Download latest distributation zip file
https://repo.spring.io/release/org/springframework/spring/

Download apache common loogs
https://commons.apache.org/proper/commons-logging/download_logging.cgi

how to create libary

rightclick-->project src project----> buildpath---confiure build path--->libeararies--->add libeary-->user libary-->next--->user libeary-->new---> give the user librariy name-->ok--now add external jar

add liberary in class path

spring-bean.realse
spring-core.realse
spring-context.realse
spring-context-support.realse
spring-expression.realse

spring common.jar

Spring IoC Containers

The spring contanier is the core of the Spring framework. The container creates the objects, waire them together, configure them, and manage their complete life cyle from creation till destruction, the spring contaier uses dependency injection(DI) to manage the components that make up an application.

Spring provides following two distinct types of containers

1. BeaFactory containers
2. ApplicationContenxt Container

1. BeanFactory Container

BeanFactory is represented by **org.springframwork.beans.factory.BeanFactory** interface. It is the main and the basic way to access the spring container. Other ways to access the spring container such as ApplicationContext, ListableBeanFactory, ConfigurableBeanFactory etc are build upon this BeanFactory.

BeanFactory interface proivedes basic functionality for the spring Container like:

1) It provides DI/IOC mechanism for the spring
2) It is built upon Factory Desing Pattern,
3)It loads the beans definitions and their property descriptions from some configuration source (from XML configuration file)
4) Instatiates the beans when they are requested

5) Wire dependencies and properties for the beans accrording configuration defined in configuration source while instatiating the beans.

6) Manage the bean life cycle by bean lifecycle interface and calling initialization and destruction methods.

(Note. That **beaFactory** does not create the objects of beans immediately when it loads the configuration for beans from configuration file. Only bean definitions and their property descriptions are loaded. Beans themseleves are instaintiated and the properties are set only when they are requested such as as by **getBean()** method)

BeanFactory Implementations:

the most important BeanFactory implementation is-

org.springframework.beans.factory.xml.XmlBeanFactory. It reads beans definitions from an XML file.

Constructor for XmlBeanFactory

XmlBeanFactory(Resource resource)

the BeanFactory interface has six methods for client code to call
1) Object getBean(String);
returns an instance of the bean registerd under the given name. Depending on how the bean was configured by the BeanFactory configuration, either a singleton and thus shared instance or newly cretated bean will be returned A BeanException will be thrown when either the bean could not be found or an exception occurred while instantiating and preparing the bean.

2) **Boolean containsBean(String)** returns true if the BeanFactory contains a bean defination or bean instance that matches the given name

3) **Object getBean(String , Class)** returns a bean, registered under the given name. The bean returned will be cast to the given Class. If the bean could not be cast, corresponding exceptions will be thrown. Furthermore, all rules of the getBean(String) method apply.

4) **class getType(String name):** returns the Class of the bean with the give name. If no bean corresponiding to the given name could be found. A **NoSuchBeanDefinitionException** will be throwns

5) **boolean isSingleton(String):** determines whether or not the bean definition or bean instance registered under the given name is a singleton. If no bean corresponding the given name could be found, a **NoSuchBeanDefinitionException** will be thrown

6) **String[] getAliases(String):** return the aliase for the given bean name, if any were defined in the bean definition

ApplicationContext Container

This container add more enterprise-specific functional such as the ability to resolve textual messages from a properties file and the ablitiy to publish application events to interested event listeners. The container is defined by the **org.springframework.contenxt.ApplicationContext** interface
The ApplicationContext container includes all functionality of the BeanFactory container, so it is generally recommended over the BeanFactory can still be used for lightweight applications like mobile devices or applet based applications where data volume and speed is significant.

Let start over first application print hello world

project name --> helloWorld
-->src
----------->beans,xml

---------->com.infotech.modal
----------> Message (bean)
---------->com.infotech.client
----------> ClientTest

bean.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="message" class="com.infotech.modal.Message">
             <property name="messgeId" value="1001"></property>
             <property name="message" value="hello world"></property>
             </bean>
             </beans>
```

Messaage.java

```
public class Message {


      private int messgeId;

      private String message;

      public int getMessgeId() {
            return messgeId;
      }
```

```
	public void setMessgeId(int messgeId) {
		this.messgeId = messgeId;
	}

	public String getMessage() {
		return message;
	}

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



}
```

Client.java (main class)

```
public class ClientTest {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub

		Resource resource = new ClassPathResource("beans.xml");

		BeanFactory factory = new XmlBeanFactory(resource);

		Object object = factory.getBean("message");

		if(object!=null){
			Message message = (Message)object;
			System.out.println(message.getMessgeId()+"
"+message.getMessage());
		}




	}

}
```

Description of each method of BeanFactory

```
System.out.println("description about each method of beanFactory");

System.out.println("-------getBean(string, class)--no need to type cast-----");

		Message message = factory.getBean("message", Message.class);


	System.out.println("--aliase method return all aliase name of bean ");
		String aliase[]=factory.getAliases("message");
```

```
            for(String obj:aliase){
                  System.out.println(obj);

            }

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

            Message message1 = factory.getBean(Message.class);

            System.out.println(message1.getMessage()+"
"+message1.getMessgeId());

System.out.println("----getType(string)---return full path of bean class----");

            System.out.println(factory.getType("message"));

      System.out.println("check is singletone or not bydfault is singltone");

            System.out.println(factory.isSingleton("message"));

            System.out.println("-----------check is prototype----");


            System.out.println(factory.isPrototype("message"));
```

----------**2. ApplicationContenxt Container**----------

the most commonly used ApplicationContext implementation are:

ClassPathXmlAplicationContext ---> this container loads the definations of the beans from an XML file. This container will look bean configuration XML file in CLASSPATH

FileSystemXmlApplicationContext---> this container loads the definition of the beans from an XML file, Here you need to provide the full/relative path of the XML bean configuration file to the constructor.

WebXmlApplicationContext:--> this container loads the XML file with definition of all beans from within a web application

A sample code for application context instantiation will look like this.

ApplicationContenxt context = new ClassPathXmlAplicationContext();
Message message = (Message)context.getBean(“message”);

----------**Bean Scope**----------

The core of Spring framework is it's bean Factory and Mechanisms to create and manage such beans inisde Spring container. The beans in spring container can be created in seven scope

Following are the spring beans scopes
1.singleton
2.prototype
3.request
4.session
5.application
6.global-session
7. websocket

Singleton

It return a single bean instance **per Spring IoContainer.** This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object if no bean scope is specifie in bean configuation file then default scope is singleton

prototype:

it return a new bean instance each time when requested, It has not store any cache version like singleton

request:
Scope a single bean defination to the lifecycle of a **single HTTP request**; that is each every HTTP request will have its own instance of a bean created off the back of a single bean definition only valid in the context of a web-aware spring applicationContext

session:
Scope in single bean definition to the **lifecycle of a HTTP Session.** Only valid in the context of a web-aware SpringApplicationContext

application
Scope a single bean definition to the **lifecycle of aServeletContext.** only valid in the context of a web-aware Spring ApplicationContext

Global session
it return a single bean instance **per global HTTP session** .it is only valid the context of a web-aware Spring ApplicationContext

Websocket
Scope a single bean definition to the **lifecycle of a webSocket**. Only valid in the context of a web-aware Spring ApplicationContext.

----------Bean Scope using Annotaton configuration----------

