ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

Java Technology

1

SPRING FRAMEWORK CORE

Agenda – what to do in this Spring

- Dependency Injection
- Overview of Spring Framework
- Various injections and their usages in detail
- Bean scope
- Bean wiring
- Inner bean
- Bean properties
- Bean life cycle
- Bean auto wiring
- Spring Annotation

High Dependency = High Responsibility


```
public class Tiger {
   public void eat() {
      System.out.println("Do not disturb");
   }
}
```

```
public class MainClass {
   public static void main(String… args) {
      Tiger t = new Tiger();
      t.eat();
   }
}
```

- **Requirement:** Change Tiger to Lion.
- **Me:** Urgggghhh. Too many changes.
  1. Standalone Lion class
  2. Change the object declaration to Lion
  3. Change the reference
  4. Compile again
  5. Test

A bit less dependency


```
public interface Animal {
  void eat();
}
```

```
public class Tiger implements Animal {
  @Override
  public void eat() {
    System.out.println("Do not disturb");
  }
}
```

```
public class MainClass {
  public static void main(String… args) {
    Animal a = new Tiger();
    a.eat();
  }
}
```

- **Requirement:** Change Tiger to Lion.
- **Me:** Ufff. Again some changes.
  1. Lion implements Animal
  2. Change the object declaration to Lion
  3. ~~Change the reference~~
  4. Compile again
  5. Test

Dependency Injection


```
public interface Animal {
 void eat();
}
```

```
public class Tiger implements Animal {
 @Override
 public void eat() {
  System.out.println("Do not disturb");
 }
}
```

```
public class MainClass {
 public static void main(String… args) {
  ApplicationContext aC = new ClassPathXmlApplicationContext("wildLife.xml");
  Animal a = (Animal) aC.getBean("animal");
  a.eat();
 }
}
```

- **Requirement:** Change Tiger to Lion.
- **Me:** Ok. Small change.
  1. Lion implements Animal
  2. Change bean class to Lion
  3. ~~Change the reference~~
  4. ~~Compile again~~
  5. Test

wildLife.xml

```
<bean id = "animal" class = "Tiger" />
```

So what is Spring?

- IoC container
- Lightweight framework
- Fully modularized (High decoupling)
- Considered an alternative / replacement for the Enterprise JavaBean (EJB) model
- Flexible
  - Programmers decide how to program
- Not exclusive to Java (e.g. .NET)
- Solutions to typical coding busywork
  - JDBC
  - LDAP
  - Web Services

What Spring offers?

- Dependency Injection (DI)
  - DI is implemented through IoC (Inversion of Control)
- Aspect Oriented Programming
  - Runtime injection-based
- Portable Service Abstractions
  - The rest of spring
    - ORM, DAO, Web MVC, Web, etc.
    - Allows access to these without knowing how they actually work
- Easily testable

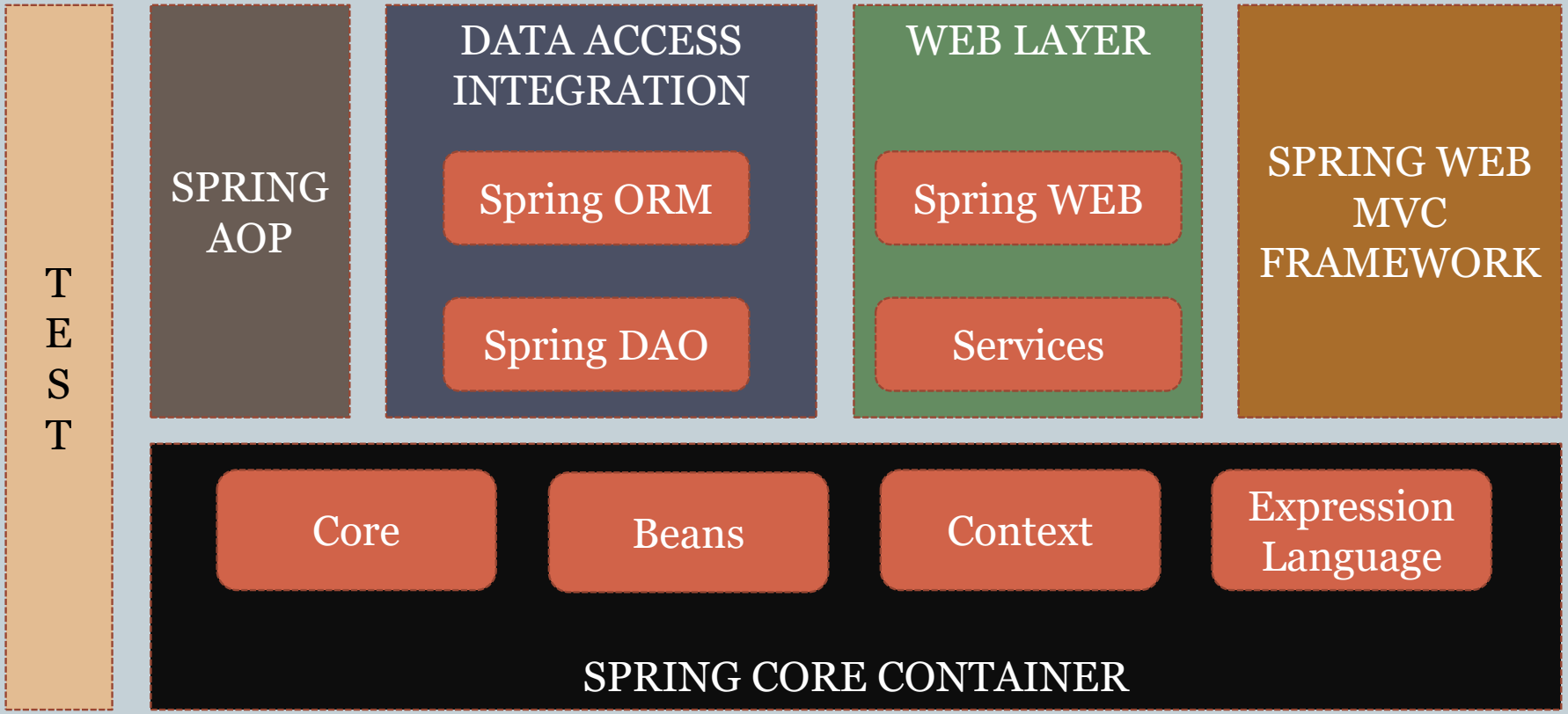
What is a bean?

