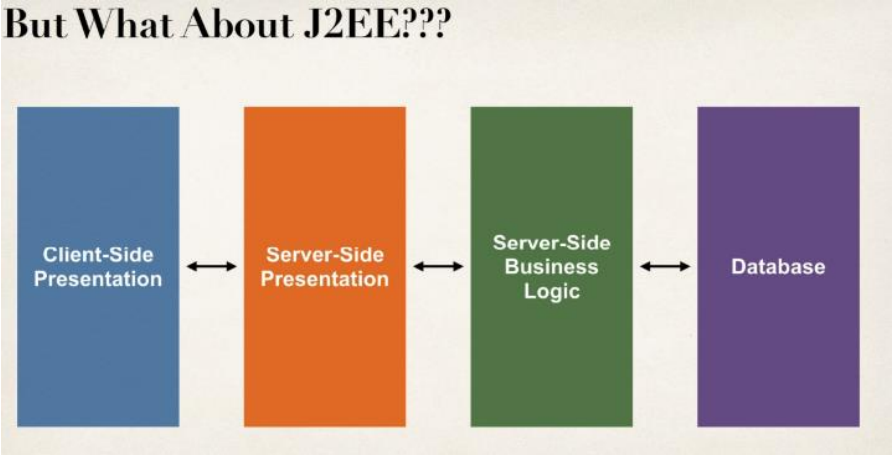
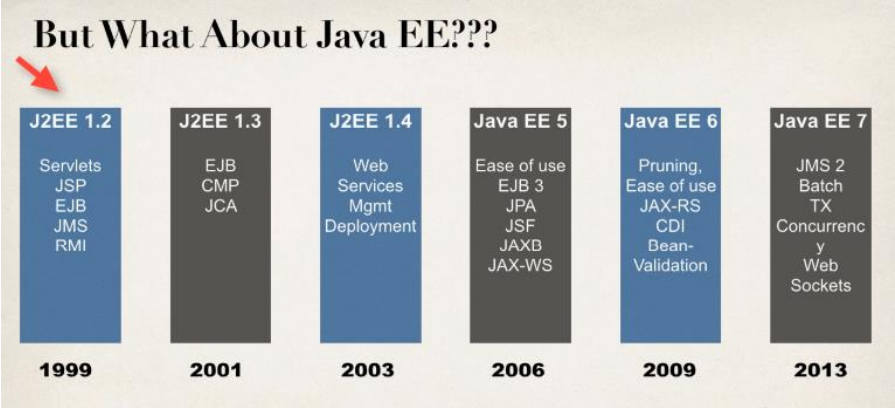
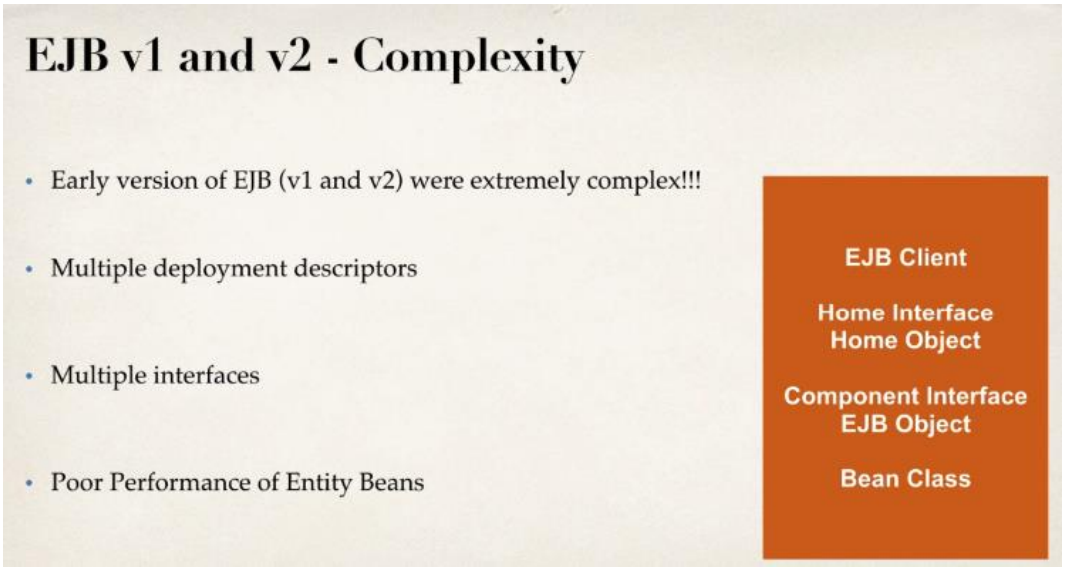
Spring Framework

Q. Why Spring?
> Spring is a very popular framework for building Java applications.
> Spring was initially a simpler & lightweight alternative to J2EE i.e., provides a large no. of helper classes… makes things easier.

