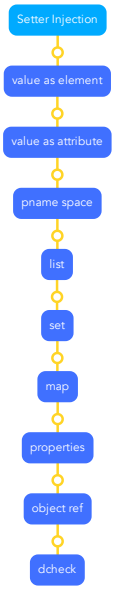
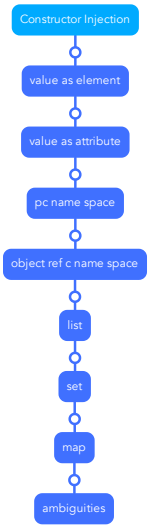
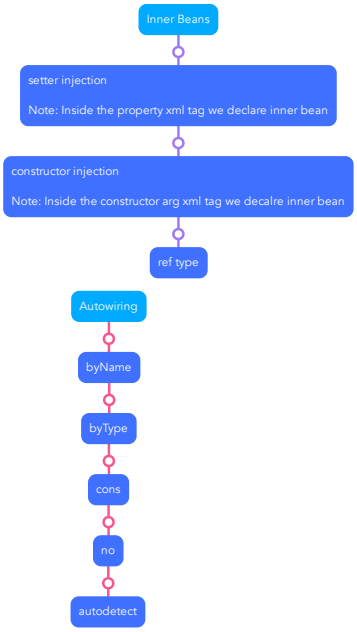
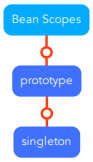
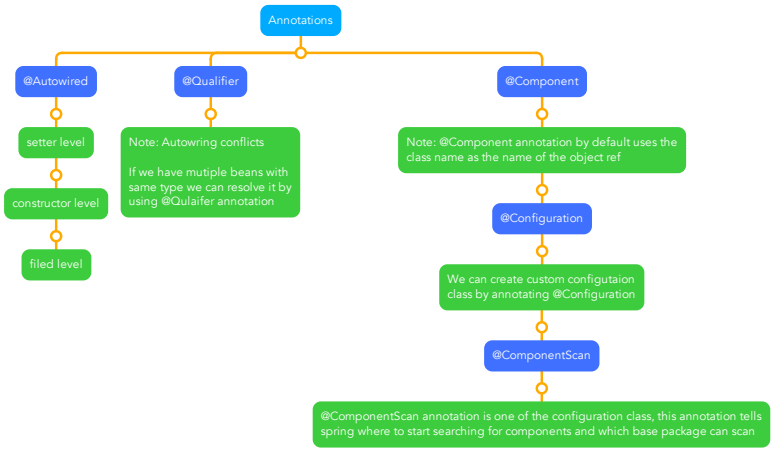
Spring Core


Setter Injection
value as element
value as attribute
pname space
list
set
map
properties
object ref
dcheck


Constructor Injection
value as element
value as attribute
pc name space
object ref c name space
list
set
map
ambiguities


Inner Beans
setter injection
Note: Inside the property xml tag we declare inner bean
constructor injection
Note: Inside the constructor arg xml tag we decalre inner bean
ref type
Autowiring
byName
byType
cons
no
autodetect


Bean Scopes
prototype
singleton


Annotations
@Autowired
setter level
constructor level
filed level
@Qualifier
Note: Autowring conflicts
If we have mutiple beans with same type we can resolve it by using @Qulaifer annotation
@Component
Note: @Component annotation by default uses the class name as the name of the object ref
@Configuration
We can create custom configutaion class by annotating @Configuration
@ComponentScan
@ComponentScan annotation is one of the configuration class, this annotation tells spring where to start searching for components and which base package can scan

Spring Core

1. Bean Scopes

1.1. prototype

1.1.1. singleton

2. Inner Beans

2.1. setter injection Note: Inside the property xml tag we declare inner bean

2.1.1. constructor injection Note: Inside the constructor arg xml tag we decalre inner bean

2.1.1.1. ref type

3. Annotations

3.1. @Autowired

3.1.1. setter level

3.1.1.1. constructor level

3.1.1.1.1. filed level

3.2. @Qualifier

3.2.1. Note: Autowring conflicts If we have mutiple beans with same type we can resolve it by using @Qulaifer annotation

3.3. @Component

3.3.1. Note: @Component annotation by default uses the class name as the name of the object ref

3.3.1.1. @Configuration

3.3.1.1.1. We can create custom configutaion class by annotating @Configuration

3.3.1.1.1.1. @ComponentScan

3.3.1.1.1.1.1. @ComponentScan annotation is one of the configuration class, this annotation tells spring where to start searching for components and which base package can scan