- A Bean is a class that has only state but no behavior
- A Bean must contain a no-argument constructor
- A Bean should be serialized

Class

Bean

Spring Framework Architecture



Spring Bean Configuration

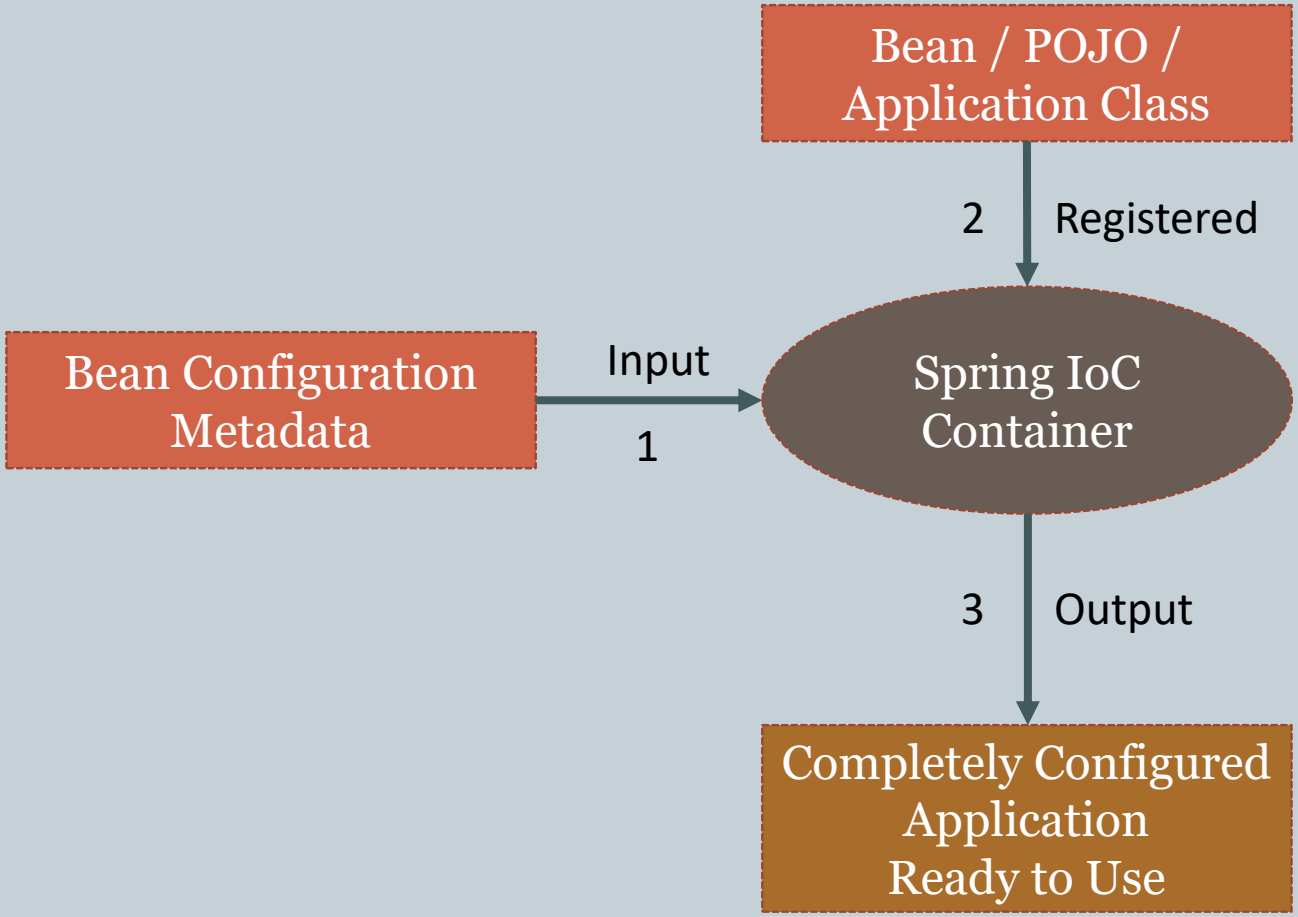

```
<?xml version="1.0" encoding="UTF-8" ?>

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

   <bean id = "animal" class = "com.pushan.spring.beans.Tiger" >
       <property name = "msg" value = "Eating now" />
   </bean>

   <!--
       .
       .
       .
   -->
</beans>
```

