

Destroy Lifecycle and Prototype Scope:

QUESTION: How can I create code to call the destroy method on prototype scope beans?

Answer:

You can destroy prototype beans but custom coding is required.

Development Process

1. Create a custom bean processor. This bean processor will keep track of prototype scoped Beans. During shutdown it will call the destroy() method on the prototype scoped beans. The custom processor is configured in the spring config file.

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

<bean id="myCoach"
      class="com.luv2code.springdemo.TrackCoach"
      scope="prototype"
      init-method="doMyStartupStuff">
  <!-- set up the constructor injection -->
  <constructor-arg ref="myFortuneService"/>
</bean>
```

2. The prototype scoped beans implement the **DisposableBean** interface. This interface defines a **destroy()** method.

```
public class TrackCoach implements Coach, DisposableBean {

    @Override
    public void destroy() throws Exception {
        // TODO Auto-generated method stub
        System.out.println("TrackCoach: inside method doMyCleanupStuffYoYo");
    }
}
```

3. The Spring configuration does not require to use the destroy-method attribute. You can safely remove it.

```
<bean id="myCoach"
      class="com.luv2code.springdemo.TrackCoach"
      scope="prototype"
      init-method="doMyStartupStuff">

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

4. The custom bean processing is handled in the MyCustomBeanProcessor class.

What are Java Annotations?

Special labels/markers added to Java Classes, and they actually give you the metadata about the class.

Basically, we have Annotations that can be processed at compile time or run time for special processing's.

Why Spring Configuration with Annotations?

1. XML configuration can be verbose for the very large projects. Imagine a project we have a very large spring project and we have 100 beans, and you have to list all those in the XML config file.
2. Instead, we can configure our spring beans with annotations.
3. Annotations will minimize the XML configuration.

So, once we add an annotation to a class then spring will actually scan the java classes that has spring annotation on it and it will automatically register it with spring container.

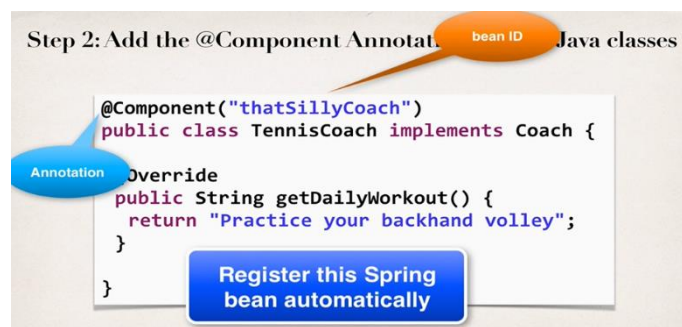
Development Process

1. Enable component scanning in Spring config file.
So Instead of listing all the beans in config file we can remove all that stuff and simply have one entry here. We will give the base package that we want to scan and spring will go and scan through the package and all the sub packages and it will try to identify the classes having annotation on it and it will register it with spring container automatically.

```
<!-- Add entry to enable component scanning -->  
<context:component-scan base-package="com.Luv2code.springdemo"/>
```

2. Add the @Component Annotation to your Java Classes

It will tell the spring that it's a special spring bean and we like spring to register it. And inside the @Component Annotation, we give the bean id that we want to use.