4. Constructor Injection

4.1. value as element

4.1.1. value as attribute

4.1.1.1. pc name space

4.1.1.1.1. object ref c name space

4.1.1.1.1.1. list

4.1.1.1.1.1.1. set

4.1.1.1.1.1.1.1. map

4.1.1.1.1.1.1.1. ambiguities

5. Autowiring

5.1. byName

5.1.1. byType

5.1.1.1. cons

5.1.1.1.1. no

5.1.1.1.1.1. autodetect

6. Setter Injection

6.1. value as element

6.1.1. value as attribute

6.1.1.1. pname space

6.1.1.1.1. list

6.1.1.1.1.1. set

6.1.1.1.1.1.1. map

6.1.1.1.1.1.1.1. properties

6.1.1.1.1.1.1.1.1. object ref

6.1.1.1.1.1.1.1.1.1. dcheck

What is Spring

Spring is a **Dependency Injection** Framework.

Spring is also called as **Light Weight Framework** and it is alternative to **J2EE.**

Spring became popular in Developing **Java Applications.**

Spring was developed by **Rod Johnson.**

Version's in Spring Framework

Version	Date
0.9	2003
1.0/ 1.2.6	2004 /2006
2.0/ 2.5	2006 /2007
3.0/ 3.1/ 3.2.5	2009 /2011 /2013
4.0 / 4.2.0 /4.2.1 /4.3	2013 /2015 /2015 /2016
5.0	2017
6.0	2022

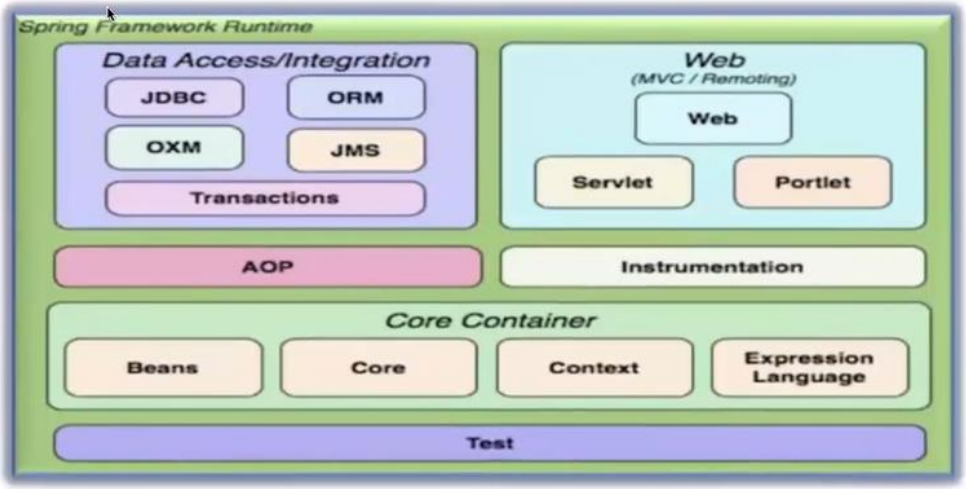
Spring Modules

Spring 1.x and 2.x
1.Spring IOC Module(Core Module)
2.Spring with JDBC Module
3.Spring with ORM Module
4.Spring with J2EE Module
5.Spring with AOP(Aspect Oriented Programming)
6.Spring with MVC Module
7.Spring with Web MVC Module

Spring 3.x and 4.x
1.Spring IOC Module
2.Spring with JDBC Module
3.Spring with ORM Module
4.Spring with J2EE Module
5.Spring with AOP Module(Aspect Oriented Programming)
6.Spring with Web MVC Module

1.x and 2.x we called them as (Spring IOC Module) and
3.x and 4.x we called them as (Dependency Injection Mechanism).

The Spring Framework contains a lot of features, which are well-organized in about twenty modules. These modules can be grouped together based on their primary features into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation and Test.



