

Spring

Why Spring:

- Alternative to EJB.
- EJB Applications are heavy weight and tightly coupled. (App server dependency)
- Spring applications are light weight and loosely coupled.(doesn't use app server)
- Spring reduces investment cost as we don't need to buy services from servers
- Spring is free source.
- Spring works with POJO classes.

Spring recommends to use associations instead of inheritance

Eg: if class D needs the functionality of class B and C then same can be achieved as below

```
Class D {
    B b = new B ();
    C c = new C();
}

class B{
}

class C{
}
```

Spring Implementation is based on 2 principles *Association* and *Runtime polymorphism*.

Instead of passing Runtime arguments through command line we can pass via XML.

For working with XML we need Container support. IOC is the required Spring container.

Spring Container:

IOC has Core, J2EE containers.

Spring MVC has web containers build on top of above 2

IOC:

Core – Bean factory (Interface)→XMLBeanFactory(class)

J2EE- ApplicationContext(Interface)→ConfigurableApplicationContext(interface)→

ClasspathXMLApplicationContext(Class)

Core container creates object on calling getBean() method→ lazy container

J2EE Container creates while loading xml→eager container

MVC:

Web- webApplicationContext(Interface)→WebApplicationContextUtils

Containers features:

- Read xml file
- Create Instances of XML declarations (java classes (POJO) or bean classes)
- Manage life cycle of bean classes

- Dynamic parameter to bean classes from XML file (****dependency injections***)

DI(dependency injection) Helps to create loose coupling

Steps needed for starting containers:

1. Driver class (any class containing main method)
2. Create Object of Container classes

Eg: Class Test{
 PSVM(string args[]){
 New XMLBeanFacory();
 New ClasspathXMLApplicationContext();
 WebApplicationContextUtils.getObject();

For web application it should be written in init()

Main components for Spring Application:

HelloWorld

1. POJO class
2. XML file
3. Driver class

Pojo class

```
Class Test{
Public static void hello(){
Sop("hello world")
}
}
```

Spring.xml

```
Dtd/xsd
<beans>
<bean class="Test" id="t" singleton="true"></bean>
</beans>
```

Driver Class:

```
Class Client{
Psvm(String args[]){
Resource rs = new ClasspathResource("Spring.xml")
BeanFactory factory = new XMLBeanFactory(r);
Test t = (Test) Factory.getBean("t");
Test t1 = (Test) Factory.getBean("t");

t.hello();
}
```

```
}
```

Note:- Spring dtd can be found at `spring-beans.jar(org, springframework.core.factory.xml.spring-beans – version.dtd)`

In the above scenario although we are using `t` and `t1` Spring will create only one object.

If we make `singleton=false` it will have multiple objects

Spring scope: singleton and prototype / request, session, context

Spring Environment Setup:

1. Create Eclipse project
2. Download Spring Jar files
3. Add jar files to Eclipse project (buildpath)

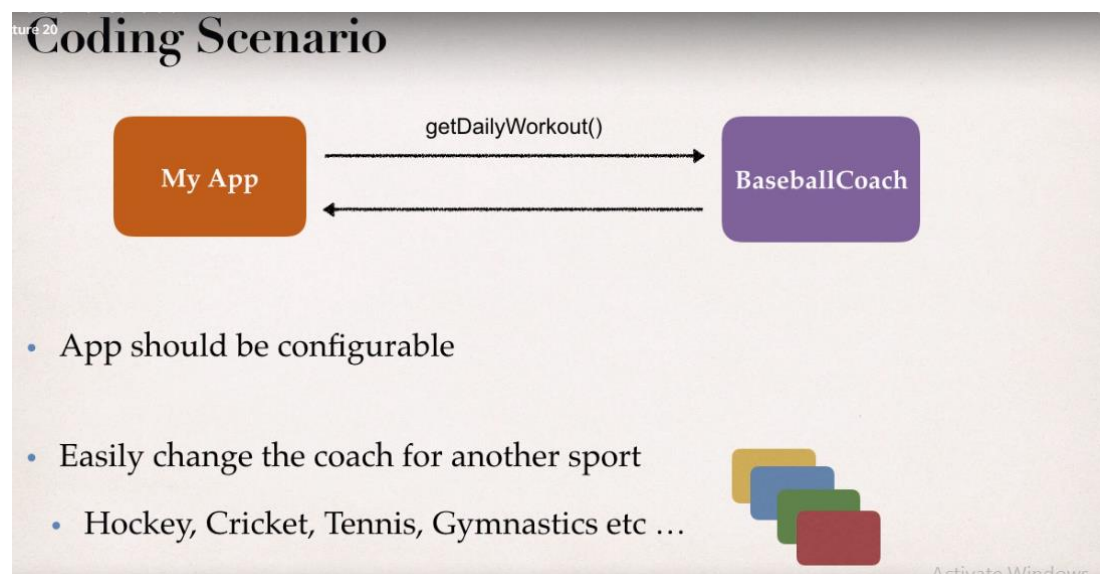
Steps:

- create a simple java project in eclipse
- download spring jar - www.luv2code.com/downloadspring
- copy all jars under lib folder
- Right click project > properties > java buildpath > add jar from lib folder

Spring IoC:

The process of outsourcing the construction and management of Objects to an Object Factory is termed as Inversion of Control (IoC)

Coding Scenario



ecture 20

Code Demo

- **MyApp.java:** main method
- **BaseballCoach.java**
- **Coach.java:** interface after refactoring
- **TrackCoach.java**

MyAPP → driver class

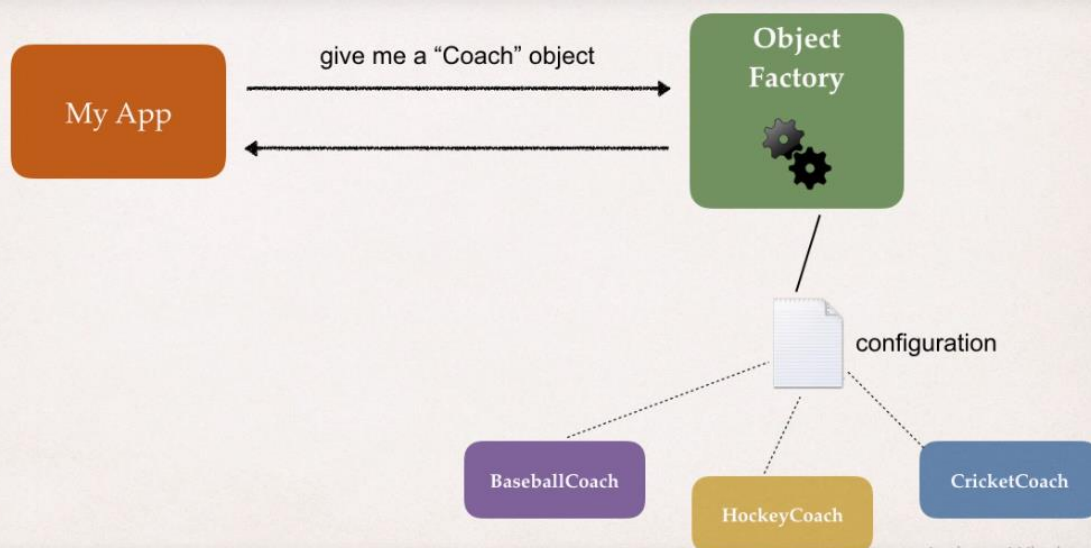
baseBallCoach → POJO class

TrackCoach → POJO class

Coach.java → Interface to achieve compatibilty

ecture 23

Ideal Solution



ure 23

Spring Container

- Primary functions
 - Create and manage objects (*Inversion of Control*)
 - Inject object's dependencies (*Dependency Injection*)

