
        vo = new CustomerVO();
```

```

        vo.setCname(name);
        vo.setCadd(address);
        vo.setpAmt(Amount);
        vo.setTime(time);
        vo.setRate(rate);
        //Create BeanFactory [IoC] container
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        //get controller class object
        controller = ctx.getBean("controller", MainController.class);
        //invoke methods
        try {
            result = controller.processCustomer(vo);
            System.out.println(result);
        } catch (Exception e) {
            System.out.println("Internal probelm : " + e.getMessage());
            e.printStackTrace();
        }
    } //main
} //class

```

Using the above "PropertyPlaceholderConfigurer" setup, we can collect and inject following values to Spring beans:

- values from the configured properties file
- system properties values like os.name, java.vm.vendor and etc.
- Environment variable values like path

```

<property name="osName" value="S{os.name}" /> //here os.name is
system property name
<property name="path" value="S{path}" /> // here path is environement
variable name

```

**Note:** Instead of configuring "PropertyPlaceholderConfigurer" as spring bean, we can add the following tag in spring bean configuration file.

```

<context:property-placeholder location="com/nt/commons/jdbc.properties" />

```

**context:** import does not allow to context.  
**property-placeholder:** get values from name space environment variables.

To work with multiple properties files:

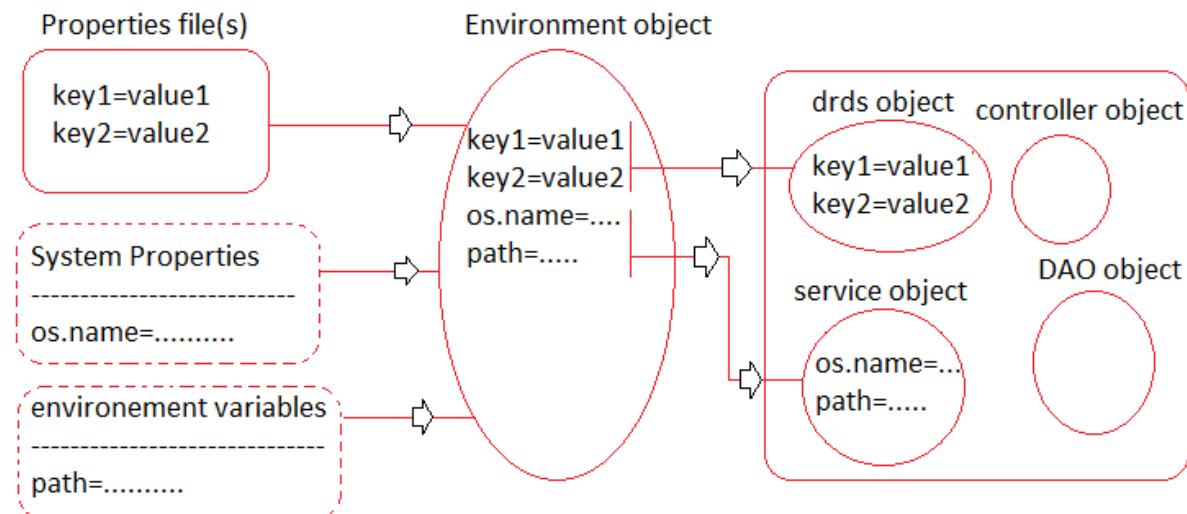
```
<context:property-placeholder location="com/nt/commons/jdbc1.properties,  
com/nt/commons/jdbc.properties "/>
```

(or)

```
<bean id="pphc"  
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  
    <property name="locations" >  
      <array>  
        <value>com/nt/commons/jdbc1.properties</value>  
        <value>com/nt/commons/jdbc1.properties</value>  
      </array>  
    </property>  
</bean>
```

**Note:** If multiple properties files are having same keys with different values then the lastly configured properties file data will override the remaining properties files data.

Internal flow:



## Support for I18n (Internationalization)

I18N (Internationalization):

- Making our App working for different locales is called 118n.
- Locale means language + country.
  - en-US: English as it speaks in US
  - en-BR: English as it speaks in Brittan
  - fr-FR: French as it speaks in France

fr-CA: French as it speaks in Canada  
de-DE: German as it speaks in Germany  
hi-IN: Hindi as it speaks in India  
tu-IN: Telegu as it speaks in India

- By enabling 118n on our App we can make our product or App giving service to multiple clients (different locale clients).
- I18n is all about changing presentation logic as required for end-user, and attracting different Locale end-user to operate the Application.
- While working with 118n we need to take care
  - a. Presentation labels (Using the support of multiple properties files)
  - b. Number formats
  - c. Currency symbols
  - d. Date, time formats
  - e. Indentation

**Note:** In JDK environment we can use `java.util.Locale` , `java.util.ResourceBundle` classes to enabled I18N support.

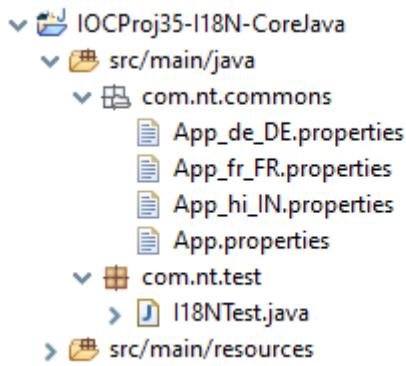
#### To enable 118n for presentation labels:

- a. Take multiple properties files on 1 locale basis including one base properties file.
- b. All properties files should have same keys and diff values collected from the google translators.
- c. Base file name should reflect in all other properties file names.
- d. All properties files extension should be .properties
  - App.properties (base file-En English)
  - App\_fr\_FR.properties ( fr- french)
  - App\_de\_DE.properties (de —german)
  - App\_hi\_IN.properties (hi- hindi)

**Note:** Java is based on Unicode character set i.e. we can render the outputs of Java App using any language alphabets including Indian languages.

#### Directory Structure of IOCProj35-I18N-CoreJava:

- Develop the below directory structure and package, class, properties file then uses the following code with in their respective file.
- Here we don't add any dependencies because it is a simple core java application, so you can create as normal application also.



### App.properties

```
#base properties file (English)
btn1.cap=insert
btn2.cap=update
btn3.cap=delete
btn4.cap=view
```

### App\_fr\_FR.properties

```
#Properties file (French)
btn1.cap = insérer
btn2.cap = mise à jour
btn3.cap = supprimer
btn4.cap = vue
```

### App\_de\_DE.properties

```
#Properties file (German)
btn1.cap = Einfügen
btn2.cap = update
btn3.cap = löschen
btn4.cap = Ansicht
```

### App\_hi\_IN.properties

```
#Properties file (Hindi)
btn1.cap = \u0921\u093E\u0932\u0928\u0947
btn2.cap = \u0905\u0926\u094D\u092F\u0924\u0928
btn3.cap = \u0939\u091F\u093E\u0928\u0947
btn4.cap = \u0926\u0943\u0936\u094D\u092F
```

## I18NTest.java

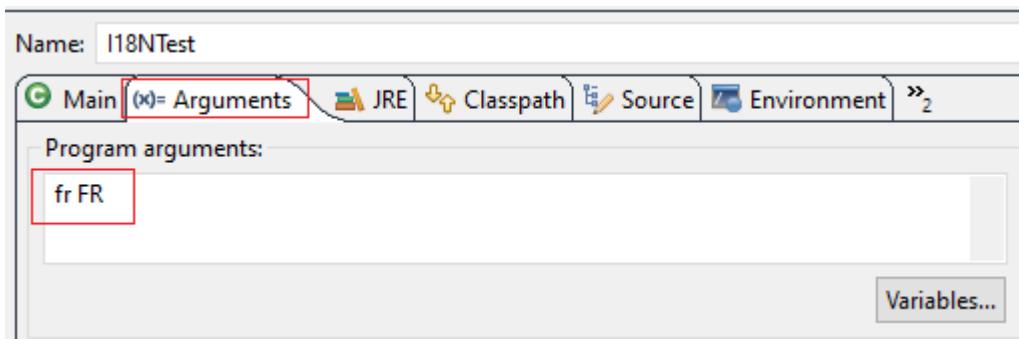
```
package com.nt.test;

import java.util.Locale;
import java.util.ResourceBundle;

public class I18NTest {

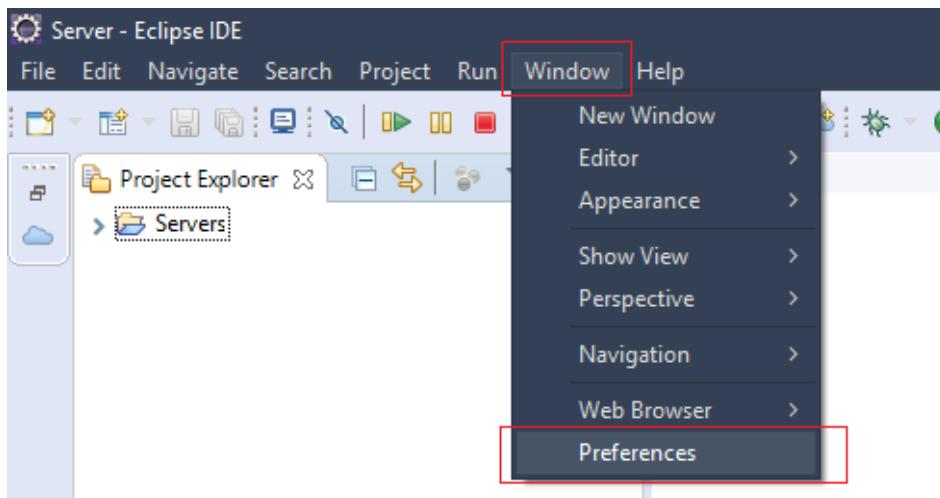
    public static void main(String[] args) {
        Locale locale = null;
        ResourceBundle bundle = null;
        //prepare local object having language and country
        locale = new Locale(args[0], args[1]);
        //create ResourceBundle object having Locale object
        bundle = ResourceBundle.getBundle("com/nt/commons/App",
        locale);
        //read and display values
        System.out.println(bundle.getString("btn1.cap")+
        "+bundle.getString("btn2.cap")+" "+bundle.getString("btn3.cap")+
        "+bundle.getString("btn4.cap"));
    }
}
```

- You have to pass argument by using Run As – Run Configurations option. And the arguments like below, then click on Run then you will get the output.

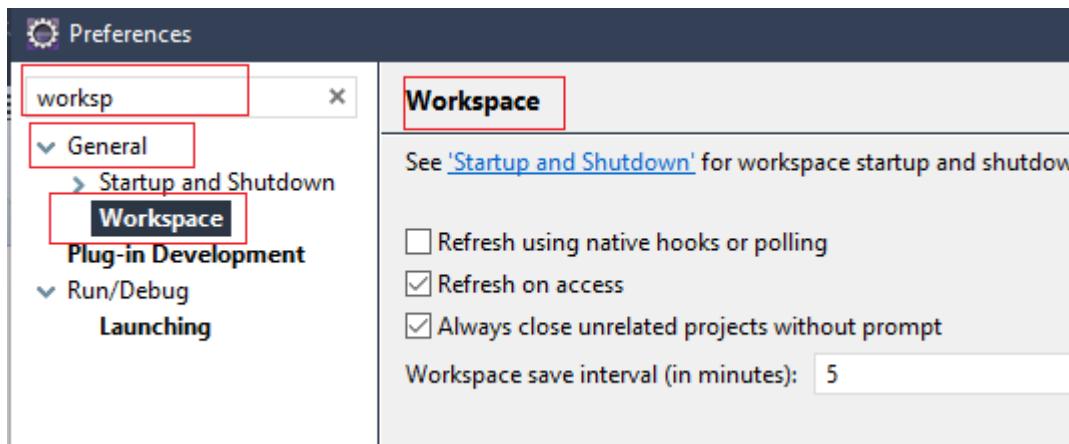


- For show the Hindi language in your console you have to change your workspace Text file encoding to UTF-8, follow the below steps.

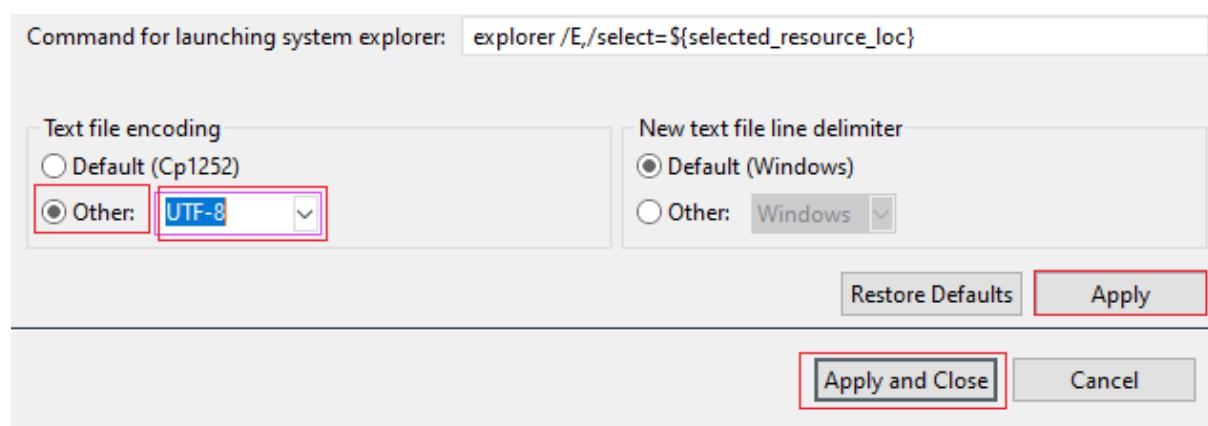
**Step #1:** Click on Window, go to Preferences



**Step #2:** Search for workspace then you will see Workspace option in General section, click on that then Workspace window will be opened.



**Step #3:** Now go bottom then you will get Text file encoding section their you choose other radio button and choose UTF-8 then click on Apply and then Apply and Close.



**Note:** java.util.Locale object holds language and country where as ResourceBundle object reads and holds specific properties Locale specific file info based on "base name" and Locale object data we supply.

**Limitations with I18N in JSE/JEE environment (Non-Spring Standalone, web application environment):**

- a. If we enable 118n in java web application, either we should create ResourceBundle object in every web component or we should place either request scope or session scope based on the requirement i.e. we must spend some time to analyze and decide the scope.
- b. If want to display messages collected from multiple properties files in a single web page then we need create multiple objects for ResourceBundle class.

```
//prepare local object having language and country
Locale locale1 = new Locale("fr", "FR");
Locale locale2 = new Locale("hi", "IN");
//create ResourceBundle object having Locale object
ResourceBundle bundle1 =
    ResourceBundle.getBundle("com/nt/commons/App", locale1);
ResourceBundle bundle2 =
    ResourceBundle.getBundle("com/nt/commons/App", locale2);
```

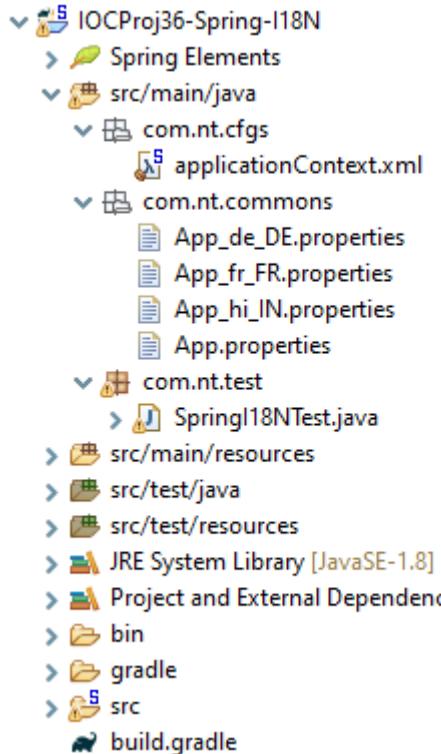
**To overcome the above problems, take the support spring based I18N:**

- Spring's ApplicationContext container gives built-in support for I18N.
- We need to use ctx.getMessage() method to get message from properties file based on Locale object data that we pass.
- We need cfg "ResourceBundleMessageSource" as spring bean having fixed bean id "messageSource" and specifying base file name.

**Note:** If matching locale specific properties file is not found for the given Locale object data, then its fallbacks to base properties file.

**Directory Structure of IOCProj36-Spring-I18N:**

- Develop the above directory structure and package, class, XML file then use the following code with in their respective file.
- Copy paste build.gradle from any other gradle project because we are using same spring-context-support dependency only.
- Copy the all-properties file from IOCProj35-I18N-CoreJava.



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

    <bean id="messageSource"
          class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename"
                  value="com/nt/commons/App"/>
    </bean>

</beans>
```

### SpringI18NTest.java

```
package com.nt.test;

import java.util.Locale;
import java.util.ResourceBundle;
```

```

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

public class SpringI18NTest {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        Locale locale = null;
        String cap1 = null, cap2 = null, cap3 = null, cap4 = null;
        //Create AC IoC container
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        //prepare local object having language and country
        locale = new Locale(args[0], args[1]);
        //get messages
        cap1 = ctx.getMessage("btn1.cap", null, locale);
        cap2 = ctx.getMessage("btn2.cap", null, locale);
        cap3 = ctx.getMessage("btn3.cap", null, locale);
        cap4 = ctx.getMessage("btn4.cap", null, locale);
        System.out.println(cap1+ " "+cap2+ " "+cap3+ " "+cap4);
    }
}

```

**Q. What is the difference ctx.getBean() and ctx.getMessage() method?**

**Ans.** ctx.getBean(-) gives spring bean class object based on given bean id whereas ctx.getMessage(-) method gives message from Locale specific properties file or Base properties file based on key and Locale object data that are supplied.

**Note:** ctx.getMessage() internally calls ctx.getBean(-) having fixed bean id "messageSource".

**Q. Why should we give fixed bean id "messageSource" for ResourceBundleMessageSource class configuration?**

**Ans.** ctx.getMessage(-) method internally called ctx.getBean(-) with fixed bean id "messageSource" to get ResourceBundleMessageSource class object and to get basename property value from it. So, we should also give same bean id as fixed bean id while configuration "ResourceBundleMessageSource".

Each Message in properties file can have max of 5 arguments {0},{1}, ..., {4} we can supply these argument values from our Application as String[] or Object[] values through to ctx.getMessage(-, -, -, -) as show below.

#### App.properties

```
btn1.cap=insert {0}
```

#### Client App

```
String cap1 = ctx.getMessage("btn1.cap", new String[] {"Student"}, "msg1",  
locale);
```

#### Above line description

cap1 - gives insert student

btn1.cap - keys or code in properties file

new String[] {"student"} - Message argument value {0}

"msg1" - Default message

locale - Locale object

#### Spring 118N Advantages:

- a. Using single object of " ResourceBundleMessageSource" we can get messages from either one locale specific properties file or from multiple locales specific properties files.
- b. In Web application environment we need not worry about keeping " ResourceBundleMessageSource object in a scope. By just creating object the IOC Container uses in multiple places.
- c. There is an ability to pass default messages as fallback messages.
- d. There is a facility to pass argument values {} to {4} to messages.

## Support for Event handling & Publishing

#### Event Handling in spring:

- Event is an action performed on the comp or object.  
e.g. clicking on the button, loading the page, opening window, creation of object, creating of container, closing of container and etc.
- Executing some logic when event is raised is called event handling, for this we need event handling methods supplied event Listeners.
- Event Listener/ Handler is a class that implements java.awt.EventListener(I) having Event handling logics in the event handling methods.
- Servlet Listeners are also some kind of Event Listeners.
- We do not call Event handling methods manually they will be called

automatically when the event is raised.

**Note:** In Java every event is an object created by JVM internally.

- For Event handling we need 4 details
  - a. Source object - Button (c) object
  - b. Event class - java.awt.ActionEvent (c) ( when we click the Button)
  - c. Event Listener - java.awt.ActionListener (l)
  - d. Event handling method –  
`public void actionPerformed(ActionEvent ae) {}`
- Event Handling on ApplicationContext container is possible and helps us to know when the ApplicationContext container is started and ended/stopped/closed. Using this we can evaluate the performance of ApplicationContext container.
- With respect to Spring ApplicationContext container
  - a. Source object - ApplicationContext object
  - b. Event class - ApplicationEvent (c) [will be raised when container created and stopped/closed]
  - c. Event Listener - ApplicationListener (l)
  - d. Event handling method:  
`public void onApplicationEvent( ApplicationEvent e) {}`  
This method is executes when IoC container creates/ started and stopped/ closed.

**Note:** Event handling methods are generally callback method i.e. we do not call them manually they will be called automatically.

#### Steps for event handling:

**Step 1:** Keep any ApplicationContext container-based App ready.

**Step 2:** Develop EventHandler/listener class implementing ApplicationListener(l) having logic in onApplicationEvent(-) method.

**Step 3:** Configure Listener class as spring bean in spring bean configuration file.

**Step 4:** Run the Client App (make sure that ctx.close() is called)

#### Directory Structure of IOCProj15-RealTimeDI-EventHandling:

- ⊕ Copy paste the Layered application and change rootProject.name to IOCProj15-RealTimeDI-CollectionInjection in settings.gradle file.
- ⊕ Need not to add a package com.nt.listener having a class IOCContainerMonitoringListener.java.
- ⊕ Use ApplicationContext container in Test class i.e. RealTimeDITest.java.

- >Add the following code in their respective files.

### IOCContainerMonitoringListener.java

```
package com.nt.listener;

import java.util.Date;

import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationListener;

public class IOCContainerMonitoringListener implements
ApplicationListener {

    private long start, end;

    @Override
    public void onApplicationEvent(ApplicationEvent event) {
        if (event.toString().indexOf("ContextRefreshedEvent")!=-1) {
            start = System.currentTimeMillis();
            System.out.println("IoC container is started at : "+new
Date());
        }
        else if (event.toString().indexOf("ContextClosedEvent")!=1) {
            start = System.currentTimeMillis();
            System.out.println("IoC container is stoped at : "+new
Date());
            System.out.println("IoC Container active duration is :
"+(end-start));
        }
    }
}
```

### applicationContext.xml

```
<bean class="com.nt.listener.IOCContainerMonitoringListener"/>
```

- All Listener classes of any environment directly or indirectly implements `java.awt.EventListener()`. After performing pre-instantiation of singleton

scope beans and related injection all Listener classes that are configured as spring beans will also be instantiated irrespective their scopes.

## Difference b/w BeanFactory container and ApplicationContext container

BeanFactory	ApplicationContext
a. Does not support pre-instantiation of spring beans.	a. Supports pre-instantiation of singleton scope beans.
b. No Direct Support for properties files and placeholders.	b. There is direct support.
c. No support for I18N.	c. Supports.
d. Does not support Event Handling and Event Publication.	d. Supports.
e. No support for closing/ stopping Container.	e. We can stop/close Container (ctx.close()/ctx.stop()).
f. Does not support Annotation based programming.	f. Supports.
g. Supports only XML driven configuration.	g. Supports xml driven, annotation driven configuration, 100%code/ java Config and Spring boot configuration.
h. No Automatic registration of BeanPostProcessors.	h. Supports Automatic registration of BeanPostProcessors.
i. No Automatic registration of BeanFactoryPostProces.	i. Supports Automatic registration of BeanFactoryPostProcessor.
j. Uses bit less memory.	j. Uses bit extra memory.
k. Cannot be used in Spring MVC based web application development.	k. Can be used.

### Q. When should we use BeanFactory Container and when should we use ApplicationContext container?

**Ans.** Always prefer using ApplicationContext container in applications because it's supporting all features of Spring but if application is memory critical Application (1 or 2 extra kilo bytes are matters) like mobile Apps, Embedded System/ Microcontroller Applications and really not using ApplicationContext container features then go for BeanFactory Container otherwise always prefer using Application Container in all kinds of Applications including standalone, web, enterprise Applications.

## P-Namespace and C-Namespace

- XML schema namespaces is a library that contains set of XML tags.
- Every XML schema namespace is identified with its namespace URI/ URL.
- To use XML schema space in our XML file we must import namespace URI/ URL in the XML file.
- Spring gives multiple built-in xml schema name spaces like,

Namespace	Namespace URI/ URL
beans	<a href="http://www.springframework.org/schema/beans">http://www.springframework.org/schema/beans</a> <Website URL>/schema/beans
c	<a href="http://www.springframework.org/schema/c">http://www.springframework.org/schema/c</a>
p	<a href="http://www.springframework.org/schema/p">http://www.springframework.org/schema/p</a>
context	<a href="http://www.springframework.org/schema/context">http://www.springframework.org/schema/context</a>

### Note:

- ✓ P namespace is given alternate to the lengthy <property> tag to perform setter Injection configurations.
- ✓ c namespace is given alternate to the lengthy <constructor-arg> to perform constructor injection configuration.

```
public class Employee {  
    private int eno;  
    private String ename;  
    private Date dob;  
    private Dept dept;  
    //setters  
    //toString()  
}
```

```
public class Department {  
    private int dno;  
    private String dname;  
    private Date dos;  
    //3 param constructor  
    .....  
    //toString()  
    .....
```

### application.xml (Old style)

```
<bean id="dob" class="java.util.Date">  
    <property name="year" value="90"/>  
    <property name="month" value="11"/>  
    <property name="date" value="30"/>  
</bean>  
<bean id="dos" class="java.util.Date">  
    <property name="year" value="100"/>  
    <property name="month" value="5"/>  
    <property name="date" value="20"/>  
</bean>  
<bean id="dept" class="pkg.Departmement">  
    <constructor-arg value="5001"/>  
    <constructor-arg value="IT"/>  
    <constructor-arg ref="dos"/>  
</bean>  
<bean id="emp" class="pkg.Employee">  
    <property name="eno" value="1001"/>  
    <property name="ename" value="rakesh"/>  
    <property name="dob" ref="dob"/>  
    <property name="dept" ref="dept"/>  
</bean>
```

**p namespace-based syntax for setter Injection:**

```
<bean p:<property name>="value" p:<property name>-ref=<bean id>/>
    for simple property    for object/ref type property
```

**c namespace-based syntax for constructor Injection:**

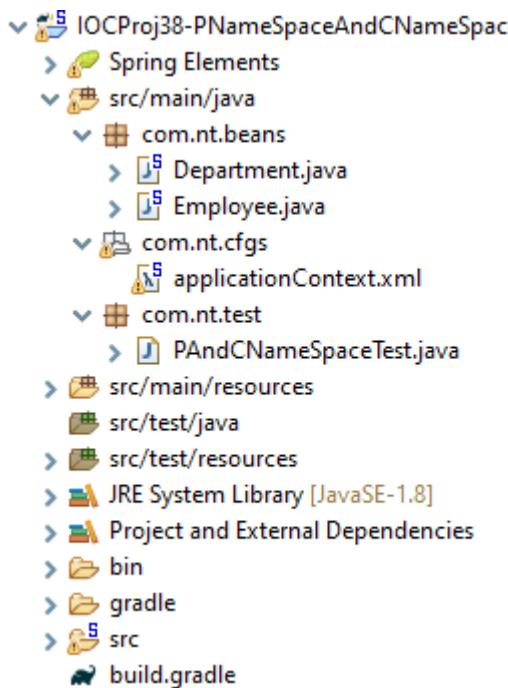
```
<bean c:<property name>="value" c:<property name>-ref=<bean id>/>
    for simple property    for object/ref type property
```

### applicationContext.xml

```
<beans>
    <bean id="dob" class="java.util.Date" p:year="90" p:month="11"
          p:date=" 30"/>
    <bean id="dos" class="java.util.Date" p:year="100" p:month="5"
          p:date=" 20"/>
    <bean id="dept" class="pkg.Department" c:dno="5001"
          c:dname="IT" c:dos-ref="dos"/>
    <bean id="emp" class="pkg.Employee" p:eno="1001"
          p:ename="rakesh" p:dob-ref="dob" p:dept-ref="dept"/>
</beans>
```

**Note:** Both Containers (BF, AC) supports P-namespace, c-namespace based Programming.

### Directory Structure of IOCProj38-PNameSpaceAndCNameSpace:



- Develop the above directory structure and package, class, XML file then use the following code with in their respective file.
- Copy paste build.gradle from any other gradle project because we are using same spring-context-support dependency only.
- Form here we use Lombok for generating dynamic setter, getter, toString, constructor and many more.

### build.gradle

```
dependencies {
    // https://mvnrepository.com/artifact/org.springframework/spring-
    context-support
    implementation group: 'org.springframework', name: 'spring-context-
    support', version: '5.2.8.RELEASE'
    // https://mvnrepository.com/artifact/org.projectlombok/lombok
    implementation group: 'org.projectlombok', name: 'lombok', version:
    '1.18.12'
}
```

### Department.java

```
package com.nt.beans;

import java.util.Date;

import lombok.AllArgsConstructor;
import lombok.ToString;

@AllArgsConstructor
@ToString
public class Department {
    private int dno;
    private String dname;
    private Date dos;
}
```

### Employee.java

```
package com.nt.beans;

import java.util.Date;
```

```

import lombok.Setter;
import lombok.ToString;

@Setter
@ToString
public class Employee {
    private int eno;
    private String ename;
    private Date dob;
    private Department dept;
}

```

### 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"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans-4.3.xsd">

    <bean id="dob" class="java.util.Date" p:year="90" p:month="11"
          p:date="30"/>
        <bean id="dos" class="java.util.Date" p:year="100" p:month="05"
              p:date="20"/>
            <bean id="dept" class="com.nt.beans.Department" c:dno="5001"
                  c:dname="IT" c:dos-ref="dos"/>
                <bean id="emp" class="com.nt.beans.Employee" p:eno="30341"
                      p:ename="NimuDEv" p:dob-ref="dob" p:dept-ref="dept"/>

</beans>

```

### PAndCNameSpaceTest.java

```

package com.nt.test;

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

```

```

import com.nt.beans.Employee;

public class PAndCNameSpaceTest {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        Employee employee = null;
        //create AC IoC container
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        //get Employee object or bean
        employee = ctx.getBean("emp", Employee.class);
        System.out.println(employee);
        ((AbstractApplicationContext) ctx).close();
    }

}

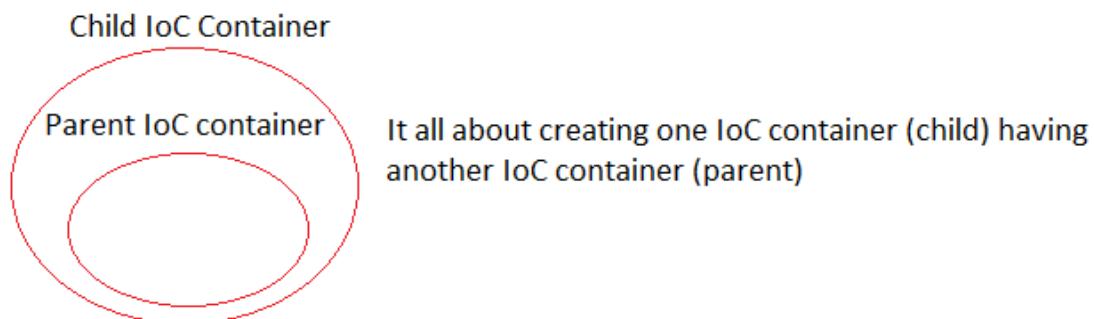
```

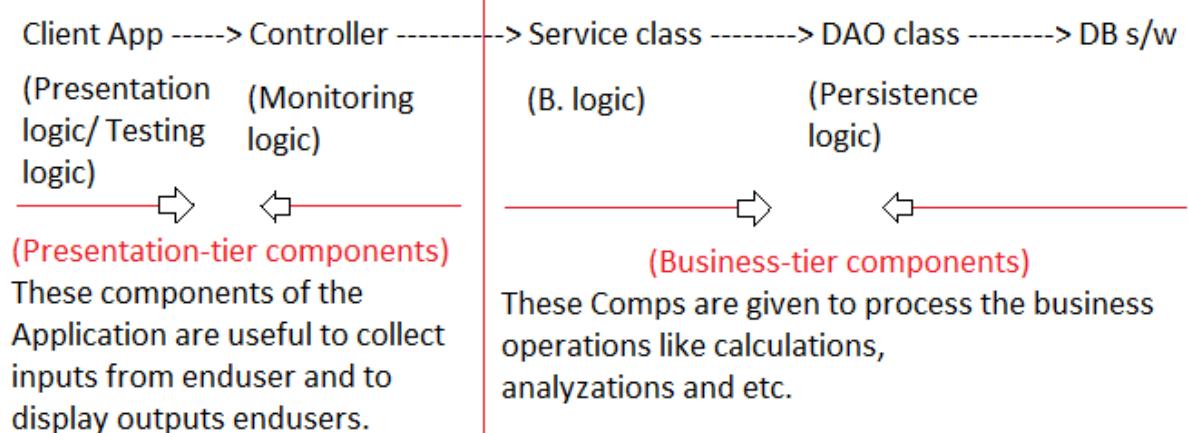
#### Limitations of P and C namespaces:

- These namespaces do not support inner beans configurations.
- These namespaces do not support collection/ Array Injections.
- These namespaces do not support Collection merging.
- These namespaces do not support Null Injections
- Does not allow to resolve constructor params by type, by index or by order (allows to resolve only by name).
- Came lately when industry is moving towards Annotation driven programming.

**Note:** We can mix-up both <property> tags with p namespace and <constructor-arg> tags with c namespace in a single application.

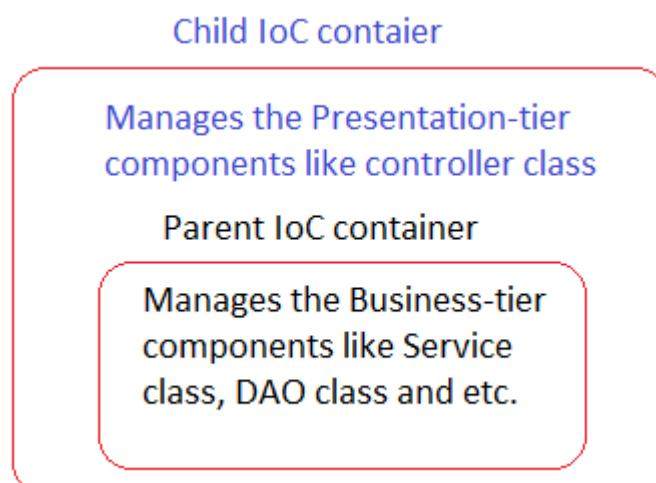
## Nested BeanFactory/ Nested ApplicationContext





#### Note:

- ✓ We should always think about developing presentation tier components and business tier components having loose coupling i.e. the degree of dependency must be very less between both tiers. This can be achieved by taking two separate IoC containers 1 for Presentation tier components and another for business tier components. This helps us to enable to disable Spring support in each tier complements irrespective other tier components.
- ✓ Taking two independent IoC containers is not going to work out completely, because we need to inject Service class object to Controller class object i.e., we need to keep Two Containers are parent and child IoC container. Since Parent IoC container bean can be injected to Child IoC container Bean and reverse is not possible So we need to take business-tier comps in parent IOC container and presentation tier comps in child IOC container. This allows to inject Service class object to Controller class object.



**To create Nested ApplicationContext container:**

```
//parent IOC container  
ApplicationContext parentCtx = new ClassPathXmlApplicationContext("com/nt/cfgs/business-beans.xml");  
//child IOC container  
ApplicationContext childCtx=new ClassPathXmlApplicationContext("com/nt/cfgs/presentation-beans.xml", parentCtx);
```

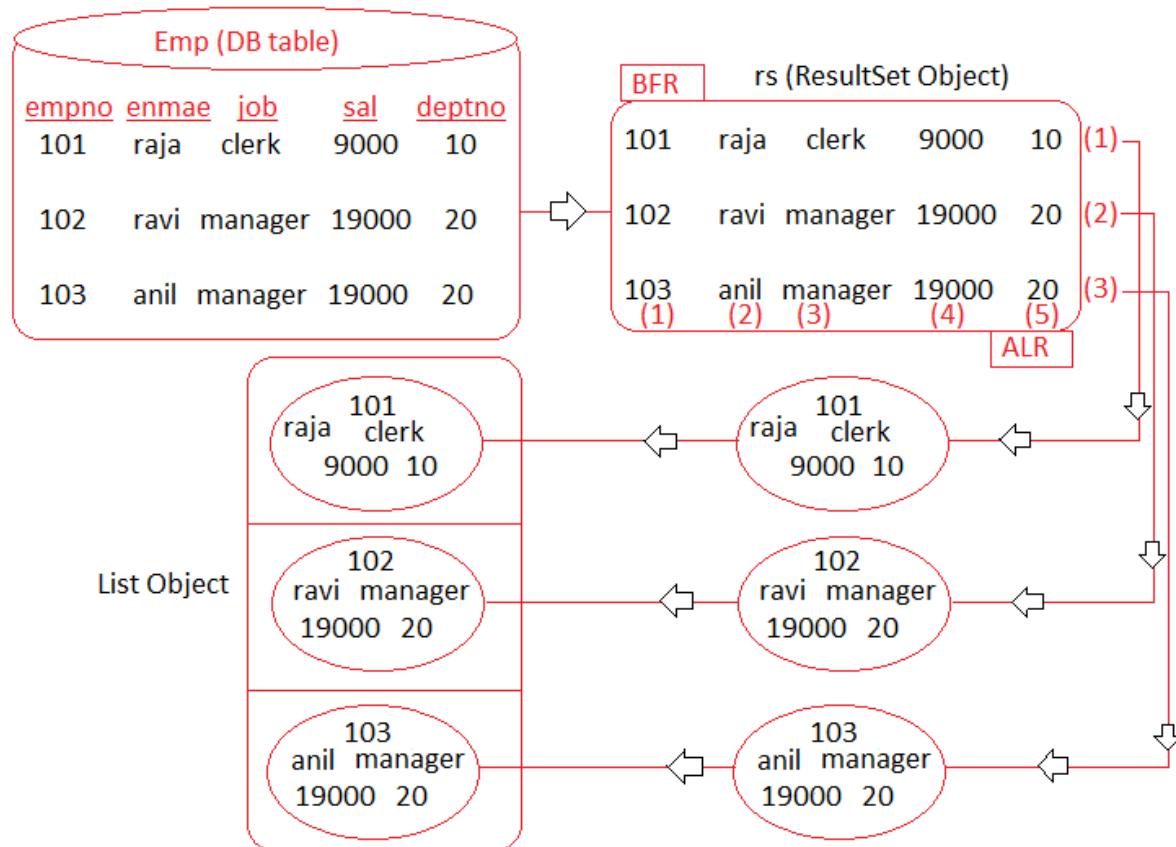
**To create Nested BeanFactory container:**