Spring IoC Container



Spring IoC Container contd…

BeanFactory	ApplicationContext
The simplest factory, mainly for DI	The advanced and more complex factory
Used when resources are limited, e.g., mobile, applets etc.	Used elsewhere and has the below features, 1> Enterprise aware functions 2> Publish application events to listeners 3> Wire and dispose beans on request
org.springframework.beans.factory.BeanFactory	org.springframework.context.ApplicationContext
`XmlBeanFactory factory = new XmlBeanFactory (new ClassPathResource("wildLife.xml"));`	`ApplicationContext aC = new ClassPathXmlApplicationContext("wildLife.xml");`

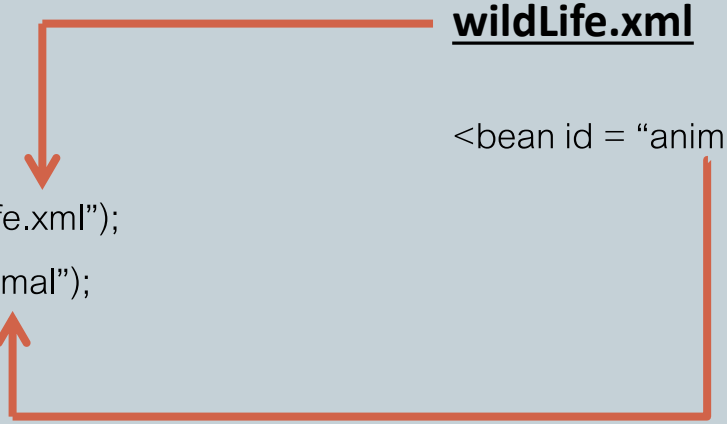
Injection Methods

- Setter Injection
  - Pass dependencies in via property setters (e.g., Spring)
- Constructor Injection
  - Pass dependencies in via constructor (e.g., Spring)
- Interface Injection
  - Pass dependencies in via interfaces (e.g., Avalon)

```
public class MainClass {
   public static void main(String… args) {
      ApplicationContext aC = new
ClassPathXmlApplicationContext("wildLife.xml");
      Animal a = (Animal) aC.getBean("animal");
      a.eat();
   }
}
```

wildLife.xml

```
<bean id = "animal" class = "Tiger" />
```



Setter Injection

- In the <bean>, Specify <property> tag.

```
public class College {
 private String collegeId;
 private int totalStudents;

 public String getCollegeId() {
  return collegeId;
 }
 public void setCollegeId(String
collegeId) {
  this.collegeId = collegeId;
 }
 public int getTotalStudents() {
  return totalStudents;
 }
 public void setTotalStudents(int
totalStudents) {
  this.totalStudents = totalStudents;
 }
}
```

```
public class MyClass {
 private static final String MY_COLLEGE = "myCollege";

 public static void main(String... args) {
  ApplicationContext ctx = new
ClassPathXmlApplicationContext("springConfig.xml");

  College col = (College) ctx.getBean(MY_COLLEGE);

  System.out.println("College Id:" + col.getCollegeId());
  System.out.println("Total Students:" +
col.getTotalStudents());
 }
}
```

```
<bean id="myCollege"
 class="com.pushan.study.spring.beans.College">
      <property name="collegeId" value="123Abc" />
      <property name="totalStudents" value="500" />
 </bean>
```