Spring

Object
Factory



Configuring Spring Container:

- XML Configuration file
- Java Annotations
- Java source code

ure 23

Spring Development Process

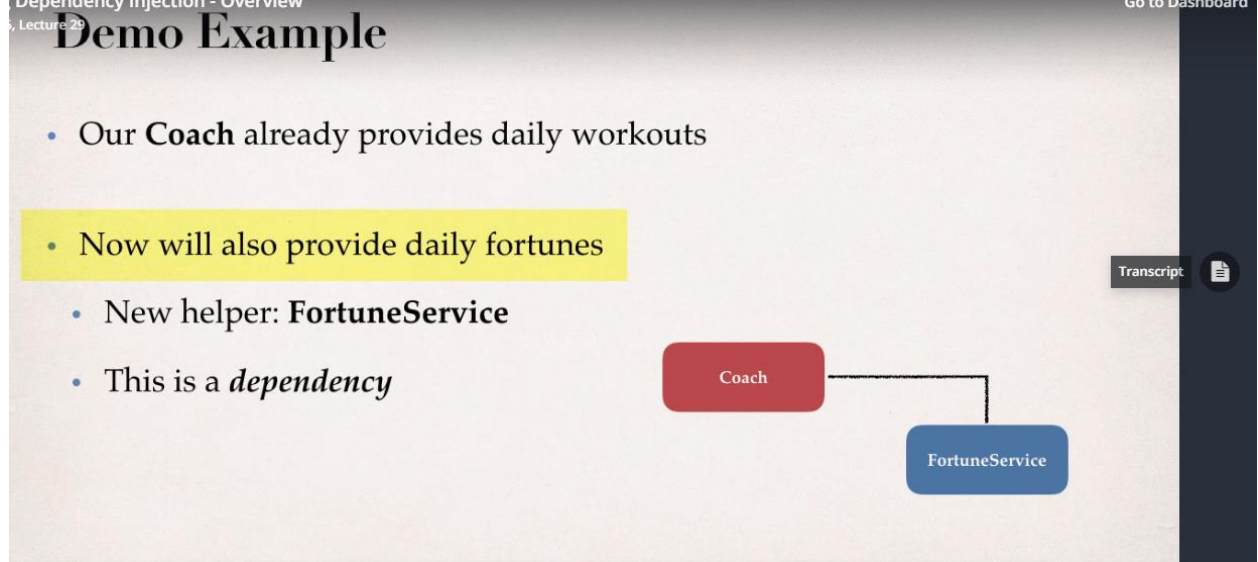
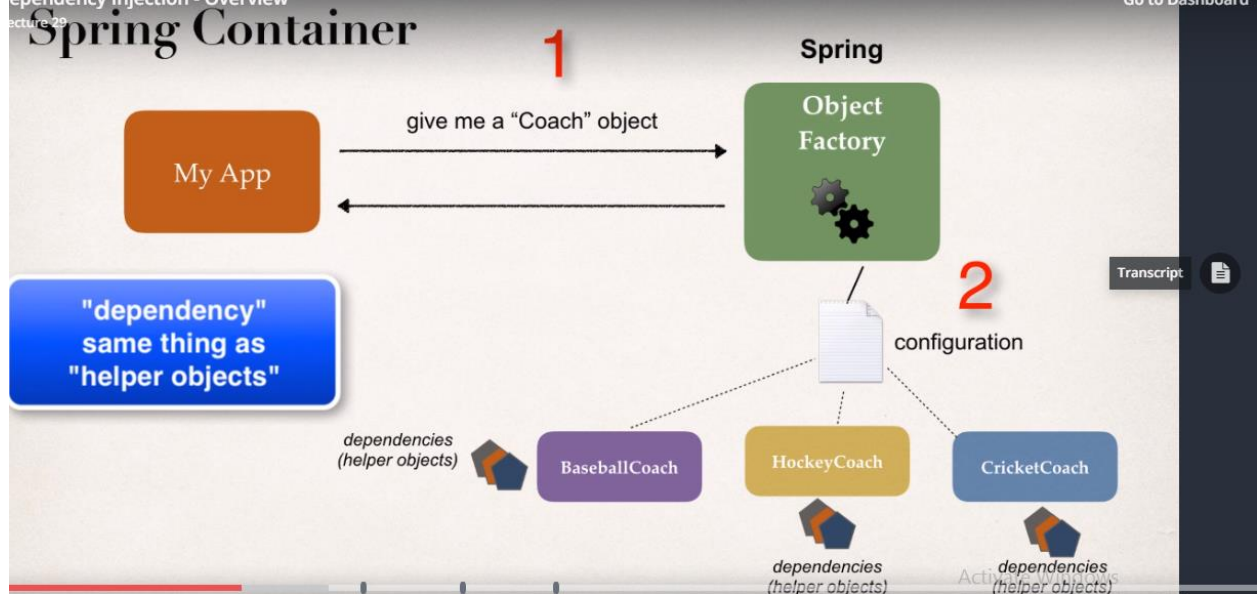
Step-By-Step