IS-A Relationship

In IS-A relationship one class is obtaining the features of another class by using Inheritance concept with extends keywords.
It means, that the child class is a type of parent class.

```
public class Honda {
public int sNumber = 101;
public String models = "Honda City, Honda Civic";
}

public class Car extends Honda{
public static void main(String[] args) {
Honda h = new Honda();
System.out.println(h.vno);
System.out.println(h.models);
}
}
```

Console:

```
101
Honda City, Honda Civic
```

HAS-A-Relationship

In Has-A relationship an object of one class is created as data member in another class the relationship between these two classes is Has-A.

```
public class Honda {
int sNumber;
String models;
public Honda(int sNumber, String models) {
this.sNumber = sNumber;
this.models = models; }}

public class Car {
Honda honda; //Created data member in another class
public Car(Honda honda) {
this.honda = honda;
}
public void display(){
System.out.println(honda. sNumber +" "+ honda.models);
}
public static void main(String[] args) {
Honda h1 = new Honda(101, " Honda City ");
Car c = new Car(h1);
c.display();
}
}
```

```
101 Honda City
```

Diff b/w Pojo Class and Bean Class

Pojo Class
Pojo Stands for Plain Old Java Object.
Here Class is **public**.
Properties are **private**.
Default Constructor is **mandatory**.
Need to have **public setters methods and getter methods**.
Now, Pojo class will not implement **Serializable interface.**

```
public class Honda{

}
```

Bean Class
All JavaBeans are POJOs but not all POJOs are JavaBeans.
Here Class is **public**.
Properties are **private**.
Default Constructor is **mandatory**.
Need to have **public setters methods and getter methods.**
Now, Bean class can implement **Serializable Interface**

```
public class Honda implements java.io.Serializable{

}
```

Poji Class

POJI stands for **Plain Old Java Interface**.
A POJI is an normal Interface without any **specialties.**
The Interfaces that do not extend from any technology/framework.

Eg:
For example all user defined Interfaces are **POJI** and an
Interface that inherits from any other API/other Technology is **not POJI.**

```
public interface Honda1 extends java.io.Serializable{
} //Another Poji
```

```
public class Honda2 implements java.io.Serializable{
} //Another Poji
```

Here **Honda1** is a **POJI**.
Here the **Interface** is extending from **Serializable Interface** but that **Serializable Interface** is not the part of any **Technology/Framework.**
It is a **Java API.**
Therefore we can say that **Honda1 is POJI in nature**.

```
public interface Honda3 extends java.rmi.Remote{
}
```

Here Honda3 are not POJI in nature. Because java.rmi.Remote is not part of API
It is Technology/Framework

Types of IoC Containers

We have two types IoC Containers.
They are **Bean Factory Container** and **ApplicationContext Container.**

Bean Factory Container:

1. BeanFactory is a **Interface**
2. Where **XmlBeanFactory** is the implementation class of **BeanFactory.**
3. The **BeanFactory** is the actual container which **instantiates, configures, and manages the number of beans.**
4. It belongs to **org.springframework.beans.factory.BeanFactory Interface.**
5. These beans typically collaborate with one another, and thus have dependencies between themselves.
6. The BeanFactory enables you to read bean definitions and access them using the bean factory.

```
BeanFactory context = new ClassPathXmlApplicationContext("com/dl/applicationContext.xml");
```

Application Context:

1. BeanFactory is the basic container, Where as **Application Context** is the **advanced container.**
2. Application Context **extends** the BeanFactory Interface.
3. Application Context provides more facilities than BeanFactory such as integration with **Spring AOP, message resource handling for i18n etc.**

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("com/dl/applicationContext.xml");
```

ApplicationContext Implementation:

3 commonly used implementations are:

1.FileSystemXmlApplicationContext:
Where these container loads the definitions of the beans from an XML file.
Here, you need to provide the full path of XML bean configuration file.
ApplicationContext context = new FileSystemApplicationContext("bean.xml");

2.ClassPathXmlApplicationContext:
Where these container loads the definitions of the beans from an XML file.
Here, you do not need to provide the full path of the XML file ,
but you need to set the CLASSPATH properly because this container will look bean configuration XML file in CLASSPATH.
ApplicationContext context = new ClassPathXmlApplicationContext t("bean.xml");

3.WebXmlApplicationContext:
This container loads the XML file with definitions of all beans from within a web application.
public interface WebApplicationContext extends ApplicationContext { ServletContext getServletContext(); }

Ways to Configure Spring Application:

We can Configure Spring Application in 3 ways
They are

1.XML based Configuration
2.Annotation based Configuration
3.Java based Configuration

No Dependency Injection

```
public interface Brand {

public String Honda();

}
```

```
public class Bike implements Brand{

@Override
public String Honda() {
return "Honda CBR";
}
}
```

```
public class Car implements Brand {

@Override
public String Honda() {
return "Honda Accord";
}
}
```

```
Car c = new Car();
System.out.println(c.Honda());