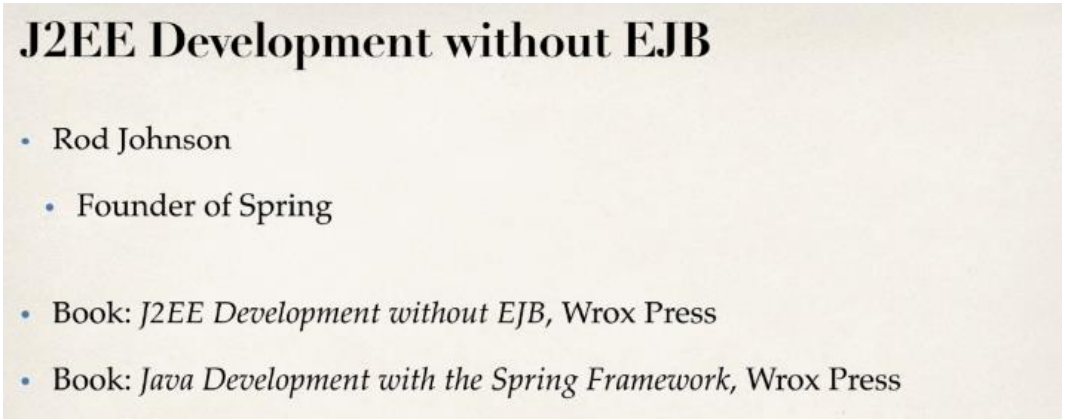




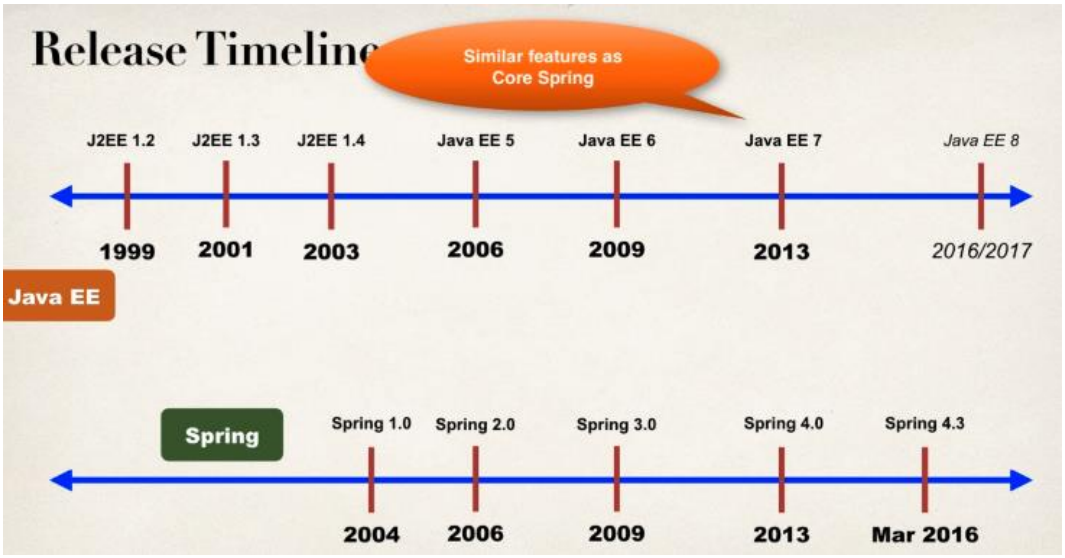
JSP – Jakarta Server Pages / JavaServer Pages
EJB – Jakarta Enterprise Beans / Enterprise JavaBeans
JMS – Java Message Service
RMI – Remote Method Invocation
CMP – Container Managed Persistence
JCA – Java Connector Architecture
JPA – Java Persistence API
JSF – Java Server Faces
JAXB – Java API for XML Binding
JAX-WS – Java Web Services (SOAP)
JAX-RS – Java Web Services (REST)
CDI – Context Dependency Injection (IoC)



> Entity Beans basically mapping between java class & Database table are just awful slow. Even on one deployment, we actually had to pull our code back out of the production & actually remove the EJB functionality because it slowed everything down.
That's a lot of developers started to continue to do J2EE but they actually would do it without Enterprise JavaBeans.