```
@Component // Employee employee = new Employee();
//@Scope("singleton")//by default
@Scope("prototype")
public class Message {

	private int messageId;
	private String message;

}
```

in message class annotated with Componet which is created the message object like Message message = new Messgage();
@Component annotation marks a java class as a bean so the component-scanning mechanism of spring can pick up it and pull it into the application context. To use this annotation,

Indicates that an annotated class is a "component". Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.

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.xsd">


  <context:component-scan base-package="com.infotech.model"></context:component-scan>
 </beans>
```

MainCLass

```
	public static void main(String[] args) {

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

		Message message = ctx.getBean("message", Message.class);

		message.setMessageId(1001);
		message.setMessage("Hello!!");
```

```
            System.out.println(message.getMessageId()
+"\t"+message.getMessage());

            Message message1 = ctx.getBean("message", Message.class);

            System.out.println(message1.getMessageId()
+"\t"+message1.getMessage());

            //((AbstractApplicationContext) ctx).close();




      }
```

----------SPRING DEPENDENCY INJECTION----------

every java based application has a few object that work together to present what the end user see as a working application , when writing java complex application, then application class should have as Independency as possible of other java class ,
To increase the possiblity of reuse these class and to test them independly of other class while doing unit testing,
dependency injection help to when these classes together and same time keeping them independent,
in spring object define there association or dependency and do not worry about how to get those dependency, know it is the responsbility of spring to provide the required dependncy for creating object by dependency injection , the responsiblity of creating object shiffted from our application code to spring container hence is called IOC container ,
spring help us creating loosly coupled application becaued of dependency injections

----------Constructor injection----------

Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on the other class.

Example

```
public class User {


   private String name;
```

```
  private int age;

  private String country;



  User(String name, int age, String country)

  {

    this.name=name;

    this.age=age;

    this.country=country;

  }



  public String toString() {

    return name + " is " + age + " years old, living in " + country;

  }

}
```

The *User* bean class has three attributes viz. *name, age and country*. All the three attributes are set thru constructor injection. The *toString()* method of the *User* bean class is overridden to display the user object.

Here the *beans.xml* file is used to do spring bean configuration. The following code shows how to set a property value thru constructor injection.

<?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="user" class="com.vaannila.User" >

    <constructor-arg value="Eswar" />

    <constructor-arg value="24"/>

    <constructor-arg value="India"/>
```

```
  </bean>

</beans>
```

The *constructor-arg* element within the *bean* element is used to set the property value thru constructor injection. Since there is only one constructor in the *User* bean class, this code will work fine. When there is more than one constructor with the same number of arguments, then the following ambiguities will occur. Conside the following code.

<bean id="user" class="com.vaannila.User" >

```
  <constructor-arg value="24"/>

  <constructor-arg value="India"/>

</bean>



</pre>

<p>

Now which constructor do you think will be invoked? The first one with the int and the String argument, right? But for your surprise it will call the second constructor with both String arguments. Though we know the first argument is of type int and the second argument is of type String, spring interprets both as String arguments.  To avoid this confusion you need to specify the <em>type</em> attribute of the <em>constructor-arg</em> element. Now with the following bean configuration, the first constructor will be invoked. </p>

<pre class="brush: xml; toolbar: false;">

<bean id="user" class="com.vaannila.User" >

  <constructor-arg type="int" value="24"/>

  <constructor-arg type="java.lang.String" value="India"/>

</bean>
```

The *constructor-arg* element within the *bean* element is used to set the property value thru constructor injection. Since there is only one constructor in the *User* bean class, this code will work fine. When there is more than one constructor with the same number of arguments, then the following ambiguities will occur. Conside the following code.

```
User(String name, int age)
```

```
{
  this.name=name;
  this.age=age;
}

User( int age, String country)
{
  this.age=age;
  this.country=country;
}
```

```
<bean id="user" class="com.vaannila.User" >
  <constructor-arg type="int" value="24"/>
  <constructor-arg type="java.lang.String" value="India"/>
</bean>
```

Now which constructor do you think will be invoked? The first one with the int and the String argument, right? But for your surprise it will call the second constructor with both String arguments. Though we know the first argument is of type int and the second argument is of type String, spring interprets both as String arguments. To avoid this confusion you need to specify the *type* attribute of the *constructor-arg* element. Now with the following bean configuration, the first constructor will be invoked.

```
  <bean id="user" class="com.vaannila.User" >
<constructor-arg index="0" type="int" value="24"/>
  <constructor-arg index="1" type="java.lang.String" value="India"/>
</bean>
```

----------Setter Injection----------

The Spring IoC container also supports setter injection, which is the preferred method of dependency injection in Spring. Setter injection uses the set* methods in a class file to garner property names that are configurable in the spring XML config.

From a configuration standpoint, setter injection is easier to read because the property name being set is assigned as an attribute to the bean, along with the value being injected.

----------Inheritance----------

A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on. A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed. Using parent and child bean definitions can save a lot of typing. Effectively, this is a form of templating.

If you work with an ApplicationContext interface programmatically, child bean definitions are represented by the ChildBeanDefinition class. Most users do not work with them on this level, instead configuring bean definitions declaratively in something like theClassPathXmlApplicationContext. When you use XML-based configuration metadata, you indicate a child bean definition by using the parentattribute, specifying the parent bean as the value of this attribute.

```
  <bean id="inheritedTestBean" abstract="true"
     class="org.springframework.beans.TestBean">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</bean>

<bean id="inheritsWithDifferentClass"
      class="org.springframework.beans.DerivedTestBean"
      parent="inheritedTestBean" init-method="initialize">

  <property name="name" value="override"/>
  <!-- the age property value of 1 will be inherited from  parent -->

</bean>
```

A child bean definition uses the bean class from the parent definition if none is specified, but can also override it. In the latter case, the child bean class must be compatible with the parent, that is, it must accept the parent's property values.

A child bean definition inherits constructor argument values, property values, and method overrides from the parent, with the option to add new values. Any initialization method, destroy method, and/or static factory method settings that you specify will override the corresponding parent settings.

The remaining settings are always taken from the child definition: depends on, autowire mode, dependency check, singleton, scope, lazy init.

The preceding example explicitly marks the parent bean definition as abstract by using the abstract attribute. If the parent definition does not specify a class, explicitly marking the parent bean definition as abstract is required, as follow

```
  <bean id="inheritedTestBeanWithoutClass" abstract="true">
 <property name="name" value="parent"/>
   <property name="age" value="1"/>
</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"
   parent="inheritedTestBeanWithoutClass" init-method="initialize">
  <property name="name" value="override"/>
  <!-- age will inherit the value of 1 from the parent bean definition-->
</bean>
```

The parent bean cannot be instantiated on its own because it is incomplete, and it is also explicitly marked as `abstract`. When a definition is `abstract` like this, it is usable only as a pure template bean definition that serves as a parent definition for child definitions. Trying to use such an `abstract` parent bean on its own, by referring to it as a ref property of another bean or doing an explicit `getBean()` call with the parent bean id, returns an error. Similarly, the container's internal `preInstantiateSingletons()` method ignores bean definitions that are defined as abstract.

Note

`ApplicationContext` pre-instantiates all singletons by default. Therefore, it is important (at least for singleton beans) that if you have a (parent) bean definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the *abstract* attribute to *true*, otherwise the application context will actually (attempt to) pre-instantiate the `abstract` bean.

----------Method injections----------