```
//parent Container  
DefaultListableBeanFactory parentFactory=new DefaultListableBeanFactory();  
XmlBeanDefinitionReader pReader=new XmlBeanDefinitionReader(parentFactory);  
pReader.loadXmlBeanDefinitions("com/nt/cfgs/parent-beans.xml");  
//child Container  
DefaultListableBeanFactory childFactory=new DefaultListableBeanFactory(parentFactory);  
XmlBeanDefinitionReader cReader=new XmlBeanDefinitionReader(childFactory);  
cReader.loadXmlBeanDefinitions("com/nt/cfgs/child-beans.xml");
```

## Mini Project 2 Discussion

- ⊕ If DAO class executes select query and gives multiple Records into ResultSet object then sending that ResultSet object to Service class is bad practice because we should place JDBC code only in DAO class not in other layers. To overcome this problem, copy ResultSet object records to the objects of BO/ Entity class and send them to Service class by keeping them in Array/ Collection (collection is recommended).
  - ⊕ After placing data into any Collection, if we are performing only read operations on that collection then prefer working with non-synchronized collections for performance.  
e.g. ArrayList, HashMap and etc.
  - ⊕ After placing data into any Collection, if we are looking perform both read and write operations simultaneously then prefer using synchronized collections for thread safety.  
e.g. Vector, Hashtable and etc.
  - ⊕ If data is key-value pairs, then go for Map collection.
  - ⊕ If want to preserve the insertion order and wants access elements using indexes then for List collection otherwise go for Set collection.

**Conclusion:** While generating reports by collecting data/ records from DB table we prefer using ArrayList to store those records as the object of BO class in the elements.



**Sample code to copy ResultSet object records to List Collection:**

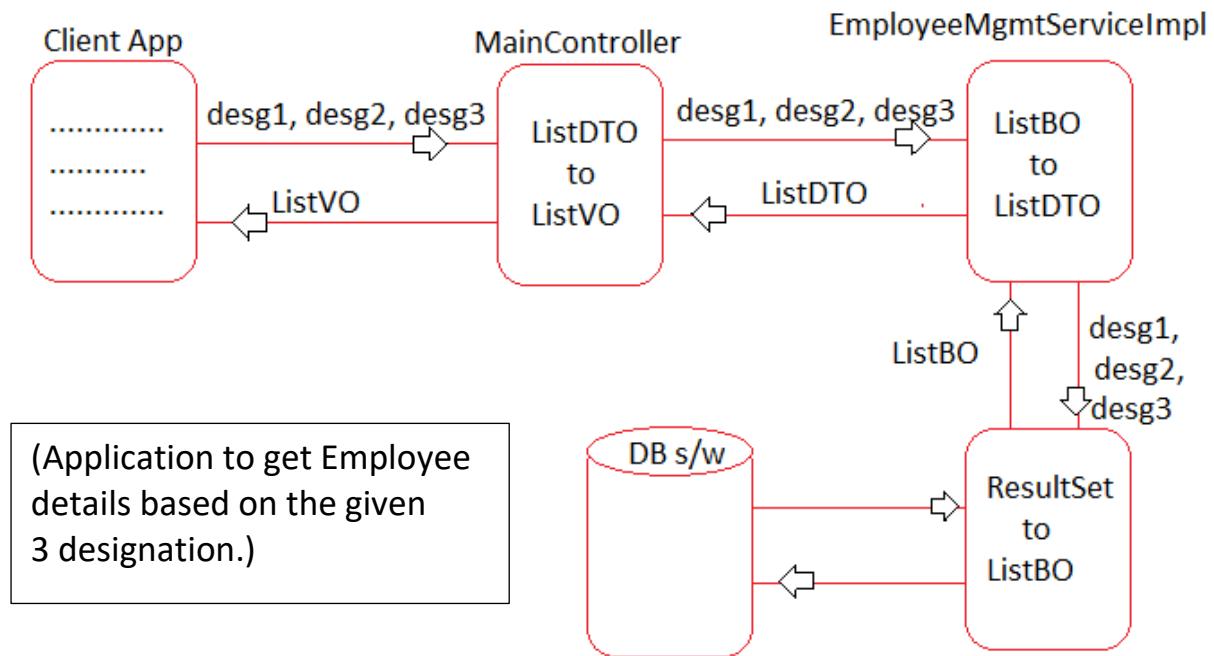
```
//execute query
PreparedStatement ps=con.prepareStatement("select empno,
                                             ename, job, sal, deptno from emp");

ResultSet rs=ps.executeQuery();
List<EmployeeBO> al = new ArrayList();
while(rs.next()){
    //copy each record of rs to each object of BO
    EmployeeBO bo=new EmployeeBO();
    bo.setEmpNo(rs.getInt(1));
    bo.setEname(rs.getString(2));
    bo.setJob(rs.getString(3));
    bo.setSalary(rs.getFloat(4));
    bo.setDeptNo(rs.getInt(5));
    //add each BO obj to List colection
    al.add(bo);
}
```

```
public class EmployeeBO {
    private Integer empNo;
    private String ename;
    private String job;
    private Float sal;
    private Integer deptno;
    //setters && getters
    .....
}
```

- In Java beans like VO, DTO, BO classes we prefer taking wrapper data types for properties, not the primitive data types because primitive variables default values like 0, 0.0 and etc. allocate memory, where wrapper variable default values are null and null does not allocate memory.
- While Persisting BO object data to DB table record the primitive variables default values like 0, 0.0 and etc. insertion will allocate memory in DB table. Where as the Wrapper variables default value null insertion does not allocate memory in DB table.

### Setup of our application:



### Story board of Mini Project 2:

(e) Internal Cache of Parent IoC container

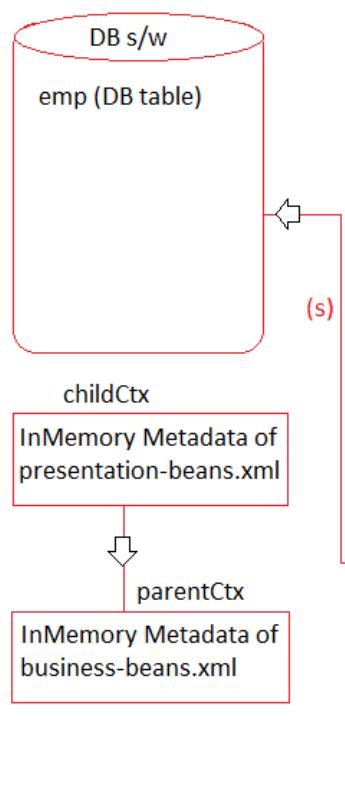
hkDs	HikariDataSource object reference
empDAO	EmployeeDAOImpl class object reference
empService	EmployeeMgmtServiceImpl class object reference

Internal Cache of Child IoC container

controller	MainController object reference
(i)	(k)?

**Note:** Taking only model/BO/Entity class passing it to DAO to service to controller to Client App is bad practice because sometimes we need add or decrease or modify persisted data collected from DB before presenting to the end-user.





```

EmployeeDAO.java
-----
public interface EmployeeDAO {
    public List<EmployeeBO> getEmpsByDesgs(String desg1, String
desg2, String desg3) throws Exception;

EmployeeDAOImpl.java
-----
public class EmployeeDAOImpl implements EmployeeDAO{
    private DataSource ds;
    public EmployeeDAOImpl(DataSource ds){
        this.ds=ds;
    }
    public List<EmployeeBO> getEmpsByDesgs(String desg1, String
desg2, String desg3) throws Exception {
        //jdbc code to ResultSet having emp details
        .....
        //convert ResultSet object records into ListBO
        .....
        return listBO; (t)
    }
}
  
```

### EmployeeMgmtService.java

```

public interface EmployeeMgmtService{
    public List<EmployeeDTO> fetchEmpsByDesgs(String desg1,String
desg2, String desg3) throws Exception;
}
  
```

### EmployeeMgmtServiceImpl.java

```

public class EmployeeMgmtServiceImpl implements EmployeeMgmt
{
    private EmployeeDAO dao;
    public EmployeeMgmtServiceImpl(EmployeeDAO dao){
        this.dao=dao;
    }
    public List<EmployeeDTO> fetchEmpsByDesgs(String desg1,String
desg2, String desg3) throws Exception {
        //convert desgs into upper case (b.logic)
        desg1=desg1.toUpperCase();
        desg2=desg2.toUpperCase();
        desg3=desg3.toUpperCase();
        //use DAO
        .....
        (u) List<EmployeeBO> listBO=dao.getEmpsByDesgs(desg1,desg2,desg3);
        //convert listBO to listDTO
        return listDTO; (v)
    }
}
  
```

### Main Controller

```
public class MainController{
    private EmployeeMgmtService service;
    public MainController(EmployeeMgmtSErvice service){
        this.service=service;
    }
    public List<EmployeeVO> getEmpByDesgs(String desg1, String desg2, String
        desg3) throws Exception{
        //use Service
        (n)
        (o)
        (w) List<EmployeeDTO> listDTO =service.fetchEmpsByDesg
            (desg1,desg2,desg3);
        //convert listDTO to listVO
        return listVO; (x)
    }
}
```

### NestedIoCContainerTest.java

```
public class NestedIoCContainerTest{ (a)
    public static void main(String args[]){
        //create parent IOC container (b) (c) Checking well formed, validness
        //and InMemory Metadata
        ApplicationContext parentCtx=new
            ClasspathXmlapplicationContext("business-beans.xml");
        //create create Child IOC container (g)Checking well formed, validness
        ApplicationContext childCtx= (f) and InMemory Metadata
            new ClasspathXmlapp icationContext(new String[]{"presentation-
                beans.xml"},parentCtx);
        //get Controller class object (j)
        (l) MainController controller = childCtx.getBean
            ("controller", MainControlle r.class);
        try{
            (m)
            (y)List<EmployeeVO> listVO=controller.gatherEmpsByDesgs("CLERK",
                "MANAGER", "SALESMAN");
            sysout(listVO); (z)
        } catch(Exception e){
            e.printStackTrace();
        }
    } //main
}
```

### business-beans.xml

```
<beans ...>          (d) Pre-instantiation of singleton scope beans
    <bean id="hkDs" class="pkg.HikariDataSource">
        .....
        .....
    </bean>
    <bean id="empDAO" class="pkg.EmployeeDAOImpl">
        <constructor-arg ref=" hkDs" />
    </bean>
    <bean id="empService" class="pkg.EmployeeMgmtServiceImpl">
        <constructor-arg ref=" empDAO"/>
    </bean>
</beans>
```

### presentation-beans.xml

```
<beans ....> (h)Pre-instantiation of singleton scope beans
    <bean id="controller" class="pkg.MainController"/>
        <constructor-arg ref=" empService" />
    </bean>
</beans>
```

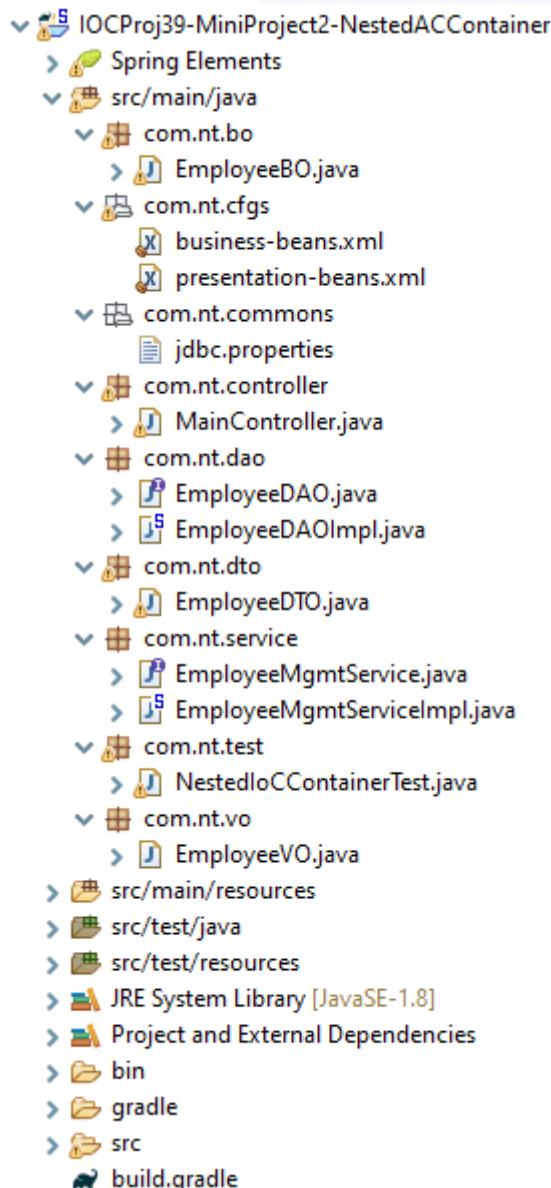
**Notes:** The Process of converting simple data type values to wrapper class objects automatically is called autoboxing and reverse is called auto unboxing (java 5 feature)

**Q. What is exception rethrowing? how does it will be used in Project Development?**

**Ans.** Catching exception raised in try block beginning from catch block and rethrowing either same exception or other exception to caller method is called exception rethrowing. For, this we need to use try/ catch, throws, throw statements together. We generally do this work DAO class methods while writing JDBC code.

**Note:** "throws" declares the exception to be thrown as part method signature (like beware of dog). "throw" actually create and throws new exception or rethrows same or new exception.

## Directory Structure of IOCProj39-MiniProject2-NestedACContainer:



- Develop the above directory structure and package, class, XML, properties file and add the jar dependencies in build.gradle file then use the following code with in their respective file.

### build.gradle

```
plugins {
    // Apply the java-library plugin to add support for Java Library
    id 'application'
}

repositories {
    . jcenter()
}
```

```
dependencies {
    // https://mvnrepository.com/artifact/org.springframework/spring-
    context-support
    implementation group: 'org.springframework', name: 'spring-context-
    support', version: '5.2.8.RELEASE'
    //
    https://mvnrepository.com/artifact/com.oracle.database.jdbc/ojdbc6
    implementation group: 'com.oracle.database.jdbc', name: 'ojdbc6',
    version: '11.2.0.4'
    // https://mvnrepository.com/artifact/org.projectlombok/lombok
    implementation group: 'org.projectlombok', name: 'lombok', version:
    '1.18.12'
    // https://mvnrepository.com/artifact/com.zaxxer/HikariCP
    implementation group: 'com.zaxxer', name: 'HikariCP', version: '3.4.5'
}
```

#### EmployeeVO.java

```
package com.nt.vo;

import lombok.Data;

@Data
public class EmployeeVO {
    private String serialNo;
    private String empno;
    private String ename;
    private String job;
    private String sal;
    private String deptno;
}
```

#### EmployeeDTO.java

```
package com.nt.dto;

import java.io.Serializable;

import lombok.Data;

@Data
```

```
public class EmployeeDTO implements Serializable {  
    private Integer serialNo;  
    private Integer empno;  
    private String ename;  
    private String job;  
    private Float sal;  
    private Integer deptno;  
}
```

### EmployeeBO.java

```
package com.nt.bo;  
  
import java.io.Serializable;  
  
import lombok.Data;  
  
@Data  
public class EmployeeBO implements Serializable {  
    private Integer empno;  
    private String ename;  
    private String job;  
    private Float sal;  
    private Integer deptno;  
}
```

### EmployeeDAO.java

```
package com.nt.dao;  
  
import java.util.List;  
  
import com.nt.bo.EmployeeBO;  
  
public interface EmployeeDAO {  
    public List<EmployeeBO> getEmployeesByDesgs(String desg1, String  
desg2, String desg3) throws Exception;  
}
```

## EmployeeDAOImpl.java

```
package com.nt.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import javax.sql.DataSource;

import com.nt.bo.EmployeeBO;

public class EmployeeDAOImpl implements EmployeeDAO {

    private DataSource ds;
    private static final String GET_EMP_DETAILS_BY_DESG = "SELECT
EMPNO, ENAME, JOB, SAL, DEPTNO FROM EMP WHERE JOB IN (?, ?, ?)
ORDER BY JOB";

    public EmployeeDAOImpl(DataSource ds) {
        this.ds = ds;
    }

    @Override
    public List<EmployeeBO> getEmployeesByDesgs(String desg1, String
desg2, String desg3) throws Exception {
        Connection con = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        List<EmployeeBO> listBO = null;
        try {
            // get Connection
            con = ds.getConnection();
            // create prepared
            ps =
con.prepareStatement(GET_EMP_DETAILS_BY_DESG);
            // set the value to query param
            ps.setString(1, desg1);
```

```

        ps.setString(2, desg2);
        ps.setString(3, desg3);
        // execute the query
        rs = ps.executeQuery();
        //Create BO object
        listBO = new ArrayList<>();
        while (rs.next()) {
            // Create BO object
            EmployeeBO bo = new EmployeeBO();
            bo.setEmpno(rs.getInt(1));
            bo.setEname(rs.getString(2));
            bo.setJob(rs.getString(3));
            bo.setSal(rs.getFloat(4));
            bo.setDeptno(rs.getInt(5));
            //add bo object to listBO
            listBO.add(bo);
        }
    }
    catch (SQLException se) {
        se.printStackTrace();
        throw se;
    } catch (Exception e) {
        e.printStackTrace();
        throw e;
    }
    finally {
        try{
            if (rs!=null)
                rs.close();
        } catch (SQLException se) {
            se.printStackTrace();
            throw se;
        }
        try{
            if (ps!=null)
                ps.close();
        } catch (SQLException se) {
            se.printStackTrace();
            throw se;
        }
    }
}

```

```

        try{
            if (con!=null)
                con.close();
        } catch (SQLException se) {
            se.printStackTrace();
            throw se;
        }
    }
    return listBO;
}

}

```

#### EmployeeMgmtService.java

```

package com.nt.service;

import java.util.List;

import com.nt.dto.EmployeeDTO;

public interface EmployeeMgmtService {
    public List<EmployeeDTO> fetchEmployeesByDesgs(String desg1,
String desg2, String desg3) throws Exception;
}

```

#### EmployeeMgmtServiceImpl.java

```

package com.nt.service;

import java.util.ArrayList;
import java.util.List;

import org.springframework.beans.BeanUtils;

import com.nt.bo.EmployeeBO;
import com.nt.dao.EmployeeDAO;
import com.nt.dto.EmployeeDTO;

public class EmployeeMgmtServiceImpl implements EmployeeMgmtService {

```

```

public EmployeeDAO dao;

public EmployeeMgmtServiceImpl(EmployeeDAO dao) {
    this.dao = dao;
}

@Override
public List<EmployeeDTO> fetchEmployeesByDesgs(String desg1,
String desg2, String desg3) throws Exception {
    List<EmployeeBO> listBO = null;
    List<EmployeeDTO> listDTO = null;
    EmployeeDTO dto = null;
    //Covert desg into upper case
    desg1 = desg1.toUpperCase();
    desg2 = desg2.toUpperCase();
    desg3 = desg3.toUpperCase();
    //use DAO
    listBO = dao.getEmployeesByDesgs(desg1, desg2, desg3);
    //covert listBO to listDTO
    listDTO = new ArrayList<>();
    for (EmployeeBO bo: listBO) {
        dto = new EmployeeDTO();
        BeanUtils.copyProperties(bo, dto);
        dto.setSal((float) Math.round(bo.getSal()));
        dto.setSerialNo(listDTO.size()+1);
        listDTO.add(dto);
    }
    return listDTO;
}
}

```

### jdbc.properties

```

#Driver Details
jdbc.driver = oracle.jdbc.driver.OracleDriver
jdbc.url = jdbc:oracle:thin:@localhost:1521:xe
jdbc.user = scott
jdbc.password = tiger
pool.maximumsize = 100
pool.minimumidle = 40000

```

## MainController.java

```
package com.nt.controller;

import java.util.ArrayList;
import java.util.List;

import org.springframework.beans.BeanUtils;

import com.nt.dto.EmployeeDTO;
import com.nt.service.EmployeeMgmtService;
import com.nt.vo.EmployeeVO;

public class MainController {

    private EmployeeMgmtService service;
    public MainController(EmployeeMgmtService service) {
        this.service = service;
    }
    public List<EmployeeVO> getEmployeeByDesg(String desg1, String
desg2, String desg3) throws Exception {
        List<EmployeeDTO> listDTO = null;
        List<EmployeeVO> listVO = null;
        EmployeeVO vo = null;
        //use service
        listDTO = service.fetchEmployeesByDesgs(desg1, desg2, desg3);
        //covert DTO to VO
        listVO = new ArrayList<>();
        for (EmployeeDTO dto : listDTO) {
            vo = new EmployeeVO();
            BeanUtils.copyProperties(dto, vo);
            vo.setSerialNo(String.valueOf(dto.getSerialNo()));
            vo.setEmpno(String.valueOf(dto.getEmpno()));
            vo.setSal(String.valueOf(dto.getSal()));
            vo.setDeptno(String.valueOf(dto.getDeptno()));
            //add vo to listVO
            listVO.add(vo);
        }
        return listVO;
    }
}
```

## business-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"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <!-- Configure Properties file -->
    <context:property-placeholder
location="com/nt/commons/jdbc.properties"/>

    <!-- HikariCP DataSource -->
    <bean id="hkDs" class="com.zaxxer.hikari.HikariDataSource">
        <property name="driverClassName" value="${jdbc.driver}"/>
        <property name="jdbcUrl" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.user}"/>
        <property name="password" value="${jdbc.password}"/>
        <property name="maximumPoolSize"
value="${pool.maximumsize}"/>
        <property name="minimumIdle"
value="${pool.minimumidle}"/>
    </bean>

    <!-- Configure to DAO -->
    <bean id="empDAO" class="com.nt.dao.EmployeeDAOImpl">
        <constructor-arg ref="hkDs"/>
    </bean>

    <!-- Configure Service -->
    <bean id="empService"
class="com.nt.service.EmployeeMgmtServiceImpl">
        <constructor-arg ref="empDAO"/>
    </bean>

</beans>
```

### [presentation-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.xsd">

    <!-- Configure the controller -->
    <bean id="controller" class="com.nt.controller.MainController">
        <constructor-arg ref="empService"/>
    </bean>

</beans>
```

### [NestedIoCContainerTest.java](#)

```
package com.nt.test;

import java.util.List;
import java.util.Scanner;

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

import com.nt.controller.MainController;
import com.nt.vo.EmployeeVO;

public class NestedIoCContainerTest {

    public static void main(String[] args) {
        ApplicationContext parentCtx = null, childCtx = null;
        MainController controller = null;
        Scanner sc = null;
        String desg1=null, desg2=null, desg3=null;
        List<EmployeeVO> listVO = null;
        //create the parent ioc container
        parentCtx = new
ClassPathXmlApplicationContext("com/nt/cfgs/business-beans.xml");
```

```

//create child IoC container
childCtx = new ClassPathXmlApplicationContext(new String[]
{"com/nt/cfgs/presentation-beans.xml"}, parentCtx);
//get controller object
controller = childCtx.getBean("controller",
MainController.class);
//get system object
sc = new Scanner(System.in);
System.out.println("Enter the Desginations");
System.out.print("Enter the Desgination 1 : ");
desg1 = sc.nextInt();
System.out.print("Enter the Desgination 2 : ");
desg2 = sc.nextInt();
System.out.print("Enter the Desgination 3 : ");
desg3 = sc.nextInt();
//invoke the method
try {
    System.out.println("\nAccording to the give designation
we get the following records\n");
    listVO = controller.getEmployeeByDesg(desg1, desg2,
desg3);
    for (EmployeeVO vo : listVO) {
        System.out.println(vo);
    }
} catch (Exception e) {
    e.printStackTrace();
}

//close containers
((AbstractApplicationContext) parentCtx).close();
((AbstractApplicationContext) childCtx).close();
}
}

```

- Then run the application and give the three designation for execution.
- Now we will have improved the application using java 8 features like
  - Lambda Expression
  - forEach() method
  - Method Reference

### Java 7 and Before:

1. Declaration interface

```
public interface Arithmetic {  
    public int sum (int x, int y);  
}
```
2. Implementation class for Interface

```
public class Calculator implements Arithmetic {  
    public int sum (int x, int y) {  
        return x + y;  
    }  
}
```
3. Creating object for implementation class  
Arithmetic ar =new Calculator ();
4. Invocation of the method  
System.out.println("result is "+ar.sum(10, 20));

### Java 8 onwards:

1. Declaration of interface with method

```
public interface Arithmetic {  
    public int sum (int x, int y);  
}
```
2. Implementation class + object creation: Lambda expression (allows to write code in short form)  
Syntax:  

```
<interface> <ref var name> = (params) -> {}
```

e.g.
  - a. Arithmetic ar = (int x, int y) -> {return x + y;}
  - b. Arithmetic ar = (int x, int y) -> return x + y; // {} is optional for single line body
  - c. Arithmetic ar = (x, y) -> return x + y; // data types are optional for params
  - d. Arithmetic ar = (a, b) -> return a + b; // param names can be changed
  - e. Arithmetic ar = (a, b) -> a + b; // if no {} is taken, return keyword optional

**Note:** () are optional for single param
3. Invocation of the Method  
System.out.println("result is "+ar.sum(10, 20));

**Note:** Lambda Expression can be applied only with Functional interfaces.

### Functional interface:

- The interface that contains single Abstract method (SAM) directly or indirectly is called Functional interface.

```
public interface Test { // valid functional interface
    public void m1 ();
}
```

**Note:** Functional interface can have multiple default methods and static method definitions.

- @FunctionalInterface does not make interface as functional interface. It makes java compiler to check current interface is functional interface or not.

```
@FunctionalInterface
public interface Test {
    public void m1 ();
}

public interface Demo { //invalid functional interface
    public void m1 ();
    public void m2 (int x, int y);
}

public interface Sample extends Test { // valid functional
}                                         interface
public interface Sample1 extends Test { //invalid functional
    public void m2 ();                      interface
}
public interface Sample2 extends Test, Sample { //valid
}                                         functional interface
```

**Note:** Functional interface should have SAM (single abstract method) directly or indirectly even after it is extending from other interfaces.

### 3 for loops in java:

- a. Traditional for loop (from java')
- b. Enhance for loop/ for each loop (from java5)
- c. forEach() method (from java 8) (Best performance)  
cannot be used on arrays, can be used only on collections

## Directory Structure of IOCProj40-MiniProject2-NestedACContainer-Java8:

- ⊕ Copy paste the Mini Project2 and change rootProject.name to IOCProj40-MiniProject2-NestedACContainer-Java8 in settings.gradle file
- ⊕ Need not to add any new file same structure as IOCProj39-MiniProject2-NestedACContainer.
- ⊕ Add the following code in their respective files as part of the replace for the particular section.

### EmployeeMgmtServiceImpl.java

```
@Override
public List<EmployeeDTO> fetchEmployeesByDesgs(String desg1,
String desg2, String desg3) throws Exception {
    List<EmployeeBO> listBO = null;
    List<EmployeeDTO> listDTO = new ArrayList<>();
    //Covert desg into upper case
    desg1 = desg1.toUpperCase();
    desg2 = desg2.toUpperCase();
    desg3 = desg3.toUpperCase();
    //use DAO
    listBO = dao.getEmployeesByDesgs(desg1, desg2, desg3);
    listBO.forEach(bo -> {
        EmployeeDTO dto = new EmployeeDTO();
        BeanUtils.copyProperties(bo, dto);
        dto.setSal((float) Math.round(bo.getSal()));
        dto.setSerialNo(listDTO.size() + 1);
        //add each dto to listDTO
        listDTO.add(dto);
    });
    return listDTO;
}
```

### MainController.java

```
public List<EmployeeVO> getEmployeeByDesg(String desg1, String
desg2, String desg3) throws Exception {
    List<EmployeeDTO> listDTO = null;
    List<EmployeeVO> listVO = new ArrayList<>();
    //use service
    listDTO = service.fetchEmployeesByDesgs(desg1, desg2, desg3);
    listDTO.forEach(dto -> {
```

```

EmployeeVO vo = new EmployeeVO();
BeanUtils.copyProperties(dto, vo);
vo.setSerialNo(String.valueOf(dto.getSerialNo()));
vo.setEmpno(String.valueOf(dto.getEmpno()));
vo.setSal(String.valueOf(dto.getSal()));
vo.setDeptno(String.valueOf(dto.getDeptno()));
//add vo to listVO
listVO.add(vo);
});
return listVO;
}

```

### NestedIoCContainerTest.java

```

//invoke the method
try {
    System.out.println("\nAccording to the give designation
we get the following records\n");
    listVO = controller.getEmployeeByDesg(desg1, desg2,
desg3);
    /*listVO.forEach(vo->{
        System.out.println(vo);
    });
    //listVO.forEach(vo->System.out.println(vo));
    listVO.forEach(System.out::println);
} catch (Exception e) {
    e.printStackTrace();
}
}

```

<ref> tag attributes:

```

<property>
    <ref bean = “....”/>      =      <property ref = “....”/>
</property>

```

```

<constructor-arg>
    <ref bean = “....”/>      = <constructor-arg ref= “....”/>
</constructor-arg>

```

Attributes of <ref> tag are:

- a. bean
- b. parent
- c. local (removed from spring 4.x onwards)

<ref bean="**<dependent bean id>**":

- Searches given bean id-based spring bean class configuration first in the current/ local IoC container spring configuration file, if not available then it searches in parent IoC container spring bean configuration file.

<ref parent="**<depending bean id>**":

- Searches only in the parent IOC container spring bean configuration file from.

<ref local="**<dependent bean id>**"> (removed spring 4.x):

- Searches only in the current/ Local/ child IOC container spring bean configuration file

⊕ Add the following changes in their respective file of the Mini project 2

EmployeeMgmtServiceImpl.java

```
public EmployeeMgmtServiceImpl(EmployeeDAO dao, String type) {  
    System.out.println("Type : "+type);  
    this.dao = dao;  
    this.type = type;  
}  
  
@Override  
public String toString() {  
    return "EmployeeMgmtServiceImpl [type=" + type + "]";  
}
```

MainController.java

```
public MainController(EmployeeMgmtService service) {  
    System.out.println(service);  
    this.service = service;  
}
```

### business-beans.xml

```
<!-- Configure Service -->
<bean id="empService"
class="com.nt.service.EmployeeMgmtServiceImpl">
    <constructor-arg ref="empDAO"/>
    <constructor-arg value="parent"/>
</bean>
```

### presentation-beans.xml

```
<!-- Configure Service -->
<bean id="empService"
class="com.nt.service.EmployeeMgmtServiceImpl">
    <constructor-arg ref="empDAO"/>
    <constructor-arg value="parent"/>
</bean>
```

## Annotation driven Spring Programming in Core module

- Annotations support is introduced to spring from spring 2.0 added more annotations incrementally in later versions.

### List of Annotations related spring core module:

#### Spring 2.0

@Configuration  
@Required  
@Repository  
@Order

#### Stereotype annotations:

@Component  
@Service  
@Controller  
@Repository

#### Spring 3.x:

@Bean  
@DependsOn  
@Lazy  
@Value  
@Import  
@ImportResource  
@ComponentScan  
@PropertySource  
@PropertySources  
@Primary  
and etc.

#### Spring 2.5

@Autowired  
@Qualifier  
@Scope

#### Spring 5.x:

@Nullable  
@NonNull  
@NonNullApi  
@NonNullFieldds  
and etc.

#### Spring 4.x:

@Lookup  
and etc.

### @Required:

- While using any parameterized constructor for constructor injection, we must configure all params of that constructor for injection otherwise exception will come.
- This restriction is not there while working with setter injection. To bring such restriction on our choice bean properties through setter injection we need go for @Required annotation.

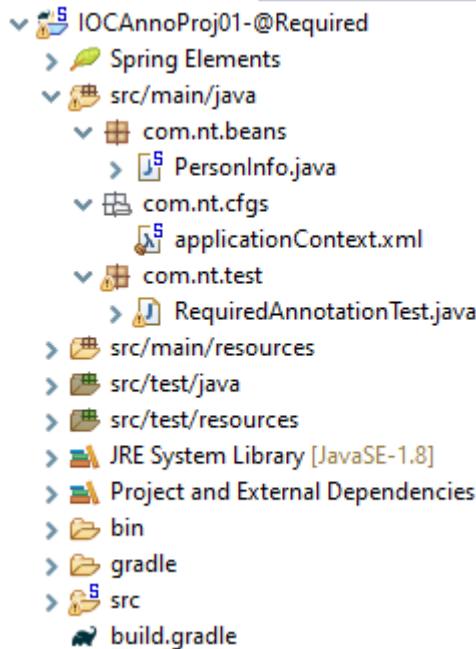
```
public class PersonInfo {  
    private int pid;  
    private String pname;  
    private String paddr;  
  
    @Required  
    public void setPid(int pid){this.pid = pid;}  
    @Required  
    public void setPname(String pname){this.pname = pname;}  
    public void setAddrs(String paddr){this.paddr = paddr;}  
    //toString()  
    .....  
}
```

### Note:

- ✓ @Required is applicable at method level.
- ✓ pid, pname properties must be configured for setter injection otherwise exception will be thrown.
- @Required is deprecated in Spring 5.1 saying go for constructor injection in order to add restriction on injections.
- The entire functionality of @Required annotation is placed in a readymade class called "RequiredAnnotationBeanPostProcessor", So we should configure this class as spring bean.

### Directory Structure of IOCAnnoProj01-@Required:

- Develop the above directory structure and package, class, XML file then use the following code with in their respective file.
- Copy paste build.gradle from any other gradle project because we are using same spring-context-support dependency only (5.0.1).



### PersonInfo.java

```
package com.nt.beans;

import org.springframework.beans.factory.annotation.Required;

public class PersonInfo {

    private int pid;
    private String pname;
    private String paddress;

    @Required
    public void setPid(int pid) {
        this.pid = pid;
    }

    @Required
    public void setPname(String pname) {
        this.pname = pname;
    }

    public void setPaddress(String paddress) {
        this.paddress = paddress;
    }
}
```

```

@Override
public String toString() {
    return "PersonInfo [pid=" + pid + ", pname=" + pname +",
paddress=" + paddress + "]";
}

}

```

### 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"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <!-- Configure the bean -->
    <bean id="personInfo" class="com.nt.beans.PersonInfo">
        <property name="pid" value="1001"/>
        <!-- <property name="pname" value="Ramji"/>
        <property name="paddress" value="ayodha"/>-->
    </bean>

    <!-- <bean
class="org.springframework.beans.factory.annotation.RequiredAnnotationB
eanPostProcessor"/> -->
    <context:annotation-config/>

</beans>

```

### RequiredAnnotationTest.java

```

package com.nt.test;

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

```

```

import com.nt.beans.PersonInfo;

public class RequiredAnnotationTest {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        PersonInfo pinfo = null;
        // Create AC IoC container
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        //get PersonInfo object
        pinfo = ctx.getBean("personInfo", PersonInfo.class);
        System.out.println(pinfo);

        ((AbstractApplicationContext) ctx).close();
    }

}

```

**Note:**

- ✓ Every BeanProcessor will be activated automatically. Once it is configured as spring bean in spring bean configuration file.  
`<bean  
class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"/>`
- ✓ Configure Bean PostProcessor for every annotation separately is complex process. To, overcome that Problem just place <context:annotation-config> in spring bean configuration file

**<context:annotation-config>:**

- Activates various annotations to be detected in bean classes Spring's @Required and @Autowired, as well as JSR 250's @PostConstruct, @PreDestroy and @Resource (if available), JAX-WS's @WebServiceRef (if available), EJB 3's @EJB (if available), and JPA's @PersistenceContext and @PersistenceUnit (if available).
- From spring 5.1, this tag is not working for deprecated annotations like @Required.
- We cannot use @Required along Lombok API generated setter methods.