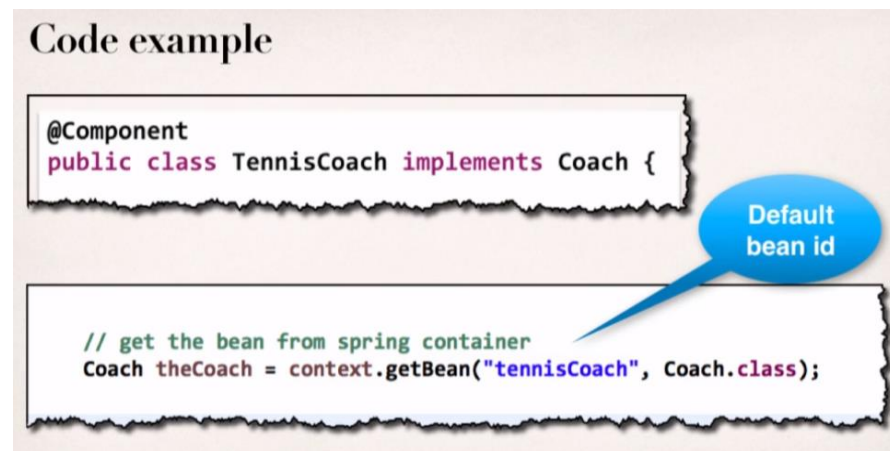
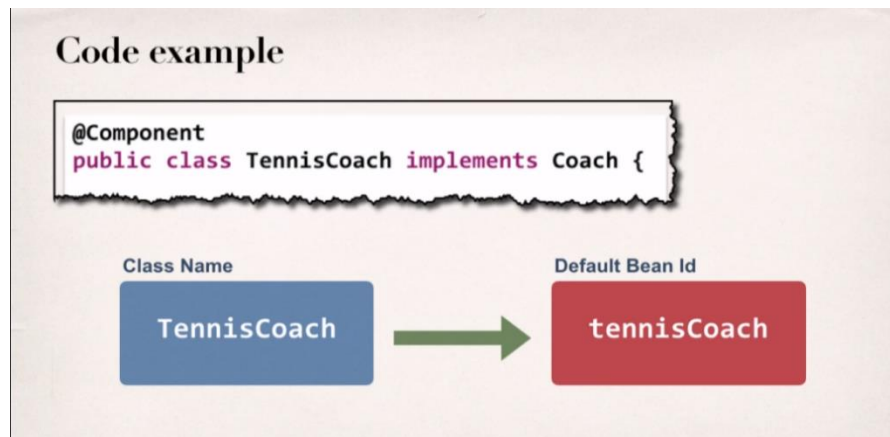
3. Retrieve bean from Spring Container.

Same as before:

```
//retrieve bean from spring container  
Coach theCoach = context.getBean("thatSillyCoach",Coach.class);
```

Spring also supports Default bean IDs:

Default bean Id: the class name, make first letter lower-case



Spring Dependency Injection with Annotations and Autowiring:

1. What is Spring AutoWiring?

For Dependency Injection Spring can use AutoWiring.

Basically, Spring will look for a class that matches the given property and it will match by Type: so, the type can be either class or interface. And once spring find the match it will automatically inject it. Hence it is called Autowired.

2. Autowiring Example:

- Spring will scan all the @Components and it will say I need to satisfy the dependency and we need to inject FortuneService.
- Spring will ask if anyone implements FortuneService interface? If so, Spring will grab that bean or component and actually inject it.

3. Autowiring Injection types:

- Constructor Injection
- Setter Injection
- Field Injections

➤ Development Process – Constructor Injection

- Define the dependency Interface and class

```
public interface FortuneService {  
    public String getYourDailyFortune();  
}  
  
@Component  
public class HappyFortuneService implements FortuneService {  
    @Override  
    public String getYourDailyFortune() {  
        return "Today is your lucky day!";  
    }  
}
```

- Create a constructor in your class for injections.

```

@Component
public class TennisCoach implements Coach {

    private FortuneService fortuneService;

    @Autowired
    public TennisCoach(FortuneService fortuneService) {

        this.fortuneService = fortuneService;
    }
}

```

We have used Constructor injection with the **@Autowired** Annotation. So, Spring will scan for the **@Component** that is implementing FortuneService interface. And in our example, that's the **HappyFortuneService** Component which will meet the requirement.

- Configure the dependency injection with **@Autowired** Annotation.

```

@Autowired
public TennisCoach(FortuneService fortuneService) {

    this.fortuneService = fortuneService;
}

@Override
public String getDailyFortunes() {
    // TODO Auto-generated method stub
    return fortuneService.getYourDailyFortune();
}

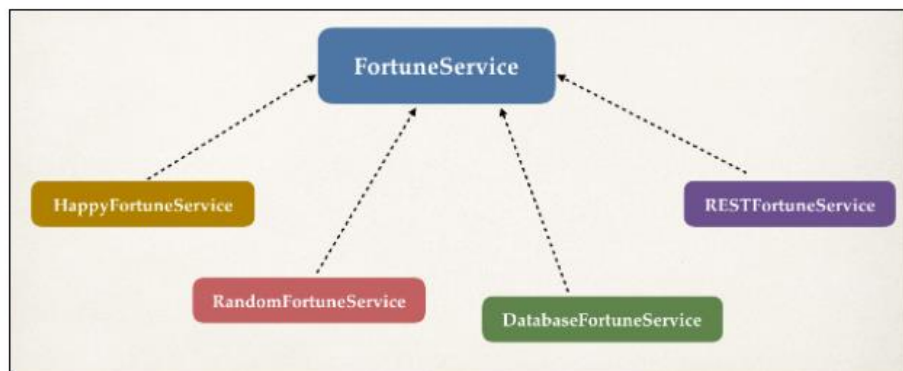
```