1. Configure your Spring Beans
2. Create a Spring Container
3. Retrieve Beans from Spring Container

HelloSpringApp

Dependency Injection:

Outsourcing the construction and Injection of Object to External entity, eg car factory.
Dependency is same thing as helper objects



Injection types:

- Constructor Injection
 - Define dependency interface and class
 - Create Constructor in your class to inject dependency
 - Configure dependency injection in Spring config file--<constructor-arg ref="name"/>

File: applicationContext.xml

```
<bean id="myFortuneService"
      class="com.Luv2code.springdemo.HappyFortuneService">
</bean>

<bean id="myCoach"
      class="com.Luv2code.springdemo.BaseballCoach">
  <constructor-arg ref="myFortuneService" />
</bean>
```

Define dependency / helper

Inject the dependency / helper using "constructor injection"

Behind the scene

Dependency Injection - Behind the Scenes

How Spring Processes your Config File

```
<bean id="myFortuneService"
      class="com.luv2code.springdemo.HappyFortuneService">
</bean>

<bean id="myCoach"
      class="com.luv2code.springdemo.BaseballCoach">
  <constructor-arg ref="myFortuneService" />
</bean>
```

Spring Framework

```
HappyFortuneService myFortuneService =
    new HappyFortuneService();

BaseballCoach myCoach =
    new BaseballCoach(myFortuneService);
```

- Setter Injection
 - Create setter methods in class.
 - Configure dependency injection in Spring config file.
`<property name="fortuneService" ref="myFortune"/>`

Property name should match with setter method name with all small letters
 Eg: `setFortuneService`

Call setter method on Java class

```
<property name="fortuneService" ref="myFortuneService" />
```

```
public void setFortuneService(...)
```

capitalize first letter
of property name:
setFortuneService

Injecting Literal values:

- Create Setter methods for Injection in Class
- Configure Injection in spring config file.

```
<property name="emailAddress" value="thebestcoach@luv2code.com" />
```

```
<property name="team" value="Sunrisers Hyderabad" />
```

```
</beans>
```

We can also set these values from properties file instead of Hardcoding the value.

In this case we need to use the below tag

- Create Properties file
- Load properties file in Spring config file
- Reference values from properties file

File: sport.properties

name

value

```
foo.email=myeasycoach@luv2code.com
```

```
foo.team=Royal Challengers Bangalore
```


File: applicationContext.xml

```
<context:property-placeholder location="classpath:sport.properties"/>
```

Bean Scopes:

- Scope refers to lifecycle of the beans
- How long does the bean live
- How many instances are created.
- How bean is shared.

