**4. Inversion of control**

The approach of outsourcing the construction and management of objects.

Spring container. Primary functions:

* Create and manage objects (Inversion of control)
* Inject object’s dependencies (Dependency Injection)

Configuring Spring Container:

* XML configuration file (legacy)
* Java Annotations (modern)
* Java Source Code (modern)

Spring Development Process

* Configure your Spring Beans
* Create a Spring Container
* Retrieve Beans from Spring Container

*Step1*.

<beans … >

<bean id =”myCoach”

class=”com.luv2code.springdemo.BaseballCoach”>

</bean>

</beans>

*Step2*. The Spring container is generally known as ApplicationContext

Specialized implementations:

* ClassPathXmlApplicationContext
* AnnotationConfigApplicationContext
* GenericWebApplicationContext
* Others

ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(“applicationContext.xml”);

*Step3*: Retrieve Beans from Container

Coach theCoach =context.getBean(“myCoach”, Coach.class);

Text

**5. Dependency Injection.**

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

Development Process – Constructor Injection:

1. Define the dependency interface and class
2. Create a constructor in your class of injections
3. Configure the dependency injection in Spring config file
   1. <bean id=”myCoach” … <constructor-arg ref=”DependencybeanId”>…

Development Process – Setter Injection:

1. Create setter method(s) in your class for injections
2. Configure the dependency in Spring config file

Development Process – Inject literal values

1. Create setter method(s) in your class for injection
2. Configure the injection in Spring config file
   1. <property name=”team” value=”Sunrisers Hyderabad”>

Development Process – inject literal values from the properties file

1. Create Properties file
2. Load properties file in spring config file

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

1. Reference values from Properties file
   1. ${the prop name}

**6**. **Spring Bean Scopes and Lifecycle**

Bean Scopes. Scope refers to the lifecycle of a bean:

* How long does the bean live?
* How many instances are created?
* How is the bean shared?

You can explicitly specify bean scope. If you want explicitly specify bean scope, then you make use of the scope attribute

Additional Spring Bean Scopes:

* 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 htttp web session. Only used for web apps.
* Global-session – scoped to a global HTTP web session. Only used for web apps.

Bean lifecycle:

1. Container started
2. Bean instantiated
3. Dependencies injected
4. Internal spring processing
5. Your custom init method (Bean is ready for use. Container is shutdown)
6. Custom destroy method

Bean lifecycle methods / Hooks: bean initialization, bean destruction:

<bean….

init-method=”doMyStartupStuff”

destroy-method=”doMyCleanupStuff”…

</bean>

**Special Note about init and destroy Method Signatures**

When using XML configuration, I want to provide additional details regarding the method signatures of the init-method  and destroy-method .

**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.

 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.

**7. Spring Configuration with Java Annotations – Inversion of Control.**

Development Process:

1. *Enable component scanning in Spring config file*
   1. <beans … >

<context:component-scan base-package=”package” />

</beans

*Spring wiil recursively scan the package*

1. *Add the @Component Annotation to your Java classes*
   1. @Component(“thatSillyCoach”)

(beanId is thatSillyCoach)

1. *Retrieve bean from Spring container*

Spring supports Default Bean IDs. Default bean id: the class name, make first letter lower case.

**8. Spring Configuration with Java Annotations – Dependency Injection.**

What is Spring AutoWiring? For dependency injection, Spring can use auto wiring. Spring will look for a class that matches the property (mathes by type: class or interface). Once it finds, it will automatically inject the dependency.

Example:

* 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
* Setter Injection
* Field Injections

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. User @Autowired annotation before constructor, which has arg with dependency

Development Process –Setter Injection:

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

Field Injections. Inject dependencies by setting field values on your class directly (even private fields).

Development Process – Fields Injection:

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

@Qualifier(“beanId”) – The desired bean id, if you have multiple implementations. Can be applied to all defined injection types.

Note 1: 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.

Note 2: Qualifier with constructors:

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

Note 3: This solution will show you how inject values from a properties file using annotatons. The values will no longer be hard coded in the Java code.

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

New text file:  src/sport.properties

foo.email=myeasycoach@luv2code.com

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**

@Value("${foo.email}")

private String email;

@Value("${foo.team}")

private String team;

---

**9. Spring configuration with Java Annotations – Bean Scopes and Lifecycle Methods.**

Explicitly specify bean scope using the @Scope annotation:

@Scope("singleton");

Scope methods/hooks. Development process:

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

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.

To get the Spring container to release resources held by prototype-scoped beans, try using a custom bean post-processor, which holds a reference to beans that need to be cleaned up.