### @Autowired:

- Performs byType, byName, constructor mode of autowiring (detecting the dependent beans dynamically without using <property>, <constructor-arg> tags).
- Can be applied at field level (instance variables), constructor level, setter method level and arbitrary method level any method.
- Cannot be used to inject values to simple properties can be used to inject values only to Object type/ref type bean properties.

```
public class Flipkart{  
    @Autowired  
    private Courier courier;  
    public String shopping (String [] items, float C] prices) {  
        .....  
        .....  
    }  
}  
public class DTDC implements Courier {  
    .....  
}
```

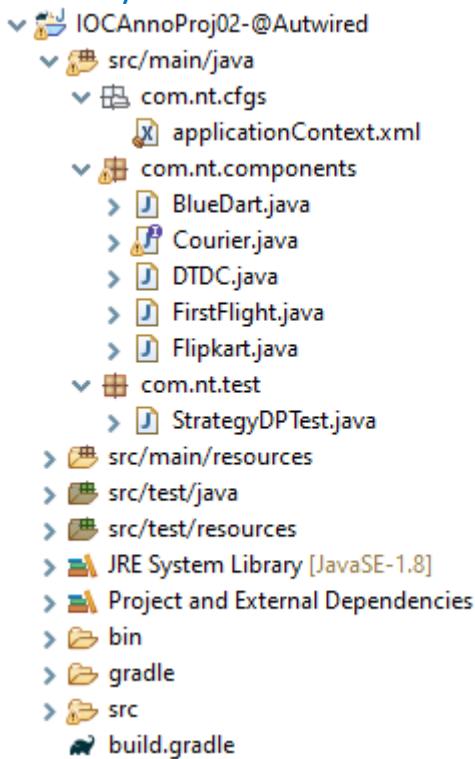
### applicationContext.xml

```
<beans>  
    <bean id="dtdc" class="pkg.DTDC"/>  
    <bean id="fpkt" class="pkg.Flipkart"/>  
    <context: annotation-config/>  
</beans>
```

- ⊕ In xml driven spring programming Dependency injection is possible only through setter methods and constructors whereas @Autowired based injections the dependency injection is possible through directly using fields, setter methods, constructor and arbitrary method.
- ⊕ <context: annotation-config/> Activates certain annotations like @Required, @Autowired and etc. by internally configure their respective BeanPostProcessors as spring beans.

**Note:** Using @Autowired at field level without having setter method, parameterized constructor is possible. In that situation it will access private property through Reflection API and performs injection on that property by seeing @Autowired Annotation.

## Directory Structure of IOCAannoProj02-@Autowired:



- Develop the above directory structure.
- Copy the commonly used packages and class from IOCProj07-StrategyPattern-Spring.
- Copy paste build.gradle from any other gradle project because we are using same spring-context-support dependency only.
- Then change and add the following code in their respective files.

### Flipkart.java

```
@Autowired  
private Courier courier;  
  
public Flipkart() {  
    System.out.println("Flipkart : Flipkart()");  
}
```

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

```

xsi:schemaLocation="http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

<!-- Dependent bean configuration -->
<bean id="dtdc" class="com.nt.components.DTDC" />
<!-- <bean id="fFlight" class="com.nt.components.FirstFlight" />
<bean id="bDart" class="com.nt.components.BlueDart" />-->

<!-- Target bean configuration -->
<bean id="fpkt" class="com.nt.components.Flipkart"/>

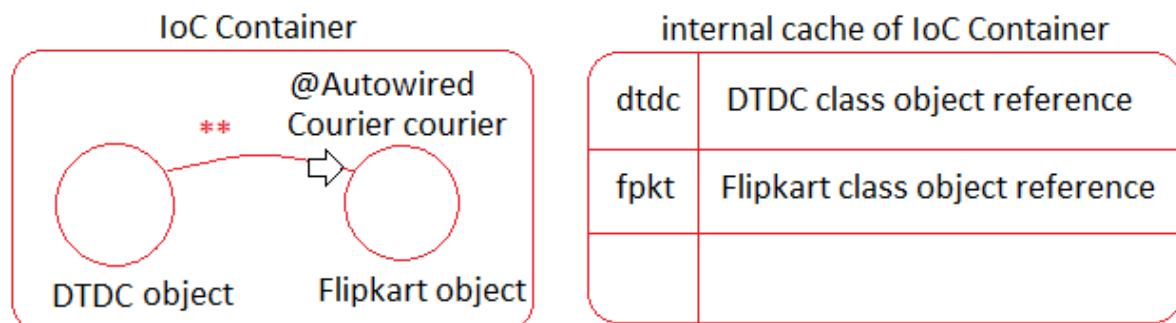
<context:annotation-config/>

</beans>

```

#### Internal workflow:

- IoC Container creation -> Loading of spring bean configuration file -> Checking well formedness, validness -> Creating InMemory Metadata of spring bean configuration file -> Pre-instantiation of all singleton scope beans -> Activation of BeanPostProcessors based <context: annotation-config> tag -> Uses Reflection API and detects in @Autowired on the top of "private Courier courier" field in Flipkart class -> Get access to that property and searches for Courier(l) type spring bean configuration in the In memory metadata of spring bean configuration file -> Finds "DTDC" configuration so takes DTDC class object and assigns/ injects to "courier" property -> Keeps all spring beans in the internal cache of IoC container.



- \*\* this injection takes place because of <context: annotation-config>

supplied/ activated @Autowired related BeanPostProcessors.

- @Autowired performs byType mode of wiring by default.
- If @Autowired does not find its dependent then it throws **Exception in thread "main"**

**org.springframework.beans.factory.UnsatisfiedDependencyException:**  
Error creating bean with name 'fpkt': Unsatisfied dependency expressed through field 'courier'; nested exception is  
**org.springframework.beans.factory.NoSuchBeanDefinitionException:** No qualifying bean of type 'com.nt.components.Courier' available: expected at least 1 bean which qualifies as autowire candidate. Dependency annotations:  
{@org.springframework.beans.factory.annotation.Autowired(required=true)}

#### applicationContext.xml

```
<!-- Dependent bean configuration -->
<bean id="dtdc" class="com.nt.components.DTDC" />

<!-- Target bean configuration -->
<bean id="fpkt" class="com.nt.components.Flipkart"/>

<context:annotation-config/>
```

#### Flipkart.java

```
@Autowired(required = false)
private Courier courier;
```

If "Courier(I)" type dependent bean configuration is not available in spring bean configuration file then "NoSuchBeanDefinitionException" will be raised.

#### Flipkart.java

```
@Autowired(required = false)
private Courier courier;
```

If "Courier(I)" type dependent bean configuration is not available in spring bean configuration file then no Exception will be raised).

- If ambiguity problem rises then it throws [org.springframework.beans.factory.UnsatisfiedDependencyException](#):  
Error creating bean with name 'fpkt': Unsatisfied dependency expressed through field 'courier'; nested exception is  
[org.springframework.beans.factory.NoUniqueBeanDefinitionException](#):  
No qualifying bean of type 'com.nt.components.Courier' available:  
expected single matching bean but found 2: dtdc,fFlight

**Q. How to resolve ambiguity Problem that comes while working @Autowired based Dependency Injection?**

**Ans.**

- Using @Qualifier (-) with dependent bean id or using <qualifier> name of dependent bean.

#### Flipkart.java

```
@Autowired
@Qualifier("bDart")
//@Qualifier("d2")
private Courier courier;
```

#### applicationContext.xml

```
<!-- Dependent bean configuration -->
<bean id="dtdc" class="com.nt.components.DTDC">
    <qualifier value="d1"/>
</bean>
<bean id="fFlight" class="com.nt.components.FirstFlight">
    <qualifier value="d1"/>
</bean>

<!-- Target bean configuration -->
<bean id="fpkt" class="com.nt.components.Flipkart"/>
```

Best Solution, indirectly it is performing byName mode of autowiring.

- By matching Target field/ property name with dependent class bean id.

#### Flipkart.java

```
@Autowired
private Courier courier;
```

### applicationContext.xml

```
<!-- Dependent bean configuration -->
<bean id="courier" class="com.nt.components.DTDC"/>
<bean id="fFlight" class="com.nt.components.FirstFlight"/>

<!-- Target bean configuration -->
<bean id="fpkt" class="com.nt.components.Flipkart"/>

<context:annotation-config/>
```

3. Using primary="true" of (in XML driven spring bean configuration) or @Primary (in annotation driven spring bean configuration) for one dependent bean.

### Flipkart.java

```
@Autowired
private Courier courier;
```

### applicationContext.xml

```
<!-- Dependent bean configuration -->
<bean id="courier" class="com.nt.components.DTDC"/>
<bean id="fFlight" class="com.nt.components.FirstFlight"
primary="true" />

<!-- Target bean configuration -->
<bean id="fpkt" class="com.nt.components.Flipkart"/>

<context:annotation-config/>
```

4. By making only one dependent bean as "autowire candidate" in spring bean configuration file.

### Flipkart.java

```
@Autowired
private Courier courier;
```

## applicationContext.xml

```
<!-- Dependent bean configuration -->
<bean id="courier" class="com.nt.components.DTDC"/>
<bean id="fFlight" class="com.nt.components.FirstFlight"
autowire-candidate="false" />

<!-- Target bean configuration -->
<bean id="fpkt" class="com.nt.components.Flipkart"/>

<context:annotation-config/>
```

- + @Autowired can be applied at field level, constructor level (parameterized), setter method level and arbitrary method level.

### At Setter Method level

```
@Autowired
@Qualifier("bDart")
public void setCourier(Courier courier) {
    System.out.println("Flipkart : setCourier(-)");
    this.courier = courier;
}
```

### At Parameterized constructor level

```
@Autowired
public Flipkart(@Qualifier("bDart") Courier courier) {
    System.out.println("Flipkart : Flipkart(-)");
    this.courier = courier;
}
```

### At Arbitrary Method level

```
@Autowired
@Qualifier("bDart")
public void assign(Courier courier) {
    System.out.println("Flipkart : assign(-)");
    this.courier = courier;
}
```

### At Field Method level

```
@Autowired  
@Qualifier("bDart")  
private Courier courier;
```

**Note:** It is industry standard because we can work with Lombok API and there is no need of any setter methods, parameterized constructors, arbitrary methods supporting injection.

**Q. If we enable @Autowire on same property at multiple levels in which order the autowiring takes place?**

**Ans.** Field level -> parameter constructor level -> setter/ arbitrary method level (both will execute order cannot be decided).

**Note:** Setter method/ Arbitrary method level will come as final value.

**Important observations on @Autowired applied on the constructors:**

- a. Applying @Autowired on 0-param constructor is meaningless.
- b. If multiple overloaded constructors having @Autowired(required=true) then exception will be raised.
- c. If multiple overloaded constructors having @Autowired(required=false) then the more param constructor will execute.
- d. If multiple overloaded constructors having @Autowired(required=false) on few constructors and @Autowired(required=true) on few other constructors then also exception will be raised.
- e. If multiple overloaded constructors having @Autowired(required=false) with same number of params then IOC container picks up the Constructor randomly.

**Note:** we can apply @Autowired on multiple fields, setter methods, arbitrary methods either having required-true/ required=false, but only constructor can have @Autowired with required-true (refer the above points).

### Stereo type annotations

- ⊕ We have multiple with similar behaviour having minor differences. So, they are called Stereo type annotations.
- ⊕ @Component -> To configure java class as spring bean (with no specialties) (To instantiate java class obj and to make it as spring bean by IOC container).

- + @Service -> @Component + also make as service class by giving Tx management support.
- + @Repository -> @Component + also make as DAO class by Exception translation ability (SQLException to Spring specific exceptions).
- + @Controller -> @Component + also makes web controller class by having ability to take/process http requests. and etc.
- + To make IOC container going to different specified packages and their sub packages to search and recognize Stereo type annotation classes as spring beans (by loading them and also by instantiating them) we need to place <context: component-scan bas-package=" ..... "/> tag in spring bean configuration file (multiple packages can be specified as comma separated values).
- + All Stereo type Annotations must be applied at class level (Type level)

```
package com.nt.beans;
@Component("dtdc") //or
@Component(value = "dtdc") (d)
public class DTDC implements Courier{
.....
.....
}
```

```
package com.nt.beans;
@Primary
@Component("bDart") (d)
public class BlueDart implements Courier{
.....
```

```
package com.nt.beans1;
@Component("fpkt")
public class Flipkart { (d) Preinstantiation
    @Autowired
    private Courier courier; (e)
    public String shopping(String items[], float prices[]){
        .... (k)
        ....
        return ...; (l)
    }
}
```

#### Internal cache of IoC container

dtdt	DTDC object reference	(f)
bDart	BlueDart object reference	(h)?
fpkt	Flipkart object reference	

applicationcontext.xml

```
<beans ...>
Import <beans> ,<context> namespace
    <context:component-scan base-Package="com.nt.beans,
                                com.nt.beans1"/>
    (or)                               (c)
        <context:component-scan base-Package="com.nt.beans"/>
</beans>
```

ClientApp

```
//create IOC container (a) (b)
ApplicationContext ctx=new CPXAC("com/nt/cfgs/applicationContext.xml");
//get Bean (i)
Flipkart fpkt = ctx.getBean("fpkt", Flipkart.class); (g) //SOP (m)
//Invoke methods
S.o.p(fpkt.shopping(new String[]{".....", "...."}, new Float[]{....., .....}); (j)
```

**Note:** <context: component-scan/> tag internally maintain behaviour of <context: annotation-config/> so no need of placing explicitly.

#### Directory Structure of IOCProjAnno03-StereoTypeAnnotation:

- + Copy paste the IOCProjAnno02-@Autowired application and change rootProject.name to IOCProjAnno03-StereoTypeAnnotation in settings.gradle file.
- + Need not to add any new file same structure IOCProjAnno02- @Autowired
- + Add the following code in their respective file as part of the replace for the particular section.

BlueDart.java

```
package com.nt.beans;

import org.springframework.stereotype.Component;

@Component("bDart")
public final class BlueDart implements Courier {

    public BlueDart() {
```

```

        System.out.println("BlueDart : BlueDart()");
    }

    @Override
    public String deliver(int orderId) {
        return "BlueDart courier will deliver Order Id :" +orderId+
order products";
    }

}

```

### DTDC.java

```

package com.nt.beans;

import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Component;

@Component("dtdc")
//@Primary
public final class DTDC implements Courier {

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

    @Override
    public String deliver(int orderId) {
        return "DTDC courier will deliver Order Id :" +orderId+ " order
products";
    }

}

```

### FirstFlight.java

```

package com.nt.beans;

import org.springframework.stereotype.Component;

@Component("fFlight")

```

```

public final class FirstFlight implements Courier {

    public FirstFlight() {
        System.out.println("FirstFlight : BlueDart()");
    }

    @Override
    public String deliver(int orderId) {
        return "FirstFlight courier will deliver Order Id : "+orderId+
order products";
    }

}

```

### Flipkart.java

```

package com.nt.beans;

import java.util.Arrays;
import java.util.Random;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component("fpkt")
//@Component
public final class Flipkart {

    @Autowired
    @Qualifier("bDart")
    private Courier courier;

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

    public String shopping(String[] items, float[] prices) {
        System.out.println("Flipkart : shopping()");
        float billAmount = 0.0f;
    }
}

```

```

int orderId = 0;
String msg = null;
//calculate bill amount
for (float p : prices)
    //billAmount = billAmount+p;
    billAmount+=p;

//Generate order id dynamically as random number
orderId = new Random().nextInt(10000);
//use courier service for delivering the products
msg = courier.deliver(orderId);

return Arrays.toString(items)+" are purchased having prices
"+Arrays.toString(prices)+"\nwill bill amount "+billAmount+" .... \n"+msg;
}

}

```

### 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"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

<context:component-scan base-package="com.nt.beans"/>

</beans>

```

#### <context: component-scan>:

Scans the CLASSPATH for annotated components that will be auto-registered as Spring beans. By default, the Spring-provided @Component, @Repository, @Service, @Controller, @RestController, @ControllerAdvice, and @Configuration stereotypes will be detected.

**Note:** This tag implies the effects of the 'annotation-config' tag, activating

@Required, @Autowired, @PostConstruct, @PreDestroy, @Resource, @PersistenceContext and @PersistenceUnit annotations in the component classes.

#### List of all stereo type annotations:

@Component, @Service, @Controller, @RestController, @ControllerAdvice, @Configuration, @Repository

**Note:** In XML configuration driven spring programming the pre-instantiation of singleton scope beans take place in the order of their configuration in spring bean configuration file where as in Annotation driven configurations the pre-instantiation takes Alphabetic order of Spring bean class names (With respect to each package).

- ⊕ In Annotation driven environment the default bean id (Object name) for spring bean is "<bean class name having first letter in lower case>".  
e.g. if class name is "Flipkart" default bean id "flipkart"  
if class name is "EmployeeDAOImpl" default bean id  
"employeeDAOImpl"

```
@Component  
public final class Flipkart {  
    .....  
    .....  
}
```

#### In Client APP

```
Flipkart fpkt = ctx.getBean(Flipkart.class)
```

(or)

```
Flipkart fpkt = ctx.getBean(Flipkart.class);
```

**Note:** If class is pre-defined class, we cannot configure it as spring bean using stereo type annotations because we cannot edit source code any pre-defined class to add stereo type annotations. So, we need to configure that class as spring bean using <bean> tag in spring bean configuration file (xml)

```
@Component  
public final class Flipkart {  
    @Autowired  
    private Date date;  
}
```

#### ApplicationContext.xml

```
<beans .....>  
<context:component-scan base-  
package="com.nt.beans"/>  
    <bean id="dt" class="java.util.Date"/>  
</beans>
```

**Q. Can we do injection on static bean properties/ fields/ attributes/ member variables using any mode of injection including @Autowired?**

**Ans.** No, we cannot.

```
@Autowired  
private static Date date;
```

**INFO:** Autowired annotation is not supported on static fields: private static java.util.Date com.nt.beans.Flipkart.date

**@Lazy:**

- **@Lazy(true):** Disables pre-instantiation on singleton scope bean and enables lazy instantiation.
- **@Lazy(false):** Continues Pre-instantiation on singleton scope bean default value is "true".
- **@Lazy(true)** is equal to lazy-init="true" attribute of <bean> tag.

```
@Component("bDart")  
@Lazy(true)  
public final class BlueDart implements Courier {  
    .....  
}
```

**@Scope:**

- It gives to specify the scope of Spring bean class object the default scope is singleton scope. In standalone environment we can work with only two scopes (singleton, prototype).
- In web environment we can use singleton, prototype, session, application, request, websocket scopes.
- **@Scope** is equal to "scope" attribute of <bean> tag.

```
@Component("fkpt")  
@Scope("prototype")  
public final class Flipkart {  
    .....  
    .....  
}
```

**While developing Layered Apps/ Projects using Annotation driven configuration we need to use the following Thumb rules:**

- a. Configure pre-defined/ third party supplied classes as spring beans using <bean> of spring bean configuration file.
- b. Configure user-defined java classes as spring beans using the support of stereo type annotations or java config annotations and link them with spring bean configuration file using <context: component-scan>

**Note:**

- ✓ While working certain special features for which annotations are not available, we still need to go for XML driven configurations even through spring bean classes user-defined classes.
- ✓ The special feature are inner beans, collection injection, collection merging, bean inheritance, factory method bean instantiation, method replacer and etc.

**Q. How can you say XML driven configurations are not outdated, they are just limited in the utilization?**

**Ans.**

- a. Still we need web.xml for certain web application configurations like welcome file, context params, security configurations and etc.
- b. Still we need hibernate configuration file as xml in hibernate programming (no alternate with annotations).
- c. In Spring, for the following features we need to use XML driven configurations inner beans, collection injection, collection merging, bean inheritance, factory method bean instantiation, method replacer and etc.
- d. In order to override configurations done through annotations without touching the source code, we need to XML files-based configurations.

**Directory Structure of IOCAnnoProj04-RealTimeDI-RealTimeStrategyDP-LayeredApp:**

- ⊕ Copy paste the Layered application and change rootProject.name to IOCAnnoProj04-RealTimeDI-RealTimeStrategyDP-LayeredApp on in settings.gradle file.
- ⊕ Need not to add any new file same structure as IOCPro13-RealTimeDI-RealTimeStrategyDP-LayeredApp.
- ⊕ Add the following code in their respective files.
- ⊕ Add also HikariCP jar in build.gradle because we will used here.

## 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"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans-4.3.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <!-- Configure Data source -->
    <bean id="oracleHKDS" class="com.zaxxer.hikari.HikariDataSource">
        <property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver"/>
        <property name="jdbcUrl"
value="jdbc:oracle:thin:@localhost:1521:xe"/>
        <property name="username" value="system"/>
        <property name="password" value="manager"/>
        <property name="minimumIdle" value="400000"/>
        <property name="maximumPoolSize" value="100"/>
    </bean>

    <bean id="mysqlHKDS" class="com.zaxxer.hikari.HikariDataSource">
        <property name="driverClassName"
value="com.mysql.cj.jdbc.Driver"/>
        <property name="jdbcUrl" value="jdbc:mysql:///nssp713db"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
        <property name="minimumIdle" value="400000"/>
        <property name="maximumPoolSize" value="100"/>
    </bean>

    <context:component-scan base-package="com.nt"/>

</beans>
```

## MySQLCustomerDAOImpl.java

```
package com.nt.dao;
```

```

import java.sql.Connection;
import java.sql.PreparedStatement;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Repository;

import com.nt.bo.CustomerBO;

@Repository("mysqlCustDAO")
public final class MySQLCustomerDAOImpl implements CustomerDAO {

    private static final String CUSTOMER_INSERT_QUERY = "INSERT INTO
    CUSTOMER(CNAME, CADD, PAMT, INTERAMT) VALUES(?, ?, ?, ?)";

    @Autowired
    @Qualifier("mysqlHKDS")
    private DataSource ds;

    @Override
    public int insert(CustomerBO bo) throws Exception {
        Connection con = null;
        PreparedStatement ps = null;
        int count = 0;
        //get JDBC connection pool object
        con = ds.getConnection();
        //Create prepareStatement object
        ps = con.prepareStatement(CUSTOMER_INSERT_QUERY);
        //set the value to query param
        ps.setString(1, bo.getCname());
        ps.setString(2, bo.getcAdd());
        ps.setFloat(3, bo.getpAmnt());
        ps.setFloat(4, bo.getInterAmt());
        //execute the query
        count = ps.executeUpdate();
        //close JDBC objects
        ps.close();
        con.close();
    }
}

```

```
        return count;
    }

}
```

### OracleCustomerDAOImpl.java

```
package com.nt.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Repository;

import com.nt.bo.CustomerBO;

@Repository("oracleCustDAO")
public final class OracleCustomerDAOImpl implements CustomerDAO {

    private static final String CUSTOMER_INSERT_QUERY = "INSERT INTO
    CUSTOMER_DB VALUES(CNO_SEQ.NEXTVAL,?, ?, ?, ?, ?)";

    @Autowired
    @Qualifier("oracleHKDS")
    private DataSource ds;

    @Override
    public int insert(CustomerBO bo) throws Exception {
        Connection con = null;
        PreparedStatement ps = null;
        int count = 0;
        //get JDBC connection pool object
        con = ds.getConnection();
        //Create prepareStatement object
        ps = con.prepareStatement(CUSTOMER_INSERT_QUERY);
        //set the value to query param
```

```

        ps.setString(1, bo.getCname());
        ps.setString(2, bo.getcAdd());
        ps.setFloat(3, bo.getpAmnt());
        ps.setFloat(4, bo.getInterAmt());
        //execute the query
        count = ps.executeUpdate();
        //close JDBC objects
        ps.close();
        con.close();
        return count;
    }

}

```

#### CustomerMgmtServiceImpl.java

```

package com.nt.dao;

import java.sql.Connection;
package com.nt.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;

import com.nt.bo.CustomerBO;
import com.nt.dao.CustomerDAO;
import com.nt.dto.CustomerDTO;

@Service("custService")
public final class CustomerMgmtServiceImpl implements
CustomerMgmtService {

    @Autowired
    //@Qualifier("oracleCustDAO")
    @Qualifier("mysqlCustDAO")
    private CustomerDAO dao;

    @Override
    public String calculateSimpleInterestAmount(CustomerDTO dto)
throws Exception {

```

```

float interAmt = 0.f;
CustomerBO bo = null;
int count = 0;
//Calculate the simple interest amount from DTO
interAmt = (dto.getpAmt()*dto.getTime()*dto.getRate())/100;
//Prepare CustomerBO having persist able Data
bo = new CustomerBO();
bo.setCname(dto.getCname());
bo.setCadd(dto.getcAdd());
bo.setpAmnt(dto.getpAmt());
bo.setInterAmt(interAmt);
//use the dao
count = dao.insert(bo);
if (count==0)
    return "Customer registration failed - Insert amount is :
"+interAmt;
else
    return "Customer registration succeeded - Insert amount
is : "+interAmt;
}
}

```

### MainController.java

```

package com.nt.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;

import com.nt.dto.CustomerDTO;
import com.nt.service.CustomerMgmtService;
import com.nt.vo.CustomerVO;

@Controller("controller")
public final class MainController {

    @Autowired
    private CustomerMgmtService service;

    public String processCustomer(CustomerVO vo) throws Exception {
}

```

```

CustomerDTO dto = null;
String result = null;
//covert Customer VO to customer DTO
dto = new CustomerDTO();
dto.setCname(vo.getCname());
dto.setCadd(vo.getcAdd());
dto.setpAmt(Float.parseFloat(vo.getpAmt()));
dto.setTime(Float.parseFloat(vo.getTime()));
dto.setRate(Float.parseFloat(vo.getRate()));
//use Service
result = service.calculateSimpleInterestAmount(dto);
return result;
}

}

```

### RealTimeDITest.java

```

package com.nt.test;

import java.util.Scanner;

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

import com.nt.controller.MainController;
import com.nt.vo.CustomerVO;

public class RealTimeDITest {

    public static void main(String[] args) {
        Scanner sc = null;
        String name=null, address=null, Amount=null, time=null,
rate=null;
        CustomerVO vo = null;
        ApplicationContext ctx = null;
        MainController controller = null;
        String result = null;
        //Read inputs from end-user using scanner
        sc = new Scanner(System.in);
    }
}

```

```

        System.out.println("Enter the following Details for registration :");
    );
    System.out.print("Enter Customer Name : ");
    name = sc.next();
    System.out.print("Enter Customer Address : ");
    address = sc.next();
    System.out.print("Enter Customer Principle Amount : ");
    Amount = sc.next();
    System.out.print("Enter Customer Time : ");
    time = sc.next();
    System.out.print("Enter Customer Rate of Interest: ");
    rate = sc.next();
    //Store into VO class object
    vo = new CustomerVO();
    vo.setCname(name);
    vo.setCadd(address);
    vo.setpAmt(Amount);
    vo.setTime(time);
    vo.setRate(rate);
    //Create BeanFactory [IoC] container
    ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
    //get controller class object
    controller = ctx.getBean("controller", MainController.class);
    //invoke mmethods
    try {
        result = controller.processCustomer(vo);
        System.out.println(result);
    } catch (Exception e) {
        System.out.println("Internal probelm :
"+e.getMessage());
        e.printStackTrace();
    }
} //main

} //class

```

Directory Structure of IOCAnnoProj05-RealTimeDI-RealTimeStrategyDP-PropertiesFile:

- + Copy paste the IOCAnnoProj04-RealTimeDI-RealTimeStrategyDP-LayeredApp application and change rootProject.name to IOCAnnoProj05-RealTimeDI-RealTimeStrategyDP-PropertiesFile on in settings.gradle file.
- + Create a package com.nt.commons with a properties file jdbc.properties.
- + Add the following code in their respective files.

### jdbc.properties

```
#Database details (Oracle)
#jdbc.driver = oracle.jdbc.driver.OracleDriver
#jdbc.url = jdbc:oracle:thin:@localhost:1521:xe
#jdbc.user = system
#jdbc.password = manager
#jdbc.minIdle = 40000
#jdbc.maxPool = 100

#Database details (MySQL)
jdbc.driver = com.mysql.cj.jdbc.Driver
jdbc.url = jdbc:mysql://nssp713db
jdbc.user = root
jdbc.password = root
jdbc.minIdle = 40000
jdbc.maxPool = 100 #Database details (Oracle)
```

### 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"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans-4.3.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <!-- Configure Data source -->
    <bean id="hKDS" class="com.zaxxer.hikari.HikariDataSource">
        <property name="driverClassName" value="${jdbc.driver}"/>
        <property name="jdbcUrl" value="${jdbc.url}"/>
```

```

<property name="username" value="${jdbc.user}"/>
<property name="password" value="${jdbc.password}"/>
<property name="minimumIdle" value="${jdbc.minIdle}"/>
<property name="maximumPoolSize" value="${jdbc.maxPool}"/>
</bean>

<context:property-placeholder
location="com/nt/commons/jdbc.properties"/>

<context:component-scan base-package="com.nt"/>

</beans>

```

- ⊕ If inject values to simple properties (primitive data type, String type, wrapper datatype) in annotation driven environment use @Value annotation.

```

@Value("30")
private int age;

```

**Note:** We can get simple properties values from properties file by @Value as shown below.

```

@Value("${cust.age}")
private int age;

```

```

jdbc.properties
cust.age = 30

```

#### Note:

- ✓ Using @Value annotation we cannot supply input values to other annotation like @Qualifier, we can just inject values to simple properties either by hardcoding or by getting them properties file.
- ✓ Through xml file or properties file the String/text values injected to the simple properties will be converted to appropriate type internally by IoC container by taking the support "Property Editors"

## Java Config Annotations in Spring environment

- Working with spring supplied annotations makes our spring App code/classes as invasive code/ classes (Tight coupling with Spring API). To overcome this problem, we have java config annotations given by JSE,

JEE modules.

- The Java config Annotations will get specific behaviour based context/environment where they are being used if there are used in spring environment, they give spring specific behaviour. Similarly, in hibernate they hibernate specific behaviour and etc. This helps to make our code non-invasive code (Loosely coupled code with Spring API).
- e.g. @Named, @Inject, @Resource, @PreDestroy, @PostConstruct and etc.

**While developing annotation driven Spring Apps, we should use different annotations having the following priority:**

- a. Java config annotations (Very less)
- b. Spring supplied Annotations
- c. Third party supplied Annotations
- d. Custom Annotations

**Note:** Industry does not care this recommended because less java config annotations are available and as now there is not alternate framework for to move spring code to that framework by having non-invasive behaviour.

#### **@Inject:**

- Alternate to @Autowired having minor differences like no "required" attribute i.e. depend bean configuration is always mandatory.
- To work this annotation we need to add javax.inject-<ver>.jar file (JEE supplied annotation) and we get that from mvnrepository.com.

```
// https://mvnrepository.com/artifact/javax.inject/javax.inject  
implementation group: 'javax.inject', name: 'javax.inject', version: '1'
```

#### **@Named:**

- It is having two angels of utilization.
  - To configure java class as spring bean (As alternate to @Component).
  - To resolve ambiguity problem (as alternate to @Qualifier).
- To work this annotation we need to add javax.inject-<ver>.jar file (JEE supplied annotation) and we get that from mvnrepository.com
- It JEE supplied java config annotation.

#### **Note:**

- ✓ @Inject is applicable at Filed level, constructor level, setter method level

and arbitrary method level.

- ✓ If @Named is applied at class level then it is for configuration java class as spring bean, if it is applied at field/ constructor/ method level to resolve ambiguity problem.

#### Directory Structure of IOCProjAnno06-JavaConfigAnnotations:

- + Copy paste the IOCProjAnno03-StereoTypeAnnotation application and change rootProject.name to IOCProjAnno06-JavaConfigAnnotation in settings.gradle file.
- + Need not to add any new file same structure IOCProjAnno03-StereoTypeAnnotation.
- + Add the following code in their respective files. And their particular position changes.

#### build.gradle

```
dependencies {  
    // https://mvnrepository.com/artifact/org.springframework/spring-  
    context-support  
    implementation group: 'org.springframework', name: 'spring-  
    context-support', version: '5.2.8.RELEASE'  
    // https://mvnrepository.com/artifact/javax.inject/javax.inject  
    implementation group: 'javax.inject', name: 'javax.inject', version:  
    '1'  
}
```

#### BlueDart.java

```
@Named("bDart")  
public final class BlueDart implements Courier {
```

#### DTDC.java

```
@Named("dtdc")  
public final class DTDC implements Courier {
```

#### FirstFlight.java

```
@Named("fFlight")  
public final class FirstFlight implements Courier {
```

## Flipkart.java

```
@Named("fpkt")
public final class Flipkart {

    @Inject
    @Named("bDart")
    private Courier courier;
```

### Description:

```
@Named("fpkt") // To configure java class as spring bean
public final class Flipkart {
    @Inject // For autowiring/ implicit injection
    @Named("bDart") // To Resolve ambiguity problem
    private Courier courier;
```

### @Resource

- It is JSE/ JDK supplied java config annotation to perform autowiring/ implicit dependency injection.
- It is same as `@Autowired` but cannot be applied at Constructor level i.e. cannot perform constructor mode autowiring.
- Since `javax.annotation` package is removed from jdk9.. we should change the setup to java8 in order to work this annotation.
- Here we can resolve ambiguity problem by using "name" attribute of `@Resource` tag itself, no need of using separate `@Qualifier` or `@Named`.
- `@Inject` is JEE supplied Java config Annotation which should be used by adding `javax.inject-<ver>.jar` file but `@Resource` can be used directly without adding any jar file (but upto java8).
- `@Resource` is applicable at field level, setter method level and arbitrary method level form autowiring but not at constructor level.

**Note:** If want to use `javax.annotation` pkg from JDK 9 onwards we need to add the following jar file `javax.annotation-api-1.3.2.jar` file.

## build.gradle

```
// https://mvnrepository.com/artifact/javax.annotation/javax.annotation-
api
implementation group: 'javax.annotation', name: 'javax.annotation-api',
version: '1.3.2'
```

## Flipkart.java

```
@Named("fpkt")
public final class Flipkart {

    /*@Inject
    @Named("bDart")*/
    @Resource(name = "bDart")
    private Courier courier;
```

**Q. What are the differences among @Autowired, @Resource, @Inject?**

Ans.

@Autowired	@Inject	@Resource
a) It is spring supplied annotation.	a) It Is JEE supplied java config annotation. -	a) It is JDK supplied java config annotation (but removed from java9 onwards we should add javax.annotation.api 1.3.2.jar to the build path in order use from java9).
b) We need to add spring jars.	b) We need to add javax.inject-<ver>.jar.	b) No jars are required up to java8, but javax.annotation.api-<ver>.jar is required from java9.
c) Can perform byType, byName, constructor mode autowiring.	c) Same.	c) Can perform only byType and byName autowiring.
d) Applicable at field, method, constructor level.	d) Same.	d) Applicable at field, method for injections.
e) To solve ambiguity problem by matching name/bean id we use @Qualifier.	e) To solve ambiguity problem by matching name/bean id we use @Named.	e) To solve ambiguity problem by matching name/bean id we use "name" attribute of @Resource tag.

**Note:** Since we cannot develop entire Spring App using java config annotations because they are in limited numbers, it is recommended to use @Autowired for autowiring operations.

## Spring Bean Life Cycle

- IOC container manages spring bean life cycle from birth to death (object creation to object destruction).
- Servlet container manages Servlet comp life cycle from birth to death (object creation to object destruction).
- Container raises life cycle events while managing the life cycle.
- Servlet comp life cycle events raised by Servlet container are
  - a. Instantiation event (raises when Servlet container creates our servlet class object).
  - b. Request processing event (raises when Servlet container keeps our servlet comp ready to process the request).
  - c. Destruction event (raises when Servlet container about to destroy our servlet comp class object).
- ServletContainer calls 3 life cycle methods for 3 life cycle events, by overriding these life cycle methods in our servlet comp we can make Servlet container executing our logics as part of life cycle management.
  - a. Instantiation event -> public void init (ServletConfig cg)
  - b. Request processing event -> public void service (Servlet Request req, Servlet Response res) throws SE, IOE
  - c. Destruction event -> public void destroy () method.
- IoC Container raises two life cycle events while managing spring bean life cycle and calls two life cycle methods to handle those events.
  - a. Instantiation event (raises when IoC container creates our spring bean class object and injections are done).
  - b. Destruction event (raises when IoC container is about to destroy our spring bean class object).

**Note:** Servlet life cycle method name are fixed because every Servlet comp is tightly bound with Servlet API (implementation javax.servlet.Servlet(I) is mandatory) whereas Spring bean life cycle methods no fixed in default setup because spring beans non-invasive (not tightly coupled with Spring API).

We can configure spring bean life cycle in 3 approaches:

- a. Using Declarative Approach (XML Configurations)

- b. Using Programmatic Approach (java code - by implementing spring API interfaces here, life cycle method names fixed)
- c. Using Annotation Approach (@PostConstruct, @PreDestroy)

 In (a), (c) approaches life cycle method names are user-defined they must be configured explicitly otherwise they will not be recognized as life cycle methods.

#### Note:

- ✓ In the Init life cycle method (Instantiation event life cycle method) of spring bean we place the following 3 logics
  - a. Initializing left over properties/fields which are not participating in dependency injection.
  - b. Verifying whether important properties are injected with correct, values or not?
  - c. Bean Posting Processing (correcting injected values).
- ✓ In the destroy life cycle method (destruction event life cycle method) of spring bean we place clean up logics like nullifying properties and releasing non-java resources.

## Using Declarative Approach

- Allows to develop Spring bean class as non-invasive class.
- But we need to configure life cycle methods explicitly in Spring bean configuration by using "init-method", "destroy-method" attributes of <bean> tag.
- If the spring bean class is pre-defined class, then we need search for life cycle methods to configure them explicitly using the above said attributes.
- The init life cycle method and destroy life cycle method creation should follow rules
  - a. Must be public
  - b. Should have "void" as the return type
  - c. Should not have params

```
public void <method-name> () {
    .....
    .....//logic
    .....
}
```

## Directory Structure of IOCProj41-BeanLifeCycle-Declarative-XML:

```
IOCProj41-BeanLifeCycle-Declarative-XML
  Spring Elements
  src/main/java
    com.nt.beans
      Voter.java
    com.nt.cfgs
      applicationContext.xml
    com.nt.test
      BeanLifeTest.java
  src/main/resources
  src/test/java
  src/test/resources
  JRE System Library [JavaSE-1.8]
  Project and External Dependencies
  bin
  gradle
  src
  build.gradle
```

- Develop the above directory structure and package, class, XML file then use the following code with in their respective file.
- Copy paste build.gradle from any other gradle project because we are using same spring-context-support, and Lombok dependency only.

### 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-4.3.xsd">

  <bean id="voter" class="com.nt.beans.Voter" init-method="myInit"
        destroy-method="myDestroy">
    <property name="name" value="raja"/>
    <property name="age" value="30"/>
  </bean>

</beans>
```

### Voter.java

```
package com.nt.beans;
```

```

import java.util.Date;

import lombok.Setter;

@Setter
public class Voter {
    private String name;
    private int age;
    private Date dov;

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

    //initialization event
    public void myInit() {
        System.out.println("Voter : myInti()");
        dov = new Date(); //initializing left over properties
        boolean flag = false;
        if (name==null || name=="") { //validation
            System.out.println("Null is not allow, you have to pass
you name");
            flag = true;
        }
        if (age<0)
            age = age*-1;      //Post processing
        if (age>100) {
            System.out.println("The age limit is between 100");
            flag = true;
        }
        if (flag)
            throw new IllegalArgumentException("Invalid inputs, try
to provide correct inputs");
    }

    //Destruction event
    public void myDestroy() {
        System.out.println("Voter : myDestroy()");
        //Nullify all values
        name = null;
    }
}

```

```

        age = 0;
        dov = null;
    }

    //Business Method
    public String checkVotingEligibility() {
        System.out.println("Voter : checkVotingEligibility()");
        if (age>=18)
            return "Mr/ Miss/ Mrs "+name+" your age is
"+age+".\nYou are eligible for vote.\nValidation checking date is "+dov;
        else
            return "Mr/ Miss/ Mrs "+name+" your age is
"+age+".\nYou are not eligible for vote./n Validation checking date is "+dov;
    }
}

```

### BeanLifeCycleTest.java

```

package com.nt.test;

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

import com.nt.beans.Voter;

public class BeanLifeCycleTest {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        Voter voter = null;
        //Create IoC Container
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        //get Bean
        voter = ctx.getBean("voter", Voter.class);
        System.out.println(voter.checkVotingEligibility());
        //close container
        ((AbstractApplicationContext) ctx).close();
    }
}

```

### Limitations of Declarative Approach of Spring bean life cycle:

- a. We should remember and configure life cycle methods, otherwise no life cycle activities take place.
- b. While configuring pre-defined, third party supplied java classes as spring beans we should search, identify life cycle methods in order to configure them.