Note:- default scope for bean is singleton, when we don't specify any scope it assumes singleton.

Types - Overview
Lecture 44

What Is a Singleton?

- Spring Container creates only one instance of the bean, by default
- It is cached in memory
- All requests for the bean
 - will return a SHARED reference to the SAME bean

```
Coach theCoach = context.getBean("myCoach", Coach.class);
```

...

```
Coach alphaCoach = context.getBean("myCoach", Coach.class);
```

TrackCoach

Trans

We can explicitly specify the bean scope as below

```
<beans ... >

    <bean id="myCoach"
          class="com.luv2code.springdemo.TrackCoach"
          scope="singleton">
        ...
    </bean>

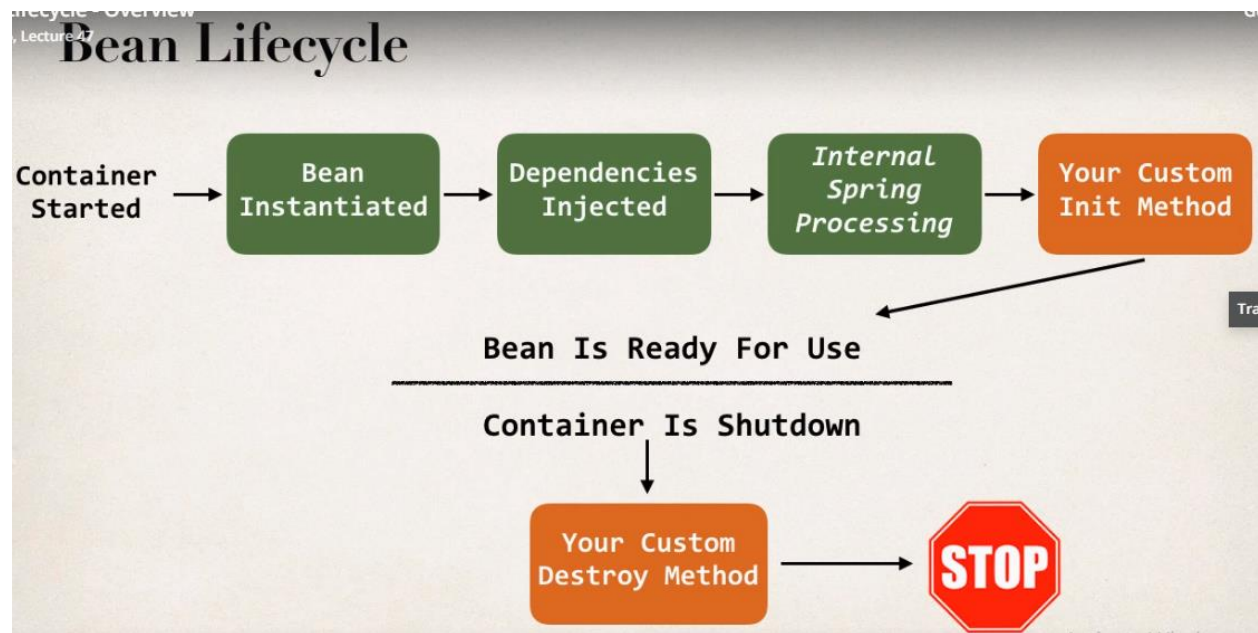
</beans>
```



Types of Scope

Scope	Description
singleton	Create a single shared instance of the bean. Default scope.
prototype	Creates a new bean instance for each container request.
request	Scoped to an HTTP web request. Only used for web apps.
session	Scoped to an HTTP web session. Only used for web apps.
global-session	Scoped to a global HTTP web session. Only used for web apps.

