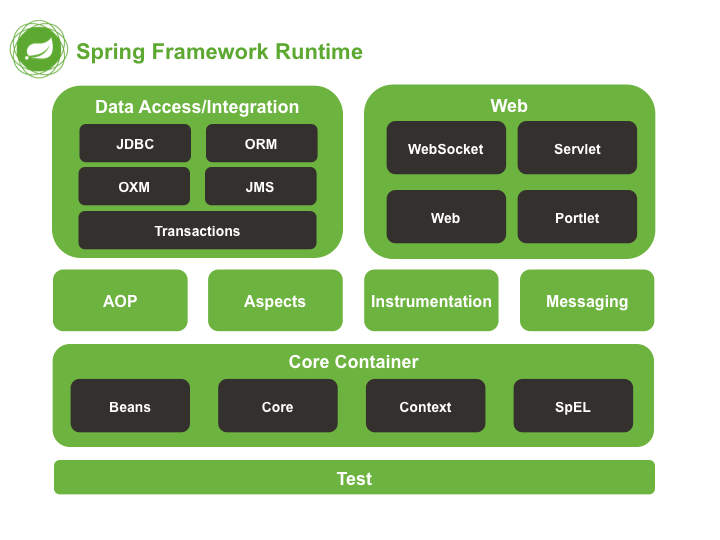
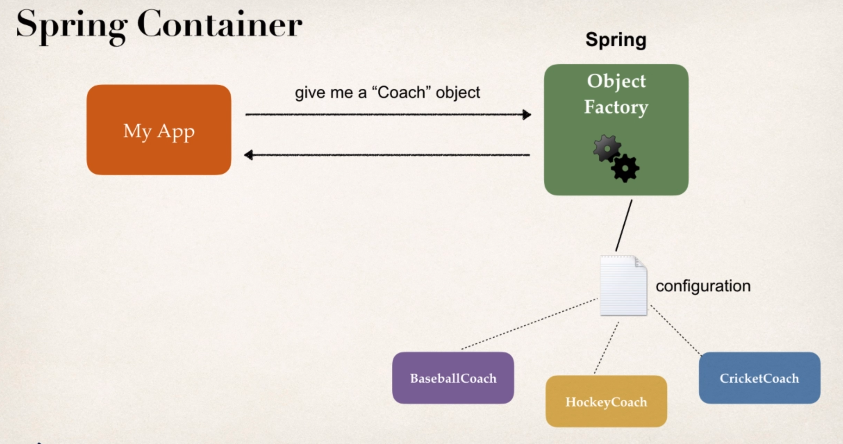
<https://repo.spring.io/release/org/springframework/spring/>



Inversion of Control: The approach of outsourcing the construction and managements of objects



**Primary Functions**

* Create and Manage Objects (Inversion of Control)
* Inject Object’s dependencies (Dependency Injection)

**There are 3 ways you can configure a Spring Container**

* XML Configuration (legacy)
* Java Annotations (Modern) + XML for component scan
* Java Source Code (Modern)

**Spring Development Process**

* Configure your Spring beans
* Create a Spring container, which is generally known as an Application Context
* Retrieve beans from the Spring container

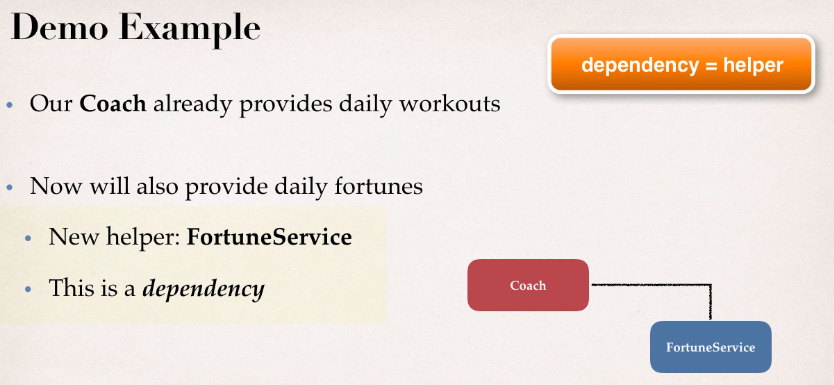
**What is a Spring Bean?**

A "Spring Bean" is simply a Java object. When Java objects are created by the Spring Container, then Spring refers to them as "Spring Beans". Spring Beans are created from normal Java classes .... just like Java objects.