### Advantages:

- a. we can develop spring bean as non-invasive class.

### Q. What is the difference between @Required or Constructor Injection and Init life cycle method?

**Ans.** Using @Required and Constructor Injection we check whether properties are participating in the injection or not but we cannot check whether valid values are injected or not. Whereas by using custom init method we can do validation bean post processing (modification of injected data) and initiating left over properties.

**Note:** @Required is deprecated from Spring 5.1 in support to constructor injection and init life cycle method.

### Note:

- ✓ Only for Singleton scope beans the destroy life cycle method executes not for other scopes because the other scope bean objects will not place in the internal cache of IoC container.

```
<bean id="voter" class="com.nt.beans.Voter" init-method="myInit"
destroy-method="myDestroy" scope="prototype">
    <property name="name" value="raja"/>
    <property name="age" value="30"/>
</bean>
```

- ✓ Instead of configuring same init, destroy life cycle methods for multiple beans we can configure only for 1 time as default init, destroy life cycle methods in the <beans> tag as shown below.

```
<beans default-init-method="myInit" default-destroy-method="myDestroy"
.....
<bean id="voter" class="com.nt.beans.Voter">
    <property name="name" value="raja"/>
```

```

        <property name="age" value="30"/>
    </bean>

    <bean id="voter1" class="com.nt.beans.Voter">
        <property name="name" value="ravi"/>
        <property name="age" value="23"/>
    </bean>

</beans>
```

**Note:** In case of BeanFactory Container the destroy life cycle method will execute only when we call factory.destroySingletons() method because there is not provision to stop or close BeanFactory container.

```

package com.nt.test;

import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;

import com.nt.beans.Voter;

public class BeanLifeCycleTest1 {

    public static void main(String[] args) {
        DefaultListableBeanFactory factory = null;
        XmlBeanDefinitionReader reader = null;
        Voter voter = null;
        //Create IoC Container
        factory = new DefaultListableBeanFactory();
        reader = new XmlBeanDefinitionReader(factory);
        reader.loadBeanDefinitions("com/nt/cfgs/applicationContext.xml");
        //get Bean
        voter = factory.getBean("voter", Voter.class);
        System.out.println(voter.checkVotingEligibility());

        //close container
        factory.destroySingletons();
    }
}
```

## Using Programmatic Approach

- We need to make our spring bean class implementing two interfaces,
  - a. InitializingBean ()  
public void afterPropertiesSet() alternate custom init method
  - b. DisposableBean ()  
public void destroy() alternate to custom destroy method
- This approach makes spring bean as invasive (tight coupling with Spring API).
- In this approached the life cycle methods will be executed automatically by seeing the implementation of interface on spring bean class. i.e. there is no need of configuring life cycle methods anywhere.

### Directory Structure of IOCProj42-BeanLifeCycle-Programmatic:

- + Copy paste the IOCProj41-BeanLifeCycle-Declarative-XML application and change rootProject.name to IOCProj42-BeanLifeCycle-Programmatic in settings.gradle file.
- + Need not to add any new file same structure as IOCProj41-BeanLifeCycle-Declarative-XML.
- + Add the following code in their respective files.

#### Voter.java

```
package com.nt.beans;

import java.util.Date;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

import lombok.Setter;

@Setter
public class Voter implements InitializingBean, DisposableBean {
    private String name;
    private int age;
    private Date dob;

    public Voter() {
        System.out.println("Voter : Voter()");
    }
}
```

```

// Business Method
public String checkVotingEligibility() {
    System.out.println("Voter : checkVotingEligibility()");
    if (age >= 18)
        return "Mr/ Miss/ Mrs " + name + " your age is " + age
               + ".\nYou are eligible for vote.\nValidation
checking date is " + dov;
        return "Mr/ Miss/ Mrs " + name + " your age is " + age
               + ".\nYou are not eligible for vote./n
Validation checking date is " + dov;
    }
    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("Voter : afterPropertiesSet()");
        dov = new Date(); // initializing left over properties
        boolean flag = false;
        if (name == null || name == "") { // validation
            System.out.println("Null is not allow, you have to pass
you name");
            flag = true;
        }
        if (age < 0)
            age = age * -1; // Post processing
        if (age > 100) {
            System.out.println("The age limit is between 100");
            flag = true;
        }
        if (flag)
            throw new IllegalArgumentException("Invalid inputs, try
to provide correct inputs");
    }
    @Override
    public void destroy() throws Exception {
        System.out.println("Voter : destroy()");
        // Nullify all values
        name = null;
        age = 0;
        dov = null;
    }
}

```

### applicationContext.xml

```
<bean id="voter" class="com.nt.beans.Voter">
    <property name="name" value="raja"/>
    <property name="age" value="30"/>
</bean>
```

### Advantages of Programmatic approach:

- a. No need of life cycle methods they will be executed automatically not.
- b. If any pre-defined class is implementing based this approach, we need to search and configure life cycle methods separately.

### Disadvantages:

- a. Makes spring bean as invasive.
- b. We cannot make third party supplied classes implementing these interfaces.

**Q. If we enable spring life cycle in both programmatic and declarative approach, which will be executed?**

**Ans.** Both will execute but first IoC container calls programmatic approach methods later it calls declarative approach methods.

For instantiation event

- a. InitializingBean's afterPropertiesSet()
- b. Custom init method

For Destruction event

- a. DisposableBean's destroy()
- b. Custom destroy method

 In DataSourceTransactionManager class (Spring supplied pre-defined class)

```
@Override
public void afterPropertiesSet() {
    if (getDataSource() == null) {
        throw new
IllegalStateException("Property 'dataSource' is required");
    }
}
```

## Using Annotation approach

- We need to configure custom init method having @PostConstruct annotation.
- We need to configure custom destroy method having @PreDestroy annotation.
- Both these annotations are Java config annotations (given JDK).
- Makes our Spring bean as non-invasive.
- @PostConstruct and @PreDestroy are part of JDK API up to Java 8 from Java 9 onwards they are removed (in fact all Java config annotations) from JDK to make the JDK s/w as the light weight s/w. So, to add jar file to the Build path.

```
//  
https://mvnrepository.com/artifact/javax.annotation/javax.annotation-api  
implementation group: 'javax.annotation', name: 'javax.annotation-  
api', version: '1.3.2'
```

- This approach cannot be used while working with BeanFactory container.
- Here there is no need of configuration life cycle methods anywhere based on the annotations that are added the life cycle method will execute automatically.

### Directory Structure of IOCProj43-BeanLifeCycle-AnnotationDriven:

- + Copy paste the IOCProj41-BeanLifeCycle-Declarative-XML application and change rootProject.name to IOCProj42-BeanLifeCycle-AnnotationDriven in settings.gradle file.
- + Need not to add any new file same structure as IOCProj41-BeanLifeCycle-Declarative-XML.
- + Add the following code in their respective files.

#### Voter.java

```
package com.nt.beans;  
  
import java.util.Date;  
  
import javax.annotation.PostConstruct;  
import javax.annotation.PreDestroy;  
  
import org.springframework.beans.factory.annotation.Value;
```

```

import org.springframework.stereotype.Component;

import lombok.Setter;

@Setter
@Component("voter")
public class Voter {

    @Value("Raju")
    private String name;
    @Value("28")
    private int age;
    private Date dob;

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

    //initialization event
    @PostConstruct
    public void myInit() {
        System.out.println("Voter : myInit()");
        dob = new Date(); //initializing left over properties
        boolean flag = false;
        if (name==null || name=="") { //validation
            System.out.println("Null is not allow, you have to pass
you name");
            flag = true;
        }
        if (age<0)
            age = age*-1;      //Post processing
        if (age>100) {
            System.out.println("The age limit is between 100");
            flag = true;
        }
        if (flag)
            throw new IllegalArgumentException("Invalid inputs, try
to provide correct inputs");
    }

    ...
}

```

```

//Destruction event
@PreDestroy
public void myDestroy() {
    System.out.println("Voter : myDestroy()");
    //Nullify all values
    name = null;
    age = 0;
    dov = null;
}

// Business Method
public String checkVotingEligibility() {
    System.out.println("Voter : checkVotingEligibility()");
    if (age >= 18)
        return "Mr/ Miss/ Mrs " + name + " your age is " + age
               + ".\nYou are eligible for vote.\nValidation
               checking date is " + dov;
    else
        return "Mr/ Miss/ Mrs " + name + " your age is " + age
               + ".\nYou are not eligible for vote./n
Validation checking date is " + dov;
}

}

```

### applicationContext.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans default-init-method="myInit" default-destroy-method="myDestroy"
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           https://www.springframework.org/schema/beans/spring-beans-4.3.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <context:component-scan base-package="com.nt.beans"/>

</beans>

```

- The methods on which apply @PostConstruct and @PreDestroy annotations should have following signature

```
public void <method> (<no params>) {  
    .....  
    .....  
}
```

#### Advantages:

- Makes spring bean as non-invasive.
- No need of configure life cycle methods anywhere based on annotations they will be executed automatically.

#### Disadvantaged:

- We cannot use this approach for third party supplied, pre-defined classes.

Q. What happens if apply all 3 approaches of spring bean life on the spring bean class?

Ans. The init and destroy life cycle methods will execute in the following order

- Annotation Driven
- Programmatic
- declarative

Q) A sub class is configured as Spring bean, life cycle methods are not in sub class, but available in super class will they execute as sub class life cycle method when the sub class is c as spring bean?

Ans. Yes.

#### Conclusion on Spring Bean Life Cycle:

- If the Spring bean is user-defined class, then go for annotation driven approach for life cycle management.
- If the spring bean is spring supplied pre-defined class, then check for InitializingBean, DisposableBean interfaces implementation. If implemented automatically programmatic approach continues, if not implemented go for declarative approach.
- If Spring bean is third party supplied class, then go for declarative approach.

Q. How did u use spring bean life cycle in your project?

Ans. In Controller, Service, DAO classes we use init life cycle method to check

whether implement bean properties are injected or not and destroy life cycle method to nullify bean properties.

For example,

In service class, we use init life cycle method to check DAO objects are injected or not and we use destroy life cycle method to nullify the injected DAO objects.

#### [Directory Structure of IOCAnnoProj07-RealTimeDI-RealTimeStrategyDP-BeanLifeCycle:](#)

- + Copy paste the IOCAnnoProj05-RealTimeDI-RealTimeStrategyDP-PropertiesFile application and change rootProject.name to IOCAnnoProj07-RealTimeDI-RealTimeStrategyDP-BeanLifeCycle in settings.gradle file.
- + Need not to add any new file same structure as IOCAnnoProj07-RealTimeDI-RealTimeStrategyDP-BeanLifeCycle.
- + Add the following code in their respective files.

#### OracleCustomerDAOImpl.java

```
@PostConstruct  
public void myInit() {  
    if (ds==null)  
        throw new IllegalArgumentException("DS is not injected");  
}  
  
@PreDestroy  
public void myDestroy() {  
    ds = null;  
}
```

#### OracleCustomerDAOImpl.java

```
@PostConstruct  
public void myInit() {  
    if (ds==null)  
        throw new IllegalArgumentException("DS is not injected");  
}  
  
@PreDestroy  
public void myDestroy() {  
    ds = null;  
}
```

### CustomerMgmtServiceImpl.java

```
@PostConstruct  
public void myInit() {  
    if (dao==null)  
        throw new IllegalArgumentException("DAO is not injected");  
}  
  
@PreDestroy  
public void myDestroy() {  
    dao = null;  
}
```

### MainController.java

```
@PostConstruct  
public void myInit() {  
    if (service==null)  
        throw new IllegalArgumentException("Service is not injected");  
}  
  
@PreDestroy  
public void myDestroy() {  
    service = null;  
}
```

## Aware Injection

- ⊕ Aware injection is also known as Interface Injection or Contextual Dependency Lookup.

Q. When should we go dependency lookup and when should go for dependency injection (Setter/ Constructor Injection)?

Ans.

- If dependent object is required only in one method of target class then go for dependency lookup.  
e.g. Vehicle (target class) Engine (Dependency)
  - |--> Engine is required only in the move () of target class not in other methods, so go for dependency lookup.
  - |--> To implement it, create an extra IOC container in the specific one method of target class and call getBean (-,-) using that extra IoC container.
- If dependent bean object is required in multiple methods of target class

then go for dependency injection (setter/ constructor injection).

e.g. Cricketer(target) Ball (dependent)

| --> Ball is required in multiple methods of Cricketer class like batting (), bowling (), fielding (), wicketkeeping () and etc. so go for dependency injection.

### Directory Structure of IOCProj44-TraditionalDependencyLookup:

- + Copy paste the IOCProj25-DependencyLookup and change rootProject.name to IOCProj44-TraditionalDependencyLookup in settings.gradle file.
- + Need not to add any new file same structure as IOCProj25-DependencyLookup.
- + Add the following code in their respective files as part of the replace for the particular section.

#### Vehicle.java

```
public void journey(String sourcePlace, String destPlace) {  
    ApplicationContext ctx = null;  
    Engine engg = null;  
    //Create BeanFactory or IoC container  
    ctx = new  
    ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");  
    //get dependent bean class object  
    engg = ctx.getBean(beanId, Engine.class);  
    engg.start();  
    System.out.println("Journey started from : "+sourcePlace);  
    System.out.println("Journey was going on from  
    "+sourcePlace+" to "+destPlace);  
    System.out.println("Journey stoped at : "+destPlace);  
}
```

#### Vehicle.java

```
public static void main(String[] args) {  
    ApplicationContext ctx = null;  
    Vehicle vechicle=null;  
    //Create BeanFactory or IoC container  
    ctx = new  
    ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");  
    //get dependent bean class object
```

```

        vechicle = ctx.getBean("vechicle", Vehicle.class);
        //invoke methods
        vechicle.journey("Odisha", "Hyd");
        vechicle.entertainment();
        vechicle.soundHorn();
        vechicle.fillFuel();

        //close the container
        ((AbstractApplicationContext) ctx).close();
    }

```

- + If we perform traditional Dependency lookup by Using Base Container and Additional Container as ApplicationContext container then we need to enable lazy-init for singleton scope target, dependent bean classes then unnecessary pre-instantiation on singleton beans for multiple times will be avoided.

#### applicationContext.xml

```

<!-- Dependent Bean Id -->
<bean id="engg" class="com.nt.beans.Engine" lazy-init="true"/>

<!-- Target bean id -->
<bean id="vechicle" class="com.nt.beans.Vehicle" lazy-init="true">
    <constructor-arg>
        <idref bean="engg"/>
    </constructor-arg>
</bean>

```

#### Limitations of Traditional dependency Lookup:

- a. Taking extra IoC container in the specific method of target class is bad.
- b. The Injected Dependent class bean id in target class is having more visibility in multiple methods of target class. (Actually, it is required only in one method of target class).
- c. If the IoC container is ApplicationContext it forces to disable pre-instantiation of singleton by using lazy-init="true" or by using @Lazy.

- + Convert the above project as annotation-based application

## Directory Structure of IOCAnnoProj08-TraditionalDependencyLookup:

- ⊕ Copy paste the IOCProj44-TraditionalDependencyLookup and change rootProject.name to IOCAnnoProj08-TraditionalDependencyLookup in settings.gradle file.
- ⊕ Need not to add any new file same structure as IOCProj44-TraditionalDependencyLookup.
- ⊕ Add the following code in their respective files as part of the replace for the particular section.

### Engine.java

```
@Component("engg")
@Lazy
public class Engine {
```

### Vehicle.java

```
@Component("vehicle")
@Lazy
public class Vehicle {

    @Value("engg")
    private String beanId;

    public Vehicle() {
        System.out.println("Vehicle : Vehicle()");
    }
}
```

### 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"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans-4.3.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <context:component-scan base-package="com.nt.beans"/>

</beans>
```

- If you want to connect the value for the beanId property we can go for properties file.
- Create a package com.nt.commons having the app.properties have the key and value and change the following things in their respective files.

### app.properties

```
#Details
dependency.id=engg
```

### Vehicle.java

```
@Value("${dependency.id}")
private String beanId;
```

### 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"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans-4.3.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <context:component-scan base-package="com.nt.beans"/>
    <context:property-placeholder
location="com/nt/commons/app.properties"/>

</beans>
```

- The Spring beans that are annotated with Stereo type annotations or @Named Annotation will be instantiated using parameterized constructor only when @Autowired is placed on the top of parametrized constructor. otherwise they will be instantiated using 0-param constructor so make sure that 0-param constructor is always available.

To overcome the above problems of traditional dependency lookup:

- Design specific method taking Client Supplied Container object.

### Vehicle.java

```
public void journey(String sourcePlace, String destPlace,  
ApplicationContext ctx) {  
    Engine engg = null;  
    //get dependent bean class object  
    engg = ctx.getBean(beanId, Engine.class);  
    engg.start();  
    System.out.println("Journey started from : "+sourcePlace);  
    System.out.println("Journey was going on from  
"+sourcePlace+" to "+destPlace);  
    System.out.println("Journey stoped at : "+destPlace);  
}
```

### DependencyLookupTest.java

```
//invoke methdos  
vechicle.journey("Odisha", "Hyd", ctx);
```

- Here @Lazy or lazy-init="true" is not required.
  - The big limitation here is assigning business method with Spring Specific IOC container object. This kills non-invasive behaviour of target class.
- b. Design the specific method target class taking Client created IOC container object and Dependent bean class id.

### Vehicle.java

```
public void journey(String sourcePlace, String destPlace,  
ApplicationContext ctx, String beanId) {  
    Engine engg = null;  
    //get dependent bean class object  
    engg = ctx.getBean(beanId, Engine.class);  
    engg.start();  
    System.out.println("Journey started from : "+sourcePlace);  
    System.out.println("Journey was going on from  
"+sourcePlace+" to "+destPlace);  
    System.out.println("Journey stoped at : "+destPlace);  
}
```

## DependencyLookupTest.java

```
//invoke methods  
vechicle.jouney("Odisha", "Hyd", ctx, "engg");
```

- Here @Lazy or lazy-init="true" not required.
  - No need of taking property in target bean class as instance variable to hold dependent bean id.
  - This is bad practice, business method and spring bean is becoming invasive because Container object, bean id.
- c. Get Client created IoC Container to Target Bean class using Aware Injection and use it specific method of target bean class.
- + IoC container injects special values to spring bean based on the XxxAware Interface that is implemented by spring bean.

### XxxAware interfaces are:

- BeanNameAware --> To inject current spring id/ name to Spring bean itself.
- BeanFactoryAware --> To inject underlying BeanFactory object.
- ApplicationContextAware --> To inject underlying ApplicationContext object.  
and etc.

### Note:

- ✓ Aware injection is not for injecting simple values or dependent bean objects, it is object injecting IOC container maintained special values.
- ✓ IoC container internally maintains some special values along with Spring bean objects like bean ids, BeanFactory object, ApplicationContext object and etc. To inject these values, go aware Injection.

```
BeanNameAware (I)  
|--> public void setBeanName(String beanId)  
BeanFactoryAware (I)  
|--> public void setBeanFactory(BeanFactory factory)  
ApplicationContextAware (I)  
|--> public void setApplicationContext(ApplicationContext ctx)
```

- These are not setter methods of setter injection. These are fixed methods declared in the interfaces.
- Setter method will be there for each property. For example, if the property name x, then setter method will be setX(-) method.

### Directory Structure of IOCProj45-TraditionalDependencyLookup-AwareInjection:

- + Copy paste the IOCProj44-TraditionalDependencyLookup and change rootProject.name to IOCProj45-TraditionalDependencyLookup-AwareInjection in settings.gradle file.
- + Need not to add any new file same structure as IOCProj44-TraditionalDependencyLookup.
- + Add the following code in their respective files as part of the replace for the particular section.

#### Vehicle.java

```
package com.nt.beans;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

public class Vehicle implements ApplicationContextAware {

    private String beanId;
    private ApplicationContext ctx = null;

    public Vehicle(String beanId) {
        System.out.println("Vehicle : Vehicle(-)");
        this.beanId = beanId;
    }

    @Override
    public void setApplicationContext(ApplicationContext ctx) throws BeansException {
        System.out.println("Vehicle.setApplicationContext()");
        this.ctx = ctx;
    }

    public void entertainment() {
```

```

        System.out.println("Vechicle is equipped with DVD player for
entertainment");
    }
    public void soundHorn() {
        System.out.println("Vechicle is equipped with skoda horn");
    }

    public void fillFuel() {
        System.out.println("Vechicle is having fuel tank of 60 liters");
    }

    public void journey(String sourcePlace, String destPlace) {
        Engine engg = null;
        //get dependent bean class object
        engg = ctx.getBean(beanId, Engine.class);
        engg.start();
        System.out.println("Journey started from : "+sourcePlace);
        System.out.println("Journey was going on from
"+sourcePlace+" to "+destPlace);
        System.out.println("Journey stoped at : "+destPlace);
    }

}

```

#### Order of execution:

- Bean Instantiations (Constructor Injection).
- Setter injection.
- Aware Injection (based on XxxAware interfaces that are implemented).
- @PostConstruct method (if available).
- InitializingBean's afterPropertiesSet () (if available).
- Custom init method (if configured)
  
- + Aware Injection is also called Interface injection because the injection is taking place only after implementing special XxxAware(I) on spring bean class.
- + Aware Injection is also called Contextual dependency lookup/ Injection because the injection is taking place only after implementing special XxxAware(I) on spring bean class as the contract/ context to be agreed/ satisfied by spring bean class.

- Servlet container injects ServletConfig object to servlet class obj whose class implementing/ satisfying javax.servlet.Servlet(I) directly or indirectly. So, we can this process as Interface injection/Contextual Dependency lookup.

### Limitations of Aware injection-based Dependency Lookup:

- makes the spring bean class as invasive (because we need to implement XxxAware).
- The Injected BeanFactory/ApplicationContext object visible through out target class though it is required in one method.
- The Injected Dependent class bean id is visible through out target class though it is required in one method.

### Advantages:

- No need of taking Extra container in target class method.
- No need of enabling lazy init on spring beans. (lazy-init-true or @Lazy not required).
- No need of designing business method with technical inputs like taking Container object, bean id as the arguments.

**Conclusion:** Go LMI (Lookup method to solve all problems) without any side effects.

## Lookup Method Injection (LMI)

**Problem:** If target spring bean scope is singleton and dependent spring bean scope is prototype somehow dependent spring bean acts as singleton only.

Target bean scope	Dependent bean scope	Resultant scope of dependent bean
Singleton	singleton	Singleton (OK)
Singleton	Prototype	Singleton (Not OK) (Use DL or AI+DL or LMI to solve this problem)
Prototype	singleton	Singleton (OK)
Prototype	Prototype	Prototype (OK)

- One ServletContainer/ WebContainer takes all requests, but it uses multiple objects of RequestHandler to handle/ process multiple requests if we design them as spring beans.

ServletContainer/ WebContainer <-----> RequestHandler  
 (target class) (singleton scope)                               (Dependent class) (prototype scope)

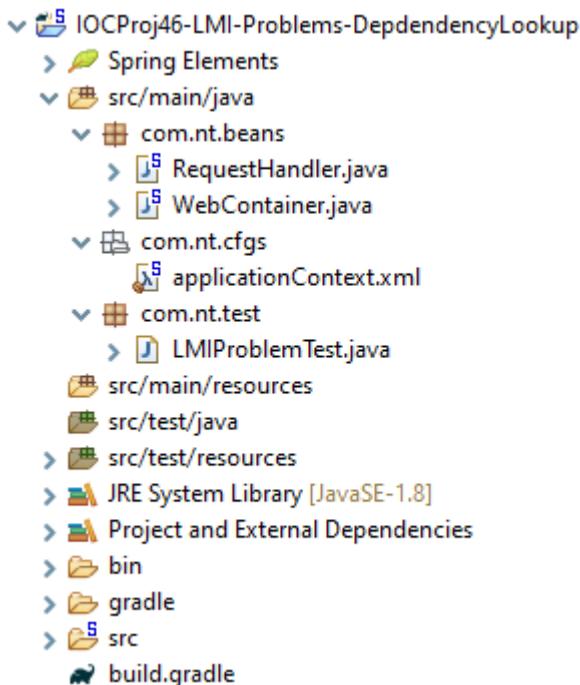
- >If we use setter/ constructor injection to inject dependent object to target object somehow prototype scope dependent bean acts as singleton scope bean).

Faculty <-----> Student  
 (target-singleton)    (dependent-prototype)

**Conclusion:** Do not use setter, constructor Injections in the following two situations.

- If Dependent object is required only in one method of target class.
- If target spring bean scope is singleton and dependent spring bean scope prototype.  
 (Here prefer using Traditional Dependency Lookup or Aware Injection + Dependency lookup or Lookup Method Injection (best)).

#### Directory Structure of IOCProj46-LMI-Problems-DependencyLookup:



- Develop the above directory structure and package, class, XML file then use the following code with in their respective file.
- Copy paste build.gradle from any other gradle project because we are using same spring-context-support dependency only.

### RequestHandler.java

```
package com.nt.beans;

public class RequestHandler {

    private static int count;

    public RequestHandler() {
        count++;
        System.out.println("RequestHandler : RequestHandler() :
"+count);
    }

    public void handleRequest(String data) {
        System.out.println("Handling request with "+data+" using the
object "+count);
    }
}
```

### WebContainer.java

```
package com.nt.beans;

public class WebContainer {

    private RequestHandler rh;

    public void setRh(RequestHandler rh) {
        this.rh = rh;
    }

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

    public void processRequest(String data) {
        System.out.println("WebContainer is processing request with
data : "+data+" by giving it to handler.");
        rh.handleRequest(data);
    }
}
```

### 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-4.3.xsd">

    <bean id="handler" class="com.nt.beans.RequestHandler"
          scope="prototype"/>

    <bean id="container" class="com.nt.beans.WebContainer"
          scope="singleton">
        <property name="rh" ref="handler"/>
    </bean>

</beans>
```

### LMIProblem.java

```
package com.nt.test;

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

import com.nt.beans.WebContainer;

public class LMIProblemTest {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        WebContainer wc = null;
        //Create ioc container
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        //get Target class bean object
        wc = ctx.getBean("container", WebContainer.class);
        wc.processRequest("hello");
        System.out.println("-----");
        wc.processRequest("hai");
```

```

        System.out.println("-----");
        wc.processRequest("123");

        //close container
        ((AbstractApplicationContext) ctx).close();
    }

}

```

To solve the above problems:

**Approach 1:** Use Traditional Lookup by taking an extra IoC container

**Directory Structure of IOCProj47-LMI-Solution1-TraditionalDependencyLookup:**

- + Copy paste the IOCProj46-LMI-Problems-DependencyLookup application and change rootProject.name to IOCProj47-LMI-Solution1-TraditionalDependencyLookup in settings.gradle file.
- + Need not to add any new file same structure as IOCProj46-LMI-Problems-DependencyLookup.
- + Add the following code in their respective files.

### WebContainer.java

```

package com.nt.beans;

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

public class WebContainer {

    private String beanId;

    public void setBeanId(String beanId) {
        this.beanId = beanId;
    }

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

    public void processRequest(String data) {

```

```

        System.out.println("WebContainer is processing request with
data : "+data+" by giving it to handler.");
        ApplicationContext ctx = null;
        RequestHandler handler = null;
        //Create extra IoC container
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        //get Dependent object using dependency lookup
        handler = ctx.getBean(beanId, RequestHandler.class);
        handler.handleRequest(data);
    }

}

```

#### applicationContext.xml

```

<bean id="handler" class="com.nt.beans.RequestHandler"
scope="prototype"/>

<bean id="container" class="com.nt.beans.WebContainer"
scope="singleton">
    <property name="beanId">
        <idref bean="handler"/>
    </property>
</bean>

```

#### Limitations of Approach1:

- Taking an extra IoC container in specific method of target bean class is bit heavy and kills the performance.
- The Injected bean id of Dependent class is visible in multiple methods of target class unnecessarily.
- Singleton scope target bean will be pre-instantiated for multiple times, because the extra IoC container (This can be solved by using lazy-init="true" or @Lazy)

**Approach 2:** Aware Injection + Traditional dependency lookup without taking extra IOC container.

#### Directory Structure of IOCProj48-LMI-Solution2-AIWithTDL:

- + Copy paste the IOCProj46-LMI-Problems-DependencyLookup application

and change rootProject.name to IOCProj48-LMI-Solution2-AIWithTDL in settings.gradle file.

- ⊕ Need not to add any new file same structure as IOCProj46-LMI-Problems-DependencyLookup.
- ⊕ Add the following code in their respective files.

### WebContainer.java

```
package com.nt.beans;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class WebContainer implements ApplicationContextAware {

    private String beanId;
    private ApplicationContext ctx = null;

    public void setBeanId(String beanId) {
        this.beanId = beanId;
    }

    @Override
    public void setApplicationContext(ApplicationContext ctx) throws
BeansException {
        System.out.println("WebContainer : setApplicationContext(-)");
        this.ctx = ctx;
    }

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

    public void processRequest(String data) {
        System.out.println("WebContainer is processing request with
data : "+data+" by giving it to handler.");
        RequestHandler handler = null;
        //get Dependent object using AI+ dependency lookup
```

```

        handler = ctx.getBean(beanId, RequestHandler.class);
        handler.handleRequest(data);
    }

}

```

### Limitations of Approach2:

- Implementation of ApplicationContextAware makes target bean as invasive.
- The Injected bean id of Dependent class is visible in multiple methods of target class unnecessarily.
- The Injected ApplicationContext object through aware Injection is visible in all the methods of Target bean unnecessarily.

### Advantages:

- No need of taking extra IOC container, so gives good performance.
- No need of enabling lazy-init="true" or @Lazy on target bean class.

### Approach 3: Use Lookup Method Injection

(Perform dependency lookup but u do not do that let the IoC container do that work internally)

**Step 1:** Take target spring bean class as abstract class having one abstract method whose return type is dependent bean class/ type. (This method is called as lookup method because IOC container internally implements this method having dependency lookup to get and return a Dependent class obj in the IOC container generated In Memory sub class for target class)

**Step 2:** Configuration the above method as lookup method in Spring bean configuration.

**Step 3:** Use the dependent class bean object in target class business method by getting that object through method call.

### Directory Structure of IOCProj49-LMI-FinalSolution3-LookupMethodInjection:

- + Copy paste the IOCProj46-LMI-Problems-DependencyLookup application and change rootProject.name to IOCProj49-LMI-FinalSolution3-LookupMethodInjection in settings.gradle file.
- + Need not to add any new file same structure as IOCProj46-LMI-Problems-DependencyLookup.
- + Add the following code in their respective files.

### WebContainer.java

```
package com.nt.beans;

public abstract class WebContainer {

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

    public abstract RequestHandler createRequestHandler();

    public void processRequest(String data) {
        System.out.println("WebContainer is processing request with
data : "+data+" by giving it to handler.");
        RequestHandler handler = null;
        //get Dependent object using dependency lookup code
generatedby the container
        handler = createRequestHandler();
        handler.handleRequest(data);
    }

}
```

### applicationContext.xml

```
<bean id="handler" class="com.nt.beans.RequestHandler"
scope="prototype"/>

<bean id="container" class="com.nt.beans.WebContainer"
scope="singleton">
    <lookup-method name="createRequestHandler"
bean="handler"/>
</bean>
```

By seeing <lookup-method> tag under <bean>, the IoC container internally generates one InMemory sub class for configuration bean class (in our case for WebContainer class) in JVM memory of RAM implemented the configuration method (in our case createRequestHandler method given in "name" attribute) having dependency lookup logic to get and return specified bean id based Dependent bean class obj (in our case it is "handler" given in "bean" attribute).

### Advantages of Approach 3:

- a. Both and target and dependent bean classes are non-invasive.
- b. Container takes care of Dependency lookup by generating one InMemory class as sub class of target class.
- c. No need of extra IOC container.
- d. No need of injecting Dependent class id.
- e. No need of enabling lazy-init or @Lazy for Target bean class.
- f. It is industry standard.

**Note:** We can use @Lookup as alternate <lookup-method> tag in annotation driven environment.

### Directory Structure of IOCAnnoProj09-LMI-FinalSolution3-

#### LookupMethodInjection:

- + Copy paste the IOCProj49-LMI-FinalSolution3-LookupMethodInjection application and change rootProject.name to IOCAnnoProj09-LMI-FinalSolution3-LookupMethodInjection in settings.gradle file.
- + Need not to add any new file same structure as IOCProj49-LMI-FinalSolution3-LookupMethodInjection.
- + Add the following code in their respective files.