Bean lifecycle Methods:



Bean Lifecycle Methods / Hooks

- You can add custom code during **bean initialization**
 - Calling custom business logic methods
 - Setting up handles to resources (db, sockets, file etc)
- You can add custom code during **bean destruction**
 - Calling custom business logic method
 - Clean up handles to resources (db, sockets, files etc)

Init: method configuration

```
<beans ... >

  <bean id="myCoach"
        class="com.Luv2code.springdemo.TrackCoach"
        init-method="doMyStartupStuff">
    ...
  </bean>

</beans>
```

Set up bean
initialization

Any method
name

Destroy: method configuration

```
<beans ... >

  <bean id="myCoach"
        class="com.Luv2code.springdemo.TrackCoach"
        init-method="doMyStartupStuff"
        destroy-method="doMyCleanupStuff">
    ...
  </bean>

</beans>
```

Set up bean
destroy method

Any method
name

Development Process:

- Define Startup and destroy methods for class
- Configure those methods in Spring Config file
-

Note:- method cannot have any arguments, it should be no arg method.
Method can have any return type but cannot capture return data.
For prototype scope destroy methods are not called.

Annotations

Annotations minimizes the xml configuration.

ecture 52

What are Java Annotations?

- Special labels/markers added to Java classes
- Provide meta-data about the class
- Processed at compile time or run-time for special processing

Boot
Color: Silver
Style: Jewel
Code: 1460
SKU: 10072090
Size US: 8
Size UK: 6

Spring will scan the java classes with special annotation and register them with Spring container

Development process:

- Enable component scanning in Spring config file

```
<beans ... >  
  
    <context:component-scan base-package="com.Luv2code.springdemo" />  
  
</beans>
```

- Add `@Component` Annotation to java class

Step 2: Add the `@Component` Annotation to Java classes

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

- Note:- providing bean name is not necessary, we can use default class name with first letter as non caps.


- Eg: `getBean("tennisCoach",Coach.class);`
- Retrieve bean from Spring Container: same as previous `context.getBean()`

Autowiring:

Spring will look for class that matches the property and will inject it automatically.

Lecture 59

Autowiring Example



```
graph TD; Coach[Coach] --- FortuneService[FortuneService];
```

- Injecting FortuneService into a Coach implementation
- Spring will scan @Components
- Any one implements FortuneService interface???
- If so, let's inject them. For example: *HappyFortuneService*

Autowiring Injection types:

- Constructor Injection

Development Process - Constructor Injection

1. Define the dependency interface and class
2. Create a constructor in your class for injections
3. Configure the dependency injection with **@Autowired** Annotation

Step-By-Step

Question

I have finished the video "Constructor Injection - Writing Code part2".

I have commented the Autowired annotation. But still it worked. How did it work?

```
//@Autowired
```

```
public TennisCoach(FortuneService theFortuneService) {  
    System.out.println(" theFortuneService " + theFortuneService);  
    fortuneService = theFortuneService;  
}
```

===

Answer

This is a new feature of Spring 4.3.

Here is the snippet from the Spring Docs.

- Method Injection

ction - Overview

ure 84

Development Process - Setter Injection

1. Create setter method(s) in your class for injections
2. Configure the dependency injection with **@Autowired** Annotation

Step-By-Step

Note:- Method name can be any name, not necessary setter.

- Field Injection

Inject dependencies by setting field values
on your class directly

(even private fields)

Accomplished by using Java Reflection

- Overview

Development Process - Field Injection

Step-By-Step

1. Configure the dependency injection with Autowired Annotation
 - ❖ Applied directly to the field
 - ❖ No need for setter methods

Question

What if there are multiple implementations of FortuneService Interface?

If we use above approach then we will get Error message
"NoUniqueBeanDefinitionException".

Solution:

With @Autowired we should use another annotation
@Qualifier(<beannname>)

Eg: @Qualifier("happyFortuneService")



Note:- when the class name is having 2nd letter CAPS it should be used as it is.