Bike b = new Bike();
System.out.println(b.Honda());
}
```

We have simple Bike, Car classes that provides a message
Instead of using Spring Dependency Injection mechanism, we direct create a instance of Bike and Car classes in the main method and call its methods

Dependency Injection

```
public interface Brand {

public String Honda();

}
```

```
public class Bike implements Brand{

@Override
public String Honda() {
return "Honda CBR";
}
}
```

```
public class Car implements Brand {

@Override
public String Honda() {
return "Honda Accord";
}
}
```

Spring XML based Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Define Java Beans here... -->
<bean class="com.dl.Bike" id="id1"></bean>
<bean class="com.dl.Car" id="id2"></bean>
</beans>
```

You can read this ApplicationContext.xml using:

```
ApplicationContext appContext = new ClassPathXmlApplicationContext("ApplicationContext.xml");
```

```
ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("com/dl/applicationContext.xml");
Brand b1 = context.getBean("id1", Brand.class);
Brand b2 = context.getBean("id2", Brand.class);
System.out.println(b1.Honda());
System.out.println(b2.Honda());
context.close();
}
```

Java Based Configuration

Java Based Configuration we use **@Configuration** annotated classes and **@Bean** annotated methods.
The **@Bean** is used to indicate that a method instantiates, configures and initializes a new Object to be managed by Spring IoC Container.
@Bean annotation plays the same role as **</bean>** element.

```
public interface Brand {

public String Honda();

}
```

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;


@Component("id1")
@Configuration
@ComponentScan("com.eagle.javabasedconfiguration")
public class Bike implements Brand{

@Override
@Bean
public String Honda() {
return "Honda CBR";
}
}
```

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

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

// ApplicationContext ctx = new AnnotationConfigApplicationContext(Bike.class);
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(Bike.class);
Bike brand = context.getBean("id1", Bike.class);
System.out.println(brand.Honda());
context.close();
}
}
```

Setter Injection

In Setter Injection we can go for

Injecting Primitive Types as Dependencies
a. Value as Elements and
b. Value as Attributes and
c. P Schema P NameSpace

Injecting Collections Types as Dependencies
a. List,
b. Set,
c. Map
d. Properties

Objects or Reference Types
<ref> ref can used as element or attribute or p:schema inside the bean.

Using Value as a Element

```
<bean name="id1" class="com.dl.valueelement.Honda">
<property name="vno"><value>9876</value>  </property>
<property name="vname"><value>Honda City</value></property>
</bean>
```

Using Value as Attribute

```
<bean name="id1" class="com.dl.valueattribute.Honda">
<property name="vno" value="9876"/>
<property name="vname" value = "Honda City"/>
</bean>
```

Using p schema/p namespace

```
<bean name="id1" class="com.dl.pnamespace.Honda" p:vno="9876" p:vname="Honda Accord"/>
```

```
public class Honda {

private int vno;
private String vname;

public int getVno() {
return vno;
}
public void setVno(int vno) {
this.vno = vno;
}
public String getVname() {
return vname;
}
public void setVname(String vname) {
this.vname = vname;
}
}
```

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/dl/applicationContext.xml");

Honda h = (Honda) ctx.getBean("id1");
System.out.println("Vechile No: " + h.getVno());
System.out.println("Vechile Name: " + h.getVname());
ctx.close();
```

Injecting Collections value as element:

```
public class Honda {

private String sname;
private List<String> models;
//setters  and getters
}
```

```
<bean name="id1" class="com.dl.list.Honda">
<property name="sname">
<value>Fortune Honda</value>
</property>
<property name="models">
<list>
<value>Honda City</value>
<value>Honda Accord</value>
<value>Honda Civic</value>
</list>
</property>
</bean>
```

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/ssit/list/applicationContext.xml");

Honda h = (Honda) ctx.getBean("id1");
System.out.println(h.getSname());
System.out.println(h.getModels());
System.out.println(h.getClass());
ctx.close();
```

Injecting Collection Types value as element

```
public class Honda {

private String sname;
private Set<String> models;
//setters and getters

}
```

```
<bean name="id1" class="com.dl.set.Honda">
<property name="sname" value = "Fotune Honda"/>

<property name="models">
<set>
<value>Honda City</value>
<value>Honda Accord</value>
<value>Honda Civic</value>
</set>
</property>
</bean>
```

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/dl/set/applicationContext.xml");

Honda h = (Honda) ctx.getBean("id1");
System.out.println(h.getSname());
System.out.println(h.getModels());
ctx.close();
```

Injecting Collection Types p schema
To enable the p-namespace feature, we need to add
xmlns:p="http://www.springframework.org/schema/p" into the XML file.