#### RequestHandler.java

```
@Component("handler")
@Scope("prototype")
public class RequestHandler {
```

#### WebContainer.java

```
import org.springframework.beans.factory.annotation.Lookup;
import org.springframework.stereotype.Component;

@Component("container")
public abstract class WebContainer {

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

    @Lookup
    public abstract RequestHandler createRequestHandler();
```

## 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"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans-4.3.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <context:component-scan base-package="com.nt.beans"/>

</beans>
```

LMI – Lookup Method Injection Internal flow:

### **RequestHandler.java**

```
-----
package com.nt.beans;

public class RequestHandler {
    private static int count;
    public RequestHandler() {
        count++;
        System.out.println("RquestHandler : RquestHandler() : "+count);
    }
    public void handleRequest(String data) {
        System.out.println("Handling request with "+data+" using the object
"+count);
    }
}
```

Internal cache of IoC container (h)

container	WebContainer\$CGLIB\$Proxy class object (j?)

### WebContainer.java

```
-----  
package com.nt.beans;  
  
public abstract class WebContainer {  
    public WebContainer() {  
        System.out.println("WebContainer : WebContainer()");  
    }  
    public abstract RequestHandler createRequestHandler();  
    public void processRequest(String data) { (m)  
        System.out.println("WebContainer is processing request with data :  
" + data + " by giving it to handler.");  
        RequestHandler handler = null;  
        //get Dependent object using dependency lookup code generated by  
the container  
        (t) handler = createRequestHandler(); (n)  
        handler.handleRequest(data); (u)  
    } (w)  
}
```

### 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-4.3.xsd">  
  
    <bean id="handler" class="com.nt.beans.RequestHandler"  
scope="prototype"/> (d) becomes ready to perform pre-instantiation  
singleton scope beans  
    <bean id="container" class="com.nt.beans.WebContainer"  
scope="singleton">  
        <lookup-method name="createRequestHandler" bean="handler"/>  
    </bean> (e)  
  
</beans>
```

### InMemoryProxy class Generated by CGLIB Libraries at runtime

```
public class WebContainer$CGLIB$Proxy extends WebContainer implements  
    ApplicationContextAware{ (f) InMemory proxy class  
        private ApplicationContext ctx; generation by seeing <lookup-method> tag  
        public void setApplicationContext(ApplicationContext ctx) {  
            this.ctx=ctx; (g) (Aware injection)  
        }  
        public RequestHandler createRequestHandler(){ (o)  
            //get Bean id from the bean attribute of <lookup-method> tag (handler)  
            .....  
            //get Dependent class object through dependency lookup operation  
            (q) new RequestHandler handler=ctx.getBean  
                ("handler",RequestHandler.class); (p)  
            return handler; (s)  
        }  
    }
```

### LMISolutionTest.java

```
public class LMISolutionTest { (a)  
    public static void main(String[] args) {  
        ApplicationContext ctx = null;  
        WebContainer wc = null;  
        //Create ioc container (b) IoC container creation  
        ctx = new ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");  
        //get Target class bean object (i) (c) creating InMemory  
        (k) wc = ctx.getBean("container", WebContainer.class); MetaData  
        (x) wc.processRequest("hello"); (l)  
        System.out.println("-----");  
        wc.processRequest("hai");  
        System.out.println("-----");  
        wc.processRequest("123");  
        //close container  
        ((AbstractApplicationContext) ctx).close();  
    }  
}
```

### Q. Can we configure abstract class as Spring bean?

**Ans.** Basically not possible because instantiation (object creation) for abstract class is not possible but we can configure it as spring bean by enabling static factory method bean instantiation or Lookup method injection. In both cases IOC container does not create object for the configuration abstract class. It will create object for other class/generated sub class.

e.g.

1. <bean id="cal" class="java.util.Calendar" factory-method="getInstance"/>

Here IOC container gets object for GregorianCalendar by calling  
Calendar.getInstance() method makes that obj as spring bean.

2. <bean id="container" class="com.nt.beans.WebContainer"

<lookup-method name="createRequestHandler" bean="handler"/>

</bean>

**Q. Can we take the target class of LMI Application as an interface instead of abstract class?**

**Ans.** Possible, but we need to make business methods as default or static methods by having Java version 8+. This indirectly allows us to configure Java 8+ interface as Spring bean but IoC container creates the implementation class of interface and makes it as spring bean object.

### WebContainer.java

```
package com.nt.beans;

public abstract class WebContainer {

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

    public abstract RequestHandler createRequestHandler();

    public void processRequest(String data) {
        System.out.println("WebContainer is processing request with
data : "+data+" by giving it to handler.");
        RequestHandler handler = null;
        //get Dependent object using dependency lookup code
generatedby the container
        handler = createRequestHandler();
        handler.handleRequest(data);
    }
}
```

**Q. Can we configure Java interface as spring bean?**

**Ans.** Not possible up to 7 but possible from Java8

**Note:** Here IC container does not create object for interface. It actually creates object for implementation class and makes it as spring bean.

## Method Injection or Method Replacer

**Environments to run the App:**

- a. Development Environment (Programmer)
- b. Testing Environment (Tester +Programmers) (Before release)
- c. UAT/ Pilot/ Sanity Test Environment (Client org -IT dept emps/ esters)
- d. Prod Environment (Project in Live end-users are using) (after release)

**To modify the Business logic for temporary period and to revert back to original logics we can use the following techniques (Without using Spring):**

- a. Comment the source of business logic to write new logic and uncomment it to revert back to original logics.
  - o We need to touch source code of main class/ target class/ service directly.
  - o Sometimes client organization will not get access to source code because s/w company does not deliver source code to Client organization (No de-compiler is 100% accurate).
  - o Any code you modify should go through testing regressively (again and again).

**Use case 1:**

Bank --> Loan Mela

Standard ROI: 24%

Offer ROI: 15% (offer period: 4)

Bank --> Loan Mela

Standard interest: compound

Offer interest: simple

**Use case 2:**

E-commerce web sites

Regular days: standard prices

Sale days/ offer days: up to 30% discount

- b. Write sub class for service class/ target class where business logic is there and override business methods in the sub class having new logics and use target class business logic to revert back to original logics.
  - No need of touching source code of target class/ main class/ service class to go to new logics or to revert back to old logic.
  - We need touch the source code of other classes like controller or client App to change from main class/ target class/ service to sub class (for new logics) and vice-versa (for reverting old logics).
  - Sometimes client organization will not get access to source code because s/w company does not deliver source code to Client organization (No de-compiler is 100% accurate).
  - Any code u modify should go through testing regressively.

**Note:**

- ✓ To overcome the above two problems in spring we can use method replacer/ method injection concept.
- ✓ Method replacer or method Injection and LMI are designed based proxy design pattern because they internally proxy class having new logics or additional responsibilities.

**Steps to work with Method Replacer/ Method Injection:**

**Step 1:** Develop target class/ main class/ service class having business logic in methods.

**Step 2:** Configure target class as Spring bean in Spring bean configuration file.

**Step 3:** Develop Method Replacer class by implementing MethodReplacer (I) keep new business logic for business method in reimplement (-, -, -) method.

**Step 4:** Configure the above Method Replacer class as Spring bean in Spring bean configuration file.

**Step 5:** Link Method replacer with target bean configuration by placing <replaced-method> under <bean> tag.

```
org.springframework.beans.factory.support.MethodReplacer (I)
    public Object reimplement (Object obj, Method method, Object [] args)
        throws Throwable;
```

**Object obj:** holds main class object reference.

**Method method:** holds target business method details

**Object [] args:** holds target method business method args value

These are given to be used in reimplement (-, -, -) while writing new logics.

Simple interest = (Principal x Rate x Term)/ 100

Compound interest = Principal x  $(1 + \text{Rate}/100)^{\text{Term}} - \text{Principal}$

### Directory Structure of IOCProj50-MethodReplacer:

```
IOCProj50-MethodReplacer
  Spring Elements
  src/main/java
    com.nt.cfgs
      applicationContext.xml
    com.nt.replacer
      BankLoanMgmt_CalculateSimpleInterestAmount.java
    com.nt.target
      BankLoanMgmt.java
    com.nt.test
      MethodInjectionTest.java
  src/main/resources
  src/test/java
  src/test/resources
  JRE System Library [JavaSE-1.8]
  Project and External Dependencies
  bin
  gradle
  src
  build.gradle
```

- Develop the above directory structure and package, class, XML file then use the following code with in their respective file.
- Copy paste build.gradle from any other gradle project because we are using same spring-context-support dependency only.

#### BankLoanMgmt.java

```
package com.nt.target;

public class BankLoanMgmt {

    public float calculateInterestAmount(float pAmt, float rate, float time) {
        System.out.println("BankLoanMgmt : calculateInterestAmount(-, -, -) - Compound interest amount");
        return (float) (pAmt * Math.pow((1+rate/100), time) - pAmt);
    }
}
```

### BankLoanMgmt\_CalculateSimpleInterest.xml

```
package com.nt.replacer;

import java.lang.reflect.Method;

import org.springframework.beans.factory.support.MethodReplacer;

public class BankLoanMgmt_CalculateSimpleInterestAmount implements MethodReplacer {

    @Override
    public Object reimplement(Object obj, Method method, Object[] args) throws Throwable {

        System.out.println("BankLoanMgmt_CalculateSimpleInterestAmount : reimplement(-, -, -) : Simple interest amount");
        float pAmt = 0.0f;
        float rate = 0.0f;
        float time = 0.0f;
        //get target class object
        pAmt = (float) args[0];
        rate = (float) args[0];
        time = (float) args[0];
        //write new business logic (Simple interest amount)
        return (pAmt*rate*time)/100.0f;
    }

}
```

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

    <bean id="bank" class="com.nt.target.BankLoanMgmt">
        <replaced-method name="calculateInterestAmount"
                         replacer="bankAccR">
            <arg-type>float</arg-type>
    
```

```

        <arg-type>float</arg-type>
        <arg-type>float</arg-type>
    </replaced-method>
</bean>

<bean id="bankAccR"
class="com.nt.replacer.BankLoanMgmt_CalculateSimpleInterestAmount"/>

</beans>
```

### MethodInjection.java

```

package com.nt.test;

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

import com.nt.target.BankLoanMgmt;

public class MethodInjectionTest {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        BankLoanMgmt bank = null;
        //Create IoC container
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        //get Target class bean
        bank = ctx.getBean("bank", BankLoanMgmt.class);
        //invoke method
        System.out.println("Interest amount:
"+bank.calculateInterestAmount(100000, 2, 12));
        //close container
        ((AbstractApplicationContext) ctx).close();
    }
}
```

**Note:** As of now no annotation is given for method replacer as equivalent.

**While working with method injection or replacer:**

- a. We cannot take target class as final class, because the generated proxy class/ InMemory class comes as the sub class of the target class and final classes can be sub classes. The generated exception is  
**Exception in thread "main"**  
**[org.springframework.beans.factory.BeanCreationException](#): Error creating bean with name 'bank' defined in class path resource [com/nt/cfgs/applicationContext.xml]: Instantiation of bean failed; nested exception is [java.lang.IllegalArgumentException](#): Cannot subclass final class com.nt.target.BankLoanMgmt**
- b. Since final methods cannot be overridden by sub classes. So, we cannot take target methods in target class (for the methods for which we want to write method replacer) as final methods. (This will not give exception but always target method will execute and method replacer logics will not execute).
- c. Though we can write new logics for multiple target methods in single reimplement (-, -) of a Method Replacer class, it is recommended to take separate replacer class for every target method. [if needed, it is recommended to take separate Method replacer for every target method/ business method].
- d. There is no annotation for <replaced-method> tag. So, while working with annotations target class must be configured using Xml <bean> in order to work with Method Replacer though the target class is user-defined class.
- e. The <arg-type> tags under <replaced-method> are use-full to configuration method replacer for only 1 form of target method even though their multiple overloaded forms of target method.
- f. The underlying IoC container generates Proxy class/ InMemory sub class for the target class only when <replaced-method> tag is present under <bean> tag that configures target spring bean class.

#### Directory Structure of IOCAnnoProj10-MethodReplacer:

- + Copy paste the IOCProj50-MethodReplacer and change rootProject.name to IOCAnnoProj10-MethodReplacer in settings.gradle file.
- + Need not to add any new file same structure as IOCProj50-MethodReplacer.
- + Add the following code in their respective files.

## 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"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <bean id="bank" class="com.nt.target.BankLoanMgmt">
        <replaced-method name="calculateInterestAmount"
replacer="bankAccR">
            <arg-type>float</arg-type>
            <arg-type>float</arg-type>
            <arg-type>float</arg-type>
        </replaced-method>
    </bean>

    <context:component-scan base-package="com.nt.replacer"/>

</beans>
```

## BankLoanMgmt\_CalculateSimpleInterestAmount.java

```
@Component("bankAccR")
public class BankLoanMgmt_CalculateSimpleInterestAmount implements
MethodReplacer {
```

Internal Flow of Method replacer:

Internal cache of IOC container (g?)

bank	BankLoanMgmt\$\$CGLIB\$\$Proxy object reference	(i?)
banCIAR	BankLoanMgmt_CaculateIntrestAmountReplacer object reference	

### BankLoanMgmt.java

```
-----  
public final class BankLoanMgmt {  
    public float calculateInterestAmount(float pAmt, float rate, float time) {  
        System.out.println("BankLoanMgmt : calculateInterestAmount(-, -, -)  
- ) - Compound interest amount");  
        return (float) (pAmt * Math.pow((1+rate/100), time) - pAmt);  
    }  
    public float getBalance() {  
        System.out.println("BankLoanMgmt : getBalance()");  
        return new Random().nextInt(1000000);  
    }  
}
```

### BankLoanMgmt\_CalculateSimpleInterestAmount.java

```
-----  
@Component("bankAccR")      (d)  
public class BankLoanMgmt_CalculateSimpleInterestAmount implements  
MethodReplacer {  
    @Override                  (o)  
    public Object reimplement(Object obj, Method method, Object[]  
args) throws Throwable {  
        System.out.println("BankLoanMgmt_CalculateSimpleInterestAmount : reimplement(-, -, -) : Simple interest amount");  
        float pAmt = 0.0f;  
        float rate = 0.0f;  
        float time = 0.0f;  
        //get target class object  
        pAmt = (float) args[0];  
        rate = (float) args[0];  
        time = (float) args[0];  
        //write new business logic (Simple interest amount)  
        (p) return (pAmt*rate*time)/100.0f;  
    }  
}
```

IOC container generated InMemory Proxy class

```
----- (f)
public class BankLoanMgmt$$CGLIB$$Proxy extends BankLoanMgmt implements ApplicationContextAware{
    private ApplicationContext ctx;
    public void setApplicationContext(ApplicationContext ctx){
        this.ctx=ctx; (g) (aware injection)
    }
    public float calculateInterestAmount(float pAmt, float rate, float time){ (l)
        //get Replacer bean id from <replaced-method> tag using xml api from
        //InMemoryMetaData of spring bean cfg file.. ( gets bankAccR)

        //get Method replacer class object. by using the above Bean id.
        MethodReplacer replacer=ctx.getBean("bankCIAR", MethodReplacer.class);
        //gets Target class object
        BankLoanMgmt target=ctx.getBean("bank", BankLoanMgmt.class);
        //get target Method info.
        Method method = target.getClass().getDeclaredMethod
            ("calcInterestAmount");
        //get target method args.
        Object args[] = new Object[]{pAmt, rate, time};
        //invoke reimplement(-,-,-) on replacer object.. (n)
        (q) float retVal=replacer.reimplement(target,method,args);
        return retVal; (r)
    } //method
} //class
```

#### applicationContext.xml

```
<beans ... >
    <bean id="bank" class="com.nt.target.BankLoanMgmt" (d)
          <replaced-method name="calculateInterestAmount"
replacer="bankAccR"> (d) Pre-instantiation
(e)           <arg-type>float</arg-type> of single scope beans
                <arg-type>float</arg-type>
                <arg-type>float</arg-type>
        </replaced-method>
    </bean>
    <context:component-scan base-package="com.nt.replacer"/> (d)
</beans>
```

```

public class MethodInjectionTest {
    (a)
    public static void main(String[] args) {
        ApplicationContext ctx = null;
        BankLoanMgmt bank = null;
        //Create IoC container
        ctx = new           (b) container creation
        ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext
        .xml");                      (c) InMemory MetaData
        //get Target class bean      (h)
        (j) bank = ctx.getBean("bank", BankLoanMgmt.class);
        //invoke method            (s)
        (k) System.out.println("Interest amount:
        "+bank.calculateInterestAmount(100000, 2, 12));

        //close container
        ((AbstractApplicationContext) ctx).close();
    }
}

```

**Note:**

- ✓ Normal classes related .java, class files allocate memory in the hard disk, whereas InMemory proxy classes source and compiled generated code takes place in JVM memory of RAM. So, we cannot see them.
- ✓ Spring IoC container use CGLIB Libraries to generate the proxy classes which are now part of spring jar/ libraries from spring 3.x (In older versions we used supply CGLIB Libraries separately)
- ✓ CGLIB stands for Code Generation Libraries.
- ✓ Each object of java.lang.reflect.Method class holds info about one java method.

**Q. Why the proxy class is coming as the sub class of Target class?**

**Ans.** Since super reference variable can refer Sub class object, they made target class as the super class for Proxy class. So, that we can use target class ref variable to refer Runtime generated proxy class object otherwise we cannot get proxy class object in our Client Apps.

## Factory Bean

Target Bean --> Normal Bean (Dependent): Injected Bean to Target Bean  
Normal Bean object.

Target Bean --> Factory Bean (Dependent): Injected Bean to target Bean is the  
Factory Bean created/ gathered Resultant object.

Factory Bean --> It is a spring bean acting selfless bean i.e. it never gives its  
object, it always gives resultant object. (This object comes because certain  
logics execution).

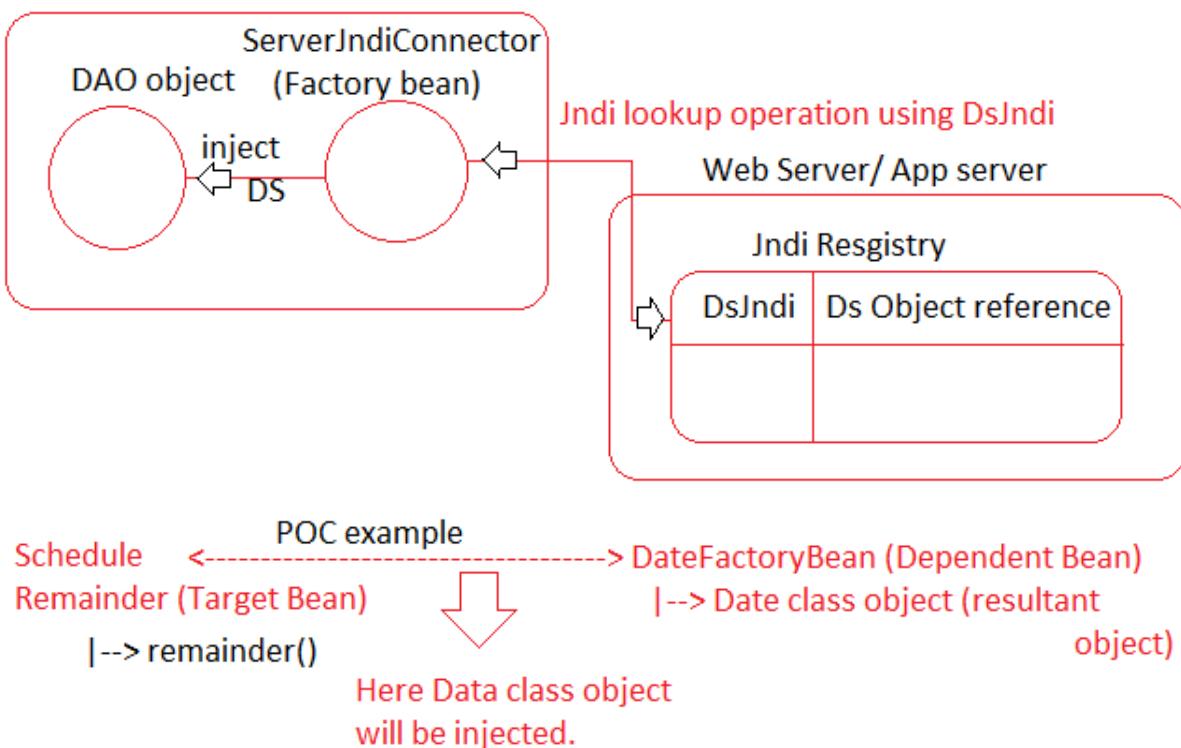
DAO (target Bean) ServerJndiConnector (dependent)

|--> gets DS object from Server managed Jndi registry  
using JNDI code  
|--> DS is resultant object.

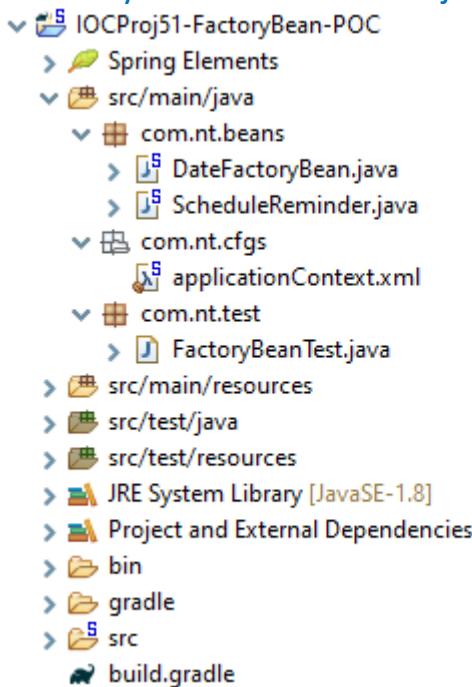
- Here we should take ServerJndiConnector as Factory Bean. So, though it  
is configured as Dependent Bean to DAO the DAO will not be injected  
with ServerJndiConnector object. It will be injected with  
ServerJndiConnector supplied DS (DataSource object) (as resultant  
object) as Dependent object.
- Java class acts Factory Bean only when it implements  
`org.springframework.beans.factory.FactoryBean<T> (I)` (java 8 interface)  
(up to spring 4 normal interface) (from spring 5 java 8 interface)
  - |--> `Object getObject ()` (Abstract method)  
(place logic to gather/create resultant object)
  - |--> `Class<?> getObjectType ()` (Abstract method)  
(place logic return object class `java.lang.Class` having  
Resultant object class name)
  - |--> `default boolean isSingleton()` (default method)  
(specifies whether the resultant object  
is singleton scope bean or not)?  
True --> singleton (scope)  
False --> prototype (scope)  
(default implementation of this method returns true always)

`java.util.Date` class most of constructors and methods  
are deprecated. So, do not use that class and prefer using the following

- a) `java.util.Calendar`
- b) `java.time.LocalDateTime` (from Java 8)
- c) `java.time.LocalDate`
- d) `java.time.LocalDateTime`



### Directory Structure of IOCProj51-FactoryBean-POC:



- Develop the above directory structure and package, class, XML file then use the following code with in their respective file.
- Copy paste build.gradle from any other gradle project because we are using same spring-context-support dependency only.

### DateFactoryBean.java

```
package com.nt.beans;

import java.time.LocalDate;

import org.springframework.beans.factory.FactoryBean;

public class DateFactoryBean implements FactoryBean<LocalDate> {
    private int year;
    private int month;
    private int dayOfMonth;

    public DateFactoryBean(int year, int month, int dayOfMonth) {
        this.year = year;
        this.month = month;
        this.dayOfMonth = dayOfMonth;
    }

    @Override
    public LocalDate getObject() throws Exception {
        return LocalDate.of(year, month, dayOfMonth);
    }

    @Override
    public Class<?> getObjectType() {
        return LocalDate.class;
    }

    @Override
    public boolean isSingleton() {
        return false;
    }
}
```

### ScheduleReminder.java

```
package com.nt.beans;

import java.time.LocalDate;

public class ScheduleReminder {

    private LocalDate date;

    public ScheduleReminder(LocalDate date) {
```

```

        this.date = date;
    }

public String checkReminder() {
    LocalDate sysDate = LocalDate.now();
    if (sysDate.isEqual(date))
        return "You have a borad meeting please read on time
and attend";
    else
        return "You dont have any remider today, relax and take
rest";
}
}

```

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

    <!-- Dependent class configuration -->
    <bean id="dfb" class="com.nt.beans.DateFactoryBean">
        <constructor-arg value="2020"/>
        <constructor-arg value="09"/>
        <constructor-arg value="20"/>
    </bean>
    <!-- Target bean configuration -->
    <bean id="reminder" class="com.nt.beans.ScheduleReminder">
        <constructor-arg ref="dfb"/></constructor-arg>
    </bean>
</beans>

```

### FactoryBeanTest.java

```

package com.nt.test;

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

```

```

import com.nt.beans.ScheduleReminder;

public class FactoryBeanTest {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        ScheduleReminder sReminder = null;
        //Create IOC container
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        //get SchedularReminder class object
        sReminder = ctx.getBean("reminder", ScheduleReminder.class);
        //invoke method
        System.out.println(sReminder.checkReminder());

        //close container
        ((AbstractApplicationContext) ctx).close();
    }

}

```

**Q. How Factory Bean class is giving Resultant object as spring bean?**

**Ans.** As Pre-instantiation or Lazy instantiation, first the Factory Bean class object will be created completes all injection process but before keeping Factory Bean class object in the internal cache of IoC container, it checks Bean is Normal Bean or Factory Bean (Based FactoryBean(I) is implemented or not), if it is FactoryBean then it will not keep current FactoryBean obj in the internal cache More ever it calls getObject () method on the current FactoryBean object to get Resultant object and keeps that Resultant object in the IOC container internal cache having the bean id of FactoryBean as shown below.

Internal cache of IoC container

dfb	LocalDate Object (Resultant object)
reminder	ScheduleReminder objet

## applicationContext.xml

```

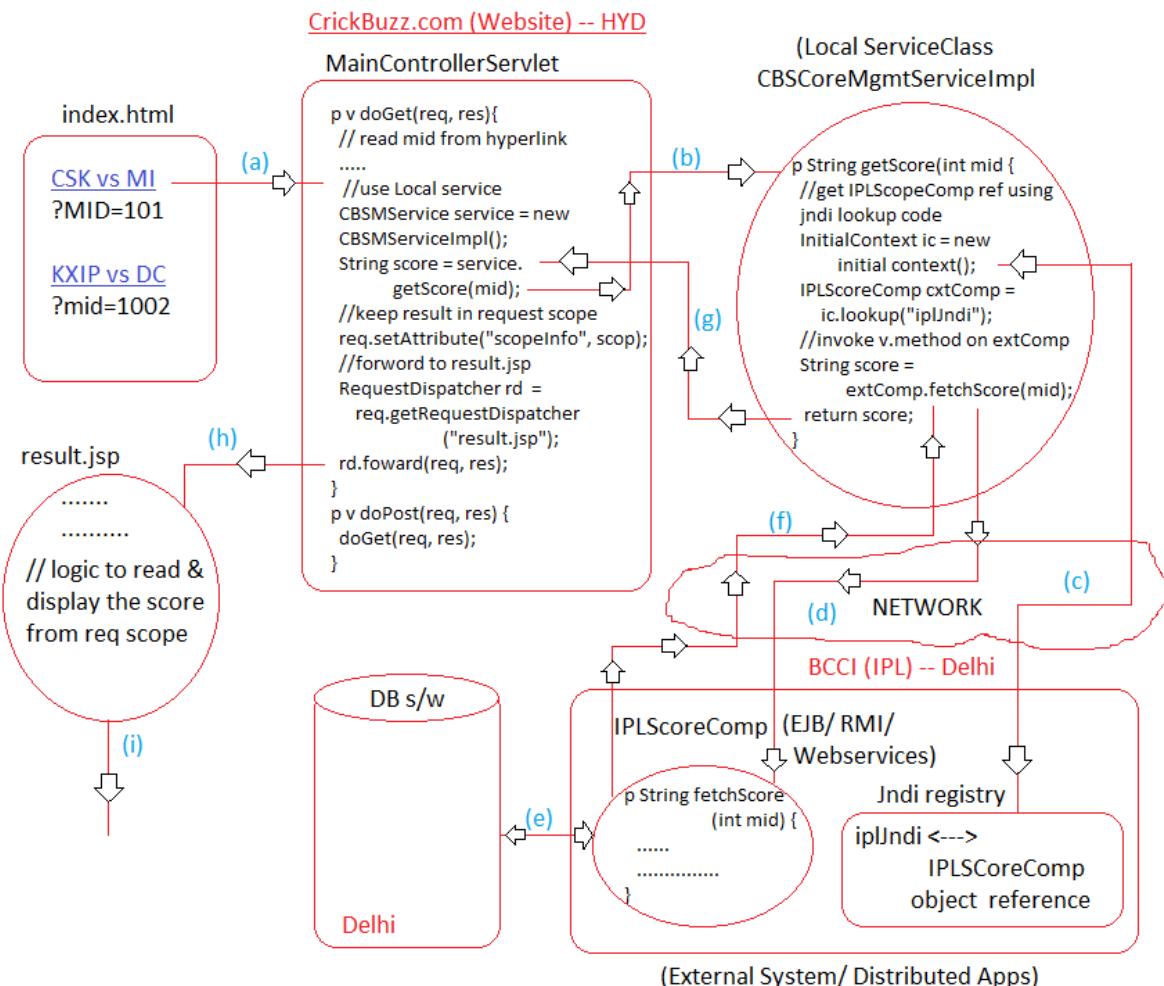
<!-- Dependent class configuration -->
<bean id="dfb" class="com.nt.beans.DateFactoryBean">
    <constructor-arg value="2020"/>
    <constructor-arg value="09"/>
    <constructor-arg value="20"/>
</bean>
<!-- Target bean configuration -->
<bean id="reminder" class="com.nt.beans.ScheduleReminder">
    <constructor-arg ref="dfb"/></constructor-arg>
</bean>

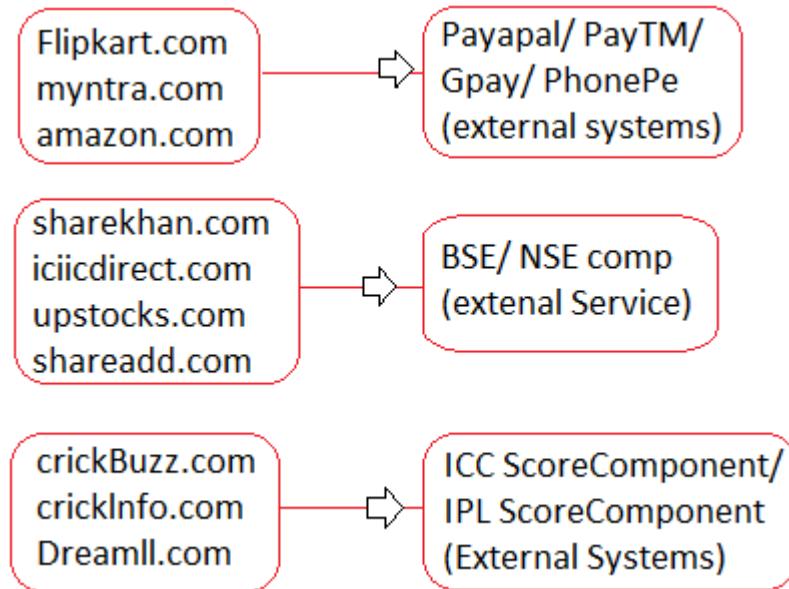
```

**Note:** IoC container creates FactoryBean class object to get call getObject() and other methods on it.. but it will not be kept in internal cache of IoC container.

## Service Locator (use case of Factory Bean)

Problem:





**Q. How CrickBuzz gets IPL match scores?**

**Ans.**

1. By keeping Person in Match Area sending score details (not recommended) (X).
2. IPL provides their DB username, password, JDBC, URL, driver class name details to CrickBuzz and CrickBuzz uses them get the score details (not recommended because CrickBuzz get unrestricted Access to DB) (X).
3. IPL develops Distributed App taking they're to DB to get the Score and keeps that distributed App reference in Jndi registry CrickBuzz gets IPLScoreComp reference (Distribute App) from Jndi registry and uses it to get the score (here IPLScoreComp will provide restricted access to DB), (Recommended - Shown in the diagram).

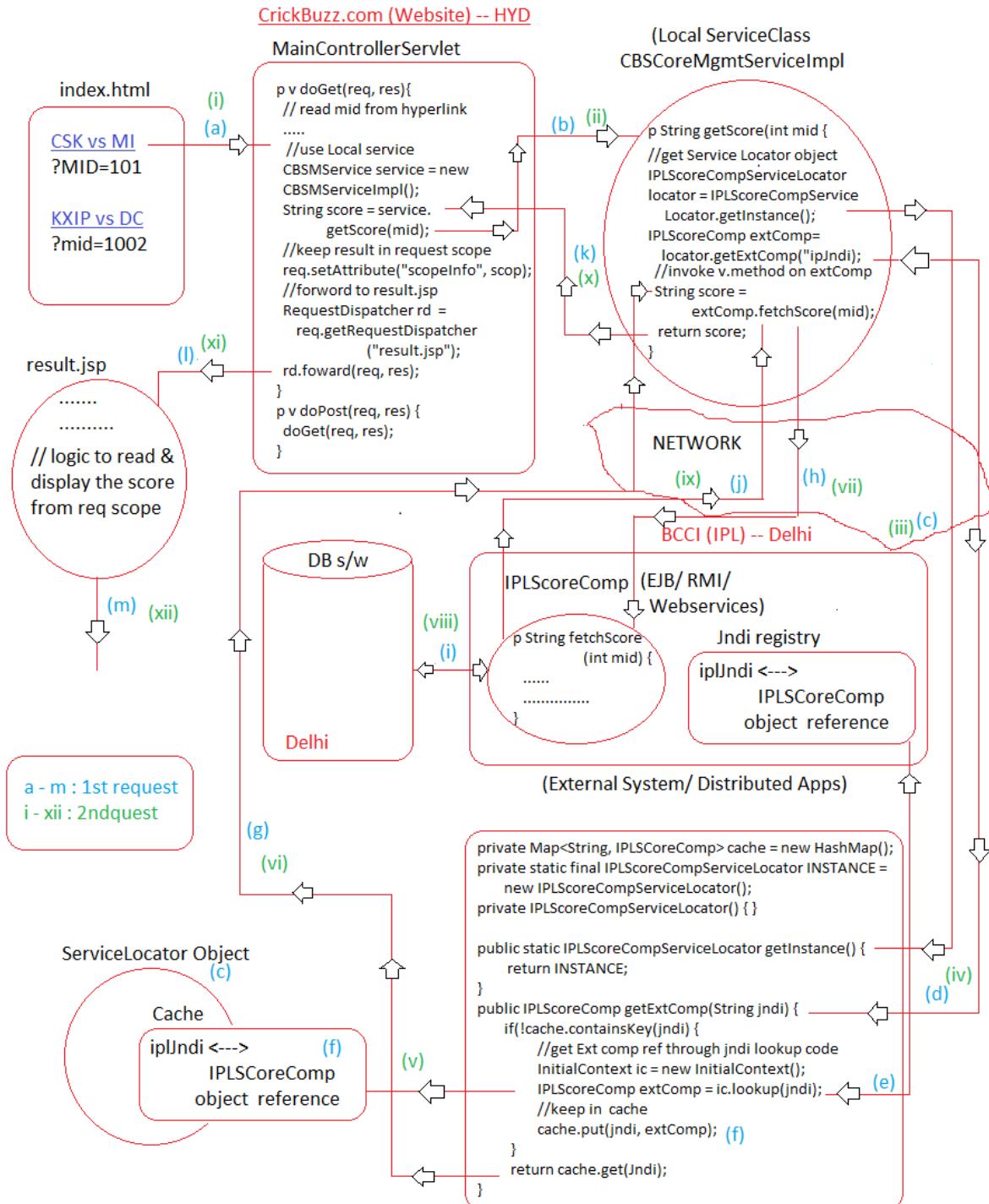
**Writing Jndi lookup code in the Local Service class Client app (like CrickBuzz is having the following limitations):**

- a. Jndi lookup code is not reusable across the multiple Service classes.
- b. If External Comp (IPLScoreComp) Technology or Location is changed we need to modify Jndi lookup code.
- c. If Jndi registry Technology or Location is changed we need to modify Jndi lookup code.
- d. Since Caching is not implemented, there more network round trips between Local Service classes (CrickBuzz) and Jndi registry to get same External Component reference for multiple times (poor performance).

**To overcome the above problems, take the support ServiceLocator Design Pattern. Which says place only Jndi lookup code that is required to get External**

component reference by talking with Jndi registry in a separate class also maintain cache having external component reference (This separate class is called ServiceLocator).

### Solution:



### Advantages of ServiceLocator Design Pattern:

- The Jndi lookup code to get External comp ref becomes reusable across the multiple local service classes.

- b. If external comp technology or location is modified, we just need to modify ServiceLocator code not the multiple local service classes code.
- c. If Jndi registry technology or location is modified, we just need to modify ServiceLocator code not the multiple local service classes code.
- d. Since the Service class Locator manages cache having component reference, the network round trips between Client App/ project and External component reference will be reduced drastically.

**Q. What is the difference Service class and ServiceLocator?**

**Ans.** Service class contains business logic where as ServiceLocator container Jndi lookup to get external component reference. In fact, Service class of client App/ client project uses ServiceLocator internally. We generally develop Service class as normal java class where we develop Service Locator as singleton class to maintain cache with external component reference.

**Note:**

- ✓ In spring environment Service class is normal Spring bean where as ServiceLocator is FactoryBean.
- ✓ If develop Client App/ Project as Spring based application, we need to take Local Service class as target and ServiceLocator class as dependent spring bean. But if we observe very carefully, Local Service class not interested in ServiceLocator object. It is actually, interested in the ServiceLocator supplied external component reference (resultant object). So, we need to take ServiceLocator as FactoryBean always.

**Advantages of taking ServiceLocator as Factory Bean class in Spring environment:**

- a. No need of developing it as Singleton java class just "singleton" scope is sufficient.
- b. No need to developing cache separately having external component reference because the Internal cache of IOC container itself maintains external component reference.

**Limitations:**

- a. ServiceLocator becomes invasive, because to make it factory bean we need to implement the Spring API supplied FactoryBean (I).

**Q. What is the difference between Local Service class and external component/ distributed App/ External service?**

**Ans.** Local Service is an ordinary java class having business logic of client App/

Client Project (like service class is Flipkart), whereas component/ service is a distributed app that can have different type remote or local client Apps accessing its services. components/ services will be developed by using Distributed Technologies like RMI, EJB, Webservices. (these are like PhonePe, GPay, PayPal and etc.).

**Q. If Servlet, JSP based MVC Web application is using Spring environment for developing Model layer service, DAO, ServiceLocator classes can u tell me where should we create IOC container that is required to Model Layer classes and how to get Local Service class object to the controller servlet?**

**Ans.** Create IOC container in the init () method ControllerServlet Comp by enabling <load-on-startup> on Controller Servlet close IOC container in the destroy () method and get Service class object to Controller Servlet in the Service (-, -)/ doGet(-, -)/ doPost(-, -) method by calling ctx.getBean(-, -) method.

[In the above setup, the Controller Servlet component pre-instantiation and Spring beans related pre-instantiation takes either during the deployment of web application or during the server startup]

**Note:** <load-on-startup> enabled on the servlet comp makes Servlet container to create Servlet class obj during the deployment of web application nothing but performs eager/ pre-instantiation of servlet component.

#### Procedure to develop Gradle based web application in Eclipse IDE:

**Step 1:** Create the Gradle Project convert it to web application

- To convert to web application  
RTC on Project --> Project facets --> Dynamic web module 4.0 --> Apply and Close.
- Add web.xml file manually (File menu new xml file name: web.xml) to WEB-INF

**Step 2:** Add the JARS/ dependencies to build.gradle by adding the "war" plugin.

- servlet-api<version>.jar
- spring-context-support<version>.jar
- lombok-project<version>.jar

**Step 3:** Add web.xml file Web Project.

RTC on WEB-INF --> new --> others --> xml file

**Step 4:** Create the packages in src/main/java folder.

**Step 5:** Develop the source code.

**Note:** It is always recommended to develop component having interface,

implementation class model. So, that interface can be given Local or remote client Projects to make them to receive and hold external component reference.

e.g. IPL People gives IPLScoreComp related interface to CrickBuzz client Project to make them to receive and hold IPLScoreComp reference.

#### Step 6: Configure Tomcat server with Eclipse IDE

window menu --> preferences --> server --> runtime environment --> add --> select apache Tomcat9 --> choose the tomcat installation folder (E:\Tomcat9.0) --> apply and ok.

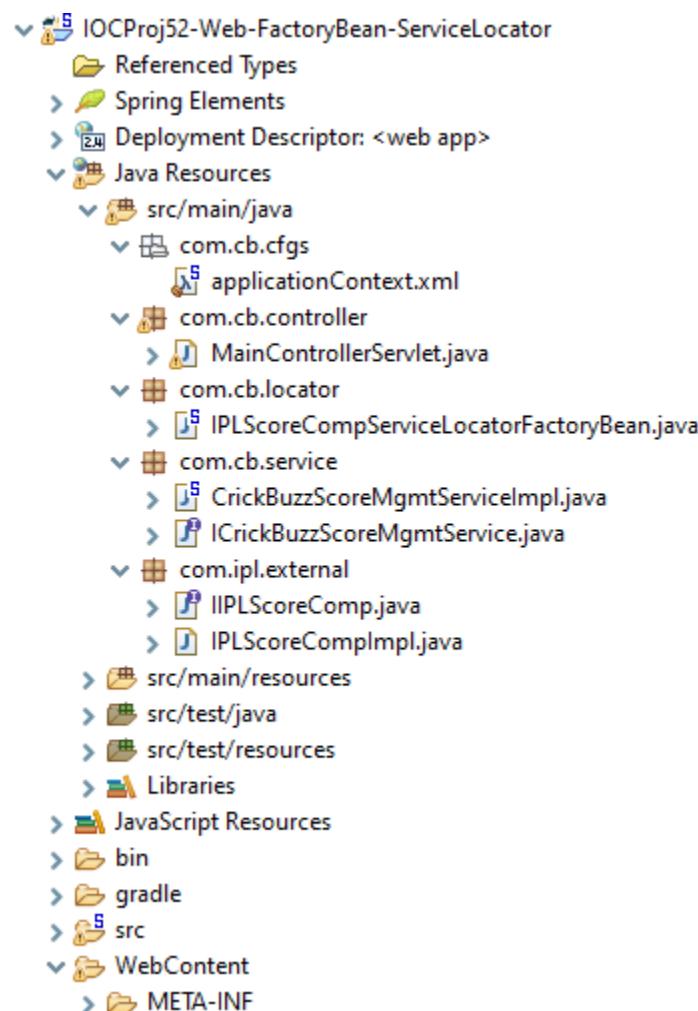
#### Step 7: Perform deployment Assembly operation on the web Project to move webcontent folder to deployment.

RTC on Project --> properties --> Deployment assembly --> add --> folder --> webcontent.

#### Step 8: Run the Application

RTC on Project --> Run as --> Run on server --> select tomcat server.

#### Directory Structure of IOCProj52-Web-FactoryBean-ServiceLocator:





- Develop the above directory structure and folder, package, class, XML, JSP, HTML file after convert to web application and add the jar dependencies in build.gradle file then use the following code with in their respective file.

### build.gradle

