

## Spring Notes

Spring is a very popular framework for building Java web application. Initially simpler and lightweight alternative to J2EE. It has large number of helper classes to make development easier.

Spring 5 which is newer version of spring has updated minimum requirement of Java8 or higher.

Spring 5 also has upgraded spring MVC to use new version of servlet API 4.0. Spring 5 still provide support for spring 4, it is just like spring 5 has got some new features.

Some specific features of Spring 5 won't work in spring 4.

### **SPRING CORE:**

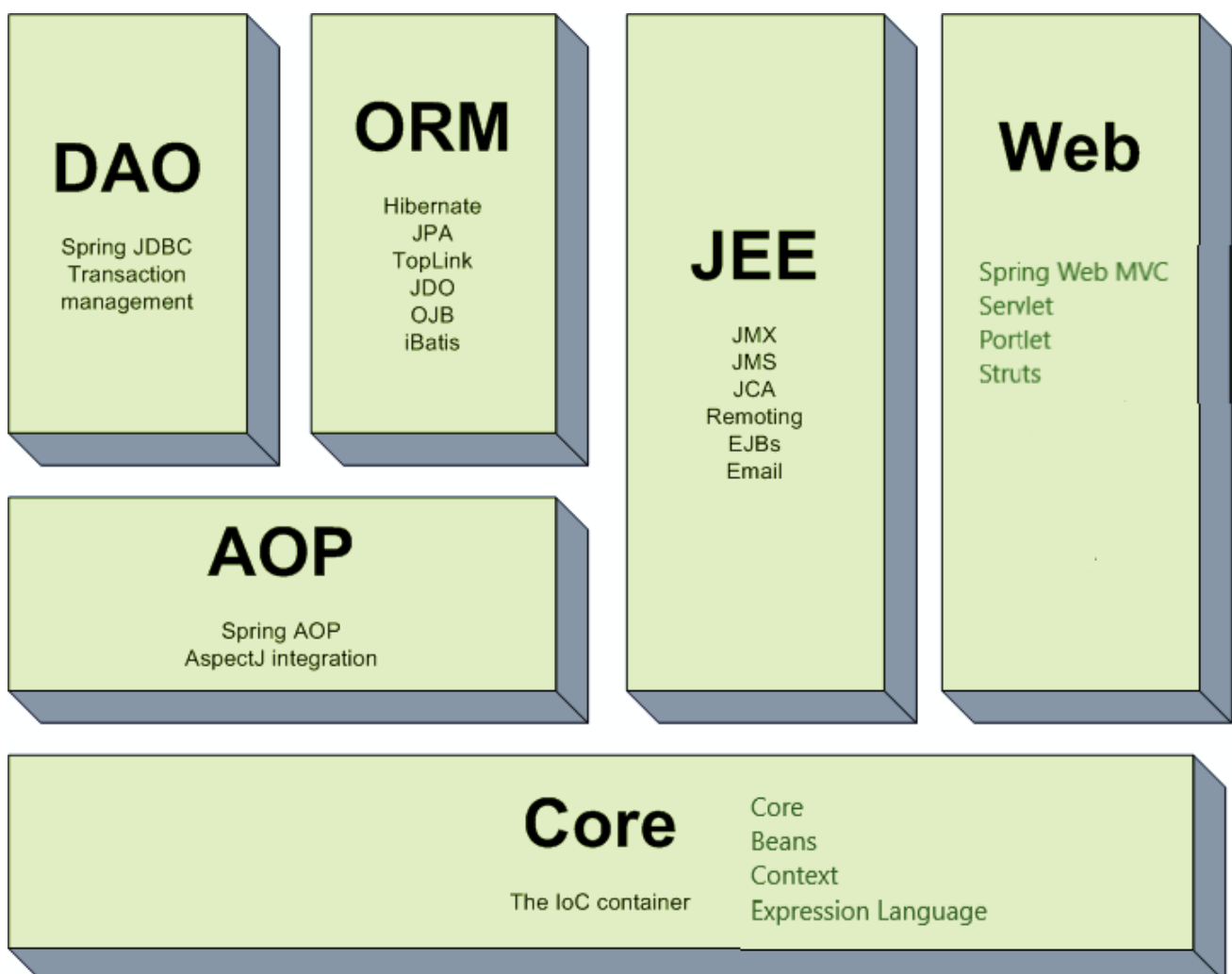
Spring provides light weight development with JAVA POJO (Plain Old Java Object).

Spring provides dependency injection to promote loose coupling.

Spring provides Declarative Programming with AOP (Aspect Object Programming)

Spring minimize the boilerplate Java Code.

### Spring Components:



Spring is a lightweight framework. It can be thought of as a framework of frameworks because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF, etc.

### IOC (Inversion of Control) & Dependency Injection:

These are design patterns that are used to remove dependency from the programming code. They make the code easier to test and maintain.