When using with Constructor it should be as below

@Autowired

```
public TennisCoach(@Qualifier("randomFortuneService")
FortuneService theFortuneService)
```

To Inject values from property file use @Value annotation

Scope:

@Scope("singleton") or

BeanLifeCycleMethods:

Just like Init and destroy we have below

Development Process

1. Define your methods for init and destroy
2. Add annotations: @PostConstruct and @PreDestroy

```
@Component
public class TennisCoach implements Coach {

    @PostConstruct
    public void doMyStartupStuff() { ... }

    ...

}
```

Code will execute after constructor
and
after injection of dependencies

```
@Component
public class TennisCoach implements Coach {

    @PreDestroy
    public void doMyCleanupStuff() { ... }

    ...

}
```

Code will execute **before**
bean is destroyed

Special Note about @PostConstruct and @PreDestroy Method Signatures

Section 9, Lecture 80

Special Note about @PostConstruct and @PreDestroy Method Signatures

I want to provide additional details regarding the method signatures of @PostConstruct and @PreDestroy methods.

Access modifier

The method can have any access modifier (public, protected, private)

Return type

The method can have any return type. However, "void" is most commonly used. If you give a return type just note that you will not be able to capture the return value. As a result, "void" is commonly used.

Method name

The method can have any method name.

Arguments

The method can not accept any arguments. The method should be no-arg.

For "prototype" scoped beans, Spring does not call the @PreDestroy method.

Configuring Spring Container using JavaCode

3 Ways of Configuring Spring Container

1. Full XML Config

```
<!-- define the dependency -->
<bean id="myFortuneService"
      class="com.luv2code.springdemo.HappyFortuneService">
</bean>

<bean id="myCoach"
      class="com.luv2code.springdemo.TrackCoach">

  <!-- set up constructor injection -->
  <constructor-arg ref="myFortuneService" />
</bean>

<bean id="myCricketCoach"
      class="com.luv2code.springdemo.CricketCoach">

  <!-- set up setter injection -->
  <property name="fortuneService" ref="myFortuneService" />
</bean>
```

2. XML Component Scan

```
<context:component-scan base-package="com.luv2code.springdemo" />
```

3. Java Configuration Class

```
package com.luv2code.springdemo;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.luv2code.springdemo")
public class SportConfig {

}
```

No XML!

Development Process

Step-By

1. Create a Java class and annotate as **@Configuration**
2. Add component scanning support: **@ComponentScan** (optional)
3. Read Spring Java configuration class
4. Retrieve bean from Spring container

1.



```
@Configuration
public class SportConfig {

}
```

2.




```
@Configuration
@ComponentScan("com.luv2code.springdemo")
public class SportConfig {

}
```

3.

```
AnnotationConfigApplicationContext context =
    new AnnotationConfigApplicationContext(SportConfig.class);
```



4. Retrieve beans

Defining beans:

Development Process

1. Define method to expose bean
2. Inject bean dependencies
3. Read Spring Java configuration class
4. Retrieve bean from Spring container

Step 1: Define method to expose bean

This method name
will be the "bean id"

No component scan

```
onfig {  
  
    @Bean  
    public Coach swimCoach() {  
        SwimCoach mySwimCoach = new SwimCoach();  
  
        return mySwimCoach;  
    }  
}
```

Step 2: Inject bean dependencies

```
@Configuration
public class SportConfig {

    @Bean
    public FortuneService happyFortuneService() {
        return new HappyFortuneService();
    }

    @Bean
    public Coach swimCoach(FortuneService fortuneService) {
        SwimCoach mySwimCoach = new SwimCoach( happyFortuneService() );

        return mySwimCoach;
    }
}
```

Method name is bean ID

To read from properties file- @PropertySource(classpath:sport.properties)
and @Value annotation, This @Value should be used in SwimCoach.java

Before Spring 4.3 It had used below for property

```
@Bean
public static PropertySourcesPlaceholderConfigurer
    propertySourcesPlaceholderConfigurer() {

    return new PropertySourcesPlaceholderConfigurer();
}
```