```
public abstract class TicketVendingMachine {

public abstract Ticket getTicketInstance();

}
```

```
public class Ticket {


	public String ticketPrint(){
		return "ticket has been printed ";
	}

}
```

```
  <bean id="ticketVendingMachine"
class="com.infotech.model.TicketVendingMachine" scope="singleton">

		<lookup-method name="getTicketInstance" bean="ticket"/>

		</bean>

		 <bean id="ticket" class="com.infotech.model.Ticket"
scope="prototype">
```

```
public class ClientTest {
public static void main(String args[]){

AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");


TicketVendingMachine tvm =
context.getBean("ticketVendingMachine",TicketVendingMachine.class);

		Ticket ticket = tvm.getTicketInstance();

		System.out.println(ticket.ticketPrint());

		context.close();



	}
}
```

----------Bean LifeCyle using CallBack Interface----------
1) Programatic approach
2)Xml File Confiuration (init-method="" and destory="") using these attribute
3)Annotation(@PostConstruct and @PreDestroy)

----------Programatic approach----------

InitializingBean and **DisposableBean** are interface which is provided by the spring

InitializingBean have one method **afterPropertiesSet**() which is execute after and bean class instantiated and bean class property value set this method we can use for execute for database connection logic or after bean initiated we want to excecute some logic

DisposableBean have one method **public void destroy**() which is executed after bean class is destory so we can put the logic for cleanup code here

Note --> impelmenting the interface in every bean class it is not good approach so avoid these problem we can go for xml configuration

```
public class Message implements InitializingBean,DisposableBean {
	private int id;
	private String message;

@Override
	public void afterPropertiesSet() throws Exception {

	System.out.println("InitializingBean(I)-put the intialized logic here");

	}
	@Override
	public void destroy() throws Exception {

System.out.println("DisposableBean(I)----put logic here for destroying bean time like close some connction");

	}
}
```

---------- Xml File Confiuration----------

---------- xml file----------

```
 <bean id="message" class="com.infotech.message.Message" init-
method="custonInit" destroy-method="customDestroy">
		<property name="id" value="10"/>
		<property name="message" value="hello every one"/>
		</bean>
```

----------Bean class----------

```
public class Message {
       private int id;
       private String message;


      public void customInit() throws Exception {

      System.out.println(put the intialized logic here");

      }

      public void customDestroy() throws Exception {

System.out.println(" logic here for destroying bean time like close some
connction");

      }
```

---------- Annotation----------

enable annotation in sprng we have to configure in xml file

search ---> ctrl+shift+T ---> search
--->**org.springframework.context.annotation.CommonAnnotationBeanPostProcessor**-->

and configure in xml file like thse

```
 <bean
class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"
></bean>
```

so spring can able to create PostConstructor and PreDestory annotation in bean class

----------xml file----------

```
 <bean
class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"
></bean>
                  or

<context:annotation-config/>
```

----------bean class----------

```
public class Message  {


       private int id;
       private String message;

      @PostConstruct
      public void custonInit() throws Exception {

            System.out.println("------InitializingBean(I)-----put the intialized
logic here ------------");
```

```
	}
	@PreDestroy
	public void customDestroy() throws Exception {

		System.out.println("--------------DisposableBean(I)----put logic here for destroying bean time like close some connction----");

	}
}
```

------------------------Collection Injections--------------------

Element	Description
<array>	This is helps in wiring or injecting a array of values with any datatype, allowing duplicationes.
<list>	This is helps in wiring or injecting a list of values with any datatypes, allowing duplicate
<set>	This is helps in wiring or injecting a set of values with any datatypes, without duplicate
<map>	This is helps in wiring or injecting a collection of name-value pairs where name and value can be any datatype
<props>	This is helps in wiring or injecting a collection of name-value pairs where name and value both are string

--------------------INJECTING NULL AND EMPTY STRING VALUES------------

if you need to pass an empty string as a value then you can pass it as follow

```
<bean id=”myBean” class=”MyBean”>
   <property name=”userName” value=””> </property>
  </bean>
```

The preceding example is equivalent to the java code: myBean.setUerName(“”);

if You wantto pass an NULL value then you can pass it as follow

```
<bean id=”myBean” class=”MyBean”>
<property name=”userName”> <null/></property>
</bean>
```

the preceding example is equivalent to the java code

```
myBean.setUserName(null);
```

Example1

----------xml file----------

```
            <bean id="orgInfo" class="com.infotech.info.OrgInfo">
            <property name="nameArr">
            <array>
            <value>A</value>
            <value>A</value>
            <value>B</value>
            <value>A</value>
            <value>C</value>
            </array>
            </property>

            <property name="empNameList">
            <list>
            <value>list A</value>
            <value>list A</value>
            <value>list B</value>
            <value>list A</value>
            <value>list C</value>
            </list>
            </property>

            <property name="empIdss">
            <set>
            <value>10</value>
            <value>20</value>
            <value>30</value>
            <value>10</value>
            <value>40</value>
            </set>
            </property>

            </bean>
```

----------Bean Class----------

```
public class OrgInfo {
      private String nameArr[] = new String [5];
      private List<String> empNameList;
      private Set<Integer> empIdss;

      public String[] getNameArr() {
            return nameArr;
      }
      public void setNameArr(String[] nameArr) {
            this.nameArr = nameArr;
      }
      public List<String> getEmpNameList() {
            return empNameList;
      }
      public void setEmpNameList(List<String> empNameList) {
            this.empNameList = empNameList;
```

```
	}
	public Set<Integer> getEmpIdss() {
		return empIdss;
	}
	public void setEmpIdss(Set<Integer> empIdss) {
		this.empIdss = empIdss;
	}
}
```

Example 2 with Custom data type (Studnet)

instance of **<value>** attribute we can use **<ref bean=””/>** attribute
<ref bean="student1"/>

```
----------------------------beans-------------------------------
public class Student {

	private Integer studentId;
	private String studnetName;
	private String email;
	private String gender;

//generate getter and setter

}

public class OrgInfo {


	private Student nameArr[] = new Student [5];
	private List<Student> empNameList;
	private Set<Student> empIdss;

//generate getter and setter

}

-----------------------------xml-------------------------------


     <bean id="student1" class="com.infotech.modal.Student">
		<property name="studentId" value="120"></property>
		<property name="studnetName" value="name1"></property>
		<property name="email" value="email1"></property>
		<property name="gender" value="gender"></property>
	      </bean>

		<bean id="student2" class="com.infotech.modal.Student">
		<property name="studentId" value="130"></property>
		<property name="studnetName" value="name2"></property>
		<property name="email" value="email2"></property>
		<property name="gender" value="gender"></property>
	      </bean>



		<bean id="orgInfo" class="com.infotech.info.OrgInfo">
		<property name="nameArr">
		<array>
		  <ref bean="student1"/>
		  <ref bean="student2"/>
		  </array>
```

```
          </property>

         <property name="empNameList">
          <list>
         <ref bean="student1"/>
            <ref bean="student2"/>
              </list>
          </property>

          <property name="empIdss">
          <set>
       <ref bean="student1"/>
             <ref bean="student2"/>
          </set>
          </property>
        </bean>
```

----------Map Injection----------

for Wrapper class

```
<map>
<entry key="" value=""></entry>
</map>
```

for Custome class

```
<map>
<entry key-ref="" value-ref=""></entry>
</map>
```

Exampel

```
public class Student {

      private Integer studentId;
      private String studnetName;
      private String email;
      private String gender;