The dependency injection principle: The client delegates to calls to another object the responsibility of providing its dependencies.

Dependency is same as helper objects.

**Dependency injection** is a technique whereby one object supplies the **dependencies** of another object. A "**dependency**" is an object that can be used, for example as a service. Instead of a client specifying which service it will use, something tells the client what service to use.



There are many types of dependency injection in Spring, two of the most commonly used are

* Constructor
* Setter

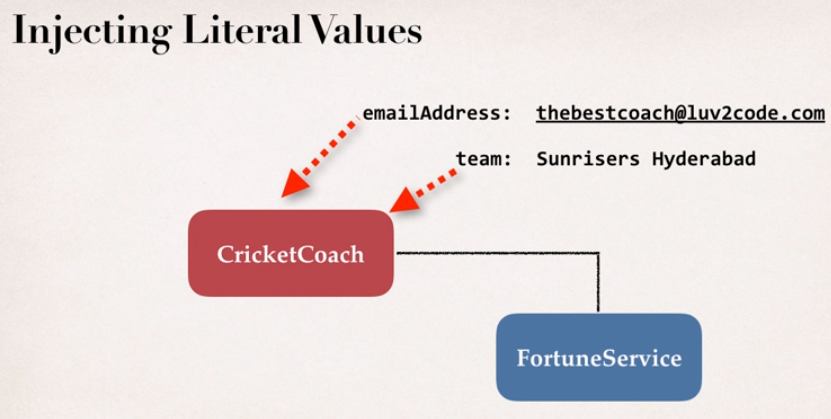
**Development Process – Constructor Injection**

* Define the dependency interface and class – FortuneService Interface and HappyFortuneService
* Create a Constructor in your class for injections – Create a constructor in BaseballCoach which will accept this dependency
* Configure the dependency injection in the Spring config file

Setter Injection: Inject dependencies by calling setter methods on your class.

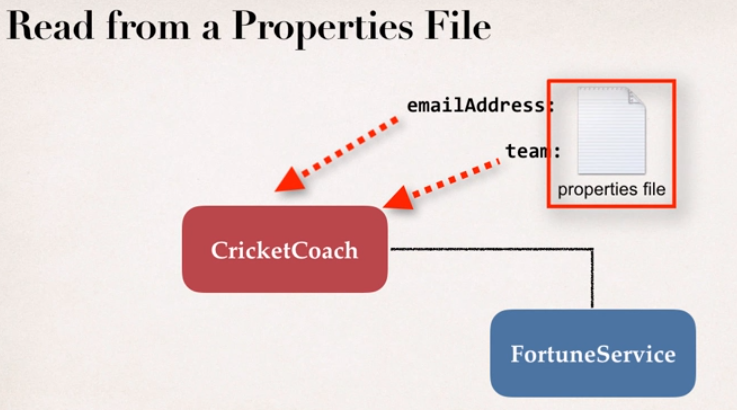
Development Process – Setter Injection

* Create Setter methods in your class for Injection
* Configure the dependency injection in Spring configuration file



Development process of injecting Literal values

* Create Setter methods in your class for Injections
* Configure the injection in the Spring configuration file



Development Process – Reading from a Properties file

* Create a Properties file
* Load Properties file in the Spring Configuration file
* Reference Properties from the Properties file