Autowiring FAQ: What if there are Multiple Implementations?

AUTOWIRING

FAQ: What if there are multiple FortuneService implementations?

When using autowiring, what if there are multiple FortuneService implementations? Like in the image below?



Answer

Spring has special support to handle this case. Use the **@Qualifier** annotation.

➤ Setter injection: Development Process

1. Create Setter Method(s) in your class for injections.

```
@Component
public class TennisCoach implements Coach {

    //define a setter method for injecting the fortuneService
    @Autowired
    public void setFortuneService(FortuneService fortuneService) {
        this.fortuneService = fortuneService;
    }
}
```

2. Configure the dependency injection with **@Autowired** Annotation.

Here we are going to add **@Autowired** annotation on the `setFortuneService()` method as previously we have set the annotation on the actual constructor but here instead we'll set the **@Autowired** annotation on setter method. Then spring goes to resolve this and go for injecting the dependency and it will search for the component that has implemented the `fortuneService` interface and if it finds it, it will make it available for dependency injection.

FAQ: How to inject properties file using Java annotations

FAQ: How to inject properties file using Java annotations

Answer:

1. Create a properties file to hold your properties. It will be a name value pair.

New text file: src/sport.properties

1. foo.email=myeasycoach@luv2code.com
2. foo.team=Silly Java Coders

Note the location of the properties file is very important. It must be stored in `src/sport.properties`

2. Load the properties file in the XML config file.

File: *applicationContext.xml*

Add the following lines:

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

This should appear just after the `<context:component-scan .../>` line

3. Inject the properties values into your Swim Coach: *SwimCoach.java*

```
1. @Value("${foo.email}")
2. private String email;
3.
4. @Value("${foo.team}")
5. private String team;
```

➤ Method Injection:

We can inject dependencies by calling any method on our class, all that we need to do is to give the **@Autowired** annotation on that method.

```
@Component
public class TennisCoach implements Coach {

    private FortuneService fortuneService;

    public TennisCoach() {
    }

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

➤ Field Injection with Annotations and Autowiring:

With field injection, we can actually inject the dependencies by setting the field values directly even for the private fields of our class. It happens behind the scene by using Java Reflection i.e. When spring creates the object, it will automatically set the field, then they don't need to go through any constructor or setter method and they can inject directly by making the use of Java Reflection technology.

Development Process: Field Injection

1. Configure the dependency injection with Autowired Annotation.

```
File: TennisCoach.java

public class TennisCoach implements Coach {

    @Autowired
    private FortuneService fortuneService;