//genrete getter and setter

}

public class OrgInfo {


      Map<Integer, String>mapValue;
      Map<Integer,Student>customStundent;

//genrete getter and setter
}
```

```
-------------------------------xml file--------------------
```

```
          <bean id="student1" class="com.infotech.modal.Student">
           <property name="studentId" value="120"></property>
           <property name="studnetName" value="name1"></property>
           <property name="email" value="email1"></property>
           <property name="gender" value="gender"></property>
          </bean>

           <bean id="student2" class="com.infotech.modal.Student">
           <property name="studentId" value="130"></property>
           <property name="studnetName" value="name2"></property>
           <property name="email" value="email2"></property>
           <property name="gender" value="gender"></property>
          </bean>



           <bean id="orgInfo" class="com.infotech.info.OrgInfo">
           <property name="mapValue">
           <map>
           <entry key="10" value="a"></entry>
           <entry key="20" value="b"></entry>
           <entry key="30" value="c"></entry>
           </map>
           </property>


             <property name="customStundent">
           <map>
           <entry key="111" value-ref="student1"></entry>
           <entry key="222" value-ref="student2"></entry>
           </map>
           </property>
           </bean>
```

SPRING – BEANS AUTO WIRING

AutoWiring Modes:
There are following autowiring modes which can be used to instruct Spring container to user autowiring for dependency injection. We use the autowire attribute of the <bean/>element to specify autowire mode for bean definition.

Mode	Description
no	this is default setting which means no autowiring and you should use explicit bean reference for wiring. You have nothing to do special for this wiring. This is what you already have seen in dependency injection.
ByName	Autowiring by property name. Spring container looks at the properties of the beans on which autowire attribute is set to byName in the xml configuration file. It then tries to match and wire its properties with the beans defined by the same names in the configuration file.
ByType	Autowiring by property datatype. Spring contaier looks at the properties of the beans on which autowire attribute is set to byType in the XML configuration file. It then tries to match and wire a property in the XML configuration file. If more than one such beans exists, a fatal exception is thrown.

Constructor	Similar to byType, but type applies to constructor arguments, if there is no exactly one bean of the cosntructor argument type in the container, a fatal error is raised.
AutoDetect	Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire by byType. This is removed from spring 4.x

----------LIMITATIONS WITH AUTOWIRING----------

Limitations	**Description**
Overrding possibility	You can still specify dependencies using<constructor-arg> and <property> setting which will always override autowiring
Primitive data type	You cannot autowire simple properties such as primities and Strings
Confusing nature	Auto wiring is less exact then explicit wiring, so if possible prefer using explicit wiring.

Examle AutoWire ByName

----------XML----------

```
 <bean id="employee" class="com.infotech.modal.Employee" autowire="byName">
           <property name="employeeId" value="120"></property>
           <property name="employeeName" value="jagasan"></property>
           <property name="email" value="jk@gmail.com"></property>

          </bean>

           <bean id="pancard" class="com.infotech.modal.Pancard">
           <property name="panCardNo" value="10000"></property>
           <property name="pandHolderName" value="rajeev gandhi"></property>
          </bean>
```

----------Beans----------

```
public class Employee {

      private int employeeId;
      private String employeeName;
      private String email;

      private Pancard panCard;

}
```

```
public class Pancard {

	private int panCardNo;

	private String  pandHolderName;

}
```

```
----------------------------Controller-------------------

public static void main(String args[]){

		AbstractApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");

		Employee emp =context.getBean("employee",Employee.class);

		System.out.println(emp);

	}
```

Note:

```
1 byName autowiring property name and id name must be same otherwise it wont be
inject
2.we can not create same id twice it throws pancard already exists
```

```
Example Employee{

Pancard pancard
}
```

```
<bean id=”employee” class=”com.infotech.Employee” autowire=”ByName”>
</bean>

<bean id=”pancard” class=”com.infotech.Pancard”>
</bean>
```

```
-------------------------AutoWiring by ByType-------------------

 <bean id="employee" class="com.infotech.modal.Employee" autowire="byType">
		<property name="employeeId" value="120"></property>
		<property name="employeeName" value="jagasan"></property>
		<property name="email" value="jk@gmail.com"></property>

	      </bean>

		<bean id="pancard" class="com.infotech.modal.Pancard">
		<property name="panCardNo" value="10000"></property>
		<property name="pandHolderName" value="rajeev gandhi"></property>
	      </bean>
```

Other beans and controller are same

Note: we can not create dependency been twice it lead to exception like

```
  <bean id="pancard" class="com.infotech.modal.Pancard">
          <property name="panCardNo" value="10000"></property>
          <property name="pandHolderName" value="rajeev gandhi"></property>
         </bean>

  <bean id="pancard1" class="com.infotech.modal.Pancard">
          <property name="panCardNo" value="10000"></property>
          <property name="pandHolderName" value="rajeev gandhi"></property>
         </bean>
```

Exception

```
Exception encountered during context initialization - cancelling refresh
attempt: org.springframework.beans.factory.UnsatisfiedDependencyException: Error
creating bean with name 'employee' defined in class path resource [beans.xml]:
Unsatisfied dependency expressed through bean property 'panCard': No qualifying
bean of type [com.infotech.modal.Pancard] is defined: expected single matching
bean but found 2: panCard,panCard1; nested exception is
org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying
bean of type [com.infotech.modal.Pancard] is defined: expected single matching
bean but found 2:
```

----------Constructor autowiring----------

```
 <bean id="employee" class="com.infotech.modal.Employee" autowire="constructor">
          <constructor-arg name="employeeId" value="120"/>
          <constructor-arg name="employeeName" value="jagasan"/>
          <constructor-arg name="email" value="jk@gmail.com"/>

         </bean>

          <bean id="panCard" class="com.infotech.modal.Pancard">
          <property name="panCardNo" value="10000"></property>
          <property name="pandHolderName" value="jagasan rajeev"></property>
         </bean>
```

// other beans and controller are same

----------SPRING EVENT HANDLING----------

ApplicationContext, which manages the complete life cycle of the beans. The ApplicationContext publishes certain types of events when loading the beans. For example, a ***ContextStartedEvent*** is published when the context is started and ***ContextStoppedEvent*** is published when the context is stopped.

Event handling in the *ApplicationContext* is provided through the *ApplicationEvent* class and ***ApplicationListener*** interface. Hence, if a bean implements the ***ApplicationListener***, then every time an ***ApplicationEvent*** gets published to the **ApplicationContext**, that bean is notified.