```
plugins {
    id 'war'
}
repositories {
    jcenter()
}
dependencies {
    // https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api
    implementation group: 'javax.servlet', name: 'javax.servlet-api',
    version: '4.0.1'
    // https://mvnrepository.com/artifact/org.springframework/spring-
    context-support
    implementation group: 'org.springframework', name: 'spring-context-
    support', version: '5.2.9.RELEASE'
}
```

### index.html

```
<h1 style="color: red; text-align: center;">ServiceLocator as
FactoryBean</h1>
<h1 style="color: cyan; text-align: center;">CrickBuzz...</h1>
<br>
<h1 style="color: blue; text-align: center;"><a
href="controller?mid=1001">CSK vs MI </a></h1>
<h1 style="color: blue; text-align: center;"><a
href="controller?mid=1002">DC vs KXIP </a></h1>
<h1 style="color: blue; text-align: center;"><a
href="controller?mid=1003">RCB vs SRH </a></h1>
```

### IPLScoreComp.java

```
package com.ipl.external;

public interface IPLScoreComp {
    public String getScore(int matchId);
}
```

### IPLScoreComImpl.java

```
package com.ipl.external;

public class IPLScoreComImpl implements IPLScoreComp {

    @Override
    public String getScore(int matchId) {
        if (matchId == 1001)
            return "CSK vs MI : MI-(162/2), CSK-(167/5) CSK won by 5
wickets";
        else if (matchId == 1002)
            return "DC vs KIXP : DC-(154/8), KIXP-(154/6) DC won by
through super over";
        else if (matchId == 1003)
            return "RCB vs SRH : RCB-(163/8), SRH-(153/10) RCB won
by 10 runs";
        else
            throw new IllegalArgumentException("Invalid match id");
    }
}
```

### ICrickBuzzScoreMgmtService.java

```
package com.cb.service;

public interface ICrickBuzzScoreMgmtService {

    public String fetchScore(int matchId);
}
```

### CrickBuzzScoreMgmtServiceImpl.java

```
package com.cb.service;

import com.ipl.external.IIPLScoreComp;

public class CrickBuzzScoreMgmtServiceImpl implements
ICrickBuzzScoreMgmtService {
    private IIPLScoreComp extComp;
    public CrickBuzzScoreMgmtServiceImpl(IIPLScoreComp extComp) {
        this.extComp = extComp;
    }
    @Override
    public String fetchScore(int matchId) {
        //use extComp
        return extComp.getScore(matchId);
    }
}
```

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

    <!-- ServiceLocator configuration -->
    <bean id="locator"
          class="com.cb.locator.IPLScoreCompServiceLocatorFactoryBean"/>
    <!-- Service configuration -->
    <bean id="cbService"
          class="com.cb.service.CrickBuzzScoreMgmtServiceImpl">
        <constructor-arg ref="locator"/>
    </bean>
</beans>
```

### MainControllerServlet.java

```
package com.cb.controller;

import java.io.IOException;
import javax.servlet.RequestDispatcher;
```

```

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AbstractApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

import com.cb.service.ICrickBuzzScoreMgmtService;

public class MainControllerServlet extends HttpServlet {
    private ApplicationContext ctx;
    @Override
    public void init() throws ServletException {
        ctx = new
ClassPathXmlApplicationContext("com/cb/cfgs/applicationContext.xml");
    }
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse
res) throws ServletException, IOException {
        int matchId = 0;
        ICrickBuzzScoreMgmtService service = null;
        String score = null;
        RequestDispatcher rd = null;
        //read additional request param value
        matchId = Integer.parseInt(req.getParameter("mid"));
        //get local service bean object
        service = ctx.getBean("cbService",
ICrickBuzzScoreMgmtService.class);
        //use local service
        score = service.fetchScore(matchId);
        //keep score in request scope
        req.setAttribute("scoreInfo", score);
        //forward to result.jsp
        rd = req.getRequestDispatcher("/result.jsp");
        rd.forward(req, res);
    }
    @Override

```

```

protected void doPost(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
    doGet(req, res);
}

@Override
public void destroy() {
    ((AbstractApplicationContext) ctx).close();
}
}

```

### IPLScoreCompServiceLocatorFactoryBean.java

```

package com.cb.locator;

import org.springframework.beans.factory.FactoryBean;

import com.ipl.external.IIPLScoreComp;
import com.ipl.external.IPLScoreComImpl;

public class IPLScoreCompServiceLocatorFactoryBean implements
FactoryBean<IIPLScoreComp> {

    private IIPLScoreComp extComp;
    public IPLScoreCompServiceLocatorFactoryBean() {
        this.extComp = new IPLScoreComImpl();
    }
    @Override
    public IIPLScoreComp getObject() throws Exception {
        /* Technically speaking here, we need add Jndi lookup
         * code to get external component reference from Jndi
         * registry, since we have external component (IPLScoreComp)
         * as an ordinary java class. we are just going to return
         * object for it without lookup operation.
        */
        return extComp;
    }
    @Override
    public Class<?> getObjectType() {
        return IIPLScoreComp.class;
    }
}

```

### web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    <servlet>
        <servlet-name>main</servlet-name>
        <servlet-
class>com.cb.controller.MainControllerServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>main</servlet-name>
        <url-pattern>/controller</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>
</web-app>
```

### result.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" isELIgnored="false"%>
<h1 style="color: red; text-align: center;">Result Page Score</h1>
<h2 style="color: cyan; text-align: center;">
    Score is : ${scoreInfo}
</h2>
<a href="index.html">home</a>
```

**Note:** org.sf.jndi.JndiObjectFactoryBean is the pre-defined FactoryBean that is given as ServiceLocator. It is very useful to get Resultant object from underlying Server Jndi registry based in Jndi name that we provide. (Especially useful to get DataSource object of Server Managed JDBC connection pool)

### applicationContext.xml

```
<bean id="jofb" class="pkg.JndiObjectFactoryBean">
    <property name="jndiName" value="DsJndi"/> | a
</bean>
<bean id="empDAO" class="pkg.EmployeeDAOImpl">
    <constructor-arg ref="jofb"/> | b
</bean>
```

- a. Get DataSource object from Jndi registry, if underlying Server based on the given Jndi name and make it as Spring bean having "jofb" as the bean id.
- b. Injects DataSource object to DAO class because "jofb" bean id is pointing to DataSource object.

**Note:** Since the Spring Bean is becoming FactoryBean because FactoryBean(I) implantation not because of any xml tag. So, there is not annotation for it.

#### Directory Structure of IOCAnnoProj11-Web-FactoryBean-ServletLocator:

- + Copy paste the IOCProj52-Web-FactoryBean-ServletLocator application and change rootProject.name to IOCAnnoProj11-Web-FactoryBean-ServletLocator in settings.gradle file.
- + Change the Web Project Settings to IOCAnnoProj11-Web-FactoryBean-ServletLocator.
- + Need Not to add anything same as MVCProj03-WishApp-AC-TwoContainer.
- + Add the following code in their respective files.

#### web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>
</web-app>
```

#### 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"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <context:component-scan base-package="com.cb"/>
</beans>
```

### MainController.java

```
@WebServlet(value="/controller", loadOnStartup = 1)
public class MainControllerServlet extends HttpServlet {
```

### IPLScoreCompServiceLocator.java

```
@Component("locator")
public class IPLScoreCompServiceLocatorFactoryBean implements
FactoryBean<IPLScoreComp> {
```

### CrickBuzzScoreMgmtServiceImpl.java

```
@Service("cbService")
public class CrickBuzzScoreMgmtServiceImpl implements
ICrickBuzzScoreMgmtService {

    @Autowired
    private IPLScoreComp extComp;
```

#### Limitation of Developing ServiceLocator as FactoryBean:

- Makes Service Locator as Invasive Spring bean because of implementing spring specific FactoryBean (I).
- Writing logics by implementing 3 methods is quite complex.
- The object created for FactoryBean class itself looks quite wasted object.

**Note:** To overcome the above problems, it is recommended to take ServiceLocator having static factory method bean Instantiation (For this no annotations are given we need to configure java class as Spring bean by keeping factory-method attribute in <bean> though spring bean is user-defined java class).

#### Directory Structure of IOCAnnoProj12-Web-StaticFactoryMethodBeanInstantiation-ServletLocator:

- + Copy paste the IOCAnnoProj11-Web-FactoryBean-ServletLocator application and change rootProject.name to IOCAnnoProj12-Web-StaticFactoryMethodBeanInstantiation-ServletLocator in settings.gradle file.
- + Change the Web Project Settings to IOCAnnoProj12-Web-StaticFactoryMethodBeanInstantiation-ServletLocator.

- ✚ Rename the IPLScoreCompServiceLocatorFactoryBean to IPLScoreCompServiceLocator.
- ✚ Add the following code in their respective files.

### 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"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <!-- Annotation Component scan -->
    <context:component-scan base-package="com.cb"/>

    <!-- Locator configuration -->
    <bean id="locator"
          class="com.cb.locator.IPLScoreCompServiceLocator" factory-
          method="getExternalComponent"/>

</beans>

```

### IPLScoreCompServiceLocator.java

```

package com.cb.locator;

import com.impl.external.IIPLScoreComp;
import com.impl.external.IPLScoreComImpl;

public class IPLScoreCompServiceLocator {

    final static IIPLScoreComp extComp = new IPLScoreComImpl();

    public static IIPLScoreComp getExternalComponent () {
        return extComp;
    }
}

```

In the following situations we need to go for xml configuration though Spring beans are user-defined classes:

- a. Inner beans
- b. Static/ instance factory method bean instantiation
- c. Collection injection
- d. Collection merging
- e. Bean Inheritance
- f. Method Replacer/Method Injection
- g. Bean Aliasing  
and etc.

Code in build.gradle to Move the WebContent folder to Deployment assembly automatically:

**Problem:** There is a bug in gradle web project when we go for gradle refresh the WebContent folder is removed from deployment assembly automatically to fix the bug and add the WebContent folder dynamically in deployment assembly we have to write the following lines of code in build.gradle.

### Solution 1

```
eclipse {  
    wtp {  
        component {  
            resource sourcePath: "/WebContent", deployPath: "/"  
        }  
    }  
}
```

### Solution 2

```
apply plugin: 'eclipse-wtp'  
webAppDirName='WebContent'
```

### Solution 3

```
plugins {  
    id 'eclipse-wtp'  
}  
  
webAppDirName='WebContent'
```

**Advantages of developing ServiceLocator by enabling static factory method bean instantiation:**

- a. Service Locator acts non-invasive.
- b. We can take service Locator as abstract class or Java8 interface i.e. not waste object will be created for ServiceLocator itself.
- c. Code becomes simple (complexity is less).
- d. There is no need of implementing Cache to hold external component reference because IoC container internal cache itself will work.

**Q. Can we configure interface as Spring Bean?**

**Ans.** Directly not possible, but we can make it possible in special situations.

- a. While Working with LMI (Lookup method), we can take target class as Java 8 Interface having business methods as default methods with.
- b. While working static factory method Bean instantiation, we can take Java 8 interface with static method as shown above.

**Note:** In the situations, we can also use abstract classes in the place Java 8 interfaces.

## BeanPostProcessor (BPP)

**Post Processing:**

- ⊕ The logic that executes after creating bean class object and after completing all injections are called post processing.
- ⊕ We can do each Spring bean object specific post processing by using customInit () or initializing Bean's afterPropertiesSet () or @PostConstruct method (Bean Life cycle init method).
- ⊕ If multiple Spring bean and their objects are looking for common processing logics then instead of writing inside every Spring bean class, we can write only for 1 time outside of all spring beans by taking the support of Bean Post Processor (BPP) as the class that implements org.springframework.beans.factory.config.BeanPostProcessor (I) [Java8 Interface].

1. default Object postProcessBeforeInitialization(Object bean,  
String beanName) throws BeansException  
This method will execute before Spring bean life cycle methods execution.

2. default Object postProcessAfterInitialization(Object bean,  
String beanName) throws BeansException  
This method will execute after Spring bean init life cycle methods execution.

### Order of execution:

- Spring Bean Instantiation (constructor Injection)
- Setter injection
- Aware Injection
- postProcessBeforeInitialization(-,-) (BPP)
- Spring bean init life cycle method(s)
- postProcessAfterInitialization(-,-) (BPP)
- And other operations like FactoryBean or LMI or Method Injection or static/ instance factory method Injection.

**Conclusion:** If Spring bean is not having init life cycle methods having Post Processing logic then we can place post processing logics in any one method of BPP class otherwise it is good to place in postProcessAfterInitialization(-,-) method.

CustomerBO  
EmployeeBO  
StudentBO

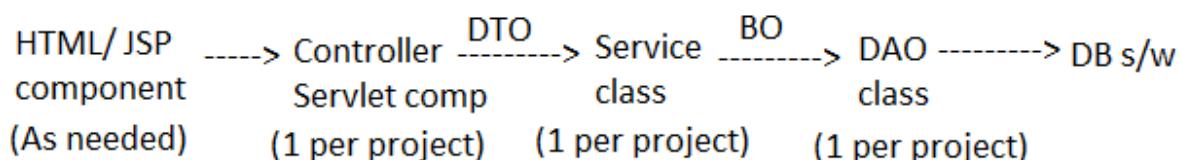
These having DOJ Property  
of java.util.Date (we want  
to have system date)

All the 3 classes configured  
as spring beans.

### Note:

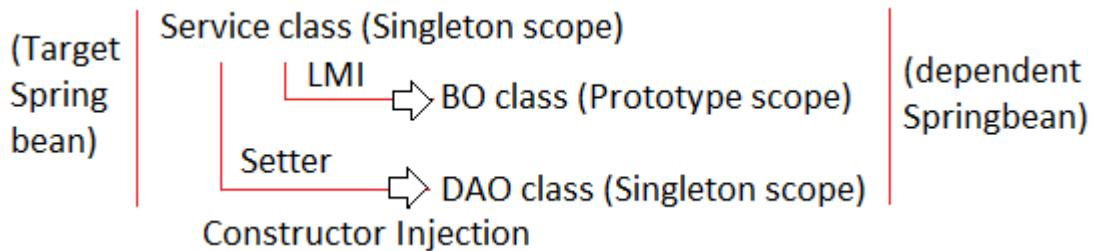
- ✓ If Spring bean init life cycle method to initialize DOJ with system date we need to write in every BO class separately (totally 3 times). By using BeanPostProcessor, the same thing can be done. Only for 1 time but it will be applied for all the 3 beans.
- ✓ @Required, @Autowired and etc. basic annotations perform their functionalities on spring beans by taking the support of BeanPostProcessor internally.
  - org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcess
  - org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcess

### Example Application using Lookup Method Injection (LMI) and BeanPostProcessor:



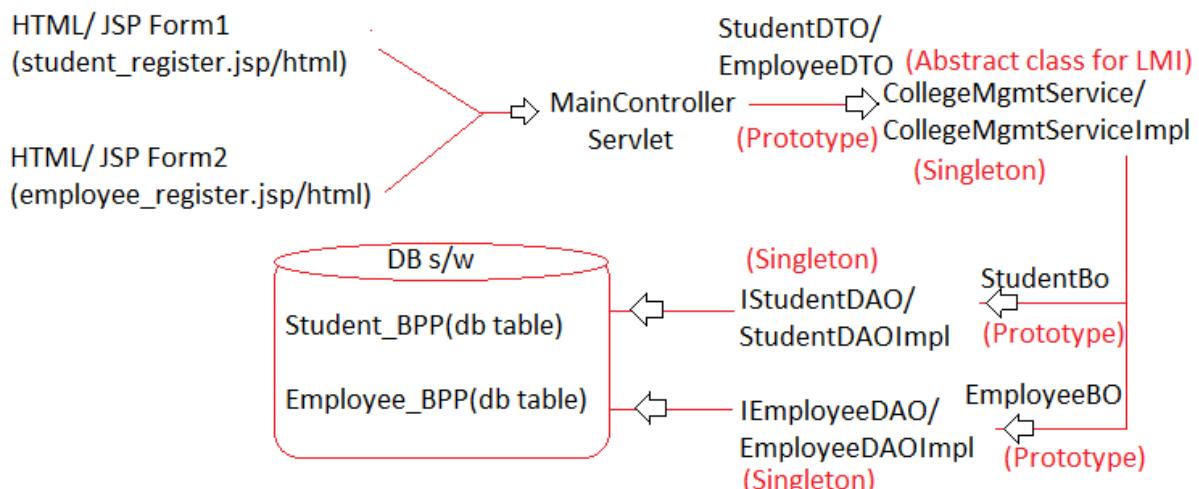
- ⊕ If want to take Service, DAO, BO classes as spring beans we generally

take Service, DAO classes as singleton scope Spring beans and BO class prototype scope Spring beans because BO class object should hold every request data separately. Here Service class is target having singleton scope and BO class dependent class having prototype scope. For this we need go for LMI. Another combination is Service class is target class and DAO class is Dependent class (both having singleton scope), For this we can go for Setter or constructor injection.



**Note:** We can apply BeanPostProcessor on java classes only when they are configured as spring beans here BO classes taken as spring beans to apply BeanPostProcessor on them.

- ⊕ DTO is required in Servlet component which cannot be configured as Spring Bean. So, Servlet component can use one of the following two approaches to get objects of DTO.
  - Create DTO class object by new operator in service (-, -)/ doXxx(-,-) methods on 1 per request basis
  - Configure DTO class as prototype scope Spring bean and get it service (-, -)/ doXxx(-,-) method by calling ctx.getBean(-,-) method. (Traditional Dependency lookup) (Best)



There are configured as Spring bean having Prototype scope	StudentDTO EmployeeDTO	doj (Common property)
	StudentBO EmployeeBO	doj (Common property)

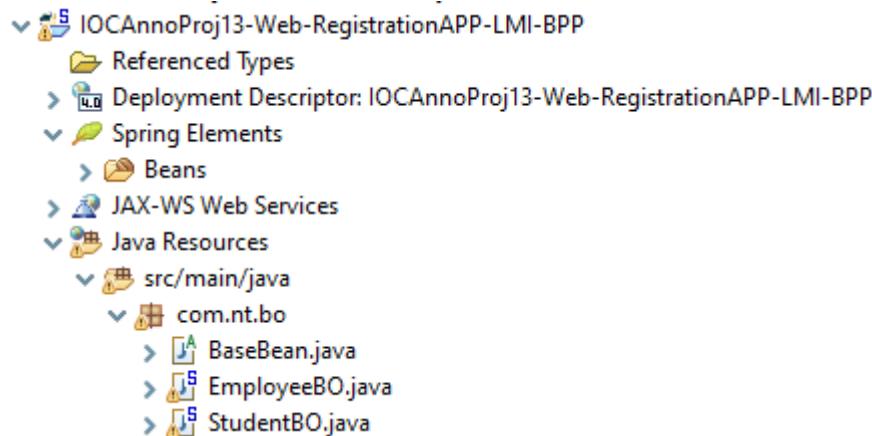
**Note:**

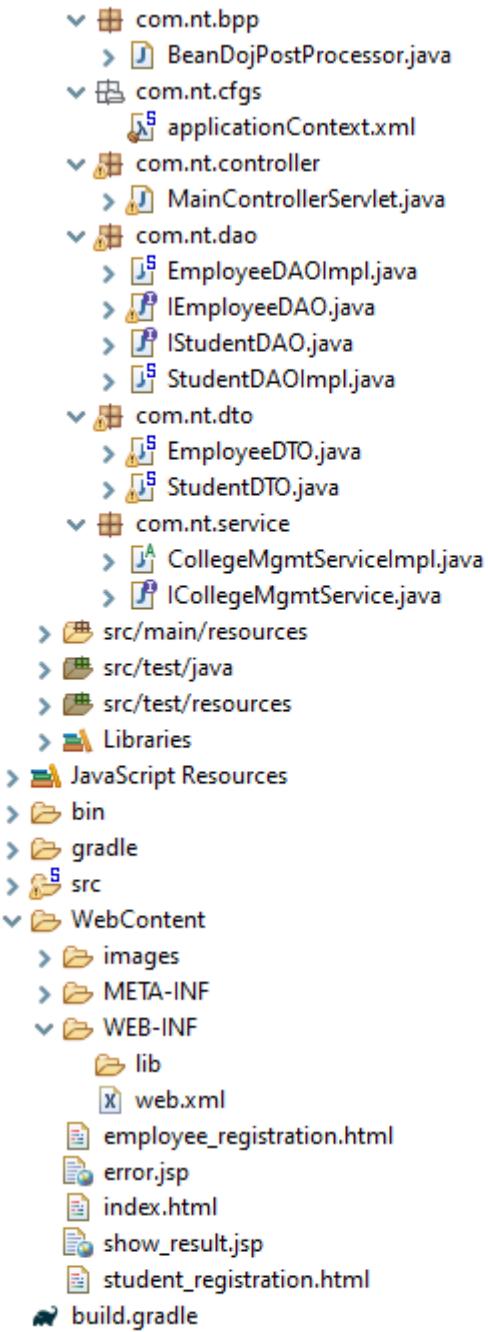
- ✓ doj will be initialized dynamically using BPP. i.e. it will not be collected.
- ✓ Instead of initializing all DTOs doj property and all BOs doj property with system date separately in every DTO or BO class, Better to Initialize only time by taking the support of single BeanPostProcessor.
- ✓ Controller Servlet gets DTO classes objects by using traditional dependency lookup.
- ✓ Service class gets BO classes objects by using LMI.
- ✓ Service class gets DAO classes objects by setter/ constructor injection.

**BeanPostProcessor (I):**

- ⊕ The BeanPostProcessor class executes for every object created for the spring bean class. If the Spring bean scope is singleton then it executes only for 1 time, if the Spring bean scope is prototype then it executes for every object created for that spring bean.
- ⊕ Ordinary Interface up to Spring 4.x.
- ⊕ Java 8 Interface with default methods from spring 5.x.  
`org.springframework.beans.factory.config.BeanPostProcessor (I)`
  - `default Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException`
  - `Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException`

**Directory Structure of IOCAnnoProj13-RegistrationAPP-LMI-BPP:**





- Develop the above directory structure and package, class, XML, JSP, HTML file then use the following code with in their respective file.
- Copy paste build.gradle from the previous project because we are using same jars.

### error.jsp

```
<h2 class="h1">Registration Form</h2>
<h2>Internal problem please! try again</h2>
<a href="index.html" style="color: gold">Home</a>
```

## index.html

```
<table>
  <tr>
    <td><a href="student_registration.html"
      style="color: gold">Student registration</a></td>
  </tr>
  <tr>
    <td><a href="employee_registration.html"
      style="color: gold">Employee registration</a></td>
  </tr>
</table>
```

## Show\_result.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1" isELIgnored="false"%>
<h2 class="h1">Registration Form</h2>
<table>
  <tr>
    <td>${resultMessage}</td>
  </tr>
  <tr>
    <td><a href="index.html" style="color:
gold">Home</a></td>
  </tr>
</table>
```

## web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" version="4.0">
  <display-name>IOCAnnoProj13-Web-RegistrationAPP-LMI-BPP</display-
  name>
  <servlet>
    <description>
    </description>
    <display-name>MainControllerServlet</display-name>
    <servlet-name>MainControllerServlet</servlet-name>
```

```
<servlet-class>com.nt.controller.MainControllerServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>MainControllerServlet</servlet-name>
    <url-pattern>/controller</url-pattern>
</servlet-mapping>
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>
</web-app>
```

### employee\_registration.html

```
<form action="controller" method="get">
    <table>
        <tr>
            <td>Employee Id </td>
            <td><input type="text" name="eid" placeholder="3567"></td>
        </tr>
        <tr>
            <td>Employee Name : </td>
            <td><input type="text" name="ename" placeholder="Nirmal  
Kumar Sahu"></td>
        </tr>
        <tr>
            <td>Company : </td>
            <td><input type="text" name="ecompany"  
placeholder="hcl"></td>
        </tr>
        <tr>
            <td>Salary : </td>
            <td><input type="text" name="esalary"  
placeholder="300000"></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" name="type" value="Employee  
Registered">
            </td>
        </tr></table> </form>
```

## student\_registration.html

```
<h2 class="h1">Student Registration Form</h2>
<form action="controller" method="get">
    <table>
        <tr>
            <td>Student Id </td>
            <td><input type="text" name="sid" placeholder="3567"></td>
        </tr>
        <tr>
            <td>Student Name : </td>
            <td><input type="text" name="sname" placeholder="Nirmal Kumar Sahu"></td>
        </tr>
        <tr>
            <td>Address : </td>
            <td><textarea name="saddress" rows="3" cols="21" placeholder="193 house hyderabad"></textarea></td>
        </tr>
        <tr>
            <td>Course : </td>
            <td><input type="text" name="scourse" placeholder="Java"></td>
        </tr>
        <tr>
            <td>Mark 1 : </td>
            <td><input type="text" name="sm1" placeholder="30"></td>
        </tr>
        <tr>
            <td>Mark 2 : </td>
            <td><input type="text" name="sm2" placeholder="30"></td>
        </tr>
        <tr>
            <td>Mark 3 : </td>
            <td><input type="text" name="sm3" placeholder="30"></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" name="type" value="Student Registered">
            </td>
        </tr>
    </table></form>
```

### BaseBean.java

```
package com.nt.bo;

import java.time.LocalDateTime;

import lombok.Data;

@Data
public abstract class BaseBean {
    private int id;
    private String name;
    private LocalDateTime doj;
}
```

### BaseBean.java

```
package com.nt.bo;

import java.time.LocalDateTime;

import lombok.Data;

@Data
public abstract class BaseBean {
    private int id;
    private String name;
    private LocalDateTime doj;
}
```

### EmployeeBO.java

```
package com.nt.bo;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

import lombok.Data;

@Data
@Component("empBO")
@Scope("prototype")
```

```
public class EmployeeBO extends BaseBean {  
    private String company;  
    private float salary;  
    private float grossSalary;  
    private float netSalary;  
}
```

### StudentBO.java

```
package com.nt.bo;  
  
import org.springframework.context.annotation.Scope;  
import org.springframework.stereotype.Component;  
  
import lombok.Data;  
  
@Data  
@Component("stuBO")  
@Scope("prototype")  
public class StudentBO extends BaseBean {  
    private String saddr;  
    private String course;  
    private int total;  
    private float avg;  
}
```

### EmployeeDTO.java

```
package com.nt.dto;  
  
import org.springframework.context.annotation.Scope;  
import org.springframework.stereotype.Component;  
  
import com.nt.bo.BaseBean;  
  
import lombok.Data;  
  
@Data  
@Component("empDTO")  
@Scope("prototype")  
public class EmployeeDTO extends BaseBean {  
    private String company;  
    private float salary;  
}
```

### StudentDTO.java

```
package com.nt.dto;

import java.io.Serializable;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

import com.nt.bo.BaseBean;

import lombok.Data;

@Data
@Component("stuDTO")
@Scope("prototype")
public class StudentDTO extends BaseBean implements Serializable {
    private String saddr;
    private String course;
    private int m1, m2, m3;
}
```

### BeanDojPostProcessorDTO.java

```
package com.nt.bpp;

import java.time.LocalDateTime;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

import com.nt.bo.BaseBean;

public class BeanDojPostProcessor implements BeanPostProcessor {

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

    @Override
    public Object postProcessBeforeInitialization(Object bean, String
beanName) throws BeansException {
        System.out.println("BeanDojPostProcessor : 
postProcessBeforeInitialization()");
    }
}
```

```

        if (bean instanceof BaseBean) {
            ((BaseBean) bean).setDoj(LocalDateTime.now());
        }
        return bean;
    }

}

```

### 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"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <context:component-scan base-package="com.nt"/>

</beans>

```

### 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"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <context:component-scan base-package="com.nt"/>

</beans>

```

### MainControllerServlet.java

```

package com.nt.controller;

import java.io.IOException;

```

```

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

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

import com.nt.dto.EmployeeDTO;
import com.nt.dto.StudentDTO;
import com.nt.service.ICollectionMgmtService;

public class MainControllerServlet extends HttpServlet {

    private ApplicationContext ctx = null;
    public void init() {
        //Create container
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
    }

    public void destroy() {
        ((AbstractApplicationContext) ctx).close();
    }

    public void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException {
        String type = null;
        StudentDTO stuDTO = null;
        EmployeeDTO empDTO = null;
        ICollectionMgmtService service = null;
        String result = null, targetPage = null;
        RequestDispatcher rd = null;
        //get service class object
        service = ctx.getBean("clgService", ICollectionMgmtService.class);
        //Evaluate which form is submitting request to different the
logics
        type = req.getParameter("type");
        if (type.equalsIgnoreCase("Student Registered")) {

```

```

//get StudentDTO class object
stuDTO = ctx.getBean("stuDTO", StudentDTO.class);
//read form data and store to DTO object
stuDTO.setId(Integer.parseInt(req.getParameter("sid")));
stuDTO.setName(req.getParameter("sname"));
stuDTO.setSadd(req.getParameter("saddress"));
stuDTO.setCourse(req.getParameter("scourse"));
stuDTO.setM1(Integer.parseInt(req.getParameter("sm1")));
stuDTO.setM2(Integer.parseInt(req.getParameter("sm2")));
stuDTO.setM3(Integer.parseInt(req.getParameter("sm3")));
try {
    //use service
    result = service.enrollStudent(stuDTO);
    //keep result in request scope
    req.setAttribute("resultMessage", result);
    targetPage = "/show_result.jsp";
} catch (Exception e) {
    targetPage = "/error.jsp";
    e.printStackTrace();
}
}
else {
    //get EmployeeDTO class object
    empDTO = ctx.getBean("empDTO", EmployeeDTO.class);
    //read form data and store to DTO object
    empDTO.setId(Integer.parseInt(req.getParameter("eid")));
    empDTO.setName(req.getParameter("ename"));
    empDTO.setCompany(req.getParameter("ecompany"));
    empDTO.setSalary(Float.parseFloat(req.getParameter("esalary")));
    try {
        //use service
        result = service.enrollEmployee(empDTO);
        //keep result in request scope
        req.setAttribute("resultMessage", result);
        targetPage = "/show_result.jsp";
    } catch (Exception e) {
        targetPage = "/error.jsp";
        e.printStackTrace();
    }
}

```

```

        //get Dispatcher
        rd = req.getRequestDispatcher(targetPage);
        //forward the request
        rd.forward(req, res);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException {
    doGet(req, res);
}

}

```

#### IEmployeeDAO.java

```

package com.nt.dao;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Repository;

import com.nt.bo.EmployeeBO;

@Repository("stuDAO")
@Scope("singleton")
public interface IEmployeeDAO {
    public int registerEmployee(EmployeeBO bo) throws Exception;
}

```

#### IStrudentDAO.java

```

package com.nt.dao;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Repository;

import com.nt.bo.StudentBO;

@Repository("stuDAO")
@Scope("singleton")
public interface IStudentDAO {
    public int registerStudent(StudentBO bo) throws Exception;
}

```

### EmployeeDAOImpl.java

```
package com.nt.dao;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Repository;

import com.nt.bo.EmployeeBO;

@Repository("empDAO")
@Scope("singleton")
public class EmployeeDAOImpl implements IEmployeeDAO {

    @Override
    public int registerEmployee(EmployeeBO bo) throws Exception {
        System.out.println("Inserting data to Employee DB table having
data : "+bo);
        return 1;
    }

}
```

### StudentDAOImpl.java

```
package com.nt.dao;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Repository;

import com.nt.bo.StudentBO;

@Repository("stuDAO")
@Scope("singleton")
public class StudentDAOImpl implements IStudentDAO {

    @Override
    public int registerStudent(StudentBO bo) throws Exception {
        System.out.println("Inserting data to Student DB table having
data : "+ bo);
        return 1;
    }

}
```

### ICollegeMgmtService.java

```
package com.nt.service;

import com.nt.dto.EmployeeDTO;
import com.nt.dto.StudentDTO;

public interface ICollegeMgmtService {

    public String enrollStudent(StudentDTO dto) throws Exception;
    public String enrollEmployee(EmployeeDTO dto) throws Exception;

}
```

### CollegeMgmtServiceImpl.java

```
package com.nt.service;

import org.springframework.beans.BeanUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Lookup;
import org.springframework.stereotype.Service;

import com.nt.bo.EmployeeBO;
import com.nt.bo.StudentBO;
import com.nt.dao.IEmployeeDAO;
import com.nt.dao.IStudentDAO;
import com.nt.dto.EmployeeDTO;
import com.nt.dto.StudentDTO;

@Service("clgService")
public abstract class CollegeMgmtServiceImpl implements
ICollegeMgmtService {

    @Autowired
    private IStudentDAO stuDAO;

    @Autowired
    private IEmployeeDAO empDAO;

    @Lookup
    public abstract StudentBO getStudentBO();

    @Lookup
    public abstract EmployeeBO getEmployeeBO();
```

```

@Override
public String enrollStudent(StudentDTO dto) throws Exception {
    int total=0;
    float avg=0.0f;
    StudentBO bo = null;
    int count=0;
    //calculate total and avg
    total = dto.getM1()+dto.getM2()+dto.getM3();
    avg = total/3;
    //get StudentBO Object
    bo = getStudentBO();
    //Copy DTO object to BO
    BeanUtils.copyProperties(dto, bo);
    bo.setTotal(total);
    bo.setAvg(avg);
    //use DAO
    count = stuDAO.registerStudent(bo);
    return (count==0)?"Student is not registered":"Student is
registered";
}
@Override
public String enrollEmployee(EmployeeDTO dto) throws Exception {
    float grossSalary=0.0f, netSalary;
    EmployeeBO bo = null;
    int count=0;
    //calculate gross and net salary
    grossSalary=(float) ((dto.getSalary()+(dto.getSalary()*0.4)));
    netSalary= (float) ((grossSalary - (grossSalary*0.2)));
    //get StudentBO Object
    bo = getEmployeeBO();
    //Copy DTO object to BO
    BeanUtils.copyProperties(dto, bo);
    bo.setGrossSalary(grossSalary);
    bo.setNetSalary(netSalary);
    //use DAO
    count = empDAO.registerEmployee(bo);
    return (count==0)?"Employee is not registered":"Employee is
registered";
}

```

**Note:**

- ✓ If Spring is having init life cycle method(s) then it is better to write BPP logics in postProcessAfterInitialization(-,-) method otherwise we can write in any method.
- ✓ instanceof is a java operator to check whether given object reference type is certain class or not.
- ✓ BeanPostProcessor class object will be created during IoC container creation automatically based on its BeanPostProcessor(I) implementation irrespective of its scope (Same thing is applicable for BeanFactoryPostProcessor and Event Listeners).

## BeanFactoryPostProcessor

- + After creating InMemory Metadata of Spring bean configuration file and before performing pre-instantiation of singleton scope beans, if we want execute some logic to change/ set data in InMemory Metadata of Spring bean configuration file we need go for BeanFactoryPostProcessor.

**Note:**

- ✓ PropertyPlaceHolderConfigurer or <context: property-placeholder> are internally BeanFactoryPostProcessor to recognize place holders \${....} in the InMemory Metadata of Spring bean configuration file and to replace them with the data collected from the properties file or system properties or environment variables.
- ✓ While working with @Value annotation the placeholder recognized and replaced by using the support of PropertyPlaceHolderConfigurer/ Support class (BeanFactoryPostProcessor).