Spring's core idea is that instead of managing object relationships yourself, you offload them to the framework. Inversion of control (IOC) is the methodology used to manage object relationships. Dependency injection is the mechanism for implementing IOC. Since these two concepts are related but different, let's consider them more closely:

#### **Inversion of Control (IOC):**

Inversion of control (IOC) does just what its name says: it inverts the traditional hierarchy of control for fulfilling object relationships. Instead of relying on application code to define how objects relate to each other, relationships are defined by the framework.

#### **Dependency injection (DI):**

It is a mechanism where the framework "injects" dependencies into your app. It's the practical implementation of IOC.

### **Spring Components:**

Core container

- Core
- Bean
- Context

Expression Language

- Aspect-oriented programming (AOP)
- AOP
- Aspects
- Instrumentation

Data access and integration

- JDBC
- JPA/ORM
- JMS
- Transactions

Web

- Web/REST

- Servlet
- Struts

### Example of Java Program without spring:

#### *Coach.java (Interface)*

```
package Stacks;  
  
public interface Coach {  
    public String getDailyWorkout();  
}
```

#### *BaseballCoach.java*

```
package Stacks;  
  
public class BaseballCoach implements Coach{  
    @Override  
    public String getDailyWorkout() {  
        return "30 minutes practice in batting";  
    }  
}
```

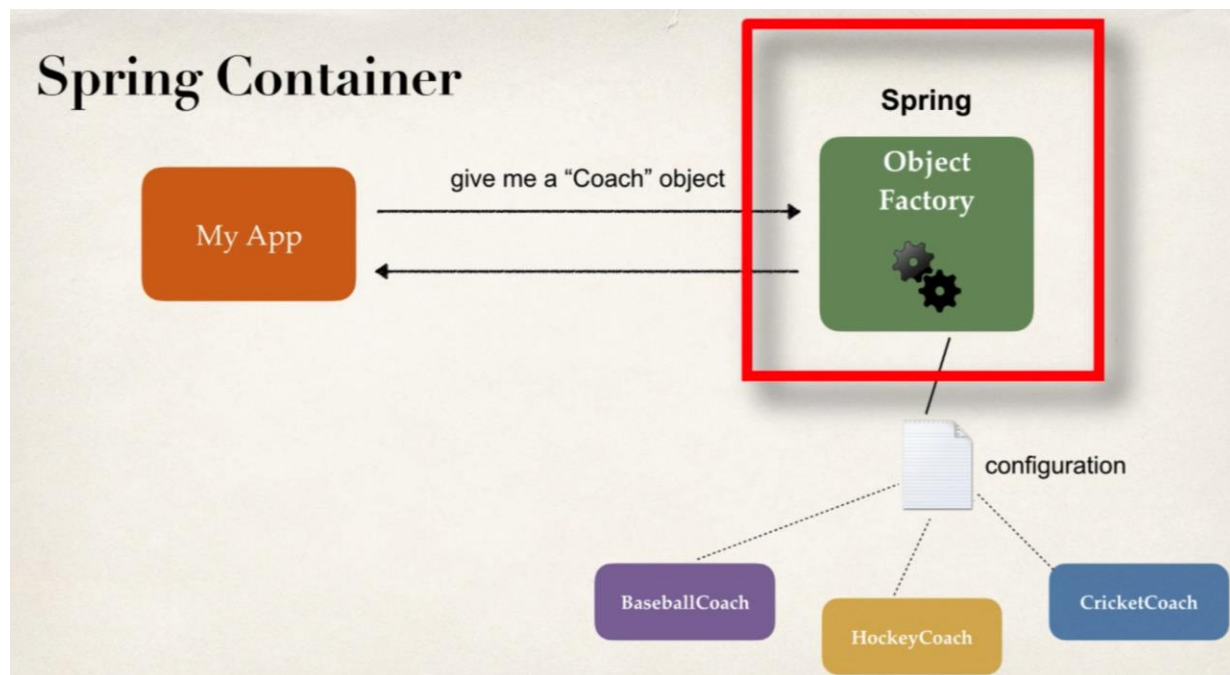
#### *MyApp.java*

```
package Stacks;  
  
public class MyApp {  
    public static void main(String[] args) {  
        Coach coach=new BaseballCoach();  
        System.out.println(coach.getDailyWorkout());  
    }  
}
```

#### **Note:**

The problem with the above example is, it is not configurable but it is hard coded. Ideally we should read the implementation from a config file and make use of it. Spring will solve this problem by using IOC (Inversion of Control).