```
Create bean for lisening the event by implements ApplicationListner implements
bean


public class ContextStartedEventHandler implements
ApplicationListener<ContextStartedEvent>{

	@Override
	public void onApplicationEvent(ContextStartedEvent cse) {
		// TODO Auto-generated method stub
		System.out.println("--------------ContextStartedEvent
Recived-------------");

		System.out.println(cse.getSource());
		ApplicationContext context = cse.getApplicationContext();
		System.out.println("----------
date-----------"+context.getStartupDate());

	}



}
-----------------------------------------------------------------------

public class ContextStopEventHandler  implements
ApplicationListener<ContextStoppedEvent>{

	@Override
	public void onApplicationEvent(ContextStoppedEvent cse) {
		// TODO Auto-generated method stub

		System.out.println("ContextStoppedEvent ----Recived-----");
		ApplicationContext contxt =cse.getApplicationContext();


		System.out.println(contxt.getApplicationName());
	}



}
```

```
-----------------------------------------------------------------
```

```
public class ClientTest {


      public static void main(String args[]){

            ConfigurableApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");

            context.start();

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

            Message emp =context.getBean("message",Message.class);

            System.out.println(emp.getMessageId()+"\t"+emp.getMessage());

            System.out.println("-----------------------------------");
            context.stop();
            context.close();

      }
}
```

```
---------------------------------XML--------------------------
```

```
          <bean id="message" class="com.infotech.modal.Message">
           <property name="messageId" value="120"/>
           <property name="message" value="jagasan"/>
          </bean>


          <bean
class="com.infotech.event.handler.ContextStartedEventHandler"></bean>
          <bean
class="com.infotech.event.handler.ContextStopEventHandler"></bean>
```

```
---------------------------Bean-------------------------------
```

```
public class Message {

private int messageId;
private String message;

//getter and setter

}
```

----------Custom event listner----------

Spring allows to create and publish custom events which – by default – **are synchronous**. This has a few advantages – such as, for example the listener being able to participate in the publisher's transaction context.

`Procedure to create custom evnet`

1)the event should extend ApplicationEvent

2)the publisher should inject an ApplicationEventPublisher object

3)the listener should implement the ApplicationListener interface

create custom class event by extends ApplicationEvent (main Event class)

```
public class CustomEvent extends ApplicationEvent{

	private static final long serialVersionUID = 1L;

	public CustomEvent(Object source) {
		super(source);

	}


  public  String toString(){
		return "--------call CustomEvent class------";
   }

}
```

---

publish the event by implements ApplicationEventPublisherAware

it create custom event object register the custom event to applicationEventPublisher

```
public class CustomEventPublisher implements ApplicationEventPublisherAware{

	ApplicationEventPublisher applicationEventPublisher;
	@Override
	public void setApplicationEventPublisher(ApplicationEventPublisher
applicationEventPublisher) {
		// TODO Auto-generated method stub

		this.applicationEventPublisher =applicationEventPublisher;
	}


	public void publishEvent(){

		CustomEvent event = new CustomEvent(this);
```

```
            applicationEventPublisher.publishEvent(event);

            }


}
```

CustomEventHandler implements ApplicationListener
customListerner listen the evnet and execute the our custom class event

```
public class CustomEventHandler implements ApplicationListener<CustomEvent>{

      @Override
      public void onApplicationEvent(CustomEvent customEvent) {

      System.out.println("call custom event method-------------");

      }



}
```

```
---------------------------------Main class-----------------
```

publish(call) the custom event in main class

```
public class ClientTest {


      public static void main(String args[]){

AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");

CustomEventPublisher handler =
context.getBean("customEventPublisher",CustomEventPublisher.class);
            handler.publishEvent();
            context.close();
      }
}
```

----------------Spring Factory method injections-----------------

Normally, Spring instantiates a class and performs dependency injection. However, sometimes it may be necessary to instantiate a class via another class (usually called a Factory class). In such a scenario, Spring should not create the class on its own but simply delegate the instantiation to the Factory class.

Spring provides a way to delegate class instantiation to another class by using *factory-method* attribute of *bean* tag.

Essentially, there is a static method defined in the Factory class that creates the instance of the required bean, hence the name factory-method.

Typically, *factory-method* is used in application integration where objects are constructed in a more complex way using a Factory class.

Also, there may be third party libraries that may need to be used within Spring and these third party libraries use Factory classes to instantiate other classes. In such case, the use of *factory-method* attribute greatly simplifies integrability of Spring based applications with third party libraries.

Factory method types

three type of factory methods

1) A **static factory method** that returns instance of **its own** class. It is used in singleton design pattern.

*<bean id="a" **class**="com.javatpoint.A" factory-method="getA"></bean>*

2) A **static factory method** that returns instance of **another** class. It is used instance is not known and decided at runtime.

*<bean id="b" **class**="com.javatpoint.A" factory-method="getB"></bean>*

3) A **non-static factory** method that returns instance of **another** class. It is used instance is not known and decided at runtime.

*<bean id="a" **class**="com.javatpoint.A"></bean>*

<bean id="b" **class**="com.javatpoint.A" factory-method="getB" factory-bean="a"></bean>

```
public class ATM {

	private Printer printer;

	public Printer getPrinter(){
		return printer;
	}

	public void setPrinter(Printer printer){

		this.printer = printer;
	}

	public void printBalanceInformation(String accountNumber){

		this.printer.printBalanceInfo(accountNumber);
	}

}
```

```
-----------------------Factory class----------------

public class PrinterFactory {

	public static Printer getPrinter(){

		return new Printer();
	}

}

-----------------------Printer class---------------------

public class Printer {

	private String accNumber;

	public void printBalanceInfo(String accNumber){

		this.accNumber = accNumber;
		System.out.println("print the balance information "+accNumber);
	}

}

----------------------XML-------------------------------------

  <bean id="atm" class="com.infotech.model.ATM">

	<property name="printer" ref="printer" />
	  </bean>
	 <bean id="printer" class="com.infotech.model.PrinterFactory" factory-
method="getPrinter">
	  </bean>




public class ClientTest {


	public static void main(String args[]){

		AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");


		ATM atm = context.getBean("atm",ATM.class);

		Printer printer = atm.getPrinter();
		printer.printBalanceInfo("myAcc-1245789");



		context.close();
```

```
	}
}
```

----------------Spring p NameSpace-----------------

Spring’s namespace is an alternative to using the property tag, By using p namespace, we can perform dependency injection by directly using the attribute of bean tag instead of using the property tag.

Benift of using p namespace are:

1)p namespace is more compact than property tag
2) Using p namespace reduces the amount of XML required in Spring configuration. The size of spring config using p namespace is typically less than one with using property tag

Notes: it is very important to discuss upfront which of the approaches(property tag or namespace) would be used in a project so that inconsistency of declaration is avoided

In practice, most projects use the property tag, P namespace are typically used reference books where compactness of spring declaration is more valuable due to space constraints.

Example

```
--------------------Beans-------------------------------

public class Student {

	private String name;
	private String age;
	private Course course;

//getter and setter
}

public class Course {

	private String courseName;

//getter and setter

}


----------------------------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"
	xmlns:p="http://www.springframework.org/schema/p"
	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.xsd">
```

```
          <bean id="student" class="com.infotech.model.Student"
          p:name="jack" p:age="20" p:course-ref="course">

          </bean>

       <bean id="course" class="com.infotech.model.Course"
p:courseName="spring">
          </bean>
 </beans>
```

--------------------------Main class-------------------------