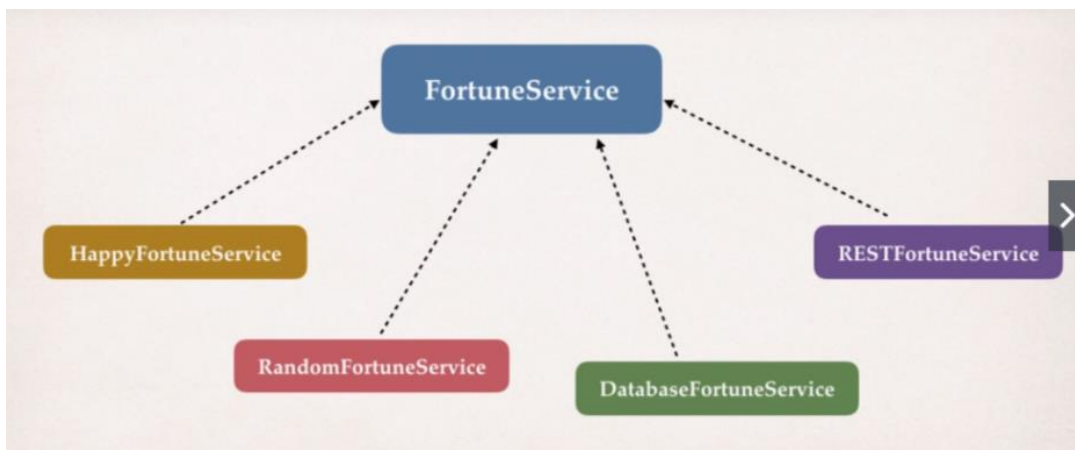
    public TennisCoach() {
    }

    // no need for setter methods
    ...
}
```

- 1.1. Applied directly to the field.
- 1.2. No need for setter methods.

Qualifiers for Dependency Injection:

Now incase we search for implementation of an interface and there is multiple component classes which implements that interface, so **which one spring will pick?**



Now in such case spring will give error messages:

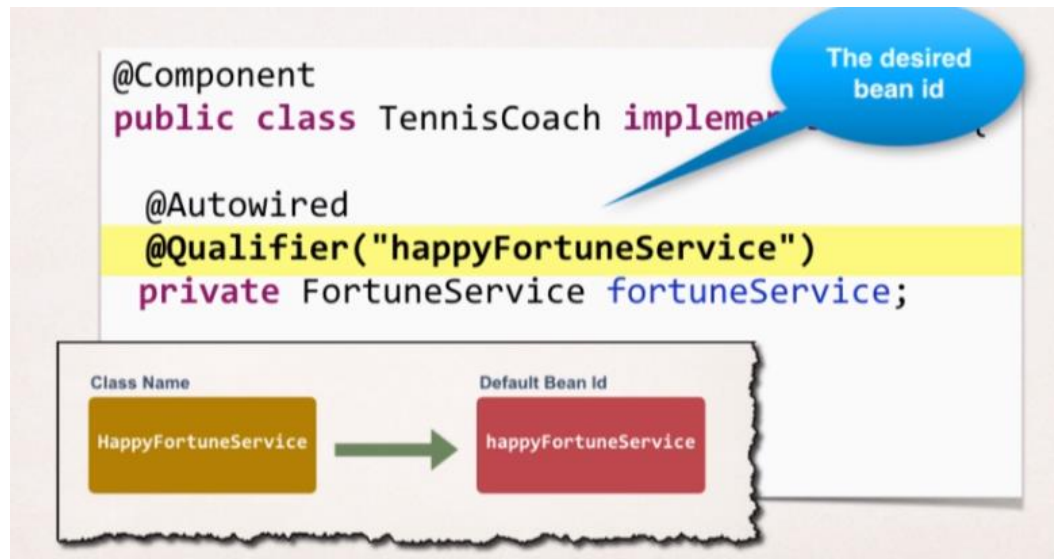
```
Error creating bean with name 'tennisCoach':
Injection of autowired dependencies failed

Caused by: org.springframework.beans.factory.NoUniqueBeanDefinitionException:
No qualifying bean of type [com.luv2code.springdemo.FortuneService] is defined:
expected single matching bean but found 4:

databaseFortuneService,happyFortuneService,randomFortuneService,RESTFortuneService
```


Spring will say that it is expecting a single matching bean but we found 4 implementations, So basically in order to resolve this we need to give spring a unique bean.

So we can use an annotation called **@Qualifier("bean ID")**: bean id of the component that we want to use. In this case we are using the default bean ID provided by the spring to the component.



Note: We can apply this @Qualifier annotation to all the injection types that we have studied so far.

Annotations - Default Bean Names - The Special Case

➤ Annotations - Default Bean Names ... and the Special Case

In general, when using Annotations, for the default bean name, Spring uses the following rule.

If the annotation's value doesn't indicate a bean name, an appropriate name will be built based on the short name of the class (with the first letter lower-cased).

For example:

HappyFortuneService --> happyFortuneService

However, for the **special case of when BOTH the first and second characters of the class name are upper case**, then the name is **NOT converted**.

For the case of RESTFortuneService

RESTFortuneService --> RESTFortuneService

No conversion since the first two characters are upper case.

Behind the scenes, Spring uses the **Java Beans Introspector** to generate the default bean name. Here's a screenshot of the documentation for the key method.

Using @Qualifier with Constructors:

@Qualifier is a nice feature , but it is tricky when used with constructors. The syntax is much different from other examples and not exactly intuitive.

You have to place the @Qualifier annotation inside of the constructor arguments.

Ex.