```
import java.util.Map;
public class Honda {

private int vno;
private Map<Integer, String> models;

public int getVno() {
return vno;
}
public void setVno(int vno) {
this.vno = vno;
}
public Map<Integer, String> getModels() {
return models;
}
public void setModels(Map<Integer, String> models) {
this.models = models;
}

@Override
public String toString() {
return "Honda [vno=" + vno + ", models=" + models + "]";
}
}
```

```
<bean name="id1" class="com.dl.map.pschema.Honda" p:vno="9876">

<property name="models">
<map>
<entry key="1" value="Honda City"></entry> <!-- Value as Attribute-->

<entry key="2">
<value>Honda Civic</value> <!-- Value as Element -->
</entry>

<entry>
<key><value>3</value></key> <!-- Value as Element -->
<value>Honda CRV</value> <!-- Value as Element -->
</entry>

</map>
</property>

</bean>
```

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/dl/map/pschema/applicationContext.xml");
Honda h = (Honda) ctx.getBean("id1");
System.out.println(h);
ctx.close();
```

Annotation Based Configuration

Annotation wiring is not turned on in the Spring container by default.
So, before we use annotation-based wiring, we need to enable in our Spring Configuration file.

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

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

@Required: The @Required Annotation applies to bean property setter methods.

@Autowired: The @Autowired annotation can apply to bean property for setter methods, non-setter methods, constructor and properties.

@Qualifier: The @Qualifier annotation along with @Autowired can be used to remove the confusion by specifying which exact bean will be wired.

@Resource, @PostConstruct and @PreDestroy annotations.

Constructor Injection

In Constructor Injection we can go for

Injecting Primitive Types as Dependencies
a. Value as Elements and
b. Value as Attributes and
c. P Schema P NameSpace
d.C Schema C NameSpace

Injecting Collections Types as Dependencies
a. List,
b. Set,
c. Map
d. Properties

Objects or Reference Types
<ref> ref can used as element or attribute or p:schema inside the bean.

Value as Element

```
public class Student {

private int rollno;
private String branch;
private String university;
//parameterized constructor
//toString
}
```

```
public class Location {

private String city;
private String state;
private Student student;
//parameterized constructor
//toString
}
```

Value as element

```
<bean name="id2" class="com.dl.valueaselement.Student">
<constructor-arg><value>9876</value></constructor-arg>
<constructor-arg><value>CSE</value></constructor-arg>
<constructor-arg><value>JNTU</value></constructor-arg>
</bean>

<bean name="id1" class="com.dl.valueaselement.Location">
<constructor-arg><value>HYD</value></constructor-arg>
<constructor-arg><value>TG</value></constructor-arg>
<constructor-arg><ref bean="id2"></ref></constructor-arg>
</bean>
```

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("com/eagle/valueaselement/applicationContext.xml");
Location e = (Location) context.getBean("id1");
System.out.println(e);
context.close();

Location [city=HYD, state=TG, student=Student [rollno=9876, branch=CSE, university=JNTU]]
```

Value as Attribute

```
public class Student {

private int rollno;
private String branch;
private String university;
//parameterized constructor
//toString
}
```

```
public class Location {

private String city;
private String state;
private Student student;
//parameterized constructor
//toString
}
```

Value as attribute

```
<bean name="id2" class="com.dl.valueasattribute.Student">
<constructor-arg value ="9876"/>
<constructor-arg value = "CSE"/>
<constructor-arg value= "JNTU"/>
</bean>

<bean name="id1" class="com.dl.valueasattribute.Location">
<constructor-arg value = "HYD"/>
<constructor-arg value = "TG"/>
<constructor-arg ref ="id2"/>
</bean>
```

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("com/dl/valueasattribute/applicationContext.xml");
Location e = (Location) context.getBean("id1");
System.out.println(e);
context.close();

Location [city=HYD, state=TG, student=Student [rollno=9876, branch=CSE, university=JNTU]]
```

P C Namespace

```
public class Student {

private int rollno;
private String branch;
private String university;
//parameterized constructor
//toString

}
```

```
public class Location {

private String city;
private String state;
private Student student;
//parameterized constructor
//toString

}
```