```
public class ClientTest {
      public static void main(String args[]){
            AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");
            Student student = context.getBean("student",Student.class);
            System.out.println(student);
            context.close();
      }
}
```

----------------Spring Expression Language----------------

In general, most of the beans declared for dependency injection using spring static and statically defined.

However,in certain advanced situations, there may be a required to perform dependency injection dynamically at runtime. Such dynamic dependency injection is possible in spring using Spring Expression Language

Using Expression Language, we can:

1. **Refer to other beans by id attribute.**
2. **Refer to the properties and invoke methods defined in other beans**
3. **Refer to the static constants and invoke static methods**
4. **Perform Mathematical operation on value.**
5. **Perform Relational and Logical comparisons**
6. **Perform Regular Expression Matching**
7. **Acessing Collections.**

The key elements of syntax of spring Expression Language are:

1 All Spring Expression should be declared inside ${...} or #{..}
2 Any bean can be directly accessed using the id attribute of the bean
3 Members and methods of a bean are accessed using the dot{.}notation. Is similar to the way members and methods are accessed in java language
4. static class is referred by using T{.....}
5. Standard mathematical operations such as +,-,*,/,% etc. Are used on numerical properties similar to java language.
6 Standard relational opertions such as <=, ==, >= etc similar to java language can be used.

7. Logical operations sucha as and, or not should be used.
8. Matching with regular expression is done using the matches keyword.
9. regular expression syntax is similar to corresponding syntax in java language
10. Individual elements within a list are accessed by using [] notation.
11. Filter operations on elements in a list are performed using ?[] notation.
12.Individual elements within a Map are accessed by referring to the corresponding key using [] notation
13. Projection of elements in a List is perormed .![] notation.

Example of
1. Refer to other beans by id attribute.
2. Refer to the properties and invoke methods defined in other beans

--------------------Beans---------------------------------

```
public class Book {

	private int bookId;
	private String bookName;
	public int getBookId() {
		return bookId;
	}
	public void setBookId(int bookId) {
		this.bookId = bookId;
	}
	public String getBookName() {
		return bookName;
	}
	public void setBookName(String bookName) {
		this.bookName = bookName;
```

```
	}
	@Override
	public String toString() {
		return "Book [bookId=" + bookId + ", bookName=" + bookName + "]";
	}

}
```

```
public class BookCollection {


	private List<Book> bookList;

	public List<Book> getBookList() {
		return bookList;
	}

	public void setBookList(List<Book> bookList) {
		this.bookList = bookList;
	}


	public Book getFirstBook(){

		return getBookList().get(0);

	}

}
```

```
public class BookLiberary {


	private List<Book> allBooks;

	private Book firstBook;

	public List<Book> getAllBooks() {
		return allBooks;
	}

	public void setAllBooks(List<Book> allBooks) {
		this.allBooks = allBooks;
	}

	public Book getFirstBook() {
		return firstBook;
	}

	public void setFirstBook(Book getFirstBook) {
		this.firstBook = getFirstBook;
	}


}

------------------------------XML---------------------------------

  <bean id="book1" class="com.infotech.model.Book">
		<property name="bookId" value="101" />
```

```
            <property name="bookName" value="java" />
          </bean>

         <bean id="book2" class="com.infotech.model.Book">
         <property name="bookId" value="102" />
          <property name="bookName" value="spring" />
           </bean>


       <bean id="bookCollection" class="com.infotech.model.BookCollection">
       <property name="bookList">
       <list>
       <ref bean="book1"/>
        <ref bean="book2"/>
       </list>
       </property>
       </bean>

 <bean id="bookLiberary" class="com.infotech.model.BookLiberary">
       <property name="allBooks" value="#{bookCollection.bookList}" />
        <property name="firstBook" value="#{bookCollection.getFirstBook()}"/>
       </bean>

-----------------------Main class-------------------------------
public static void main(String args[]){

            AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");

            BookLiberary bookLiberary =
context.getBean("bookLiberary",BookLiberary.class);

            List<Book> bookList = bookLiberary.getAllBooks();
            Book firstBook = bookLiberary.getFirstBook();

            for(Book book: bookList){
                  System.out.println(book);
            }


            System.out.println(firstBook);
            context.close();
}
```

Example 3 Refer to the static constants and invoke static methods

```
public class RandomNumberGenerator {


      private double randomNumber;
      private double pi;
      public double getRandomNumber() {
            return randomNumber;
      }
      public void setRandomNumber(double randomNumber) {
            this.randomNumber = randomNumber;
      }
      public double getPi() {
            return pi;
      }
```

```
	public void setPi(double pi) {
		this.pi = pi;
	}



}

----------------------------XML-------------------------------------

  <bean id="randomNumber" class="com.infotech.model.RandomNumberGenerator">
	<property name="randomNumber" value="#{T(java.lang.Math).random()}" />
	 <property name="pi" value="#{T(java.lang.Math).PI}"/>
	</bean>

---------------------------Main class-------------------------------


 public static void main(String args[]){

		AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");


		RandomNumberGenerator randomNumber =
context.getBean("randomNumber",RandomNumberGenerator.class);

		System.out.println(randomNumber.getPi());
		System.out.println(randomNumber.getRandomNumber());
		context.close();



	}
```

---------4 Spring Expression Language-Mathematical Operators----------

```
public class Rectangle {


	private int length;

	private int breadth;
//getter and setter

}
```

```
public class PerimitorCalculator {
	private int perimiter;

 //getter and setter
}

--------------------------------xml-----------------------

 <bean id="rectangle" class="com.infotech.model.Rectangle">
```

```
         <property name="length" value="10" />
          <property name="breadth" value="20" />
        </bean>


       <bean id="perimitorCalculator"
class="com.infotech.model.PerimitorCalculator">
       <property name="perimiter" value="#{2*(rectangle.length +
rectangle.breadth)}" />

       </bean>

-------------------main class------------

public static void main(String args[]){

            AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");

            PerimitorCalculator calculator =
context.getBean("perimitorCalculator",PerimitorCalculator.class);
            System.out.println(calculator.getPerimiter());

            context.close();
      }
```

---------5. Perform Relational and Logical comparisons----------

```
public class MarkSheet {


      private String stundentName;
      private Integer marksInMath;
      private Integer marksInPhysics;
      private Integer marksInChemistry;
//getter and setter

}

public class ExamineResult {

private Boolean hasPassed;

private String resultMessage;

//getter and setter
}
```

```
----------------------------XML----------------------------

   <bean id="markSheet" class="com.infotech.model.MarkSheet">

         <property name="stundentName" value="jack" />
          <property name="marksInMath" value="50" />
           <property name="marksInPhysics" value="60" />
          <property name="marksInChemistry" value="70" />
        </bean>


       <bean id="examineResult" class="com.infotech.model.ExamineResult">
       <property name="hasPassed" value="#{markSheet.marksInMath >=33 and
         markSheet.marksInPhysics>=33 and markSheet.marksInChemistry>=33 }" />

        <property name="resultMessage" value="#{markSheet.marksInMath >=33 and
         markSheet.marksInPhysics >=33 and markSheet.marksInChemistry >=33 ?
passedMessage : failedMessage }" />

       </bean>

       <bean id="passedMessage" class="java.lang.String">

       <constructor-arg>
          <value> Congratulations: You have passed!!</value>
       </constructor-arg>
       </bean>

        <bean id="failedMessage" class="java.lang.String">
         <constructor-arg>
       <value>Congratulation: You have Failed</value>
       </constructor-arg>
       </bean>
```

```
------------------------main class-------------------------

public static void main(String args[]){

            AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");

            ExamineResult result =
context.getBean("examineResult",ExamineResult.class);
            System.out.println(result.getHasPassed());
            System.out.println(result.getResultMessage());

            context.close();
}
```

------6.Spring Expression Language-Regular Expressions----------

```
public class Student {


      private String email;
```

```
	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}
}
-------------------------------------------------


public class EmailValidator {

	private Boolean isEmailValide;

	public Boolean getIsEmailValide() {
		return isEmailValide;
	}

	public void setIsEmailValide(Boolean isEmailValide) {
		this.isEmailValide = isEmailValide;
	}