```
@Autowired
public TennisCoach(@Qualifier("randomFortuneService") FortuneService fortuneService) {
    System.out.println(">> TennisCoach: inside constructor using @Autowired and @Qualifier");
    this.fortuneService = fortuneService;
}
```

Section 9: Spring Configuration with Java Annotations - Bean Scopes and Lifecycle Methods

0 / 8 | 17min

@Scope Annotation

```
@Component
@Scope("singleton")
public class TennisCoach implements Coach {
    ...
}
```

Prototype scope: new object for each request

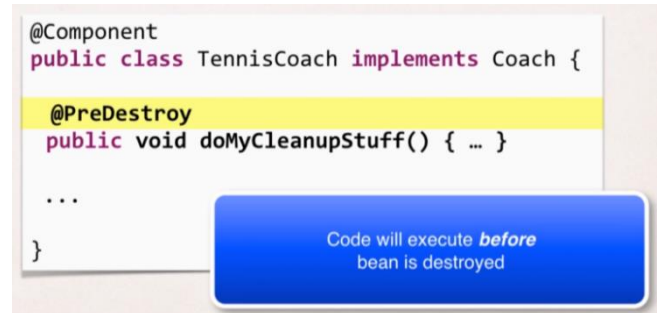
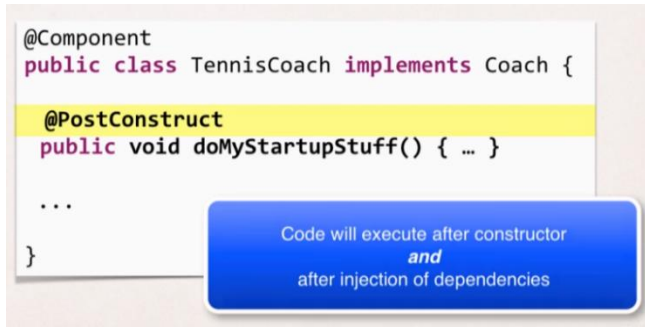
```
@Component
@Scope("prototype")
public class TennisCoach implements Coach {
    ...
}
```

Bean LifeCycle Methods/Hooks

➤ Development Process

- Define your methods for init and destroy. So for initialization we can choose any method name and given any method name we can simply annotated with @Postconstruct. And likewise, we can do for destroy.

- Add annotations: @PostConstructor and @PreDestroy



HEADS UP - FOR JAVA 9 USERS - @PostConstruct and @PreDestroy

If you are using Java 9 or higher, then you will encounter an error when using `@PostConstruct` and `@PreDestroy` in your code.

Error

- Eclipse is unable to import `@PostConstruct` or `@PreDestroy`
- This happens because of Java 9 or higher.
- When using Java 9 or higher, `javax.annotation` has been removed from its default classpath. That's why Eclipse can't find it.

Solution

1. Download the `javax.annotation-api-1.3.2.jar` from

<https://search.maven.org/remotecontent?filepath=javax/annotation/javax.annotation-api/1.3.2/javax.annotation-api-1.3.2.jar>

2. Copy the Jar file to the **lib folder** of your project
3. Use the following Steps to add it to your Java Build Path.
 - Right-click your project, select **Properties**.
 - On left-hand side, click **Java Build Path**.
 - In Top-corner of dialog, click **Libraries**.
 - Click **classpath** and then click **Add jars...**
 - Navigate to the JAR File **<your-project>/lib/javax.annotation-api-1.3.2.jar**
 - Click **OK** then click **Apply and Close**.
4. **Eclipse** will perform a rebuild of your project and it will resolve the related build errors.

Section 10: Spring Configuration with Java Code (no xml)

0 / 13 | 40min

We Will configure the spring container using Java code i.e., instead of using XML file we will use java code to configure.

There are actually 3 ways to configure the Spring container:

1. Full XML Config
2. XMLComponent Scan
3. Java Configuration Class

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!

➤ Java Configuration Development Process:

- Create a Java Class and annotate as **@Configuration**.