P namesaoce and C namespace

```
<bean name="id2" class="com.dl.pcnamespace.Student"
p:rollno="9876" p:branch="CSE" p:university="JNTU" />

<bean name="id1" class="com.dl.pcnamespace.Location"
c:city="HYD" c:state="Telanagana" c:student-ref="id2"
/>
```

Should be Matching Names or else get an error

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("com/dl/valueaselement/applicationContext.xml");
Location e = (Location) context.getBean("id1");
System.out.println(e);
context.close();

Location [city=HYD, state=Telanagana, student=Student [rollno=9876, branch=CSE, university=JNTU]]
```

List

```
public class Honda {

private String sname;
private List<String> models;
//parameterized constructor
//toString

}
```

```
<bean name="id1" class="com.dl.list.Honda">

<constructor-arg value="Fortune Honda"></constructor-arg><!-- Value as Attribute -->

<constructor-arg>
<list>
<value>Honda City</value>  <!-- Value as Element -->
<value>Honda Accord</value>
<value>Honda Civic</value>
<value>Honda CRV</value>
</list>
</constructor-arg>
</bean>
```

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/eagle/list/applicationContext.xml");
Honda h = (Honda) ctx.getBean("id1");
System.out.println(h);
ctx.close();

Honda [sname=Fortune Honda, models=[Honda City, Honda Civic, Honda Accord, Honda CRV]]
```

Set

```
public class Honda {

private String sname;
private Set<String> models;
//parameterized constructor
//toString

}
```

```
<bean name="id1" class="com.dl.set.Honda">
<constructor-arg value="Fortune Honda"></constructor-arg>
<constructor-arg>
<set>
<value>Honda City</value>
<value>Honda Accord</value>
<value>Honda Civic</value>
</set>
</constructor-arg>
</bean>
```

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/dl/set/applicationContext.xml");
Honda h = (Honda) ctx.getBean("id1");
System.out.println(h);
ctx.close();

Honda [sname=Fortune Honda, models=[Honda City, Honda Civic, Honda Accord]]
```

Map

```
public class Honda {

private int vno;
private Map<Integer, String> models;

}
```

```
<bean name="id1" class="com.dl.map.Honda">

<constructor-arg value ="9876"/> <!-- Value as Attribute -->

<constructor-arg>
<map>
<entry key="1" value="Honda City" /><!-- Value as Element -->
<entry key="2" value="Honda Accord" />
<entry key="3" value="Honda Civic" />
</map>
</constructor-arg>
</bean>
```

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/dl/map/applicationContext.xml");
Honda h = (Honda) ctx.getBean("id1");
System.out.println(h);
ctx.close();

Honda [vno=101, models={1=Honda City, 2=Honda Civic, 3=Honda Accord}]
```

Ref

```java
public class Models {

private String carName;
private Double cost;
private String generation;
private String type

}
```

```java
public class Honda {

private Models models;
}
```

```
<bean name="models" class="com.dl.ref.cnamespace.Models"
 c:carName="Honda City" c:cost="900000" c:generation="G6" c:type="Manual"/>

<bean name="id1" class="com.dl.ref.cnamespace.Honda" c:models-ref="models"/>
```

```java
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/dl/ref/cnamesapce/applicationContext.xml");
Honda h = (Honda)ctx.getBean("id1");
System.out.println(h);
ctx.close();
```

```
<bean name="id1" class="com.dl.ref.cons.Honda"/>

<bean name="id2" class= "com.dl.ref.cons.Models">
   <constructor-arg name= "carName" value="Honda City"/>
   <constructor-arg name= "cost"  value="900000"/>
   <constructor-arg name ="generation"  value="G6"/>
   <constructor-arg name = "type" value="Manual"/>
</bean>

<bean name="id3" class="com.dl.ref.cons.Honda" parent="id1">
<constructor-arg ref ="id2" />
</bean>
```

```java
Honda h = (Honda)ctx.getBean("id3");
```

Ambiquities:

```java
public class Student {

public Student(int rollno, double stipend, String university) {

System.out.println(rollno);
System.out.println(stipend);
System.out.println(university);
}
}
```

```
<bean
class="com.dl.ambiguities.Student"name="id1">
<constructor-arg value="2000.00" type="double" index="1" name="stipend"/>
<constructor-arg value="9876" type="int" index="0" name="rollno"/>
<constructor-arg value="JNTU" type="String" index="2" name="university"/>

</bean>
```

```java
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/eagle/ambiguities/applicationContext.xml");
Student st = (Student) ctx.getBean("id1");
System.out.println(st.getClass());
ctx.close();
```

```
101
2000.0
JNTU
class com.dl.ambiguities.Student
```

```
Ambiguous argument values for parameter of type [int] - did you specify the correct bean references as arguments?
Exception in thread "main" org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'id1' defined in class path resource
[com/eagle/ambiguities/applicationContext.xml]: Unsatisfied dependency expressed through constructor parameter 0: Ambiguous argument values for parameter
of type [int] - did you specify the correct bean references as arguments?
```