## Ideal Solution:



## Spring Container:

Spring Container primary functions are:

- To create and manage objects (*Inversion of Control*)
- Inject object dependencies (*Dependency Injection*)

There are three ways for configuring spring container:

- XML Configuration file (*Legacy, but most legacy apps still use this*)
- Java Annotations (*Modern*)
- Java source code (*Modern*)

## Spring Development process:

1. Configure your Spring beans
2. Create a Spring container
3. Retrieve beans from container

### Step\_1: Configuring your spring beans

```
<!-- Define your beans here -->
<bean id="myCoach"
      class="Stacks.BaseballCoach">
</bean>
```

**Note:** Here ID is an alias name which java application uses to get the class object.

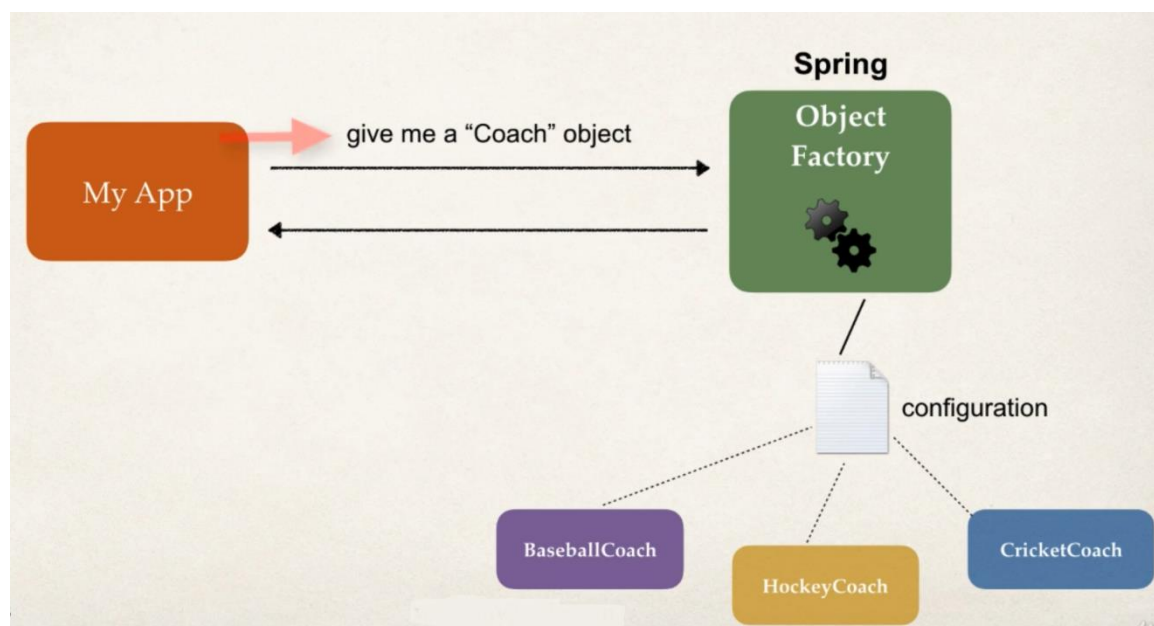
## Step\_2: Create a Spring Container

Spring container is generically known as **ApplicationContext**

Following syntax is used to create a spring container

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

## Step\_3: Retrieve beans from container



```
Coach coach=context.getBean("myCoach",Coach.class);  
System.out.println(coach.getDailyWorkout());
```

## Example of Java Program with spring:

*Coach.java (Interface)*

```
package Stacks;  
  
public interface Coach {  
    public String getDailyWorkout();  
}
```

### BaseballCoach.java

```
package Stacks;

public class BaseballCoach implements Coach{

    @Override
    public String getDailyWorkout() {
        return "30 minutes practice in batting";
    }

}
```

### ApplicationContext.xml

```
package Stacks;

import org.springframework.context.support.ClassPathXmlApplicationContext;

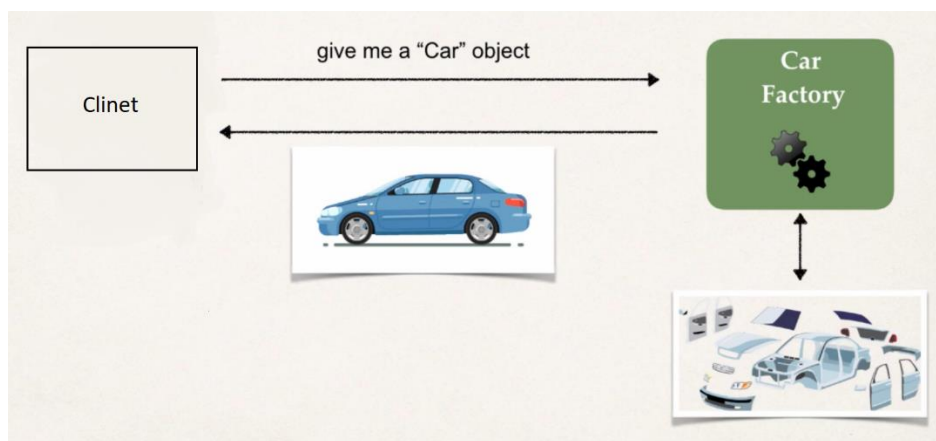
public class MyApp {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context=
            new ClassPathXmlApplicationContext("applicationContext.xml");
        Coach coach=context.getBean("myCoach",Coach.class);
        System.out.println(coach.getDailyWorkout());
    }
}
```