```
@Value("${<key>}")  
private int age;
```
- ✓ In ApplicationContext container BeanFactoryPostProcessor will be recognized and applied automatically once it configured as Spring bean. In BeanFactory Container we must register it explicitly.

**Q. Can we use Properties file and place holder while working with BeanFactory IoC container?**

**Ans.** Directly not possible because BeanFactory Container does not register/ recognize PropertyPlaceHolderConfigurer/ Support as BeanFactoryPostProcessor directly. We should go for explicit registration.

### Example Application:

- a. Keep any application ready that using properties file and place holders \${...}.
- b. Configure PropertyPlaceHolderConfigurer in the Spring bean configuration file specifying the name and location of properties file(s).
- c. Take BeanFactory container in client App and explicitly register Container with PropertyPlaceHolderConfigurer.

### Directory Structure of IOCProj53-PropertiesFile-BeanFactoryContainer-BFPP:

- ⊕ Copy paste the IOCProj34-RealTimeDI-PropertiesFile application and change rootProject.name to IOCProj53-PropertiesFile-BeanFactoryContainer-BFPP in settings.gradle file.
- ⊕ Need not to add any new file same structure as IOCProj34-RealTimeDI-PropertiesFile.
- ⊕ Add the following code in their respective files.

#### CollegeMgmtServiceImpl.java

```
public class RealTimeDITest {

    public static void main(String[] args) {
        Scanner sc = null;
        String name=null, address=null, Amount=null, time=null,
rate=null;
        CustomerVO vo = null;
        DefaultListableBeanFactory factory = null;
        XmlBeanDefinitionReader reader = null;
        PropertyPlaceholderConfigurer ppvc = null;
        MainController controller = null;
        String result = null;
        //Read inputs from end-user using scanner
        sc = new Scanner(System.in);
        System.out.println("Enter the following Details for registration:");
        System.out.print("Enter Customer Name : ");
        name = sc.next();
        System.out.print("Enter Customer Address : ");
        address = sc.next();
        System.out.print("Enter Customer Principle Amount : ");
        Amount = sc.next();
        System.out.print("Enter Customer Time : ");
        ..
        ..
    }
}
```

```

time = sc.next();
System.out.print("Enter Customer Rate of Interest: ");
rate = sc.next();
//Store into VO class object
vo = new CustomerVO();
vo.setCname(name);
vo.setCadd(address);
vo.setpAmt(Amount);
vo.setTime(time);
vo.setRate(rate);
//Create BeanFactory [IoC] container
factory = new DefaultListableBeanFactory();
reader = new XmlBeanDefinitionReader(factory);

reader.loadBeanDefinitions("com/nt/cfgs/applicationContext.xml");
//get PropertyPlaceHolderConfigure
pphc = factory.getBean("pphc",
PropertyPlaceholderConfigurer.class);
pphc.postProcessBeanFactory(factory);
//get controller class object
controller = factory.getBean("controller", MainController.class);
//invoke methods
try {
    result = controller.processCustomer(vo);
    System.out.println(result);
} catch (Exception e) {
    System.out.println("Internal probelm :
"+e.getMessage());
    e.printStackTrace();
}
} //main

} //class

```

## PropertyEditor

- It is useful to convert configured values to as required for the bean properties i.e. talks about auto conversion of values.  
e.g. <property name="age" value= "30">  
<property name="avg" value="45.66f"/>

- IoC container internally uses PropertyEditor concept to convert XML file/ properties file supplied String inputs to appropriate type as required for bean properties.
- Multiple Built-in PropertyEditor are already registered with both IOC containers (BF, AC Containers).
- Every PropertyEditor is a java class that implements `java.beans.PropertyEditor` (I) directly or indirectly.

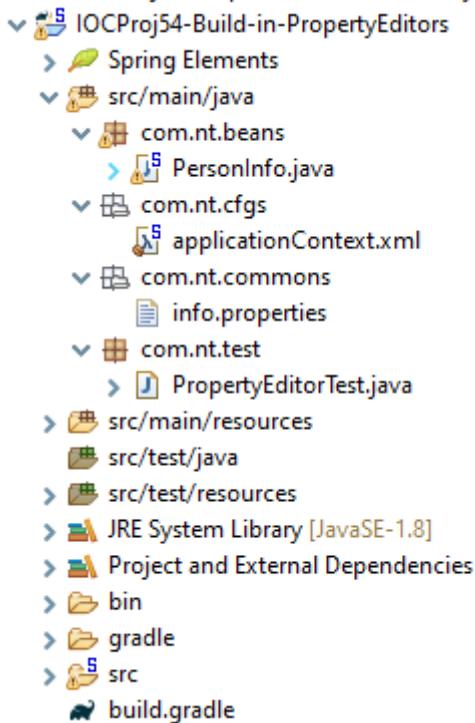
#### List of class `PropertyEditor`:

- `ByteArrayPropertyEditor`
- `CharacterEditor`
- `CharArrayPropertyEditor`
- `CharsetEditor`
- `ClassArrayEditor`
- `ClassEditor`
- `CurrencyEditor`
- `CustomBooleanEditor`
- `CustomCollectionEditor`
- `CustomDateEditor`
- `CustomMapEditor`
- `CustomNumberEditor`
- `FileEditor`
- `InputSourceEditor`
- `InputStreamEditor`
- `LocaleEditor`
- `PathEditor`
- `PatternEditor`
- `PropertiesEditor`
- `ReaderEditor`
- `ResourceBundleEditor`
- `StringArrayPropertyEditor`
- `StringTrimmerEditor`
- `TimeZoneEditor`
- `URIEditor`
- `URLEditor`
- `UUIDEditor`
- `ZoneIdEditor`

Built-in `PropertyEditor` belonging to `org.sf.beans.propertyeditor` package.

**Note:** We can develop and register Custom PropertyEditor with IoC containers.

### Directory Structure of IOCProj54-Build-in-PropertyEditors:



- Develop the above directory structure and package, class, XML file then use the following code with in their respective file.
- Copy paste build.gradle from any other gradle project because we are using same spring-context-support and lombok API dependency.

#### PersonInfo.java

```
package com.nt.beans;

import java.io.File;
import java.net.URL;
import java.util.Currency;
import java.util.Date;
import java.util.Locale;
import java.util.Properties;
import java.util.TimeZone;

import lombok.Setter;
import lombok.ToString;

@Setter
@ToString
```

```

public class PersonInfo {
    private Long aadharNo;
    private String pname;
    private String[] addresses;
    private float salary;
    private File photoPath;
    private Currency countryCurrency;
    private Date dob;
    private Class javaClass;
    private Class[] javaClasses;
    private URL fbUrl;
    private Locale currentLocale;
    private TimeZone timeZone;
    private Properties props;
}

```

### 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-4.3.xsd">
    <bean id="pinfo" class="com.nt.beans.PersonInfo">
        <property name="aadharNo" value="4577567567"/>
        <property name="pname" value="Nimu"/>
        <property name="addresses" value="hyd, odisha"/>
        <property name="salary" value="64833"/>
        <property name="photoPath" value="E://files//photo.jpg"/>
        <property name="countryCurrency" value="INR"/>
        <property name="dob" value="11/23/1990"/>
        <property name="javaClass" value="java.lang.System"/>
        <property name="javaClasses" value="java.lang.System,
java.lang.String"/>
        <property name="fbUrl"
value="http://facebook.com/?userid=raja"/>
        <property name="currentLocale" value="hi-IN"/>
        <property name="timeZone" value="Asia/Calcutta"/>
        <property name="props" value="name=raja, age=30,
address=hyd"/>
    </bean>
</beans>

```

## PropertyEditorTest.java

```
package com.nt.test;

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

import com.nt.beans.PersonInfo;

public class PropertyEditorTest {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        PersonInfo info = null;
        //Create IoC container
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        //get Bean
        info = ctx.getBean("pinfo", PersonInfo.class);
        System.out.println(info);
        //close container
        ((AbstractApplicationContext) ctx).close();
    }
}
```

## Custom PropertyEditor

- We can develop Java class Custom PropertyEditor by making that class implementing java.beans.PropertyEditor (I).
- java.beans.PropertyEditorSupport (c) is the implementation class of PropertyEditor(I) having null method definitions. So, we create Custom PropertyEditor extending PropertyEditorSupport and we can override only those methods in which we are interested in.

e.g.

```
java class implementing Servlet(I)
    |--> Should implement 5 methods
java class extending from GenericServlet
    |--> should implement only 1 method
java class extending from HttpServlet
    |--> Can override its choice methods
```

Custom PropertyEditor class implementing java.beans.PropertyEditor (I)  
|--> should implement all the 12 methods.  
Custom PropertyEditor class extending from java.beans.PropertyEditorSupport  
(c)  
|--> Can override only required methods.

```
java.beans.PropertyEditor(I)
    |implements
    |
java.beans.PropertyEditorSupport (c)
    |-> setAsText(String text) most imp method to override
```

[Example showing need of Custom PropertyEditor:](#)

#### Dependent class

```
public class LoanAmtDetails{
    private float pAmt;
    private float rate;
    private float time;
    //setters methods & getter methods
    .....
}
```

#### target class

```
public class LoanIntrestAmtCalculator{
    private LoanAmtDetails details;
    public void setDetails(LoanAmtDetails details){
        this.details=details;
    }
    public float calIntrestAmt(){
        return (details.getPAmt()* details.getRate()*details.getTime());
    }
}
```

#### applicationContext.xml (Actual code we should write)

```
<beans ..... >
    <bean id="laDetails" class="pkg.LoanAmtDetails">
        <property name="pAmt" value="10000"/>
        <property name="rate" value="2"/>
        <property name="time" value="10"/>
    </bean>
```

```

<bean id="laiCalculator" class="pkg.LoanAmtIntrestCalculator">
    <property name="details" ref="laDetails"/>
</bean>
</beans>

```

### [applicationContext.xml \(we want like this\)](#)

```

<beans .... >
    <bean id="laiCalculator" class="pkg. LoanAmtIntrestCalculator ">
        <property name="details" value="10000, 2, 10"/>
    </bean>
</beans>

```

Since "details" is Object type bean property we cannot inject simple/ String value to it. To make it possible we need to develop and configure Custom PropertyEditor to convert comma separated String values to LoanAmtDetails object.

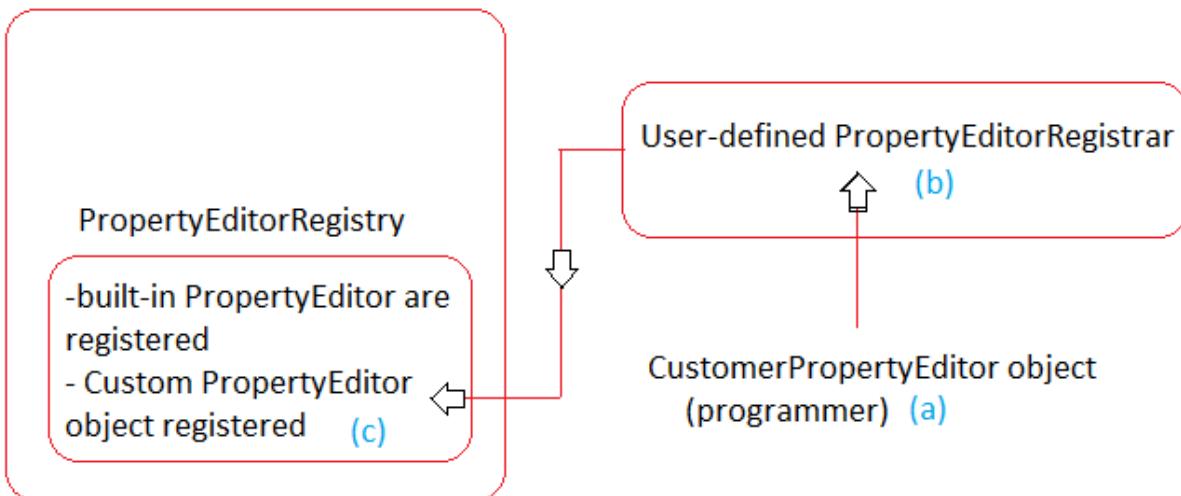
A spring bean class can have 4 types of bean properties:

- a. Simple type (primitive type, String type, wrapper type)
- b. Object/ Reference type
- c. Array type
- d. Collection type

**Note:** Every Customer PropertyEditor must registered IoC container (for both BF, AC container).

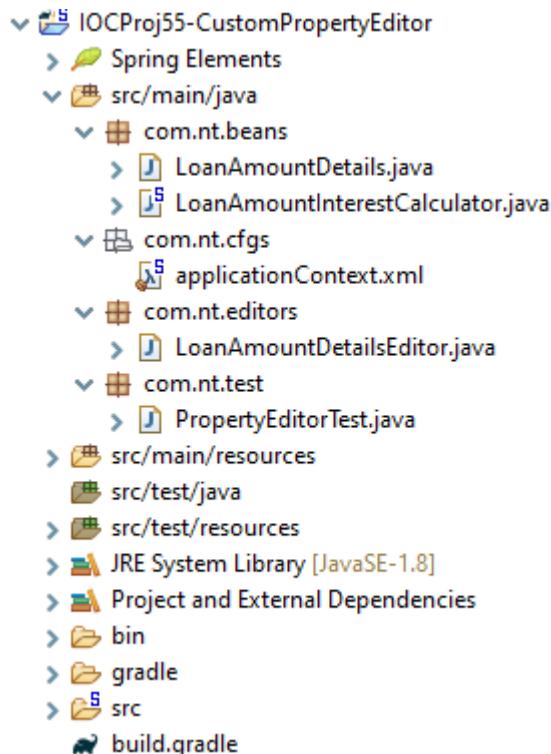
### PropertyEditorRegistry

- + IoC Container maintains PropertyEditorRegistry where Custom Property Editors can be registered. To get Access to this registry we need to create PropertyEditorRegistrar explicitly.



**Note:** We can get create our own PropertyEditorRegistrar by making our class implementing import org.springframework.beans.PropertyEditorRegistrar (I) and this gives access PropertyEditorRegistry. So, that we can register any number of Custom Property Editors by specifying their property types for whom they should be applied.

### Directory Structure of IOCProj55-CustomPropertyEditor:



- Develop the above directory structure and package, class, XML file then use the following code with in their respective file.
- Copy paste build.gradle from any other gradle project because we are using same spring-context-support dependency and Lombok API.

#### LoanAmountDetails.java

```
package com.nt.beans;  
  
import lombok.Data;  
  
@Data  
public class LoanAmountDetails {  
    private float pAmount;  
    private float rate;  
    private float time;  
}
```

```

package com.nt.beans;

import lombok.AllArgsConstructor;
@AllArgsConstructor
public class LoanAmountInterestCalculator {

    private LoanAmountDetails details;

    public float calculateInterestAmount() {
        return
details.getPAmount()*details.getRate()*details.getTime();
    }

}

```

#### 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-4.3.xsd">

    <bean id="laiCalculator"
          class="com.nt.beans.LoanAmountInterestCalculator">
        <constructor-arg name="details" value="100000,2,12"/>
    </bean>

</beans>

```

#### LoanAmountDetailsEditor.java

```

package com.nt.editors;

import java.beans.PropertyEditorSupport;

import com.nt.beans.LoanAmountDetails;

public class LoanAmountDetailsEditor extends PropertyEditorSupport {

    @Override
    public void setAsText(String text) throws IllegalArgumentException {

```

```

float pAmount=0.0f, rate=0.0f, time=0.0f;
String info[] = null;
LoanAmountDetails details = null;
//split text into pAmount, rate, time
/*pAmount = Float.parseFloat(text.substring(0,
text.indexOf(",")));
rate = Float.parseFloat(text.substring(text.indexOf(",") + 1,
text.lastIndexOf(",")));
time = Float.parseFloat(text.substring(text.lastIndexOf(",") + 1,
text.length()));*/
info = text.split(",");
pAmount = Float.parseFloat(info[0]);
rate = Float.parseFloat(info[1]);
time = Float.parseFloat(info[2]);
//create LoanAmountDetails object
details = new LoanAmountDetails();
details.setPAmount(pAmount);
details.setRate(rate);
details.setTime(time);
//set details object to Bean Property as value
setValue(details);
}
}

```

### PropertyEditorTest.java

```

package com.nt.test;

import org.springframework.beans.PropertyEditorRegistrar;
import org.springframework.beans.PropertyEditorRegistry;
import
org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanDefinitionReader;

import com.nt.beans.LoanAmountDetails;
import com.nt.beans.LoanAmountInterestCalculator;
import com.nt.editors.LoanAmountDetailsEditor;

public class PropertyEditorTest {

```

```

public static void main(String[] args) {
    DefaultListableBeanFactory factory = null;
    XmlBeanDefinitionReader reader = null;
    LoanAmountInterestCalculator lamount = null;
    //Create Container
    factory = new DefaultListableBeanFactory();
    reader = new XmlBeanDefinitionReader(factory);
    reader.loadBeanDefinitions("com/nt/cfgs/applicationContext.xml");
    //add custom PropertyEditorRegistrar to IoC container
    factory.addPropertyEditorRegistrar(new CustomRegistrar());
    //get bean object
    lamount = factory.getBean("laiCalculator",
    LoanAmountInterestCalculator.class);
    //invoke method
    System.out.println("Interest amount :
"+lamount.calculateInterestAmount());
}

private static class CustomRegistrar implements
PropertyEditorRegistrar {
    @Override
    public void registerCustomEditors(PropertyEditorRegistry
registry) {
        registry.
        registerCustomEditor(LoanAmountDetails.class, new
LoanAmountDetailsEditor());
    }
} //inner class
}

```

We have 4 types of inner classes:

- Normal inner class (Use it when its logics are required in multiple non-static methods of outer class).
- Nested/ Static inner class (Use it when its logics are required in multiple static methods of outer class)
- Local inner class (Use it when its logics are required only in specific method definition of outer class).
- Anonymous inner class (Use it when its logics are required only in specific method calls of outer class).

Note:

- ✓ `factory.addPropertyEditorRegistrar(new CustomRegistrar());` -> This method takes the given `CustomPropertyEditorRegistrar` object and calls `registerCustomerEditors(-)` having `PropertyEditorRegistry` object as the argument value because of  
`registry.registerCustomEditor(LoanAmtDetails.class, new LoanAmtDetailsEditor());` method the Customer `PropertyEditor` (`LoanAmtDetailsEditor` class object) will be registered with `PropertyEditorRegistry` against the `Property` of type `LoanAmtDetails`).
- ✓ This Registered `CustomPropertyEditor` (`LoanAmtDetailsPropertyEditor` object) will be used by IOC Container (`BeanFactory`) container when it is performing Lazy Instantiation of target bean class (`LoanAmtIntrestCalculator`) for method call  
`factory.getBean("laiCalculator", LoanAmtIntrestCalculator.class);`

[Anonymous inner class-based Logic to register CustomPropertyEditor with BF container:](#)

#### [PropertyEditorTest.java](#)

```
factory.addPropertyEditorRegistrar(new PropertyEditorRegistrar() {  
    @Override  
    public void registerCustomEditors(PropertyEditorRegistry registry) {  
        registry.registerCustomEditor(LoanAmountDetails.class,  
        new LoanAmountDetailsEditor());  
    }  
});
```

[With respect to above code:](#)

- One Anonymous (nameless) inner class is created implementing `PropertyEditorRegistrar (I)`.
- In that Anonymous inner class `registerCustomEditor(-)` is method implemented having logic to register custom `PropertyEditor`.
- Object of the anonymous inner class created using `new` operator and passed it as the argument value of `factory.addPropertyEditorRegistrar(-)` method.

[Lambda Expression Logic to register CustomPropertyEditor with BF container:](#)

- + Since `PropertyEditorRegistrar (I)` is java 8 functional interface (interface with 1 method declaration) we can use LAMDA Expression based Anonymous inner class code as shown below.

### PropertyEditorTest.java

```
factory.addPropertyEditorRegistrar(registry -> {
    registry.registerCustomEditor(LoanAmountDetails.class, new
LoanAmountDetailsEditor());
});
```

- + While working with ApplicationContext container there is no provision to call addPropertyEditorRegistrar(-) on its object because this method is not basically available in BeanFactory (I), ApplicationContext (I) interfaces. This method is available in ConfigurableBeanFactory (I) and implemented by DefaultListableBeanFactory class and not implemented by any ApplicationContext container classes. So, we use other approach for ApplicationContext container that is working with a readymade BeanFactoryPostProcessor called "CustomEditorConfigurer" as shown below.

### applicationContext.xml

```
<bean
class="org.springframework.beans.factory.config.CustomEditorConfigurer">
<property name="customEditors">
<map>
<entry key="com.nt.beans.LoanAmountDetails"
value="com.nt.editors.LoanAmountDetailsEditor"/>
</map>
</property>
</bean>
```

**Note:** ApplicationContext container automatically registers the BeanFactoryPostProcessor right after creating InMemory Metadata of Spring bean configuration file and before performing pre-instantiation singleton scope beans. In this process the above CustomEditorConfigurer internally creates one PropertyEditorRegistrar to access to PropertyEditorRegistry and registers the given CustomPropertyEditor against given property type.

- + Since we can activate/ use BeanFactoryPostProcessor in BeanFactory container by registering them explicitly we can use the above "CustomEditorConfigurer" in BeanFactory container also as shown below.
- + Any how the configuration of CustomEditorConfigurer is the but we have

to add some code in client application.

### PropertyEditorTest.java

```
//add custom PropertyEditorRegistrar to IoC container  
configure = factory.getBean(CustomEditorConfigurer.class);  
configure.postProcessBeanFactory(factory);
```

## 100% code Driven Spring App development

- Also known as Java Config Approach of Spring App Development

### Advantages:

- XML based configuration can be avoided in maximum cases.
- Improves the readability.
- Debugging becomes easy.
- Foundation to learn Spring Boot.

### Thumb rule:

- Configured user-defined classes as Spring beans using stereo type annotations and link them with Configuration class (alternate to Spring bean configuration file (XML file) using @ComponetScan).  
**Note:** Java class that is annotated with @Configuration automatically becomes Configuration class.
- Configured pre-defined classes as Spring beans using @Bean Methods (method that is annotated with @Bean) of @Configuration class (1 method for 1 object of bean class).
- Use AnnotationConfigApplicationContext class to create IoC container having @Configuration class as the input class name.

WishMessageGenerator (target class) -----> LocalDateTime (dependent class)  
(use-defined class -> @Component)      (pre-define class -> @Bean method)

**Q. Why we cannot use stereo type annotations to configured pre-defined classes as spring beans?**

**Ans.** We cannot open the source code of pre-define class to add stereo type annotations on the top of the class. So, go for @Bean methods of @ Configuration class

**Note:** @Bean methods of @Configuration class will be called automatically as part of IoC container singleton scope beans pre-instantiation.

### Sample Configuration class

```
@Configuration  
 @ComponentScan (basePackages="<pkg name(s)>")  
 public class AppConfig {  
     @Bean(name="<beanId>")  
     public <b><class/ interface></b> <b><method> () {  
         ..... //logic to create object and to set data  
         .....  
         return <b><object of class/ implementation class></b>  
     }  
 }
```

The Object returned by this method becomes Spring bean having the given bean id

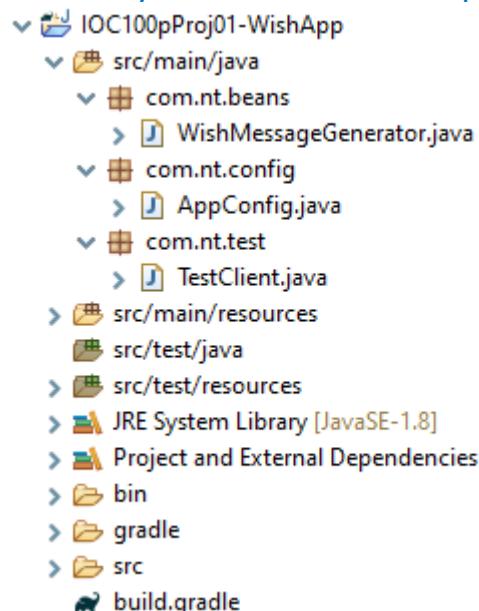
### IoC container creation

```
ApplicationContext ctx=new AnnotationConfigApplicationContext (AppConfig.  
 class);
```

(Configuration class)

**Note:** We cannot @Bean Methods as overloaded methods in the @Configuration class.

### Directory Structure of IOC100pProj01-WishApp:



- Develop the above directory structure and package, class then use the following code with in their respective file.
- Copy paste build.gradle from any other gradle project because we are using same spring-context-support dependency only.

### WishMessageGenerator.java

```
package com.nt.beans;

import java.time.LocalDateTime;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component("wmg")
public class WishMessageGenerator {

    @Autowired
    private LocalDateTime date;

    public String generateWishMessage(String user) {
        int hour = 0;
        // get current hour of the date
        hour = date.getHour();
        // generate wish message (Business logic)
        if (hour < 12)
            return "Good Morning : " + user;
        else if (hour < 16)
            return "Good Afternoon : " + user;
        else if (hour < 20)
            return "Good Evening : " + user;
        else
            return "Good Night : " + user;
    }
}
```

### AppConfig.java

```
package com.nt.config;

import java.time.LocalDateTime;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "com.nt.beans")
public class AppConfig {
```

```

static {
    System.out.println(" AppConfig : static {}");
}

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

@Bean(name = "dt")
public LocalDateTime createSystemDateTime() {
    System.out.println(" AppConfig : createSystemDateTime()");
    return LocalDateTime.now();
}
}

```

### AppConfig.java

```

package com.nt.test;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.AbstractApplicationContext;

import com.nt.beans.WishMessageGenerator;
import com.nt.config.AppConfig;

public class TestClient {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        WishMessageGenerator generator = null;
        //create IoC container
        ctx = new
AnnotationConfigApplicationContext(AppConfig.class);
        // get Target bean class object
        generator = ctx.getBean("wmg", WishMessageGenerator.class);
        // invoke the method
        System.out.println("Wish Message is : " +
generator.generateWishMessage("Nimu"));
    }
}

```

## Flow Execution:

WishMessageGenerator.java

```
package com.nt.beans; (e)
@Component("wmg") (f) Object creation
public class WishMessageGenerator {
    @Autowired (g) looks
    private LocalDateTime date;
    ..... (j) .....
} (injection completed)
```

AppConfig.java

```
@Configuration
@ComponentScan(basePackages = "com.nt.beans")
public class AppConfig { (c) configuration class instantiation
```

```
    @Bean(name = "dt") (h)
    public LocalDateTime createSystemDateTime() {
        System.out.println("AppConfig : createSystemDateTime()");
        return LocalDateTime.now(); (i)
    }
}
```

TestClient.java

```
public class TestClient { (a)
    public static void main(String[] args) {
        ApplicationContext ctx = null;
        WishMessageGenerator generator = null;
        //create IoC container (b)
        ctx = new AnnotationConfigApplicationContext(AppConfig.class);
        // get Target bean class object (l)
        (n) generator = ctx.getBean("wmg", WishMessageGenerator.class);
        // invoke the method
        System.out.println("Wish Message is : " +
generator.generateWishMessage("Nimu"));
    }
}
```

**Note:** 100% Code driven Approach first pre-instantiates user-defined singleton scope Spring beans that are referred through `@ComponentScan` and later pre-instantiates `@Bean` methods related singleton scope Spring beans.

Approach	Default bean Id	Example
XML driven configuration	<code>&lt;fully qualified classname&gt;#&lt;n&gt;</code>	<pre>&lt;bean class="com.nt.beans.WishMessageGenerator"&gt; default bean Id is: com.nt.beans.WishMessageGenerator#0</pre>
Annotation configuration	bean class name having first letter in lower case	<pre>@Component public class WishMessageGenerator{     ..... } default bean Id is: wishMessageGenerator</pre>
<code>@Bean Methods</code>	method name	<pre>@Bean public LocalDateTime createSysDateTime() {     ..... } default bean Id is: createSysDateTime</pre>

**Note:** `@Streo` type annotations cannot be applied at method level. Similarly, `@Bean` cannot be applied at class level.

While working with following features of Spring core module there are no annotations. So, use XML driven configuration and link that XMI file with `@Configuration` class using `@ImporResource` Annotations. The features are

- 1) Bean inheritance
- 2) Collection merging
- 3) Inner beans
- 4) Method replacer
- 5) Factory method bean instantiations
- 6) collection Injection and etc.

#### Directory Structure of IOC100pProj02-MethodReplacer:

- ⊕ Copy paste the IOCProj50-MethodReplacer project and change `rootProject.name` to `IOC100pProj02-MethodReplacer` in `settings.gradle` file.

- + Add com.nt.config package with a class AppConfig.java.
- + Add the following code in their respective files.

### AppConfig.java

```
package com.nt.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;

@Configuration
@ImportResource("classpath:com/nt/cfgs/applicationContext.xml")
@ComponentScan(basePackages = "com.nt.replacer")
public class AppConfig {

}
```

### MethodInjectionTest.java

```
public class MethodInjectionTest {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        BankLoanMgmt bank = null;
        //Create IoC container
        ctx = new
        AnnotationConfigApplicationContext(AppConfig.class);
        //get Target class bean
        bank = ctx.getBean("bank", BankLoanMgmt.class);
        //invoke method
        System.out.println("Interest amount:
"+bank.calculateInterestAmount(100000, 2, 12));

        //close container
        ((AbstractApplicationContext) ctx).close();
    }
}
```

Now we are going to convert a Mini Project to 100% code driven application/ Java config driven application.

## Directory Structure of IOC100pProj03-MiniProject2-NestedACContainer-Java8:

- + Copy paste the IOCProj40-MiniProject2-NestedACContainer-Java8 Project and change rootProject.name to IOC100pProj02-MethodReplacer in settings.gradle file.
- + Add com.nt.config package with a class BusinessAppConfig.java and PresentationAppConfig.java remove the com.nt.cfgs and com.nt.common package along with the file.
- + Add the following code in their respective files.

### EmployeeDAOImpl.java

```
@Repository("empDAO")
public class EmployeeDAOImpl implements EmployeeDAO {

    @Autowired
    private DataSource ds;
```

### EmployeeMgmtServiceImpl.java

```
@Service("empService")
public class EmployeeMgmtServiceImpl implements EmployeeMgmtService {
    @Autowired
    public EmployeeDAO dao;
```

### MainController.java

```
@Controller("controller")
@Lazy
public class MainController {

    @Autowired
    private EmployeeMgmtService service;
```

### PresentationAppConfig.java

```
package com.nt.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "com.nt.controller")
public class PresentationAppConfig { }
```

### BusinessAppConfig.java

```
package com.nt.config;

import javax.sql.DataSource;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import com.zaxxer.hikari.HikariDataSource;

@Configuration
@ComponentScan(basePackages = {"com.nt.service", "com.nt.dao"})
public class BusinessAppConfig {

    @Bean(name="hkDs")
    public DataSource createDS() {
        HikariDataSource hkDs = null;
        //Create DataSource
        hkDs = new HikariDataSource();
        hkDs.setDriverClassName("oracle.jdbc.driver.OracleDriver");
        hkDs.setJdbcUrl("jdbc:oracle:thin:@localhost:1521:xe");
        hkDs.setUsername("system");
        hkDs.setPassword("manager");
        hkDs.setMinimumIdle(10);
        hkDs.setMaximumPoolSize(100);
        return hkDs;
    }

}
```

### NestedIoCContainerTest.java

```
public class NestedIoCContainerTest {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext parentCtx = null, childCtx
= null;
        MainController controller = null;
        List<EmployeeVO> listVO = null;
        //create the parent ioc container
        parentCtx = new
        AnnotationConfigApplicationContext(BusinessAppConfig.class);
```

```

    //create child IoC container
    childCtx = new
AnnotationConfigApplicationContext(PresentationAppConfig.class);
    childCtx.setParent(parentCtx);
    //get controller object
    controller = childCtx.getBean("controller",
MainController.class);
    //invoke the method
    try {
        listVO = controller.getEmployeeByDesg("clerk",
"manager", "salesman");
        listVO.forEach(System.out::println);
    } catch (Exception e) {
        e.printStackTrace();
    }

    //close containers
((AbstractApplicationContext) parentCtx).close();
((AbstractApplicationContext) childCtx).close();
}
}

```

**Note:** While creating Nested container using 100p code driven approach we get problem towards injecting service class of parent container to controller class child container. To solve that problem.

- Enable @Lazy on controller class and write the following code to create Nested container in client app.