}
```

```
 <bean id="student" class="com.infotech.model.Student">
	<property name="email" value="jagasan.dansena@gmail.com" />
	</bean>

	 <bean id="emailValidator" class="com.infotech.model.EmailValidator">
	<property name="isEmailValide" value="#{student.email matches '[\w]+.[\w]
+@[\w]+.com'}" />
	</bean>
```

```
public static void main(String args[]){

		AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");

		EmailValidator validator =
context.getBean("emailValidator",EmailValidator.class);
		System.out.println(validator.getIsEmailValide());
		context.close();
	}

--------------------7 Accessing Collections--------------------------

public class Student {

	private int marks;
	private String name;

}
```

```
public class StudentListAccessor {

  private Student thirdStudent;

  private List<Student> faliStudentList;
```

```
  private List<String> stundentNameList;
}
```

```
public static void main(String args[]){

            AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");

            StudentListAccessor studentListAccessor =
context.getBean("studentListAccessor",StudentListAccessor.class);

            Student thirdStudent = studentListAccessor.getThirdStudent();

            List<Student> studentList =
studentListAccessor.getFaliStudentList();
            List<String>nameList = studentListAccessor.getStundentNameList();
}
```

```
  <bean id="student1" class="com.infotech.model.Student">
        <property name="name" value="raja" />
        <property name="marks" value="70" />
        </bean>

         <bean id="student2" class="com.infotech.model.Student">
        <property name="name" value="mohan" />
        <property name="marks" value="60" />
        </bean>

          <bean id="student3" class="com.infotech.model.Student">
        <property name="name" value="diviya" />
        <property name="marks" value="80" />
        </bean>
         <bean id="student4" class="com.infotech.model.Student">
        <property name="name" value="rama" />
        <property name="marks" value="25" />
        </bean>

         <bean id="student5" class="com.infotech.model.Student">
        <property name="name" value="jack" />
        <property name="marks" value="30" />
```

```
        </bean>

         <bean id="studentList" class="java.util.ArrayList">
        <constructor-arg>
        <list>
        <ref bean="student1"/>
         <ref bean="student2"/>
          <ref bean="student3"/>
           <ref bean="student4"/>
             <ref bean="student5"/>
        </list>
        </constructor-arg>
        </bean>

        <bean id="studentListAccessor"
class="com.infotech.model.StudentListAccessor">
         <property name="thirdStudent" value="#{studentList[2]}" />
          <property name="faliStudentList" value="#{studentList.?[marks lt
40]}" />
           <property name="stundentNameList" value="#{studentList.![name]}" />


        </bean>
```

```
Map

public class TelephoneDirectoryAccessor {

  int telephoneNumber;

public int getTelephoneNumber() {
      return telephoneNumber;
}

public void setTelephoneNumber(int telephoneNumber) {
      this.telephoneNumber = telephoneNumber;
}

@Override
public String toString() {
      return "TelephoneDirectoryAccessor [telephoneNumber=" + telephoneNumber
                  + "]";
}

}
```

```
<bean id="telephoneDirectory" class="java.util.HashMap">
       <constructor-arg>
       <map>
      <entry key="jack" value="55458454"/>
      <entry key="johan" value="784512"/>
      <entry key="mohan" value="124578"/>
       </map>
       </constructor-arg>
       </bean>

       <bean id="TelAccessor"
class="com.infotech.model.TelephoneDirectoryAccessor">
   <property name="telephoneNumber" value="#{telephoneDirectory['jack']}" />
       </bean>
```

```
public static void main(String args[]){

AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");

            TelephoneDirectoryAccessor teleAccessor =
context.getBean("TelAccessor",TelephoneDirectoryAccessor.class);

            System.out.println(teleAccessor.getTelephoneNumber());

            context.close();
      }

}
```

----------SPRING ANNOTATION----------

https://dzone.com/refcardz/spring-annotations

we can use annotation waring in spring instance of xml configuration it is better and easy way to configure a beans in springs

Rules:---

1 annotation configure in not enable bydefault in spring container so we have enable the annotation we must configure in xml file

```
<?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.xsd">
```

we have must add in xml schema
http://www.springframework.org/schema/context

```
<context:annotation-config/>
```

once <context:annotation-config/> is configured, you can start annotation your code to indicate that spring should automatically wire values into properties, and constructors.

2 annotation waring is apply first then XML configuration if we can use both xml and annotation based configuration then xml is override the annotation

Core Spring annoations

@Autowired
use--> Constructor, Field, Method
Descriptions
Declares a constructor, field, setter method, or configuration method to be autowired by type. Items annotated with @Autowired do not have to be public.

@Configurable
use-->Type
Descriptions
Used with <context:springconfigured> to declare types whose properties should be injected, even if they are not instantiated by Spring. Typically used to inject the properties of domain objects.

@Order
use-->Type, Method, Field
Descriptions
Defines ordering, as an alternative to implementing the org.springframework.core.Ordered interface.
@Qualifier
uses-->Field, Parameter, Type, Annotation Type
Descriptions
Guides autowiring to be performed by means other than by type.
The @Qualifier annotation along with @Autowired can be used to remove the confusion by specifiying which exact bean will be wired.

@Required
uses-->Method (setters)
Description-->Specifies that a particular property must be injected or else the configuration will fail.(must have value if it not throws exceptions)

@Scope
uses-->Type
Description-->
Specifies the scope of a bean, either singleton, prototype, request, session, or some custom scope.

----------Stereotyping Annotations----------

These annotations are used to stereotype classes with regard to the application tier that they belong to. Classes that are annotated with one of these annotations will **automatically be registered in the Spring application** context if **<context:component-scan>** is in the Spring XML configuration.

In addition, if a PersistenceExceptionTranslationPostProcessor is configured in Spring, any bean annotated with @Repository will have SQLExceptions thrown from its methods translated into one of Spring's unchecked DataAccessExceptions.

@Component
uses-->Type
Descriptions

Generic stereotype annotation for any Spring-managed component.

@Controller
uses-->Type
Descriptions

Stereotypes a component as a Spring MVC controller.

@Repository
uses-->Type
Descriptions

Stereotypes a component as a repository. Also indicates that SQLExceptions thrown from the component's methods should be translated into Spring DataAccessExceptions.

@Service
uses-->Type
Descriptions

Stereotypes a component as a service.

Example of @Required

```
public class Message {


	String messageId;
	String messageName;
	public String getMessageId() {
		return messageId;
	}

	public void setMessageId(String messageId) {
		this.messageId = messageId;
	}
	public String getMessageName() {
		return messageName;
	}
	@Required
	public void setMessageName(String messageName) {
```

```
            this.messageName = messageName;
      }
      @Override
      public String toString() {
            return "Message [messageId=" + messageId + ", messageName="
                        + messageName + "]";
      }

}
```

----------XML----------

```
  <context:annotation-config/>
       <bean id="message" class="com.infotech.model.Message">
        <property name="messageId" value="1245789" />
         <property name="messageName" value="hi every one !!" />
       </bean>
```

```
public static void main(String args[]){

            AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");

            Message message = context.getBean("message",Message.class);

            System.out.println("message id "+message.getMessageId()+ "  message
"+message.getMessageName());

            context.close();
      }
```

----------@Autowired----------

autowired annotation create the object of dependecy class and inject them

```
public class Employee {


      private int employeeId;
      private String employeeName;
      @Autowired(required=false)
      private PanCard pancard;