Bean Scopes

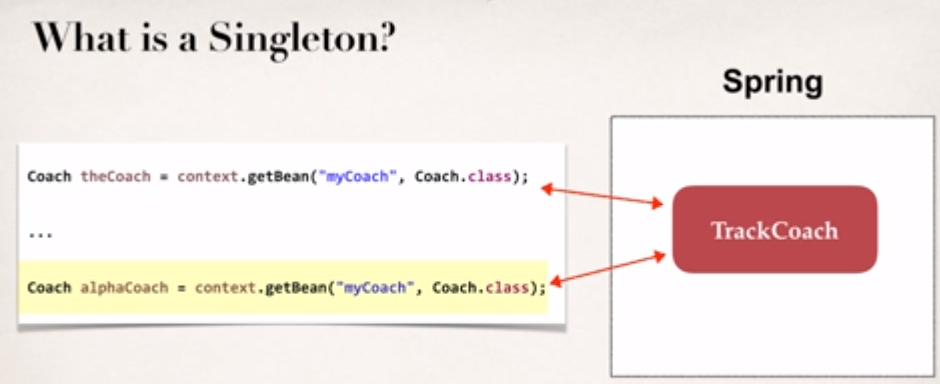
Scope Refers to the lifecycle of the bean, Default scope is **Singleton**

What is Singleton?

* Spring Container creates only one instance of the bean by default
* Its cached in memory
* All requests for the bean, will return the shared reference to the same bean.

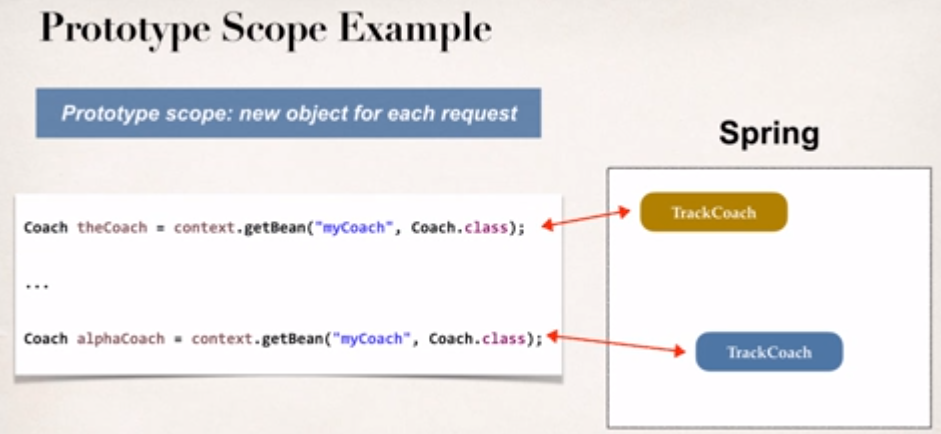
In the below diagram, both the calls to the TrackCoach will return the same instance, i.e. theCoach and alphaCoach will point to the same instance.

Best usecase for using Singleton will be Stateless Bean.



Additional Spring bean scopes

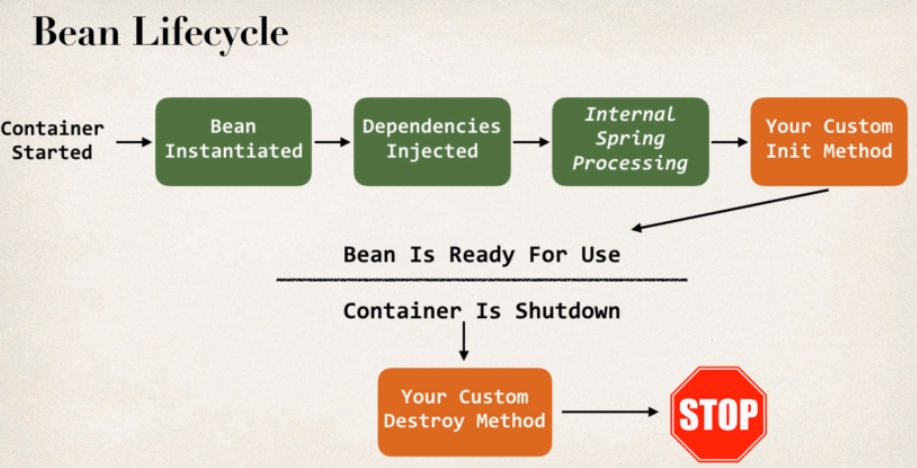
|  |  |
| --- | --- |
| singleton | Create a single shared instance bean, Default scope |
| prototype | Create a new bean instance for each container request |
| request | Scoped to a HTTP web request, only used for web applications |
| session | Scoped to a HTTP web session, only used for web applications |
| global-session | Scoped to a global HTTP web session, only used for web applications |



