Spring Core Notes

**Inversion of Control (IOC):**

The practice of outsourcing object creation and management is called inversion of control. Spring framework provides Object factory for this.

**Spring Container:**

Primary functions of spring container are

* Creating and managing Objects (Inversion of Control)
* Injecting objects into application (Dependency injection)

Spring Bean Factory

Give me baseball coach object

A

Spring Container

Configuration

Baseball

Coach

Cricket

Coach

Track

Coach

Here is your object…

My App

Dependencies

Dependencies

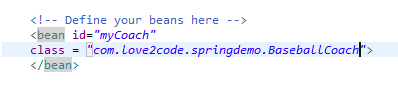
**Three ways to Configure Spring Container**

1. XML Configuration (Old way of doing)
2. Java Annotations (Modern)
3. Java Source Code (Modern)

**Spring Development process**

**XML Configuration method:**

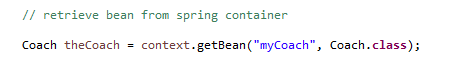
1. Configure Spring beans

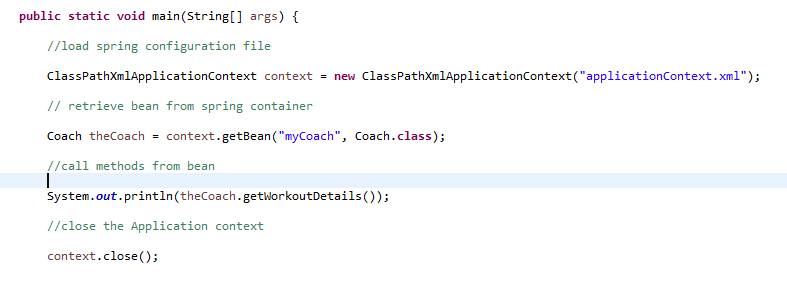


1. Create a Spring Container



1. Retrieve beans from Spring Container





Note: **Spring container** is often referred to as **ApplicationContext** in java world

**Spring Bean**: A “Spring Bean” is simply a Java Object. Spring Beans are normal Java objects created from Java Classes. When Java Objects are created by Spring Container then Spring refers to them as Spring Bean.

**Spring Dependency Injection**

Dependency Injection is a fundamental aspect of the Spring framework, through which the Spring container "injects" objects into other objects or "dependencies".

Car Factory

Give me a car

Customer



Assemble parts and make car



**Injection Types:**

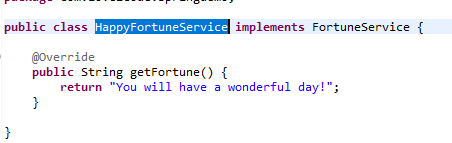
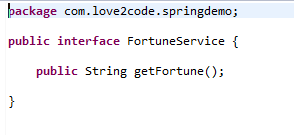
Two most common types

* Constructor Injection
* Setter Injection

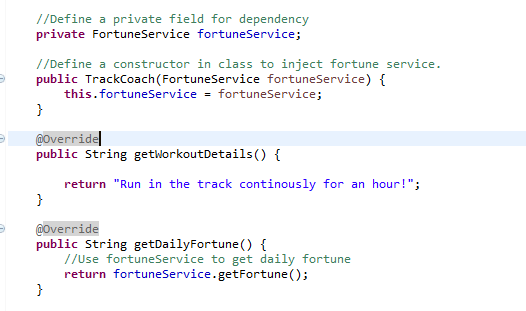
In Annotations we also have something called “auto-wiring”

**Constructor Injection steps**

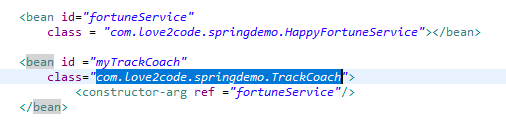
* Define dependency interface and class



* Create constructor in your class for injection

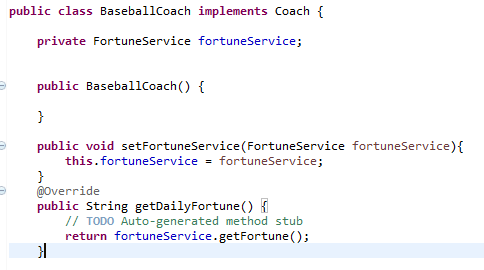


* Configure dependency injection in spring config file

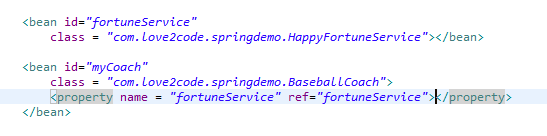


**Setter Injection Process**

* Creating setter methods in our class for injection

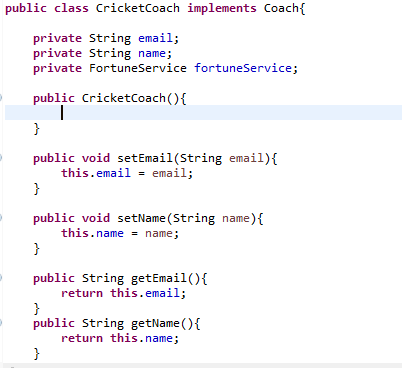


* Configure the dependency in Spring config file.