Constructor Injection

- In the <bean>, Specify <constructor-arg>

```
public class College {

 private String collegeId;
 private int totalStudents;

 public void College (int tS,
String cl) {
  this.totalStudents = tS;
  this.collegeId = cl;
 }

 public String
getCollegeId(){
  return collegeId;
  }
  public int
getTotalStudents() {
  return totalStudents;
 }
```

```
public class MyClass {
 private static final String MY_COLLEGE = "myCollege";

 public static void main(String... args) {
  ApplicationContext ctx = new
ClassPathXmlApplicationContext("springConfig.xml");

  College col = (College) ctx.getBean(MY_COLLEGE);

  System.out.println("College Id:" + col.getCollegeId());
  System.out.println("Total Students:" +
col.getTotalStudents());
 }
}
```

```
<bean id="myCollege"
class="com.pushan.study.spring.beans.College">
      <constructor-arg type="int" value="500"/>
      <constructor-arg type="java.lang.String"
value="123Abc"/>
 </bean>
```

Constructor Injection Illustrated

- The ‘value’ attribute is mandatory and rest are optional, e.g., ‘type’
- Constructor Injection can automatically cast the value to the desired ‘known’ type
- By default the ‘type’ of the ‘value’ is ‘java.lang.String’ (if not specified explicitly)
- Constructor injection does not interpret ordering of the arguments specified

Constructor Injection Ambiguity 1


```
<bean id="myCollege" class="com.pushan.College">
 <constructor-arg value="500"/>
 <constructor-arg value="123Abc"/>
</bean>
```

```
public class College {
 private String collegeId;
 private int totalStudents;
 private String collegeAdd;

 public College (int totalStudents, String collegeId){
  this.totalStudents = totalStudents;
  this.collegeId = collegeId;
 }

 public College (String collegeAdd, String collegeId){
  this.collegeAdd = collegeAdd;
  this.collegeId = collegeId;
 }
}
```

Which constructor will be called?

Constructor Injection Ambiguity 1 Solution


```
<bean id="myCollege" class="com.pushan.College">
 <constructor-arg value="500" type="int"/>
 <constructor-arg value="123Abc" type="java.lang.String"/>
</bean>
```

```
public class College {
 private String collegeId;
 private int totalStudents;
 private String collegeAdd;

 public College (int totalStudents, String collegeId){
  this.totalStudents = totalStudents;
  this.collegeId = collegeId;
 }

 public College (String collegeAdd, String collegeId){
  this.collegeAdd = collegeAdd;
  this.collegeId = collegeId;
 }
}
```

The '**type**' of the value is specified

Constructor Injection Ambiguity 2


```
<bean id="myCollege" class="com.pushan.College">
 <constructor-arg value="500" type="int"/>
 <constructor-arg value="123Abc" type="java.lang.String"/>
</bean>
```

```
public class College {
 private String collegeId;
 private int totalStudents;
 private String collegeAdd;

 public College (int totalStudents, String collegeId){
  this.totalStudents = totalStudents;
  this.collegeId = collegeId;
 }

 public College (String collegeAdd, int totalStudents){
  this.totalStudents = totalStudents;
  this. collegeAdd = collegeAdd;
 }
}
```

Which constructor will be called?

Constructor Injection Ambiguity 2 Solution


```
<bean id="myCollege" class="com.pushan.College">
 <constructor-arg value="500" type="int" index="0"/>
 <constructor-arg value="123Abc" type="java.lang.String" index="1"/>
</bean>
```

```
public class College {
 private String collegeId;
 private int totalStudents;
 private String collegeAdd;

 public College (int totalStudents, String collegeId){
  this.totalStudents = totalStudents;
  this.collegeId = collegeId;
 }

 public College (String collegeAdd, int totalStudents){
  this.totalStudents = totalStudents;
  this. collegeAdd = collegeAdd;
 }
}
```

The '**index**' of the value is specified

Bean Scope 1 – Object Type (Part I)

Bean Scope	Description
singleton	Single instance of bean in every getBean() call [Default]
prototype	New instance of bean in every getBean() call
request	Single instance of bean per HTTP request
session	Single instance of bean per HTTP session
global-session	Single instance of bean per global HTTP session
thread	Single instance of bean per thread
custom	Customized scope

We will look into 'singleton' and 'prototype' scopes only

Added in Spring 3.0

Valid in the context of web-aware ApplicationContext

Bean Scope 1 – Object Type (Part II)

- Mainly bean can be of two types, viz.,
  1. **Singleton** (e.g., <bean scope="singleton" ... />)
  2. **Prototype** (e.g., <bean scope="prototype" ... />)
- A 'singleton' bean is created once in the Spring container. This 'singleton' bean is given every time when referred. It is garbage collected when the container shuts down.
- A 'prototype' bean means a new object in every request. It is garbage collected in the normal way, i.e., when there is no reference for this object.
- By default every bean is singleton if not specified explicitly.