Custom Init and Destroy methods, these are also called hooks.

Development Process

* Define the custom init and destroy methods
* Specify the method names in the Spring configuration file



Access Modifier – can be any modifier, public, protected or private

Return Type – Return type cannot be accessed, hence void is the most commonly used, but it can be anything.

Method name – Can be anything

Arguments – Does not accept any parameters, should be no arguments.

Note: 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.

What are Java Annotations?

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

Example, @Override is processed at compile time.

XML configuration can be verbose in case of many beans.

Scanning for Component classes

* Spring will scan the java class for the special annotations
* Automatically register the beans with the Spring Container

Development Process for Annotations

* Enable component scanning in Spring configuration file
* Add **@Component** annotation in the Java Class
* Retrieve bean from the Spring Container

Default bean id in the @Component annotation will be class name, with the first letter, changed to lowercase

**What is Auto Wiring?**

* For dependency injection, spring can use auto-wiring
* Spring will look for class that matches the Property, matches by type, class or interface
* Spring will inject it automatically, hence auto-wiring

**Auto Wiring Injection types**

* Constructor Injection
* Setter Injection (Method Injection)
* Field Injections

**Auto Wiring example**

* Injecting FortuneService into Coach implementation
* Spring will scan @Components
* Any class that implements FortuneService Interface
* If so, inject the class, HappyFortuneService

**Development Process – Constructor Injection**

* Define the dependency Interface and Class
  + FortuneService & HappyFortuneService
* Create a Constructor in the class for Injection
* Configure the dependency injection with **@AutoWired** Annotation

Note: The @Autowired annotation is optional, please read the note below:

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.

**Setter Injection**, Inject dependencies by call setter methods in the class.

Development Process: Setter Injection

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

**Method Injection**, inject dependency by calling any method on the class.

**Field Injections**: Inject field dependencies by setting field values directly on the class (even private fields)

Accomplished using **Java Reflection**.

Development Process

* Configure the dependency injection with Autowired Annotation, applied directly to the field.
* No Need for setter methods.

**@Qualifier** Annotation in case there are multiple classes inheriting from the FortuneService

**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.

**@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.

Here's an example from our classroom example. I updated it to make use of constructor injection, with @Autowired and @Qualifier

@Autowired

public TennisCoach(**@Qualifier("randomFortuneService")** FortuneService theFortuneService) {

System.out.println(">> TennisCoach: inside constructor using @autowired and @qualifier");

fortuneService = theFortuneService;

}

Specifying the scope using annotations

@Scope(“singleton”)

@Scope(“prototype”)

Bean Lifecycle method annotations

* Define your methods for init and destroy
* Add annotations **@PostConstruct** and **@PreDestroy**

Using Java Configuration to configure the Spring container instead of using XML.

Development Process for Java Configuration

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

Development Process – Dependency injection using Java Class

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

Injecting the values from the Properties file

* Create the properties file
* Load Properties file in the Spring configuration file
* Reference values from the Properties file

Required for Version 4.2 and lower

// add support to resolve ${...} properties

@Bean

public static PropertySourcesPlaceholderConfigurer

propertySourcesPlaceHolderConfigurer() {

return new PropertySourcesPlaceholderConfigurer();

}