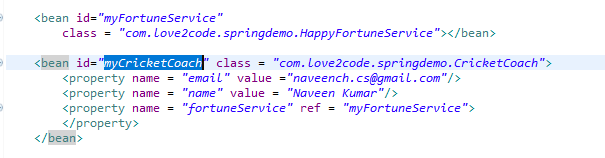


**Injecting Literal Values**

* Create a setter in class to inject literal values



* Configure the dependency in Spring config file.



**Injecting values from Properties file**

In the above injecting literals from config file, there were hardcode. So to avoid this we can read these properties from Properties files as well.

Development process:

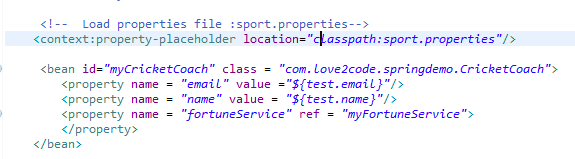
* Create Property file under src directory



* Load properties file into Spring configuration file



* Reference values from property file.



**Bean Scopes**

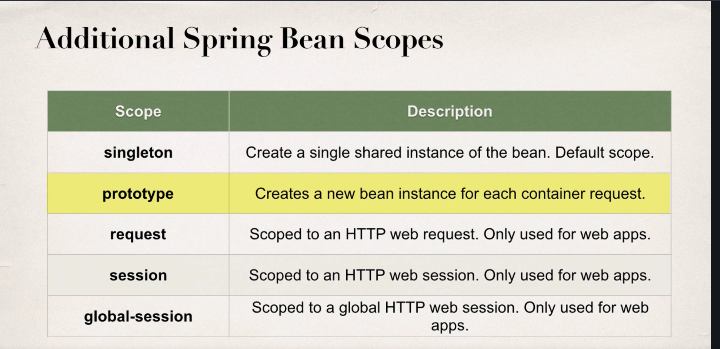
Scope refers to life cycle of a bean, which could be how long a bean lives? How many instances are created? How is the bean shared? Etc.

Default scope of bean is **Singleton**

What is a Singleton?

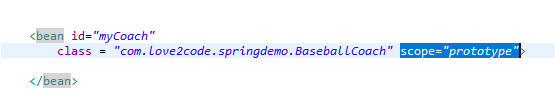
Spring container creates only one instance of a bean, by default and stores it in the cache memory.

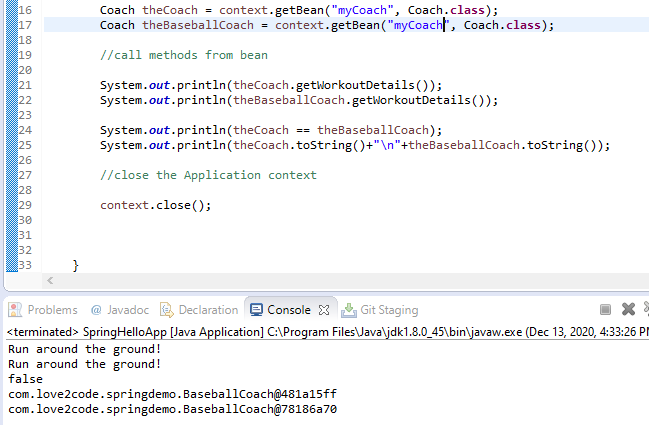
All requests to this bean will be given the shared reference to the same bean.



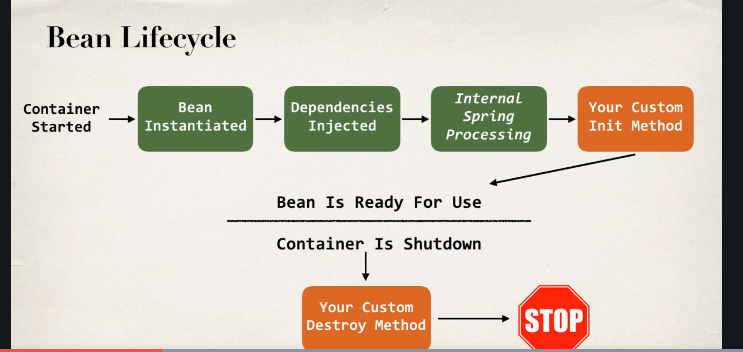
Prototype Scope:

A new object is created for each request.