```
@Configuration
public class SportConfig {

}
```

- Add component scanning support **@ComponentScan** (optional).

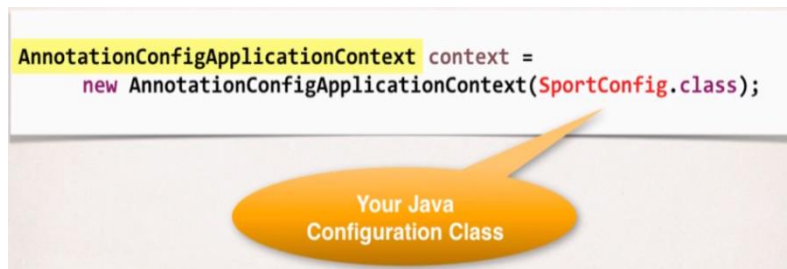


```
@Configuration
@ComponentScan("com.luv2code.springdemo")
public class SportConfig {
}
```

Package to scan

Optional

- Read Spring Java configuration class.



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

Your Java Configuration Class

- Retrieve bean from Spring container.



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

Defining Beans with Java Code:

Let's Introduce a new class here SwimCoach which will implement Coach interface.



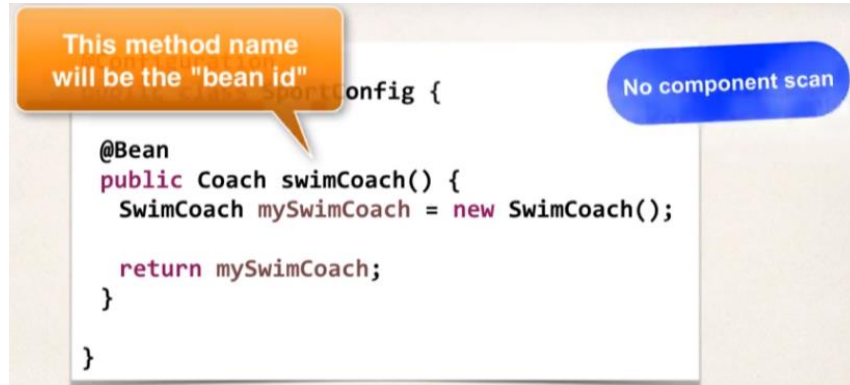
Here this SwimCoach will also have FortuneService implementation and we will learn how to inject that fortuneService using just raw Java code.

➤ Development Process:

- Define Method to Expose bean

In our configuration class here i.e., **SportsConfig class** we are going to use a new annotation called **@Bean** and this will basically define the bean and here we will provide a method called `swimCoach()` which will internally creates an object of bean `SwimCoach` and will return it.

And in this Example we are not using **@ComponentScan()** and in this configuration we will actually Define each bean individually within this configuration class.



- Inject bean dependencies

Here the method name is the “**bean id**” which spring will use when we register this bean with the application context and `happyFortuneService()` method is returning a new instance of `HappyFortuneService` implementation and this bean can be simply used when we create the `SwimCoach` instance and we will call the bean reference of `happyFortuneService`. (Constructor Dependency Injection).

Spring will now intercept the dependency and will give the object accordingly based on the bean scope.



- Read Spring Java Configuration class

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


- Retrieve bean from Spring Container.



FAQ: How @Bean works behind the scenes

Question:

During All Java Configuration, how does the @Bean annotation work in the background?

Answer

This is an advanced concept. But I'll walk through the code line-by-line.

For this code:

```
1. @Bean
2. public Coach swimCoach() {
3.     SwimCoach mySwimCoach = new SwimCoach();
4.     return mySwimCoach;
5. }
```

At a high-level, Spring creates a bean component manually. By default, the scope is singleton. So, any request for a "swimCoach" bean, will get the same instance of the bean since singleton is the default scope.

However, let's break it down line-by-line

```
1. @Bean
```

The @Bean annotation tells Spring that we are creating a bean component manually. We didn't specify a scope so the default scope is singleton.

```
1. public Coach swimCoach(){
```

This specifies that the bean will have an id of "swimCoach". The method name determines the bean id. The return type is the Coach interface. This is useful for dependency injection. This can help Spring find any dependencies that implement the Coach interface.

The @Bean annotation will intercept any requests for "swimCoach" bean. Since we didn't specify a scope, the bean scope is singleton. As a result, it will give the same instance of the bean for any requests.