```

//create the parent IoC container
parentCtx = new
AnnotationConfigApplicationContext(BusinessAppConfig.class);
//create child IoC container
childCtx = new
AnnotationConfigApplicationContext(PresentationAppConfig.class);
    childCtx.setParent(parentCtx);

```

- Write following code in the client without any enabling @Lazy on controller class.

```
//create child IoC container
childCtx = new AnnotationConfigApplicationContext();
childCtx.setParent(parentCtx);
childCtx.register(PresentationAppConfig.class);
childCtx.refresh();
```

- While developing Layered applications instead of configuring all DAO classes, service classes, AOP classes and Controller class as Spring beans in single Spring bean configuration file (XML file), It recommended to take multiple XML files and link them to single xml file.

persistence-beans.xml --> DAO classes, DS configuration  
 service-beans.xml --> Service classes configuration  
 aop-beans.xml --> AOP classes configuration  
 controller-beans.xml --> controller class configuration

[Link with,](#)

[applicationContext.xml](#)

```
<beans>
  <import resource="persistence-beans.xml"/>
  <import resource="service-beans.xml"/>
  <import resource="aop-beans.xml"/>
  <import resource="controller-beans.xml"/>
</beans>
```

[In 100p Code driven Configurations:](#)

@Configuration  
 PersistenceConfig class --> DAO classes, DS configuration  
 @Configuration  
 ServiceConfig class --> Service classes configuration  
 @Configuration  
 AOPConfig class --> AOP classes configuration  
[Link them with single Configuration class,](#)  
 @Configuration  
 @Import (value={PersistenceConfig.class, ServiceConfig.class, AOPConfig.class})  
 AppConfig class --> Main configuration class

**Note:** By taking multiple Spring bean configuration files or multiple Configuration classes as shown above, we can decrease the possibility getting conflicts in the team environment while working with code Repository/ Management tools like GIT/ SVN and etc.

**Q. What is the difference between @Import and @ImportResource?**

**Ans.** @ImportResource given to link Spring bean configuration file (XML file) with @Configuration class.

@Import given to link Helper configuration classes with Main @Configuration class.

e.g.

1. @Configuration

```
@Import (value={PersistenceConfig.class, ServiceConfig.class,  
AOPConfig.class})
```

```
public class AppConfig class {....}
```

2. @Configuration

```
@ImportResource(.com/n%cfgs/applicationContext.xml")
```

```
public class AppConfig class {....}
```

**Note:**

- ✓ In every IoC container creation one built-in object will be maintained that "Environment" object having system property values (like os.name and etc.) and given properties files values and profiles Info.
- ✓ This Environment object can be injected to our Spring beans to read and use its data. by submitting key to get value.

### BusinessAppConfig.java

```
@Configuration  
@ComponentScan(basePackages = {"com.nt.service", "com.nt.dao"})  
@PropertySource(value="com/nt/commons/jdbc.properties")  
//@PropertySource(value= {"com/nt/commons/jdbc.properties",  
"com/nt/commons/jdbc.properties"})  
public class BusinessAppConfig {  
    @Autowired  
    private Environment env;  
    @Bean(name="hkDs")  
    public DataSource createDS() {  
        HikariDataSource hkDs = null;  
        //Create DataSource  
        hkDs = new HikariDataSource();  
        hkDs.setDriverClassName(env.getRequiredProperty("jdbc.driver"));  
        hkDs.setJdbcUrl(env.getRequiredProperty("jdbc.url"));  
        hkDs.setUsername(env.getRequiredProperty("jdbc.user"));  
        hkDs.setPassword(env.getRequiredProperty("jdbc.password"));  
    }  
}
```

```

        hkDs.setMinimumIdle(Integer.parseInt(env.getRequiredProperty("po
ol.minIdle")));
        hkDs.setMaximumPoolSize(Integer.parseInt(env.getRequiredProperty
("pool.maxSize")));
        return hkDs;
    }
}

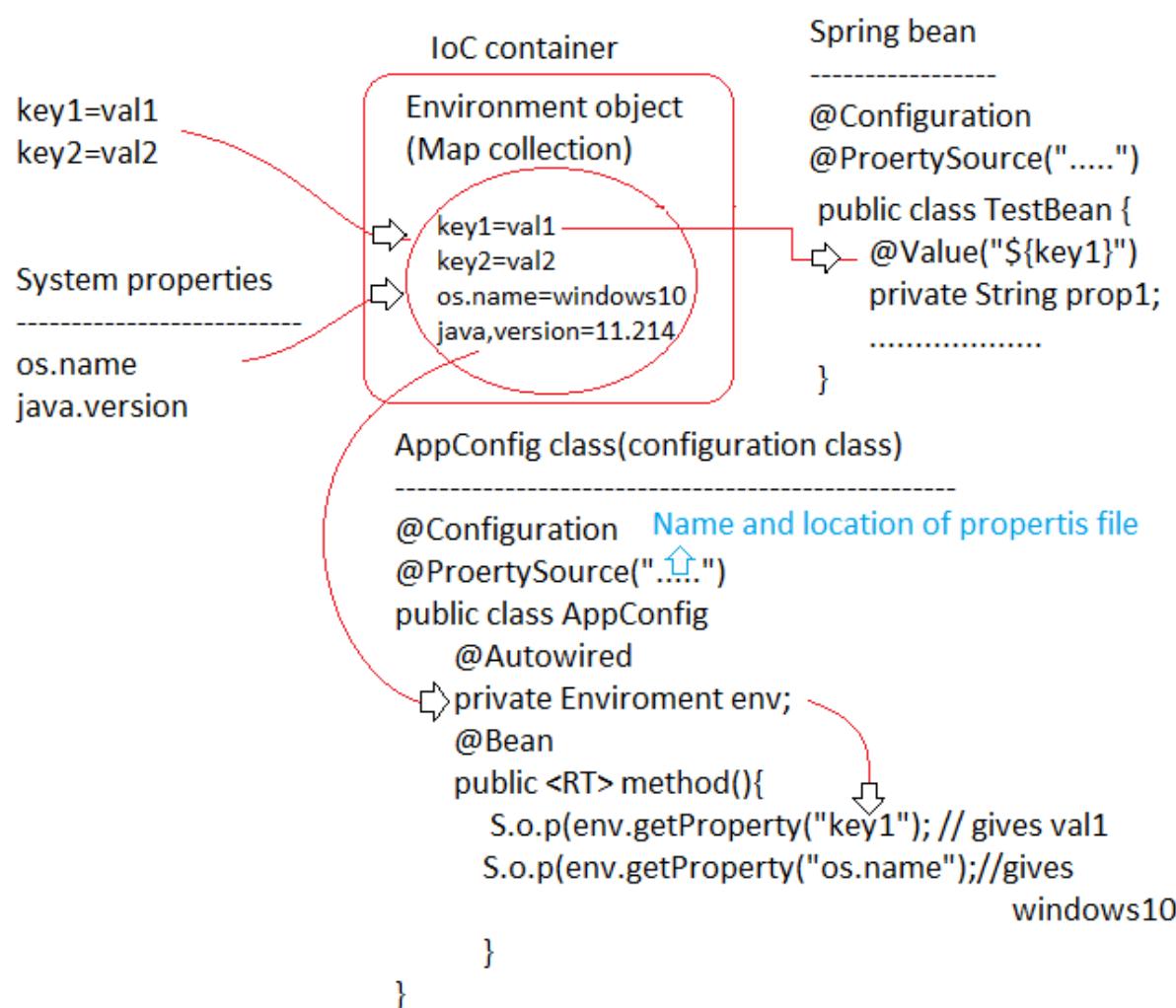
```

### jdbc.properties

```

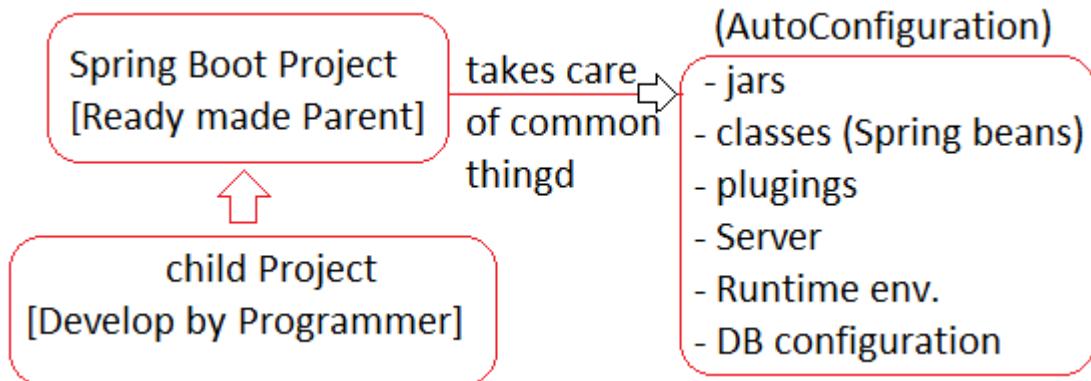
#Oracle details
jdbc.driver=oracle.jdbc.driver.OracleDriver
jdbc.url=jdbc:oracle:thin:@localhost:1521:xe
jdbc.user= system
jdbc.password=manager
pool.minIdle=10
pool.maxSize=100

```



## Spring Boot

- Spring boot is a Spring Project that act as parent project to Programmer developed Child Project taking care common things application/ Project development through a concept called AutoConfiguration.



**Note:** Due to auto configuration the number of lines we write in child project will be reduced drastically.

### Q. What is AutoConfiguration?

**Ans.** Spring Boot providing jars, classes (Spring beans), Servers, Runtime environment plugins, DB configuration and etc. based on the "Spring boot starters" that we add to child project is called AutoConfiguration.

Project1 (Flipkart.com)



Developer 1

Project 2 (GPay)



Developer 2

common things both projects are

- Tomcat server
- DB configuration
- Jars
- Plugins
- and etc.

- If both Projects are developed by using Spring f/w, Programmers only should take care of these common things.
- If both Projects are developed by using spring Boot (extension of Spring) Programmers need not to take care of these common things because Spring boot will generate them dynamically based on the "spring boot starters" that we added to the Project (Readymade partial code).

### DB Connectivity Using Spring:

@Configuration

@ComponentScan(basePackage="com.nt.dao")

```
public class PersistenceConfig {          #Programmer
    @Bean
    public DataSource createDs(){
        HikariDataSource ds=new HikariDataSource();
        ds.setDriverClassName(" .... ");
        ds.setUrl(" .... ");
        ....
        return ds;
    }
}
```

#### Jars in CLASSPATH #Programmer

- spring-context-support-<ver>.jar
- spring-jdbc-<ver>.jar
- hikaricp-<ver>.jar
- ojdbc8.jar

#### DAO class #Programmer

```
@Repository("empDAO")
public class EmployeeDAOImpl implements
    @Autowired
    private DataSource ds;
    public int insert(EmployeeBO bo){
        //use ds here
        .....
        .....
    }
}
```

#### DB Connectivity Using Spring Boot:

- Just add "spring-boot-starter-jdbc" to the Project to take care Autoconfiguration.
- Spring-boot-starter-jdbc gives Spring Jars (Spring-context-support, Spring JDBC, and all dependency jars), Spring beans (DS, JdbcTemplate and etc., HikariCP Jars).

#### application.properties #Programmer

```
spring.datasource.driver-class-name= .....
spring.datasource.url= .....
spring.datasource.username= .....
spring.datasource.password= .....
```

```

DAO class           #Programmer
@Repository("empDAO")
public class EmployeeDAOImpl implements
    @Autowired
    private DataSource ds;
    public int insert(EmployeeBO bo){
        //use ds here
        .....
        .....
    }
}

```

**Thumb rule while working with Spring boot:**

- + Configured user-defined classes as Spring beans using stereo type annotations.
- + Configured pre-defined classes as Spring beans using @Bean methods in configuration class only if they are not coming through autoconfiguration based on Spring starters that we have added.
- + Give instructions to Autoconfiguration process using application.properties/ yml file.
  - yaml: yet another markup language
  - yaml: YAML Ain't Markup L
  - (Another approach of writing file nothing key=value pair)

These are ready made Spring boot starters and we need them either in Gradle - build.gradle (or) Maven - pom.xml:

spring-boot-start-<\*> naming convention

- spring-boot-starter-jdbc
- spring-boot-starter-mail
- spring-boot-starter-aop
- spring-boot-starter-web
- spring-boot-starter-data-jpa and etc.

URL for starters: [\[Starters\]](#)

### build.gradle

```
//https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-jdbc
implementation group: 'org.springframework.boot', name: 'spring-boot-starter-jdbc',
version: '2.3.4.RELEASE'
```

## pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <version>2.3.4.RELEASE</version>
</dependency>
```

- ⊕ Working with java technologies (like JDBC, Servlet, JSP and etc.):  
Washing clothes manually.
- ⊕ Working with Spring f/w: semi-automated washing machine.
- ⊕ working with Spring boot: fully automated washing machine

### Note:

- ✓ Old projects (small, medium and large Scale) are in spring.  
(Now there are in Maintenance mode/Enhancement mode)
- ✓ New Projects (small, medium Scale) are in spring f/w.
- ✓ New Projects (Larger Scale) are in spring Boot.
- ✓ Migration Projects (Spring to Spring Boot)

Spring Boot = Spring f/w - XML files + AutoConfiguration + Embedded Servers + Embedded DB + .....

- ⊕ Spring Boot gives single useful annotations by combining related multiple annotations  
`@SpringBootApplication = @Configuration + @ComponentScan + @EnableAutoConfiguration`

### We can develop Spring boot Apps in multiple ways:

- a. Using CLI (Command Line Interface)
- b. Using start.spring.io website
- c. Using Eclipse IDE having STS Plugin + Gradle/ Maven support (best)

- ⊕ Every Spring Boot App main class/ runner class (the class that is having main (-) method) must be annotated with `@SpringBootApplication`.

### `@SpringBootApplication` contains:

- a. `@ComponentScan`: To recognize Java classes/ Spring beans and configuration classes automatically that are there in the same package or sub packages of main class package.

```

com.nt (root package)    Main class/ runner class/ starter class
    |--> <MainAppClass>.java having @SpringBootApplication +
          main
    |--> com.nt.dao
        |--> all dao classes having @Repository
    |--> com.nt.service
        |--> all service classes having @Service
    |--> com.nt.controller
        |--> controller class having @Controller
    |--> com.nt.beans
        |--> helper classes having @Component
    |--> com.nt.config
        |--> PersistenceConfig ,ServiceConfig and etc. having
              @Configuration with @Bean methods

```

- b. **@Configuration:** To make main class/ runner class/ starter class as Configuration class.
- c. **@EnableAutoConfiguration:** To provide common things like jars, pre-defined classes as spring beans, plugins, servers, runtime and etc. automatically based on spring boot starters that are added.

### [org.springframework.boot.](#)

#### [SpringApplication.run\(-,-\)](#)

- Bootstraps the spring Application by creating multiple objects internally like ApplicationContext object (IoC container) and etc.
- Refreshes the ApplicationContext object (IoC container) by loading and pre-instantiating all singleton scope beans.
- Returns the internally created ApplicationContext obj (IoC container). So, that we can use it to call ctx.getBean(-) methods.

### [Sample main/ starter in Spring Boot Application:](#)

```

package com.nt;
@SpringBootApplication
public class SpringBootAppTester {
    public static void main(String args[]){
        ApplicationContext ctx=null;
        //get IOC container
        ctx=SpringApplication.run(SpringBootAppTester.class, args);
        //get Beans
        .....
    }
}

```

```

        //close container
        ctx.close();
    }
}

```

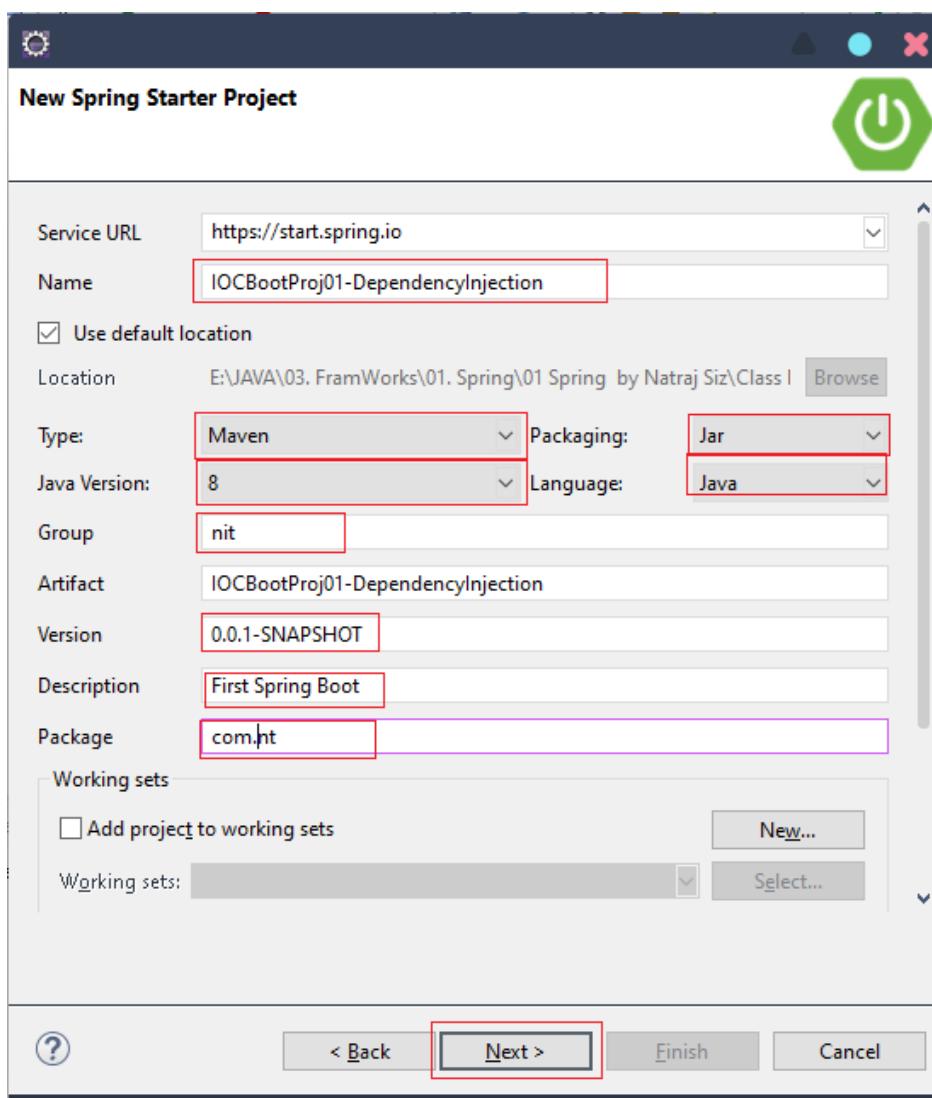
**Note:** Spring BootAppTester.class for internally created AnnotationConfigApplicationContext object the current class will be passed as the configuration class.

### Procedure First Application using spring boot support in Eclipse IDE:

**Step 1:** Make sure that STS plugin and maven/ gradle plugins are added to Eclipse IDE.

**Step 2:** Create Spring boot starter project

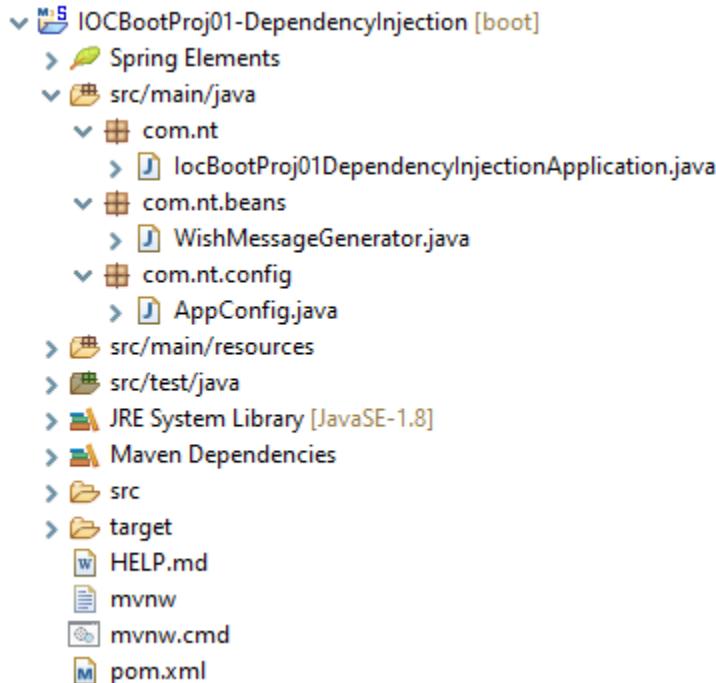
File menu -> new -> spring starter project fill -> details -> Next -> Next -> Finish



**Step 3:** Develop the package and cod.

**Step 4:** Run the project.

#### Directory Structure of IOCBootProj01-DependencyInjection:



- Develop the above directory structure using Spring Starter Project option and package and classes also.
- Many jars dependencies will come automatically in pom.xml because so need not to add any jars now.
- Then use the following code with in their respective file.

#### AppConfig.java

```
package com.nt.config;

import java.time.LocalTime;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {

    @Bean("time")
    public LocalTime getTime() {
        return LocalTime.now();
    }
}
```

### WishMessageGenerator.java

```
package com.nt.beans;

import java.time.LocalTime;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component("wmg")
public class WishMessageGenerator {

    @Autowired
    private LocalTime time;

    public String generateWishMessage(String user) {
        int hour = 0;
        // get current hour of the date
        hour = time.getHour();
        // generate wish message (Business logic)
        if (hour < 12)
            return "Good Morning : " + user;
        else if (hour < 16)
            return "Good Afternoon : " + user;
        else if (hour < 20)
            return "Good Evening : " + user;
        else
            return "Good Night : " + user;
    }
}
```

### locBootProj01DependencyInjectionApplication.java

```
package com.nt;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

import com.nt.beans.WishMessageGenerator;

@SpringBootApplication
public class locBootProj01DependencyInjectionApplication {
```

```

public static void main(String[] args) {
    ApplicationContext ctx = null;
    WishMessageGenerator generator = null;
    ctx =
SpringApplication.run(IocBootProj01DependencyInjectionApplication.class,
args);
    //get Bean
    generator = ctx.getBean("wmg", WishMessageGenerator.class);
    //invoke method
    System.out.println(generator.generateWishMessage("nimu"));
}
}

```

**Note:**

- ✓ Based on the starters that we have added to the BUILDPATH/ CLASSPATH the Spring boot performs auto configuration by giving certain pre-defined classes as Spring beans and other operations. In this process if we want to provide instructions to Spring boot/ starters then we can give use application.properties.(main/java/resources). It is part of Spring boot ecosystem, i.e. we need not configure it separately.
- ✓ if we add spring-boot-starter-jdbc to BUILDPATH using pom.xml or build.gradle then we get the following pre-defined classes through Autoconfiguration including their jar files.
  - a. HikariDataSource
  - b. JdbcTemplate, NamedParameterJdbcTemplate
  - c. DataSourceTransactionManager

pom.xml

```

<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-
boot-starter-jdbc -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <version>2.3.4.RELEASE</version>
</dependency>

```

build.gradle

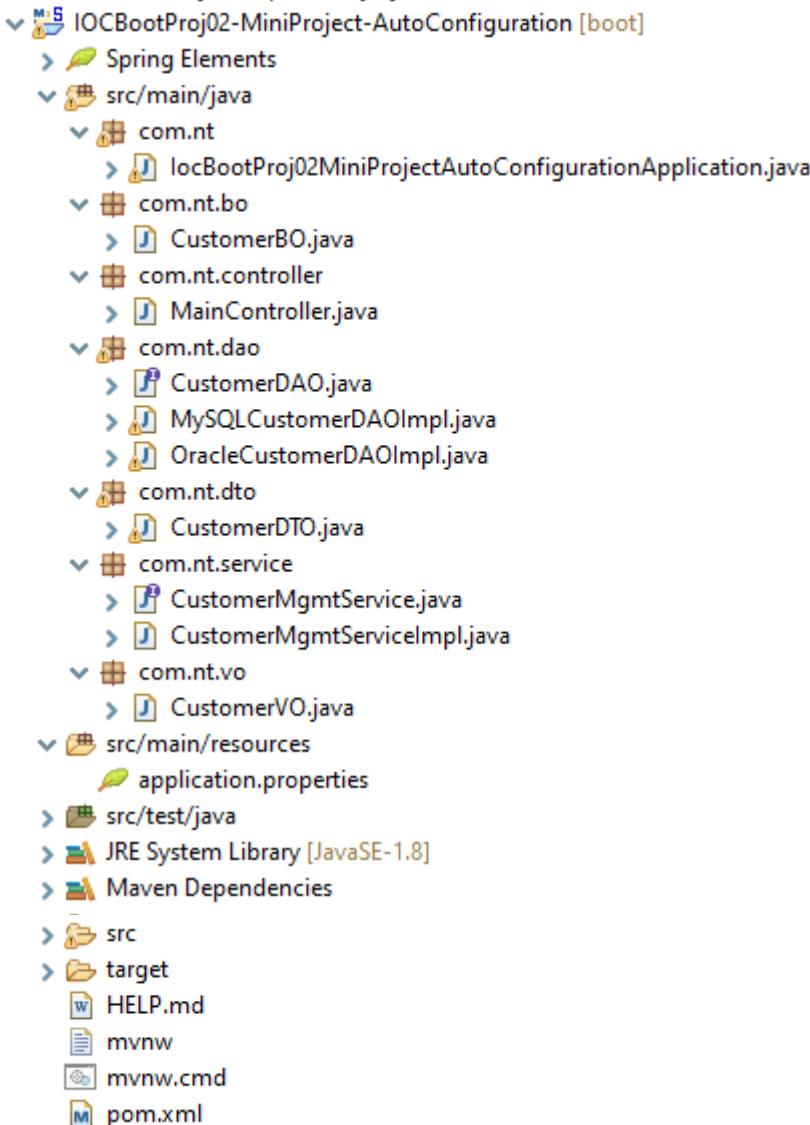
```

// https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-
starter-jdbc

```

implementation group: 'org.springframework.boot', name: 'spring-boot-starter-jdbc', version: '2.3.4.RELEASE'

### Directory Structure of IOCBootProj02-DependencyInjection:



- Develop the above directory structure using Spring Starter Project option and package and classes also.
- Choose the following Spring Stater Project Dependencies
  - JDBC API
  - MySQL Driver
  - Oracle Driver
  - Lombok
- Then use the following code with in their respective file.
- Copy the rest packages and classes from IOCAnnoProj04-RealTimeDI-RealTimeStrategyDP-LayeredApp.

## application.properties

```
#DataSource configuration
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=system
spring.datasource.password=manager
```

## locBootProj02MiniProjectAutoConfigurationApplication.java

```
package com.nt;

import java.util.Scanner;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AbstractApplicationContext;

import com.nt.controller.MainController;
import com.nt.vo.CustomerVO;

@SpringBootApplication
public class locBootProj02MiniProjectAutoConfigurationApplication {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        MainController controller = null;
        String name=null, address=null, Amount=null, time=null,
rate=null;
        CustomerVO vo = null;
        String result = null;
        Scanner sc = null;
        //Read inputs from end-user using scanner
        sc = new Scanner(System.in);
        System.out.println("Enter the following Details for registration :
");
        System.out.print("Enter Customer Name : ");
        name = sc.next();
        System.out.print("Enter Customer Address : ");
        address = sc.next();
        System.out.print("Enter Customer Principle Amount : ");
        Amount = sc.next();
        ...
        ...
    }
}
```

```

        System.out.print("Enter Customer Time : ");
        time = sc.next();
        System.out.print("Enter Customer Rate of Interest: ");
        rate = sc.next();
        //Store into VO class object
        vo = new CustomerVO();
        vo.setCname(name);
        vo.setCadd(address);
        vo.setpAmt(Amount);
        vo.setTime(time);
        vo.setRate(rate);
        //get controller class object
        //get container
        ctx =
SpringApplication.run(locBootProj02MiniProjectAutoConfigurationApplication.class, args);
        //get controller class
        controller = ctx.getBean("controller", MainController.class);
        //invoke mmethods
        try {
            result = controller.processCustomer(vo);
            System.out.println(result);
        } catch (Exception e) {
            System.out.println("Internal probelm :
"+e.getMessage());
            e.printStackTrace();
        }
        //close container
        ((AbstractApplicationContext) ctx).close();
    }
}

```

**Note:** Here we do not configure HikariDataSource class using @Bean method anywhere because it is coming as spring bean through auto Configuration.

**Note:** Spring Boot 2.x uses two DataSource as part of AutoConfiguration hierarchy.

- HikariCP
- ApacheDBcp2 (If HikariCP jars not there in build path)

**Q. How to work with Apache DBCP2 in Spring boot application?**

**Ans.** Make sure that HikariCP jars excluded from build path with respect spring-boot -starter- jdbc and add Apache DBCP2 jar file to build path by adding following entries to pom.xml file.

**pom.xml**

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <exclusions>
        <exclusion>
            <groupId>com.zaxxer</groupId>
            <artifactId>HikariCP</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<!--
https://mvnrepository.com/artifact/org.apache.commons/commons-dbc2
-->

<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
    <version>2.8.0</version>
</dependency>
```

**Note:** If both are there, it will take HikariCP. To break the default DataSource algorithm of Spring boot and to configure your choice DataSource class as default DataSource class of auto configuration then specify that DataSource class name in application.properties also add relevant jar file in pom.xml

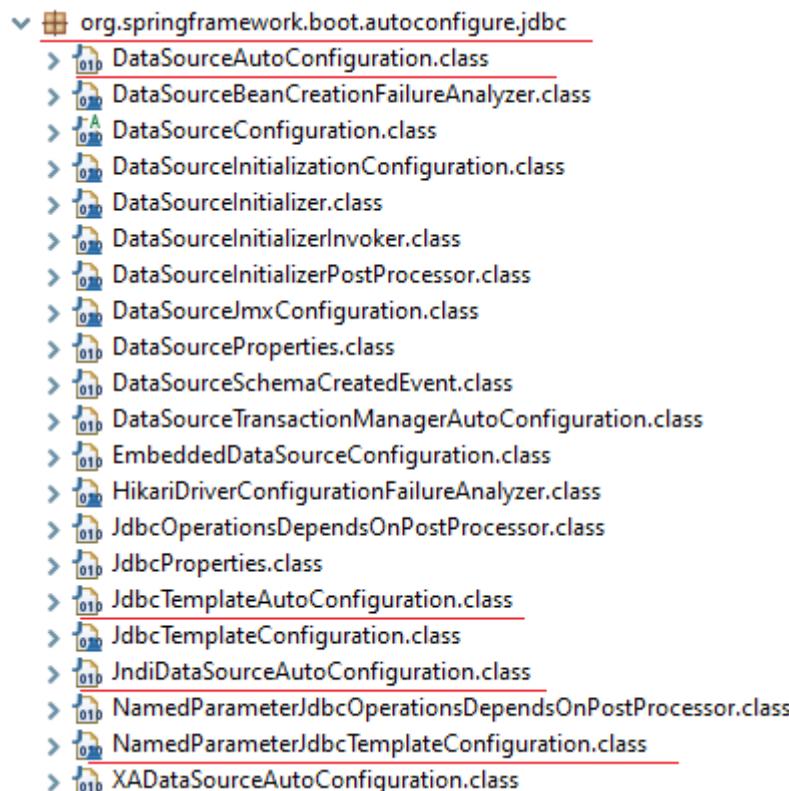
**pom.xml**

```
<!-- https://mvnrepository.com/artifact/com.mchange/c3p0 -->
<dependency>
    <groupId>com.mchange</groupId>
    <artifactId>c3p0</artifactId>
    <version>0.9.5.5</version>
</dependency>
```

## application.properties

```
#TO change the Defeualt DataSource type of autoconfiguration  
spring.datasource.type=com.mchange.v2.c3p0.ComboPooledDataSource
```

- ⊕ To find out the list of pre-defined classes that comes as Spring beans through Autoconfiguration, we can use XxxxAutoConfiguration class names of different packages from spring-boot-autoconfigure-<version>.jar



- ✓ To make certain classes of any Spring boot starter, not coming through AutoConfiguration we need to use "exclude" param of `@SpringBootApplication` annotation.  
`@SpringBootApplication(exclude = {  
 JdbcTemplateAutoConfiguration.class,  
 DataSourceTransactionManagerAutoConfiguration.class})  
public class locBootProj02MiniProjectAutoConfigurationApplication {  
 .....  
 .....  
 .....  
}`

To disable spring boot banner:

```
#To disable Spring boot banner  
spring.main.banner-mode=off
```

values: console for on, log for log file

To Add custom start-up banner to Spring boot application:

Step 1: Get custom banner content from online [\[visit\]](#)

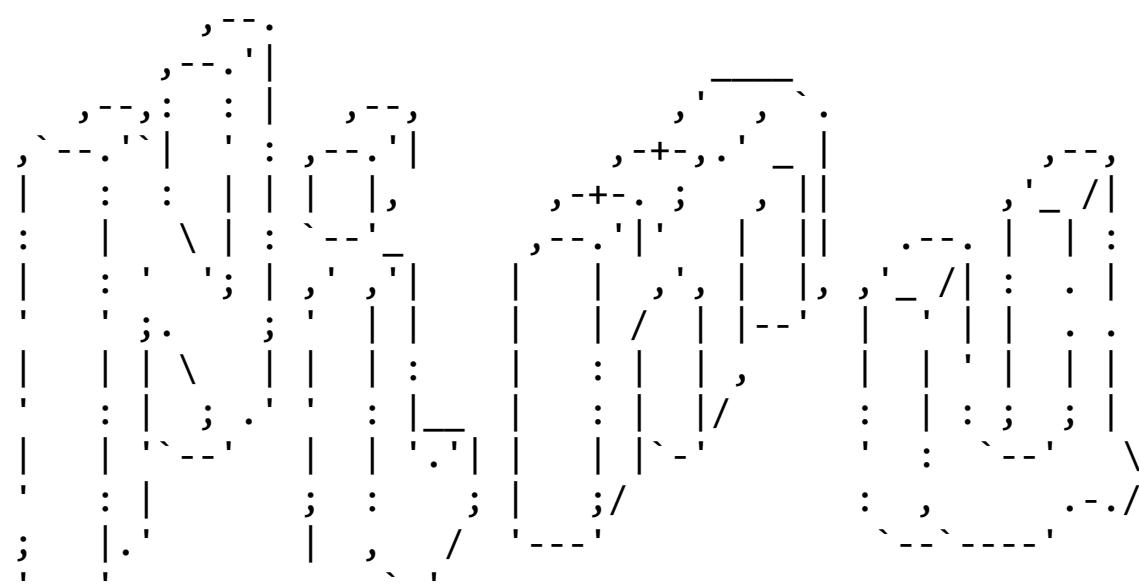
Step 2: Copy and paste the above banner content to a txt file

src/main/java

|--> com/nt/banner/mybanner.txt

Step 3: Enable Spring boot banner and specify the above file location in application properties.

mybanner.txt



application.properties

```
#To enable spring bot banner  
spring.main.banner-mode=console
```

```
#Custom banner location  
spring.banner.location=classpath:/com/nt/banner/mybanner.txt
```

Another approach of Bootstrapping Spring boot application from the main class/ starter class:

```
SpringApplication app = new SpringApplication();
app.setBannerMode(Banner.Mode.OFF);
ctx = app.run(locBootProj02MiniProjectAutoConfigurationApplication.class,
args);
```

Note:

- + The spring-boot-starter-parent that we add to our pom.xml file will inherit spring boot parent project to our spring boot project (child project) using maven inheritance gives multiple dependencies, maven plugins and configuration properties to child project (our project) from parent boot project.

- + Add following line in our project (child project) pom.xml in order to inherit from spring boot parent project available in maven central repository.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.3.4.RELEASE</version>
  <relativePath /> <!-- lookup parent from repository -->
</parent>
```

- + The spring-boot-starter-parent project is a special starter project - that provides default configurations for our application and a complete dependency tree to quickly build our Spring Boot project. It also provides default configuration for Maven plugins such as maven-failsafe-plugin, maven-jar-plugin, maven- sure fire-plugin, maven-war-plugin.

- + Beyond that, it also inherits dependency management from spring-boot-dependencies which is the parent to the spring-boot-starter-parent.

- + For more reference [\[Visit\]](#)

- + Spring boot starter parent pom.xml [\[Visit\]](#)

The END