### Note:

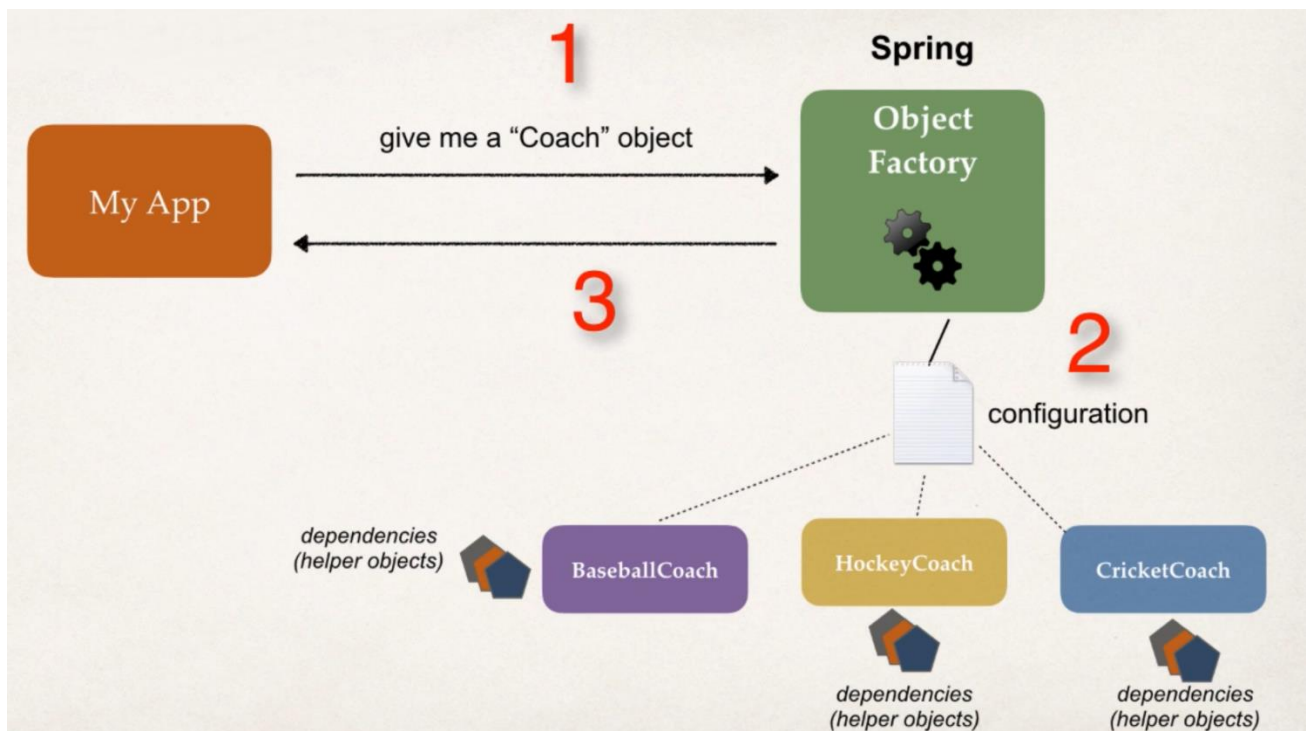
The problem has been solved, now the app is configurable it is no more hard coded. We can easily change Coach to another sport. Spring has solved this problem using IOC.

### Dependency Injection:

The client delegates to calls to another object the responsibility of providing its dependencies.



We are simply outsourcing the construction of car (object) to an external entity (object factory). In case of spring we are asking for an object, and the object will be constructed by spring object factory and given to you as readymade.



### Dependency Injection Example:

Consider a scenario now, Coach wants to provide another service called **FortuneService**. So FortuneService is a helper class or we call it as dependency. It means Coach is dependent on **FortuneService** to give the fortune service to user.

### **Injection Types:**

There are many types of injections with spring.