**Bean Life Cycle Methods/ Hooks:**



You can add custom code during bean initialization like calling business logic methods, setting up handles for resources like db, files, sockets etc.

You can also do similar thing during the bean destruction like clean up handles to resources.

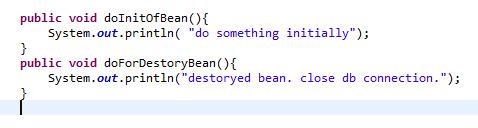
Development process:

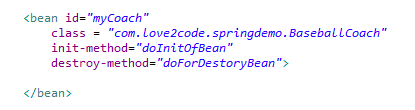
* Create methods for init and destroy in the class
* Configure the method names in config file.

**Special Note:** Defining init and destroy methods - Method Signatures

When using XML configuration,

* The method can have any access modifier (public, protected, private).
* 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.
* The method can have any method name.
* The method cannot accept any arguments. The method should be no-arg.





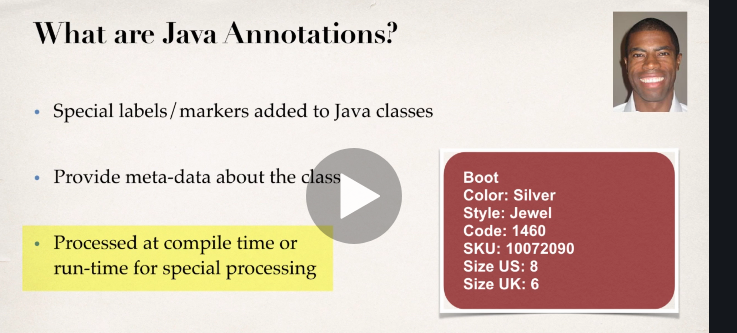
There is a subtle point you need to be aware of with "prototype" scoped beans.

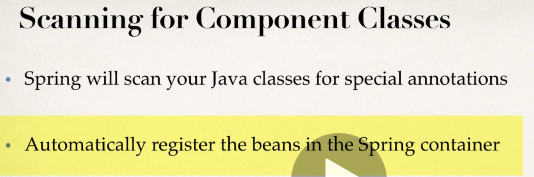
**For "prototype" scoped beans, Spring does not call the destroy method.  Gasp!**

In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean: the container instantiates, configures, and otherwise assembles a prototype object, and hands it to the client, with no further record of that prototype instance.

Thus, although initialization lifecycle callback methods are called on all objects regardless of scope, in the case of prototypes, configured destruction lifecycle callbacks are not called. The client code must clean up prototype-scoped objects and release expensive resources that the prototype bean(s) are holding.

**Spring Configuration with Java Anotations**





Development process

* Enable component scanning in spring config file



* Add @Component annotation to Java classes



You can give bean id with component annotation. But if we don’t give bean id explicitly then java will take default id as class name with first letter as lower case. For example in the above case it would be swimmingCoach.

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.

* Retrieve bean from spring container.

This is same as in xml configuration.

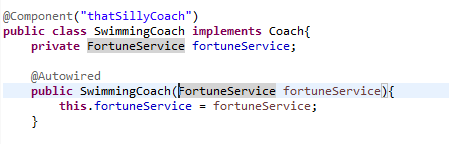
**Spring Dependency Injection using Annotations and Autowiring**

Autowiring Injection types

* Constructor injection
* Setter injection
* Field injection

Development process for construction injection:

* Define the dependency interface and class
* Create a constructor in your class for injection

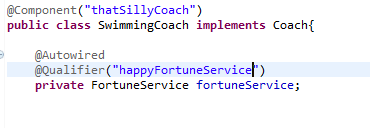


* Configure the dependency injection using Autowired annotation.

Here Spring will first scan all the components in the package configured in spring config file. Then if any of the component classes has @Autowired annotation on a constructor, then spring will check if there are any classes that are implementing the interface being used. If yes then spring will inject that bean into the original component.

But what if there are multiple classes (components) implementing the same interface?

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

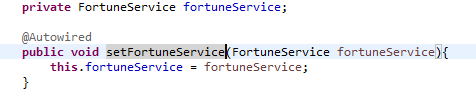


Note: As of Spring Framework 4.3, an @Autowired annotation on such a constructor is no longer necessary if the target bean only defines one constructor to begin with. However, if several constructors are available, at least one must be annotated to teach the container which one to use.