Inner Bean

Inner Bean are defined inside with in the scope of another bean.
Thus, a <bean/> element inside the <property/> or <constructor-arg/> elements is called inner bean.

```
public class HondaCars {

private String carModels;

}
```

```
public class Honda{

HondaCars hondacars;

}
```

```
<bean class="com.eagle.innerbean.si.Honda" name="id1">
<property name="hondacars">
<bean class="com.eagle.innerbean.si.HondaCars" p:carModels="Honda City, Honda Civic"/>
</property>
</bean>
```

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/eagle/innerbean/si/applicationContext.xml");
Honda h = (Honda)ctx.getBean("id1");
System.out.println(h); //Honda [hondacars=HondaCars [carModels=Honda City, Honda Civic]]
ctx.close();
```


Inner Bean

Inner Bean are defined inside with in the scope of another bean.
Thus, a <bean/> element inside the <property/> or <constructor-arg/> elements is called inner bean.


```
public class HondaCars {

private String carModels;

}
```

```
public class Honda{

HondaCars hondacars;

}
```

```
<bean class="com.eagle.innerbean.ci.Honda" name="id1">
<constructor-arg name="hondaCars">
<bean class="com.eagle.innerbean.ci.HondaCars“ p:carModels="Honda City, Honda Civic"></bean>
</constructor-arg>
</bean>
```

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/eagle/innerbean/si/applicationContext.xml");
Honda h = (Honda)ctx.getBean("id1");
System.out.println(h); //Honda [hondaCars=HondaCars [carModels=Honda City, Honda Civic]]
ctx.close();
```

Inner Bean Ref Tag

Now, if we want to refer another bean in the configuration file, then we can go for <ref> tag.

```
<bean id ="id2" class="com.sajeed.innerbean.ref.HondaCars">
<property name="carModels" value="Honda Civic, Honda Accord"></property>
</bean>

<bean id ="id1" class="com.sajeed.innerbean.ref.Honda">
    <property name= "hondacars">
      <ref bean= "id2"/>
    </property>
 </bean>
```

Scopes:

In Spring we have 5 types of Scope.
They are

singleton	Creates only one instance of the bean. Singleton is a default scope.
prototype	Creates a new Instance
request	single bean instance per HTTP request
session	single bean instance per HTTP session
global Session	single bean instance per global HTTP session.

We have two ways to work on Scopes.
One is XML
Two is Annotations

Singleton: Only one instance is created

```
public class Honda {

private int vno;
private String vname;

//setters and getters
}
```

```
<bean name="id1" class="com.eagle.singleton.Honda" scope="singleton">
<property name="vno">
<value>101</value>
</property>
<property name="vname">
<value>Honda City</value>
</property>
</bean>
```

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/eagle/singleton/applicationContext.xml");
Honda h1 = (Honda)ctx.getBean("id1");
System.out.println(h1.getVno());
System.out.println(h1.getVname());
System.out.println(h1.hashCode());

Honda h2 = (Honda)ctx.getBean("id1");
System.out.println(h2.getVno());
System.out.println(h2.getVname());
System.out.println(h2.hashCode());
ctx.close();
```

```
101
Honda City
1433666880
101
Honda City
1433666880
```

Auto wiring
Auto wiring enables you to inject the object dependency implicitly.
It internally uses setter or constructor injection.
Auto wiring can't be used to inject primitive and string values. It works with reference only.

Mode	Description
no	Default is no, It is not auto wired
byName	The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method.
byType	The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.
constructor	The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.
autodetect	It is deprecated since Spring 3.

We have two ways to work on Auto wiring
One is XML Two is Annotations @Autowired, @Qualifier

without byType (same as injecting reference setter injection)

```
<bean name="id2" class="com.eagle.ref.withoutbyType.Models" p:models="Honda City" p:type="Automatic"/>

<bean name="id1" class="com.eagle.ref.withoutbyType.Honda" p:models-ref="id2"/>
```