- Setter injection
- Constructor injection

### **Spring Development process: (Constructor Injection):**

1. Define the dependency interface and class
2. Create a constructor in your class for injection
3. Configure the dependency injection in Spring config file

### *Coach.java (Interface)*

```
package Stacks;

public interface Coach {
    public String getDailyWorkout();
    public String getDailyFortune();
}
```

### *BaseballCoach.java*

```
public class BaseballCoach implements Coach{

    private FortuneService fortuneService;
    public BaseballCoach(FortuneService theFortuneService) {
        this.fortuneService=theFortuneService;
    }

    @Override
    public String getDailyWorkout() {
        return "30 minutes practice in batting";
    }

    @Override
    public String getDailyFortune() {
        return fortuneService.getDailyFortune();
    }
}
```

### *FortuneService.java*

```
package Stacks;

public interface FortuneService {
    public String getFortune();
}
```

### *HappyFortune.java*

```
package Stacks;

public class HappyFortune implements FortuneService {

    @Override
    public String getFortune() {
        return "Today you will meet with your soulmate";
    }
}
```



### ApplicationContext.xml

```
<!-- Define your beans here -->
<bean id="myFortuneService" class="Stacks.HappyFortune">

</bean>
<bean id="myCoach"
      class="Stacks.BaseballCoach">
    <constructor-arg ref="myFortuneService"/>
</bean>
```

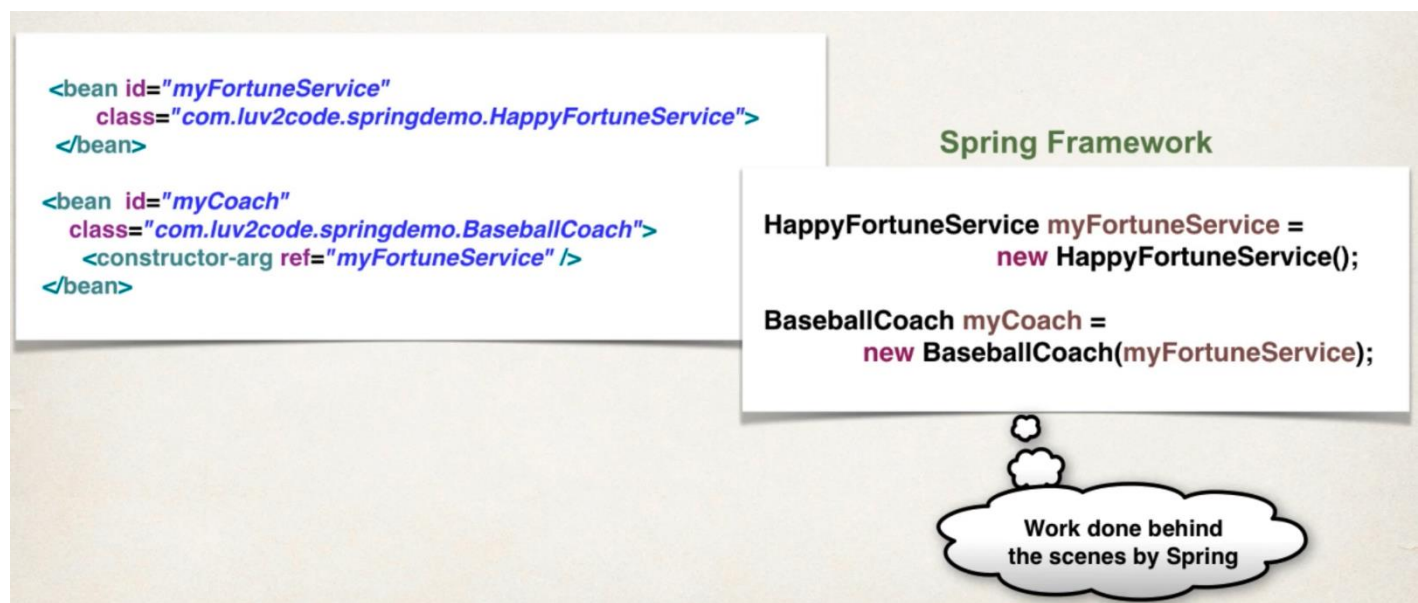
### MyApp.java

```
package Stacks;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyApp {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context=
            new ClassPathXmlApplicationContext("applicationContext.xml");
        Coach coach=context.getBean("myCoach",Coach.class);
        System.out.println(coach.getDailyWorkout());
        System.out.println(coach.getDailyFortune());
    }
}
```

### BEHIND THE SCENE:



**Note:** We need no-arg constructor in `TrackCoach`, otherwise it will throw an error.

## Spring Setter Injection:

### *CricketCoach.java*

```
public class CricketCoach implements Coach{

    private FortuneService fortuneService;

    @Override
    public String getDailyWorkout() {
        return "Bowling practice for 1 hr";
    }

    public void setFortuneService(FortuneService fortuneService) {
        this.fortuneService= fortuneService;
    }
    @Override
    public String getDailyFortune() {
        return fortuneService.getDailyFortune();
    }
}
```

### *ApplicationContext.xml*

```
<bean id="myFortuneService" class="Stacks.HappyFortune"></bean>

<bean id="myCoach" class="Stacks.CricketCoach">
    <property name="fortuneService" ref="myFortuneService"></property>
</bean>
```