Bean Scope 2 - Inheritance


```
<bean id="a" abstract="true">
  <property name="msg1" value="Tiger runs" />
  <property name="msg2" value="Tiger eats" />
</bean>
```

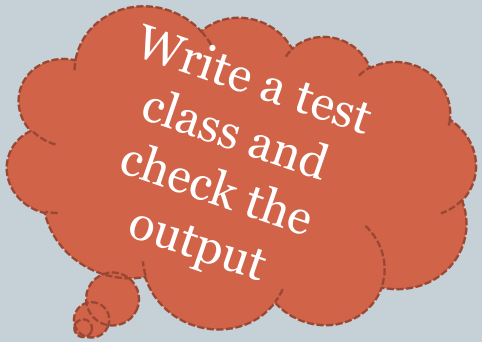
```
<bean id="b" class="B" parent="a">
  <property name="msg1" value="Lion runs" />
  <property name="msg3" value="Lion sleeps" />
</bean>
```

```
public class A {
  private String msg1;
  private String msg2;

  //  getters and setters ...
}
```

```
public class B {
  private String msg1;
  private String msg2;
  private String msg3;

  //  getters and setters ...
}
```



Bean Reference (wiring)

- Through setter or constructor injection we can refer to another bean which has its own separate definition.

```
public class Person{
 //   ...
}
```

```
<bean id="person"
class="Person" />
```

```
public class Location{
 private String city;

 public Location (String c)
  this.city = c;
 }

 //   ...
}
```

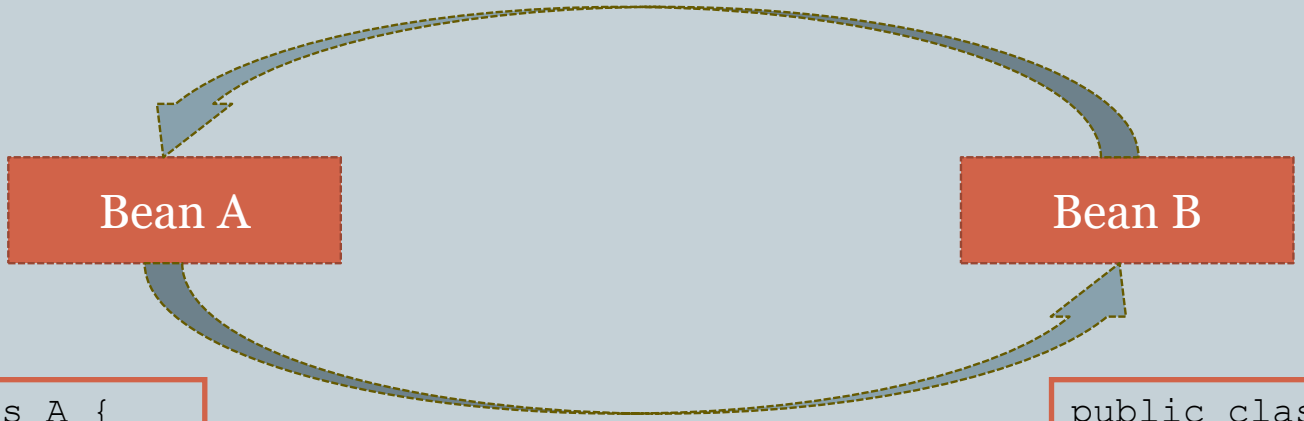
```
<bean id="location"
class="Location">
  <constructor-arg
value="Kolkata"
type="java.lang.String" />
</bean>
```

```
public class A {
 private String msg;
 private Person owner;
 private Location address;

 //   getters and setters ...
}
```

```
<bean id="a" class="A">
 <property name="msg" value="hello"/>
 <property name="owner" ref="person"/>
 <property name="address">
  <ref bean="location" />
 </property>
</bean>
```

Circular Dependency




```
public class A {
   private B b;
   public A (B b) {
      this.b = b;
   }
}
```

```
public class B {
   private A a;
   public B (A a) {
      this.a = a;
   }
}
```

```
<bean id="a" class="A">
   <constructor-arg ref="b" />
</bean>

<bean id="b" class="B">
   <constructor-arg ref="a" />
</bean>
```

Inner Bean

- Inner bean is also a bean reference.
- A bean defined within another bean.
- The inner bean is fully local to the outer bean.
- The inner bean has *prototype* scope.

```
<bean id="a" class="A">
 <property name="address">
  <bean class="Location">
   <property name="city" value="Kolkata"/>
   <property name="zip" value="700006"/>
  </bean>
 </property>
</bean>
```

```
public class A {
 private Location address;

 // getters and setters …
}

public class Location {
 private String city, zip;

 // getters and setters …
}
```