**Injection using Setter methods:**

Development process

* Create a setter method in your class for injection
* Configure the dependency injection using Autowired annotation.



**Field Injection with Annotations and Autowiring:**

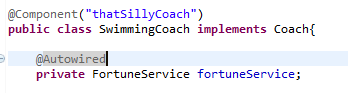
Inject dependencies by setting field values on your class directly. Even for the private fields.

This is done in the background using some java technology called java reflection.

Development process:

Configure the dependency injection using Autowired annotation

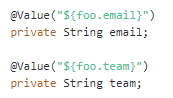
* Applied directly to the field
* No need for setter methods.



**Inject property values:**

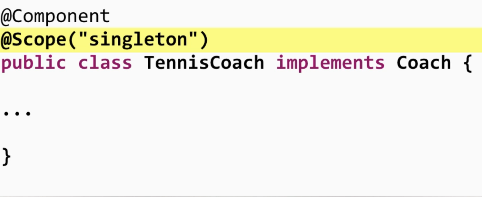
Creating the properties file and configuring it in spring config is same as for xml method.

But injection is done as below.

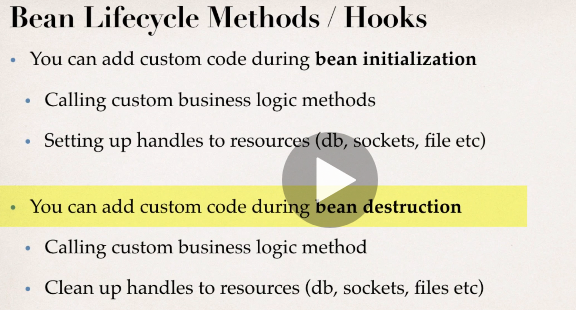


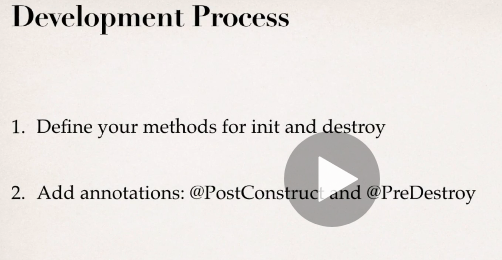
**Bean Scopes with Annotations:**

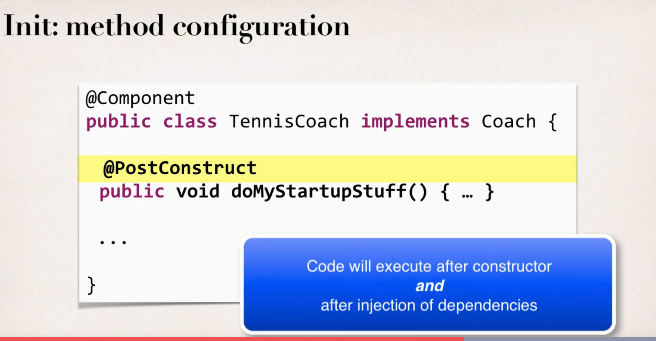
Default scope is Singleton here as well.

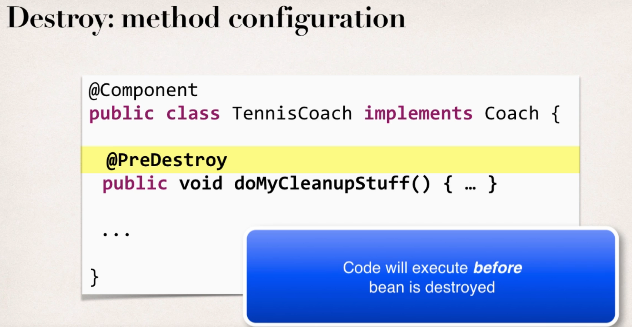


Bean lifecycle









**Spring configuration with Java Code (no XML):**

Instead of configuring Spring container using xml, configure it with purely Java.

Development process:

* Create a Java Class and annotate it as @Configuration
* Add component scanning support using @ComponentScan (optional)
* Read Spring Java configuration class
* Retrieve bean from Spring container.

Defining beans in Spring without xml:

In this process, we don’t use @Component and @ComponentScan annotations. But we directly define beans in Java Config file.

Development process:

* Define method to expose bean
* Inject bean dependencies
* Read Spring java configuration class
* Retrieve the bean from spring container.

**Inject values from properties file:**

Development process:

* Create properties file
* Load properties file in Spring config
* Reference value from properties file.