### *MyApp.java*

```
/*Taking CricketCoach Object*/
Coach coach1=context.getBean("myCoach", Coach.class);
System.out.println(coach1.getDailyWorkout());
System.out.println(coach1.getDailyFortune());
```

## BEHIND THE SCENE:



## Spring Literal values Injection:

*CricketCoach.java*

```

public class CricketCoach implements Coach{

    private FortuneService fortuneService;
    private String emailAddress;
    private String team;

    public CricketCoach() {
    }
    public String getEmailAddress() {
        return emailAddress;
    }

    public void setEmailAddress(String emailAddress) {
        System.out.println("CricketCoach inside setter - emailAddress");
        this.emailAddress = emailAddress;
    }

    public String getTeam() {
        return team;
    }

    public void setTeam(String team) {
        System.out.println("CricketCoach inside setter - team");
        this.team = team;
    }

    @Override
    public String getDailyWorkout() {
        return "Bowling practice for 1 hr";
    }

    public void setFortuneService(FortuneService fortuneService) {
        this.fortuneService= fortuneService;
    }
    @Override
    public String getDailyFortune() {
        return fortuneService.getDailyFortune();
    }
}

```

*ApplicationContext.xml*

```

<bean id="myFortuneService" class="Stacks.HappyFortune"></bean>

<bean id="myCoach" class="Stacks.CricketCoach">
    <property name="fortuneService" ref="myFortuneService"></property>
    <property name="emailAddress" value="syedzafar@gmail.com"></property>
    <property name="team" value="Sun Risers Hyderabad"></property>
</bean>

```

### Spring\_Literal\_Injection.java

```
public class Spring_Literal_Injection {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context=
            new ClassPathXmlApplicationContext("applicationContext.xml");
        CricketCoach coach=context.getBean("myCoach", CricketCoach.class);
        System.out.println(coach.getDailyWorkout());
        System.out.println(coach.getDailyFortune());
        System.out.println(coach.getEmailAddress());
        System.out.println(coach.getTeam());
    }
}
```

### Spring values Injection from properties file:

Instead of injecting hardcoded values in config file, we can inject values from properties file.

#### teamDetails.properties

```
sports.email=jonson@gmail.com
sports.team=Royal Chlленege Bangalore
```

#### ApplicationContext.xml

```
<!-- Loading the properties file first-->
<context:property-placeholder location="classpath:teamDetails.properties"/>

<!-- Define Beans here -->
<bean id="myFortuneService" class="Stacks.HappyFortune"></bean>

<bean id="myCoach" class="Stacks.CricketCoach">
    <property name="fortuneService" ref="myFortuneService"></property>
    <property name="emailAddress" value="${sports.email}"></property>
    <property name="team" value="${sports.team}"></property>
</bean>
```

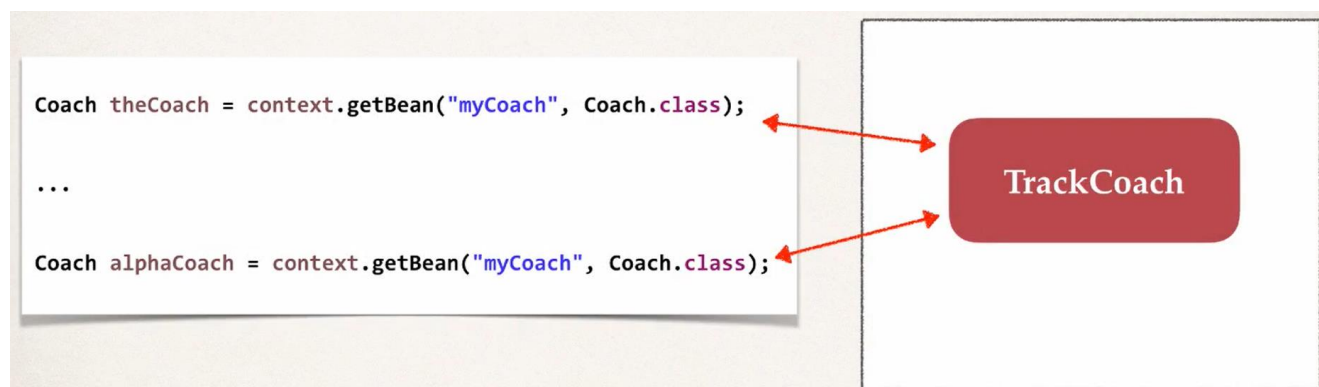
### Spring\_values\_Injection.java

```
public class Spring_Literal_Injection {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context=
            new ClassPathXmlApplicationContext("applicationContext.xml");
        CricketCoach coach=context.getBean("myCoach", CricketCoach.class);
        System.out.println(coach.getDailyWorkout());
        System.out.println(coach.getDailyFortune());
        System.out.println(coach.getEmailAddress());
        System.out.println(coach.getTeam());
    }
}
```