Injecting Collection

Tag Name	Inner Tag Name	Java Collection Type	Specification
<list>	<value>	java.util.List<E>	Allows duplicate entries
<map>	<entry>	java.util.Map<K, V>	Key-Value pair of any object type
<set>	<value>	java.util.Set<E>	Does not allow duplicate entries
<props>	<prop>	java.util.Properties	Key-Value pair of type 'String'

So, lets inject some collection then …


```
<bean id="animalCollection" class="AnimalCollection">
  <property name="animalList">
    <list>
      <value>Tiger</value>
      <value>Lion</value>
      <value>Tiger</value>
    </list>
  </property>
  <property name="animalSet">
    <set>
      <value>Lion</value>
      <value>Tiger</value>
      <value>Lion</value>
    </set>
  </property>
  <property name="animalMap">
    <map>
      <entry key="1" value="Tiger"/>
      <entry key="2" value="Lion"/>
      <entry key="3" value="Tiger"/>
    </map>
  </property>
  <property name="animalProp">
    <props>
      <prop key="one">Lion</prop>
      <prop key="two">Tiger</prop>
      <prop key="three">Lion</prop>
    </props>
  </property>
</bean>
```

Bean definition for DI

```
public class AnimalCollection {
  List<String> animalList;
  Set<String> animalSet;
  Map<String, String> animalMap;
  Properties animalProp;

  //  getters and setters ...
}
```

Bean

```
public class MainClass {
  public static void main(String… args) {
    ApplicationContext ctx = new
ClassPathXmlApplicationContext("wildLife.xml");

    AnimalCollection aC =(AnimalCollection)
ctx.getBean("animalCollection");

    System.out.println("animalList :" + aC.getAnimalList());
    System.out.println("animalSet:" + aC.getAnimalSet());
    System.out.println("animalMap:" + aC.getAnimalMap());
    System.out.println("animalProp:" + aC.getAnimalProp());
  }
}
```

Test class

```
animalList :[Tiger, Lion, Tiger]
animalSet :[Lion, Tiger]
animalMap :{1=Tiger, 2=Lion, 3=Tiger}
animalProp :{one=Lion, two=Tiger, three=Lion}
```

Output

Constructor v/s Setter Injection

- Setter injection gets preference over constructor injection when both are specified
- Constructor injection cannot partially initialize values
- *Circular dependency can be achieved by setter injection*

- Security is lesser in setter injection as it can be overridden
- Constructor injection fully ensures dependency injection but setter injection does not
- Setter injection is more readable

Bean Properties

Tag Name	Description	Example
id	Unique Id	<bean id="person" ... />
name	Unique Name	<bean name="lion" ... />
class	Fully qualified Java class name	<bean class="a.b.C" ... />
scope	Bean object type	<bean scope="singleton" ... />
constructor-arg	Constructor injection	<constructor-arg value="a" />
property	Setter injection	<property name="a" ... />
autowire	Automatic Bean referencing	<bean autowire="byName" ... />
lazy-init	Create a bean lazily (at its first request)	<bean lazy-init="true" ... />
init-method	A callback method just after bean creation	<bean init-method="log" ... />
destroy-method	A callback just before bean destruction	<bean destroy-method="log" ... />

Bean Life Cycle Callbacks

org.springframework.beans.factory.InitializingBean

void afterPropertiesSet () throws Exception;

```
public class MyBean implements
InitializingBean {
  public void afterPropertiesSet () {
    //   Initialization work
  }
}
```

```
<bean id=“myBean"
      class=“MyBean" init-method="init"/>
```

```
public class MyBean {
  public void init () {
    //   Initialization work
  }
}
```

org.springframework.beans.factory.DisposableBean

void destroy () throws Exception;

```
public class MyBean implements
DisposableBean {
  public void destroy () {
    //   Destruction work
  }
}
```

```
<bean id=“myBean"
      class=“MyBean" destroy-method=“des"/>
```

```
public class MyBean {
  public void des () {
    //   Destruction work
  }
}
```

Beans Auto-Wiring

- Spring container can auto wire beans
- In this case, there is no need to specify <property/> and/or <constructor-arg/> tags
- It decreases the amount of xml configuration

Removed in Spring 3.0

Mode	Description	Example
no	No auto wiring of beans	Default
byName	Auto wire beans by property name	<bean autowire="byName" ... />
byType	Auto wire beans by property type	<bean autowire="byType" ... />
constructor	Auto wire beans by constructor	<bean autowire="constructor" ... />
~~autodetect~~	~~First try by constructor, then by type~~	~~<bean autowire="autodetect" ... />~~