```
1. SwimCoach mySwimCoach = new SwimCoach();
```

This code will create a new instance of the SwimCoach.

```
1.  return mySwimCoach;
```

This code returns an instance of the swimCoach.

Now let's step back and look at the method in its entirety.

```
1.  @Bean
2.  public Coach swimCoach() {
3.      SwimCoach mySwimCoach = new SwimCoach();
4.      return mySwimCoach;
5.  }
```

It is important to note that this method has the @Bean annotation. The annotation will intercept ALL calls to the method "swimCoach()". Since no scope is specified the @Bean annotation uses singleton scope. Behind the scenes, during the @Bean interception, it will check in memory of the Spring container (applicationContext) and see if this given bean has already been created.

If this is the first time the bean has been created then it will execute the method as normal. It will also register the bean in the application context. So that it knows that the bean has already been created before. Effectively setting a flag.

The next time this method is called, the @Bean annotation will check in memory of the Spring container (applicationContext) and see if this given bean has already been created. Since the bean has already been created (previous paragraph) then it will immediately return the instance from memory. It will not execute the code inside of the method. Hence this is a singleton bean.

The code for

```
1.  SwimCoach mySwimCoach = new SwimCoach();
2.  return mySwimCoach;
```

is not executed for subsequent requests to the method public Coach swimCoach() . This code is only executed once during the initial bean creation since it is singleton scope.

That explains how @Bean annotation works for the swimCoach example.

=====

Now let's take it one step further.

Here's your other question

>> Please explain in detail what's happening behind the scene for this statement.

```
1.  return new SwimCoach(sadFortuneService())
```


The code for this question is slightly different. It is injecting a dependency.

In this example, we are creating a SwimCoach and injecting the sadFortuneService().

```
1.      // define bean for our sad fortune service
2.      @Bean
3.      public FortuneService sadFortuneService() {
4.          return new SadFortuneService();
5.      }
6.
7.      // define bean for our swim coach AND inject dependency
8.      @Bean
9.      public Coach swimCoach() {
10.         SwimCoach mySwimCoach = new SwimCoach(sadFortuneService());
11.
12.         return mySwimCoach;
13.     }
```

Using the same information presented earlier

The code

```
1.      // define bean for our sad fortune service
2.      @Bean
3.      public FortuneService sadFortuneService() {
4.          return new SadFortuneService();
5.      }
```

In the code above, we define a bean for the sad fortune service. Since the bean scope is not specified, it defaults to singleton.

Any calls for sadFortuneService, the @Bean annotation intercept the call and checks to see if an instance has been created. First time through, no instance is created so the code executes as desired. For subsequent calls, the singleton has been created so @Bean will immediately return with the singleton instance.

Now to the main code based on your question.

```
1.     return new SwimCoach(sadFortuneService())
```

This code creates an instance of SwimCoach. Note the call to the method sadFortuneService(). We are calling the annotated method above. The @Bean will intercept and return a singleton instance of sadFortuneService. The sadFortuneService is then injected into the swim coach instance.

This is effectively dependency injection. It is accomplished using all Java configuration (no xml).

This concludes the line-by-line discussion of the source code. All of the behind-the-scenes work.

FAQ: What is a real-time use case for @Bean?

Here is a real-time use case of using @Bean: *You can use @Bean to make an existing third-party class available to your Spring framework application context.*

For example, I was recently working on a global real-time project using Amazon Web Services. The project made use of the [Amazon Simple Storage Service \(AWS S3\)](#). This is remote service that provides object storage in the cloud. You can think of AWS S3 at a high-level as a remote file server for storing files (pdfs, pngs etc).

Our Spring application needed to integrate with AWS S3 and store pdf documents. Amazon provides an AWS SDK for integrating with AWS S3. Their API provides a class, [S3Client](#). This is a regular Java class that provides a client interface to the AWS S3 service. We needed to share the S3Client object in various services in our Spring application. However, the S3Client does not have the @Component annotation. The S3Client does not use Spring.

Since the S3Client is part of the AWS framework, we can't modify the source code for the S3Client directly. We can't simply add the @Component annotation to the S3Client source code. As a result, we need an alternative solution.

But no problem, by using the @Bean annotation, I can wrap this third-party class, S3Client, as a Spring bean. And then once it is wrapped using @Bean, it is as a singleton object and available in our Spring framework application context. I can now easily share this bean in my app using dependency injection and @Autowired. So, think of the @Bean annotation was a wrapper / adapter for third-party classes. You want to make the third-party classes available to your Spring framework application context.

Here's a real-time example

Here is a snippet from our @Configuration class. We create an instance of the S3Client and wrap it as a Spring bean. The default scope is singleton. It is now available in our application context and we can inject it to other parts of our Spring application using @Autowired.