## Spring Bean Scope:

- Scope of bean refers to the life cycle of bean
- How long does the bean live
- How many instances will be created
- How is the bean shared

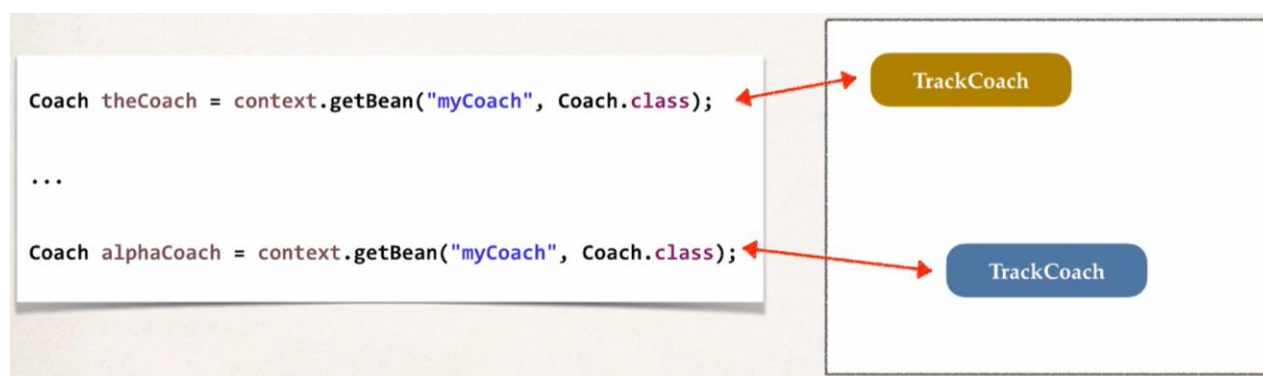
Default scope of a bean is **Singleton**: Only one instance will be created by spring container. It will be cached in memory. All requests for the bean will return the SHARED references to the SAME bean.



Spring Container

## Proto type Scope of a Bean:

In this scope one bean will be created separate for each request, and it is specially used in the scenario where you want to maintain the session of a user.



Spring Container

## Singleton Bean Example:

*HelloBean.java*

```
class HelloBean{
    private String name;

    public String getName() {
        return name;
    }

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

### ApplicationContext.xml

```
<!-- Define your beans here -->

<bean id="myHelloBean" class="Stacks.HelloBean" scope="singleton">

</bean>
```

### BeanScope.java

```
public class BeanScope_Demo {
    public static void main(String[] args) {
        ApplicationContext ap=new ClassPathXmlApplicationContext("applicationContext2.xml");
        HelloBean b1=(HelloBean)ap.getBean("myHelloBean");
        b1.setName("Request_1");

        HelloBean b2=(HelloBean)ap.getBean("myHelloBean");
        b2.setName("Request_2");

        System.out.println(b1.getName());
        System.out.println(b2.getName());

        //Comparing the references to check whether these two are pointing to the same object or different
        System.out.println("Request_1 & Request_2 referring to same OBJ="+b1==b2);
        System.out.println("Request_1 Address: "+b1);
        System.out.println("Request_2 Address: "+b2);
    }
}
```

### Prototype Bean Example:

#### ApplicationContext.xml

```
<!-- Define your beans here -->

<bean id="myHelloBean" class="Stacks.HelloBean" scope="prototype">

</bean>
```

#### OUTPUT:

```
Request_1
Request_2
Request_1 & Request_2 referring to same OBJ=false
Request_1 Address: Stacks.HelloBean@527740a2
Request_2 Address: Stacks.HelloBean@13a5fe33
```

**Note:** Bean life cycle has two methods as follows.

1. *init()*
2. *destroy()*



## Configuring spring using Annotations:

Annotations are used to minimize the XML configuration. This is the latest approach to configure beans. When we write the annotations, in the background spring will scan out java class for annotations. It automatically register the beans in the spring container and then finally retrieve bean from container.

### Development Process:

1. Enable component scanning in spring config file
2. Add the @component annotation to your java classes
3. Retrieve beans from spring container

### TennisCoach.java

```
@Component("myTennisBean")
public class TennisCoach implements Coach{
    public String getDailyWorkout() {
        return "Practice your backhand volley";
    }
}
```

### ApplicationContext.xml

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

### App.java

```
public class App {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context=new
        ClassPathXmlApplicationContext("applicationContext.xml");
        Coach coach=context.getBean("myTennisBean", Coach.class);
        System.out.println(coach.getDailyWorkout());
    }
}
```

## Spring Configuration with Java Annotation (Dependency Injection):

Spring uses auto wiring for dependency injection. Spring will look for the class that matches the property.

Matches by type *class/interface*

Auto wiring can be of three type

1. Constructor injection
2. Setter injection
3. Field injection



### Example: Constructor injection

#### *HockeyCoach.java*