```
public class Models {

private String models;
private String type;
//setters getters
}
```

```
public class Honda {

private Models models;
//setters getters
}
```

property name and bean name can be different

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/eagle/ref/withoutbyType/applicationContext.xml");
Honda h = (Honda)ctx.getBean("id1");
System.out.println(h);
ctx.close();
```

```
Honda [models=Models [models=Honda City, type=Automatic]]
```

with byType

```
<bean name="id2" class="com.eagle.ref.byType.Models" p:models="Honda City, Honda Civic" p:type="Automatic"/>
<bean name="id1" class="com.eagle.ref.byType.Honda" autowire="byType"></bean>
```

```
public class Models {

private String models;
private String type;
//setters getters
}
```

```
public class Honda {

private Models models;
//setters getters
}
```

property name and bean name can be different

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/eagle/ref/byType/applicationContext.xml");
Honda h = (Honda)ctx.getBean("id1");
System.out.println(h);
ctx.close();
```

```
Honda [models=Models [models=Honda City, Honda Civic, type=Automatic]]
```

with byName

```
<bean name="models" class="com.eagle.ref.byName.Models" p:models="Honda City, Honda Civic" p:type="Automatic"/>
<bean name = "id1" class="com.eagle.ref.byName.Honda" autowire="byName"></bean>
```

```
public class Models {

private String models;
private String type;
//setters getters
}
```

```
public class Honda {

private Models models;
//setters getters
}
```

property name and bean name can be same

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("com/eagle/ref/byType/applicationContext.xml");
Honda h = (Honda)ctx.getBean("id1");
System.out.println(h);
ctx.close();
```

```
Honda [models=Models [models=Honda City, Honda Civic, type=Automatic]]
```

@Autowire at Setter Injection:

```
public class Models {
private String models;
private String type;
//setters getters
}
```

```
import org.springframework.beans.factory.annotation.Autowired;

public class Honda {

private Models models;
public Models getModels() {
return models;
}
@Autowired
public void setModels(Models models) {
this.models = models;
}
```

```
<context:annotation-config/>

<bean name="model" class="com.sajeed.autowire.setterinjection.Models" p:models="Honda City" p:type="Automatic"/>

<bean name="id1" class="com.sajeed.autowire.setterinjection.Honda"/>
```

```
ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext("com/sajeed/autowire/setterinjection/applicationContext.xml");
Honda h = (Honda)ctx.getBean("id1");
System.out.println(h);
ctx.close();
```

```
Honda [models=Models [models=Honda City, Honda Civic, type=Automatic]]
```

disabled @Autowire at Setter Injection:

```
public class Models {
private String models;
private String type;
//setters getters
}
```

```
import org.springframework.beans.factory.annotation.Autowired;

public class Honda {

private Models models;
public Models getModels() {
return models;
}
//@Autowired
public void setModels(Models models) {
this.models = models;
}
```

```
<context:annotation-config/>

<bean name="model" class="com.sajeed.autowire.setterinjection.Models" p:models="Honda City" p:type="Automatic"/>

<bean name="id1" class="com.sajeed.autowire.setterinjection.Honda"/>
```

```
ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext("com/sajeed/autowire/setterinjection/applicationContext.xml");
Honda h = (Honda)ctx.getBean("id1");
System.out.println(h);
ctx.close();
```

```
Honda [models=null]
```

@Autowire at ConstructorInjection:

```
public class Models {
private String models;
private String type;
//setters getters
//default cons
//para cons
}
```

```
import org.springframework.beans.factory.annotation.Autowired;

public class Honda {
private Models models;
//setters getters
//default cons
//para cons
@Autowired
public Honda(Models models) {
this.models = models;
}
```

```
<context:annotation-config/>

<bean name="model" class="com.sajeed.autowire.setterinjection.Models" p:models="Honda City" p:type="Automatic"/>

<bean name="id1" class="com.sajeed.autowire.setterinjection.Honda"/>
```

```
ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext("com/sajeed/autowire/setterinjection/applicationContext.xml");
Honda h = (Honda)ctx.getBean("id1");
System.out.println(h);
ctx.close();
```

```
Honda [models=Models [models=Honda City, Honda Civic, type=Automatic]]
```

@Autowire at Field Level

```
public class Honda {

@Autowired //using on field level
private Models model;

}
```

```
<context:annotation-config/>

<bean name="model" class="com.sajeed.autowire.filedlevel.Models" p:models="Honda City"
p:type="Automatic"/>

<bean name="id1" class="com.sajeed.autowire.filedlevel.Honda"/>
```