```
1. @Bean
2.     public S3Client remoteClient() {
3.
4.         // Create an S3 client to connect to AWS S3
5.         S3Client s3Client = S3Client.builder().region(Region.of(region))
6.             .credentialsProvider(StaticCredentialsProvider.create(awsCreds)).build();
7.
8.         return s3Client;
9.     }
```

In the code below, this is a Spring service that uses the S3Client. The service @Service annotation is a subclass of @Component. This code uses @Autowired to inject the bean named "remoteClient". This bean was created in the configuration code above using @Bean.

Once the bean is injected, then our method can use this to interact with the Amazon S3 service. In this real-time project, we were processing insurance claims. We store the PDF invoices in the cloud using the AWS S3 service.

```
1. @Service
2.     public class InsuranceClaimsServiceImpl implements ClaimsService {
3.
4.         @Autowired
5.         private S3Client remoteClient;
6.
7.         ...
8.
9.         public void processClaim(Claim theClaim) {
10.
11.
12.             // read claim data
13.             FileData fileData = theClaim.getFileData("payerInvoice");
14.             String fileName = theClaim.getSubmittedFileName();
15.
16.             // get the input stream and file size
17.             InputStream fileInputStream = fileData.getInputStream();
18.             long contentLength = fileData.getSize();
19.
20.             //
21.             // store claim data in AWS S3
22.             //
23.
24.             // Create a put request for the object
25.             PutObjectRequest putObjectRequest = PutObjectRequest.builder()
26.                 .bucket(bucketName)
27.                 .key(subDirectory + "/" + fileName)
28.                 .acl(ObjectCannedACL.BUCKET_OWNER_FULL_CONTROL).build();
29.
30.             // perform the putObject operation to AWS S3 ... using our autowired bean
31.             remoteClient.putObject(putObjectRequest, RequestBody.fromInputStream(fileInputStre
32. am, contentLength))
33.         }
```

As you can see, I was able to wrap a third-party class as a Spring bean. The AWS S3Client object was not originally annotated with @Component. The S3Client is not aware of Spring. But I could manually wrap it using @Bean. By doing this, the object is now available in our Spring application context. We can now share/reuse this bean in other areas of our Spring app by using dependency injection and @Autowired.

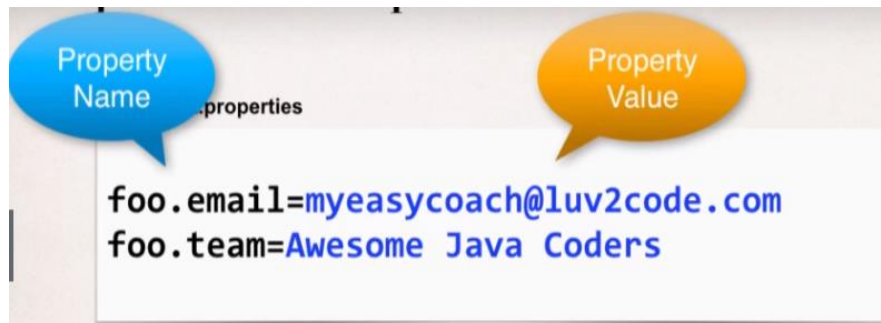
For other services in our application, if they need access to the S3client (singleton) then they can simply inject it using @Autowired. No need for each service to create a new instance of the S3Client every time. This keeps the application efficient in terms of memory and performance.

In summary: ***You can use @Bean to make an existing third-party class available to your Spring framework application context.***

Injecting values from Properties File:

Development Process

1. Create Properties File.



2. Load Properties File in Spring config.



3. Reference the Values from the Properties file