Auto wire by Name


```
public class Person {
  //   ...
}
```

```
<bean id="person"
class="Person" />
```

```
public class Location {
  private String city;

  public Location (String city) {
    this.city = city;
  }

  //   ...
}
```

```
<bean id="location"
class="Location">
  <constructor-arg value="Kolkata"
type="java.lang.String" />
</bean>
```

```
public class A {
  private String msg;
  private Person person;
  private Location location;

  //   getters and setters ...
}
```

```
<bean id="a" class="A“ autowire=“byName”>
  <property name="msg" value="hello" />
</bean>
```

Auto wire by Type


```
public class Person {
  //   ...
}
```

```
<bean id="person"
class="Person" />
```

```
public class Location {
  private String city;

  public Location (String city) {
    this.city = city;
  }

  //   ...
}
```

```
<bean id="location"
class="Location">
  <constructor-arg value="Kolkata"
type="java.lang.String" />
</bean>
```

```
public class A {
  private String msg;
  private Person owner;
  private Location address;

  //   getters and setters ...
}
```

```
<bean id="a" class="A“ autowire=“byType”>
  <property name="msg" value="hello" />
</bean>
```

<!-- The below bean definition will destroy the uniqueness of beans byType, hence Spring will give exception. -->

```
<bean id=“person2" class=“Person”/>
```

Auto wire by Constructor


```
public class Person {
  //   ...
}
```

```
<bean id="person"
class="Person" />
```

```
public class Location {
  private String city;

  public Location (String city) {
    this.city = city;
  }

  //   ...
}
```

```
<bean id="location"
class="Location">
  <constructor-arg value="Kolkata"
type="java.lang.String" />
</bean>
```

```
public class A {
  private String msg;
  private Person owner;
  private Location address;

  public A (String m, Person o, Location a){
    …
  }
}
```

```
<bean id="a" class="A“ autowire=“constructor”>
  <property name="msg" value="hello" />
</bean>
```

Spring Auto-wiring Bottleneck

- Spring auto wiring has certain disadvantages or limitations, given below.

Disadvantage	Description
Overriding beans configuration	If the bean configuration is specified explicitly then it overrides the bean auto wiring configuration
Unable to wire primitive types	Auto wiring is applicable only for beans and not for simple properties like primitives, String etc.
Auto wiring has confusing nature	Explicit wiring or manual wiring of beans is easier to understand than auto wiring. It is preferred to use explicit wiring if possible.
Partial wiring is not possible	Auto wiring will always try to wire all the beans through setter or constructor. It cannot be used for partial wiring.

@Spring (annotation = start)

- Evolved in Spring 2.0 (@Required).
- Developed and became famous from Spring 2.5
- “Old wine in new bottle” - an alternative of the xml configuration for bean wiring
- Not enabled by default (need explicit enabling)
- XML overrides annotation for bean configuration
- Sometimes gets treated as an anti-pattern since change of annotation configuration needs compilation of code
- IDE support

Enabling Spring Annotation


```
<?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-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd" >

   <context:annotation-config/>

   <!--
      <bean ... />
      <bean ... />
          .
          .
          .
      <bean ... />
   -->
</beans>
```

@Required

- **Applicable to only setter methods.**

```
public class Boy {
 private String name;
 private int age;

 @Required
 public void setName(String name){
  this.name = name;
 }

 @Required
 public void setAge(int age){
  this.age = age;
 }

 //   getters ...
}
```

- **Values by name.**

```
<bean id="boy" class="Boy">
 <property name="name" value="Rony"/>
 <property name="age" value="10"/>
</bean>
```

- **Strict checking.**

```
<bean id="boy" class="Boy">
 <property name="name" value="Rony"/>
</bean>
```

```
Property 'age' is required for bean
'boy'
```

@Autowired

- **Applicable to methods, fields and constructors**

```
public class Boy {
 private String name;
 private int age;

 //   getters and setters ...
}

public class College {
 private String collegeId;
 @Autowired
 private Boy student;

 public void setCollegeId(String cI){
  this.collegeId = cI;
 }

 //   getters ...
}
```

No setter needed for autowiring on field

- **Wiring by type.**

```
<bean id="boy" class="Boy">
 <property name="name" value="Rony"/>
 <property name="age" value="10"/>
</bean>
<bean id="college" class="College">
 <property name="collegeId" value="1A"/>
</bean>
```

- **'required' attribute**

```
public class College {
 private String collegeId;
 @Autowired(required=false)
 private Boy student;

 //   ...
}
```

'student' auto wiring becomes optional

@Qualifier

- **Solution to @Autowired for type ambiguity**

```
public class Boy {
 private String name;
 private int age;

 //   getters and setters ...
}

public class College {
 private String collegeId;
 @Qualifier(value="tony")
 @Autowired
 private Boy student;

 public void setCollegeId(String cI){
  this.collegeId = cI;
 }

 //   getters ...
}
```