```
@Component
public class HockeyCoach implements Coach{

    private FortuneService fortuneService;
    public String getDailyWorkout() {
        return "Practice for Hocky Long shot";
    }
    @Autowired
    public HockeyCoach(FortuneService fortuneService) {
        this.fortuneService=fortuneService;
    }

    public String getDailyFortune() {
        return fortuneService.getDailyFortune();
    }
}
```

#### *LuckFortune.java*

```
@Component
public class LuckFortune implements FortuneService{

    public String getDailyFortune() {
        return "Today you will meet your partner";
    }
}
```

#### *MyLuckApp.java*

```
public class MyLuckApp {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context=
            new ClassPathXmlApplicationContext("applicationContext.xml");
        Coach coach=context.getBean("hockyCoach", Coach.class);

        System.out.println(coach.getDailyWorkout());
        System.out.println(coach.getDailyFortune());
    }
}
```

### Example: Setter injection

```
@Component
public class HockyCoach implements Coach{

    private FortuneService fortuneService;
    public String getDailyWorkout() {
        return "Practice for Hocky Long shot";
    }
    /*@Autowired
    public HockyCoach(FortuneService fortuneService) {
        this.fortuneService=fortuneService;
    }*/

    @Autowired
    public void setFortuneService(FortuneService fortuneService) {
        this.fortuneService=fortuneService;
    }

    public String getDailyFortune() {
        return fortuneService.getDailyFortune();
    }

}
```

### Example: Method injection

```
/*@Autowired
public void setFortuneService(FortuneService fortuneService) {
    this.fortuneService=fortuneService;
}*/

@Autowired
public void doMyStuff(FortuneService fortuneService) {
    this.fortuneService=fortuneService;
}

public String getDailyFortune() {
    return fortuneService.getDailyFortune();
}
```

### Example: Field injection

```
@Component
public class HockyCoach implements Coach{

    @Autowired
    private FortuneService fortuneService;

    public String getDailyWorkout() {
        return "Practice for Hocky Long shot";
    }
    public String getDailyFortune() {
        return fortuneService.getDailyFortune();
    }

}
```

## Qualifier for Dependency Injection:

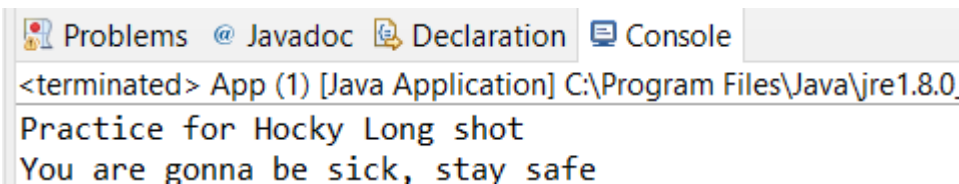
If we have multiple implementation for one interface, in that case spring will give run time exception saying `UnsatisfiedDependencyException`. To solve this problem we use `@Qualifier` annotation to get the specific implemented class using bean ID.

```
public class HockeyCoach implements Coach{

    @Qualifier("healthFortune")
    @Autowired
    private FortuneService fortuneService;

    public String getDailyWorkout() {
        return "Practice for Hockey Long shot";
    }
    public String getDailyFortune() {
        return fortuneService.getDailyFortune();
    }
}
```

OUTPUT:



```
<terminated> App (1) [Java Application] C:\Program Files\Java\jre1.8.0.
Practice for Hockey Long shot
You are gonna be sick, stay safe
```

**Note:** We can assign values from properties file instead of hardcoded values using the following annotation.

```
@Val("{sports.team}")
```

We can use `@Scope` annotation to specify the scope of the bean.

```
@Scope ("singleton")
public class Tennis Coach implements Coach{
}
```

## Bean Life cycle method annotations:

There are basically two methods we have as a part of bean life cycle, `init()` and `destroy()`. We can use `@PostConstruct` and `@PreDestroy` annotation for initialization and destruction respectively.

```
@PostConstruct
public void doMyStuff() {
}
//Code will be executed after bean initialization
```

```
@PreDestroy
public void doMyStuffYo() {
}
//Code will be executed before bean is destroyed
```

### Configuring Beans using Java Code (no xml):

Without using XML configuration we can configure beans using Java code using @ComponentScan annotation.

```
@Configuration
@ComponentScan("com.annotation.compscan")
public class SportsConfig {
}

```

#### *App.java*

```
public class App
{
    public static void main( String[] args )
    {
        AnnotationConfigApplicationContext context=
            new AnnotationConfigApplicationContext("SportsConfig.class");
        Coach coach=context.getBean("hockeyCoach", Coach.class);

        System.out.println(coach.getDailyWorkout());
        System.out.println(coach.getDailyFortune());
    }
}
```