//getter and setter

}

public class PanCard {
```

```
      public String panCardNumber;
      public String panCardHolder;

//getter and setter

}

-------------------------XML file------------------

  <context:annotation-config/>

       <bean id="employee" class="com.infotech.model.Employee">
        <property name="employeeId" value="1245789" />
         <property name="employeeName" value="Rajesh" />

       </bean>

         <bean id="pancard" class="com.infotech.model.PanCard">
        <property name="panCardNumber" value="PAN-1245789" />
         <property name="panCardHolder" value="Holder details " />

       </bean>
```

Note: in above xml file we are not wired a bean class in employee bean like

```
  <property name="employeeName" ref="pancard" />
```

we are creating object using @Autowired annotation

```
public static void main(String args[]){

            AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");

            Employee employee = context.getBean("employee",Employee.class);
            PanCard panCard = employee.getPancard();

System.out.println(employee.getEmployeeId()+"  "+employee.getEmployeeName());
System.out.println(panCard.getPanCardNumber() + " "+panCard.getPanCardHolder());
      }
```

Note:---

1) @Autowire annotaton injecting time serach instance variable equals id in xml file if instance variable and id not match then check Class Type is avilable in configuration file if both are not match then throws exceptions

2)@Autowire are apply only reference type not primitve type

Example Constructor @Autowired

other is same as previous some point only changed in below

```
public class Employee {


      private int employeeId;
      private String employeeName;
```

```
	private PanCard pancard;

	public Employee(){

	}

	@Autowired
	public Employee(int employeeId, String employeeName, PanCard pancard) {
		super();
		this.employeeId = employeeId;
		this.employeeName = employeeName;
		this.pancard = pancard;
	}

}
```

----------xml----------

```
 <context:annotation-config/>

	<bean id="employee" class="com.infotech.model.Employee">
	 <constructor-arg name="employeeId" value="1245789" />
	  <constructor-arg name="employeeName" value="Rajesh" />

	</bean>

	  <bean id="pancard" class="com.infotech.model.PanCard">
	 <property name="panCardNumber" value="PAN-1245789" />
	  <property="panCardHolder" value="Holder details " />

	</bean>
```

----------@Qualifier----------

qualifer annotation we can use when configure file Same Type(Same class Name twice) then ambigutey probem be occure because Same class type is represent twice so avoid the ambiguty problem we can use **@Qualifer(bean id)** to sepreate the same type by using bean id

@qualifer alway use with @Autowired we can not use along

example ambiguty problem

bean class

```
public class Employee {


	private int employeeId;
	private String employeeName;


	@Autowired
```

```
      private PanCard pancard;
//setter and getter
}
-----------------------------XML file--------------------------
```

we are using Same Type(Same Class name) twice in id are not match in instance variable name in Employee class

```
 <bean id="employee" class="com.infotech.model.Employee">
       <constructor-arg name="employeeId" value="1245789" />
        <constructor-arg name="employeeName" value="Rajesh" />

       </bean>

        <bean id="pancard1" class="com.infotech.model.PanCard">
       <property name="panCardNumber" value="PAN-1245789" />
        <property name="panCardHolder" value="Holder details " />

       </bean>

        <bean id="pancard2" class="com.infotech.model.PanCard">
       <property name="panCardNumber" value="PAN-1245789" />
        <property name="panCardHolder" value="Holder details " />

       </bean>
-------------------------Solutions-----------------------------
use quantifer with beans id name to avoid the ambiguty problem
public class Employee {
      private int employeeId;
      private String employeeName;
      @Autowired
      @Qualifier("pancard2")
      private PanCard pancard;
//setter and getter

}

public class PanCard {
      public String panCardNumber;
      public String panCardHolder;
//setter and getter
}

-----------------------------XML file------------------
  <context:annotation-config/>
       <bean id="employee" class="com.infotech.model.Employee">
        <property name="employeeId" value="1245789" />
         <property name="employeeName" value="Rajesh" />

       </bean>

        <bean id="pancard1" class="com.infotech.model.PanCard">
       <property name="panCardNumber" value="PAN-1245789" />
        <property name="panCardHolder" value="Holder details " />

       </bean>

        <bean id="pancard2" class="com.infotech.model.PanCard">
       <property name="panCardNumber" value="PAN-1245789" />
        <property name="panCardHolder" value="Holder details " />
```

```
       </bean>

-----------------------main class--------------------------

      public static void main(String args[]){

            AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");

            Employee employee = context.getBean("employee",Employee.class);
            PanCard panCard = employee.getPancard();


System.out.println(employee.getEmployeeId()+"  "+employee.getEmployeeName());
System.out.println(panCard.getPanCardNumber() + " "+panCard.getPanCardHolder());
            context.close();
      }
```

----------@Resource----------

@Resource annotation as same as **@Autowaired** but difference is resource doen't check Class Type it only check bydefaul bean id and instance variable name must be same
otherwise we can provide explicity name like **@Resource(name="beanId")**

Example

```
public class Employee {


      private int employeeId;
      private String employeeName;

      @Resource(name="pancard2")
      private PanCard pancard;
}

public class PanCard {
      public String panCardNumber;
      public String panCardHolder;
}
```

----------XML----------

```
<bean id="employee" class="com.infotech.model.Employee">
```

```
        <property name="employeeId" value="1245789" />
         <property name="employeeName" value="Rajesh" />

       </bean>

        <bean id="pancard2" class="com.infotech.model.PanCard">
        <property name="panCardNumber" value="PAN-1245789" />
         <property name="panCardHolder" value="Holder details " />

       </bean>
```

```
      public static void main(String args[]){

            AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");

            Employee employee = context.getBean("employee",Employee.class);
            PanCard panCard = employee.getPancard();


System.out.println(employee.getEmployeeId()+"  "+employee.getEmployeeName());
System.out.println(panCard.getPanCardNumber() + " "+panCard.getPanCardHolder());
            context.close();
      }
```

----------COMPONENT ANNOTATION----------

the **@Component** annotation marks a java class as a begning so the component-scanning mechanism of spring can pick it up and pull it into the application contex. To use annotation,
use we are using Component annotation then we are not need to xml configuration of bean
like
<bean id="some Id" class="className">
</bean>

@Component annotation automatically ceate the object of bean and pass to container

apply it overclass as below:

```
@Component
public Employee{

}
```

Example----->

----------Bean----------

```
@Component //Employee employee = new Employee();
```

```
public class Employee {

	@Value("124578")
	private int employeeId;
	@Value("jack")
	private String employeeName;

	@Autowired
	private PanCard pancard;
//setter and getter
}

@Component
public class PanCard {
	@Value("#{employee.employeeName}")
	public String panCardNumber;

	@Value("PAN-1245")
	public String panCardHolder;

}
```

------------------------------XML file----------------------

```
<context:component-scan base-package="com.infotech.model"/>
```

**@Componet scanner scan the base package for creating the object
configure path of our bean class for creating the object by container using
@component**

------------------------------Main clas------------------------------

```
public static void main(String args[]){

		AbstractApplicationContext  context = new
ClassPathXmlApplicationContext("beans.xml");

		Employee employee = context.getBean("employee",Employee.class);
		PanCard panCard = employee.getPancard();


System.out.println(employee.getEmployeeId()+"  "+employee.getEmployeeName());
System.out.println(panCard.getPanCardNumber() + " "+panCard.getPanCardHolder());
		context.close();
	}
```