Qualifier value

```
<bean id="boy1" class="Boy">
 <qualifier value="rony"/>
 <property name="name" value="Rony"/>
 <property name="age" value="10"/>
</bean>

<bean id="college" class="College">
 <property name="collegeId" value="1A"/>
</bean>

<bean id="boy2" class="Boy">
 <qualifier value="tony"/>
 <property name="name" value="Tony"/>
 <property name="age" value="8"/>
</bean>
```

- **If <qualifier/> is not configured, then searches with bean id/name. But if specified, it will always search with qualifier only.**

@Resource, @PostConstruct, @PreDestroy

- @PostConstruct is an alternative of the init-method.
- @PreDestroy is an alternative of the destroy-method.
- @Resource(name="<beanName>") is used for auto wiring by name.
- @Resource can be applied in field, argument and methods.
- If no 'name' attribute is specified in @Resource, then the name is derived from the field, setter method etc.

```
<bean id="boy1" class="Boy">
 <property name="name" value="Rony"/>
 <property name="age" value="10"/>
</bean>

<bean id="college" class="College">
 <property name="collegeId" value="1A"/>
</bean>

<bean id="boy2" class="Boy">
 <property name="name" value="Tony"/>
 <property name="age" value="8"/>
</bean>
```

```
public class College {
 private String collegeId;

 @Resource(name="boy1")
 private Boy student;

 //   getters and setters ...
}
```

@Configuration & @Bean (Part I)


```
@Configuration
public class AnimalConfig{
 @Bean
 public Lion lion(){
  return new Lion();
 }

 @Bean
 public Tiger tiger(){
  Tiger t = new Tiger();
  t.doInit();
  return t;
 }
}
```

```
 <bean id="lion" class="Lion" />

<bean id="tiger" class="Tiger"
 init-method="doInit" />
```

```
public class MainTest{
 public static void main(String[] a){
  ApplicationContext aC = new
AnnotationConfigApplicationContext(An
imalConfig.class);

  Tiger t = aC.getBean(Tiger.class);
  Lion l = aC.getBean(Lion.class);

  //   ...
 }
}
```

@Configuration & @Bean (Part II)


```
@Configuration
public class LionConfig{
 @Bean(destroyMethod="clear")
 public Lion lion(){
  return new Lion();
 }
}

@Configuration
public class TigerConfig{
 @Bean(initMethod="doInit")
 public Tiger tiger(){
  return new Tiger();
 }
}
```

```
<bean id="lion" class="Lion"
destroy-method="clear" />

<bean id="tiger" class="Tiger"
init-method="doInit" />
```

```
public class MainTest{
 public static void main(String[] a){
  ApplicationContext aC = new
AnnotationConfigApplicationContext();

  aC.register(LionConfig.class,
TigerConfig.class);
  aC.refresh();

  //   ...
 }
}
```

@Configuration & @Bean (Part III)


```
@Configuration
public class TigerConfig{
 @Bean
 public Tiger tiger(){
  return new Tiger();
 }
}
```

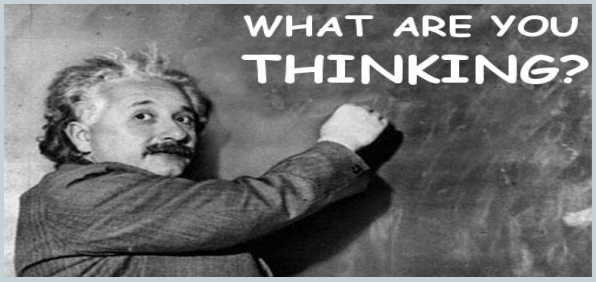
```
@Configuration
@Import(TigerConfig.class)
public class LionConfig{
 @Bean
 public Lion lion(){
  return new Lion();
 }
}
```

```
@Configuration
public class TigerConfig{
 @Bean(name="cat")
 @Scope("prototype")
 public Tiger tiger(){
  return new Tiger();
 }
}
```

```
<bean id="cat"
class="Tiger"
scope="prototype" />
```

Try Yourself

- Are ‘Dependency Injection’ and ‘Inversion of Control’ same? Give reason for the answer.
- How to achieve circular dependency via both setter and constructor injection?
- How to declare the <property> of the bean together with the bean definition in a single line?
- Why does an inner bean have the default scope as ‘prototype’?



Helping Resources

- Spring source documentation (http://www.springsource.org/documentation)
- Tutorials Point tutorial (http://www.tutorialspoint.com/spring)
- Mkyong tutorial (http://www.mkyong.com/tutorials/spring-tutorials)
- DZone (http://java.dzone.com/articles/case-spring-inner-beans)