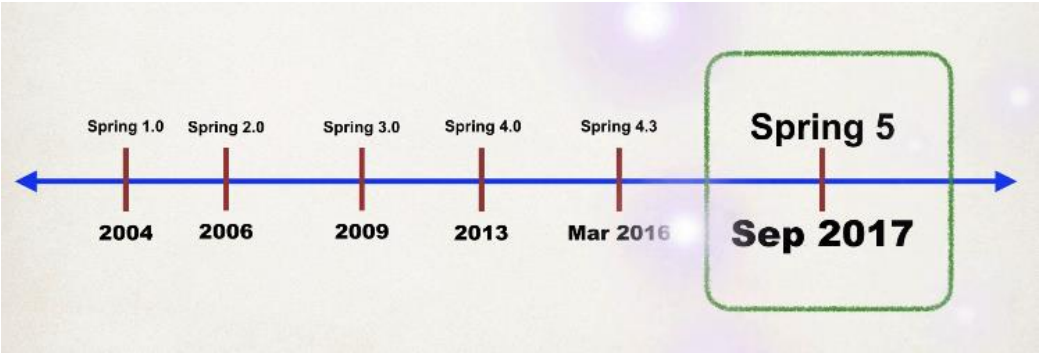
Release Timeline of J2EE & Spring



EJB 3.1 in Java EE 6 was much easier to use i.e., in J2EE 6, added CDI Context Dependency Injection IoC.
So, from **Java EE 7**, you can do everything as **Spring** can do but the only problem though is that they were just a little bit too late & unfortunately, EJB just kind of has a bad name & also, Spring has huge momentum, huge market share.

> Spring is a lightweight framework, simple, easy to use & a lot of developers like it.

What's New in Spring 5?



> Updated minimum requirements for Java 8 or higher.
> Deprecated legacy integration for: Tiles, Velocity, Portlet, Guava etc.
> Upgraded Spring MVC to use new versions of Servlet API 4.0
> Added new reactive programming framework: Spring WebFlux

Spring Framework Overview

> Spring official Website: - www. spring.io

> Goals of spring

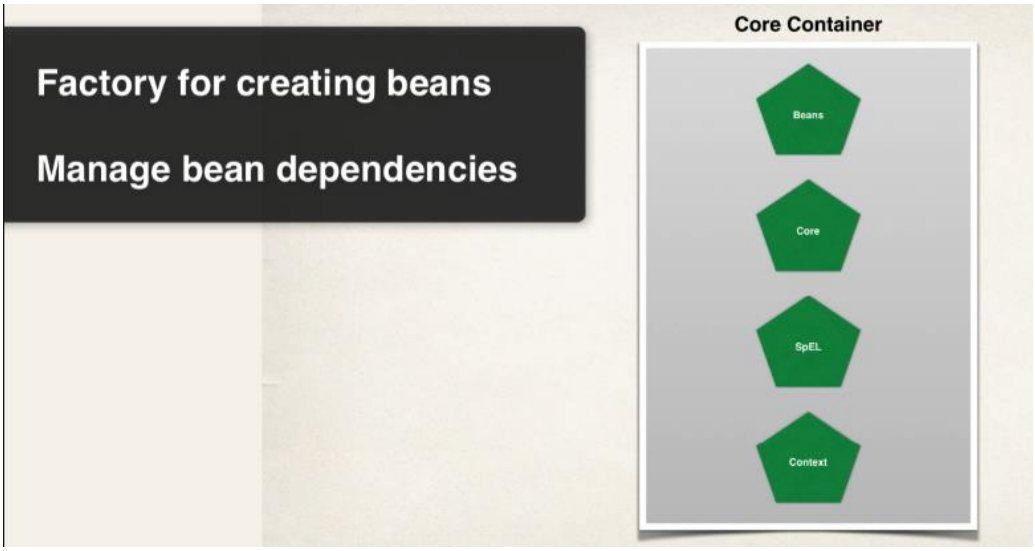
a) Lightweight development with Java POJOs (Plain – Old – Java – Objects))
   [Make it much simpler to build, as compared to the heavyweight EJBs from the early versions of J2EE]

b) Dependency injection to promote loose coupling
   [So instead of hard wiring your objects together, you simply specify the wiring via a configuration file or annotations.]

c) Declarative programming with Aspect – Oriented – Programming (AOP)
   [Basically allow you to add some application wide services to your given objects]

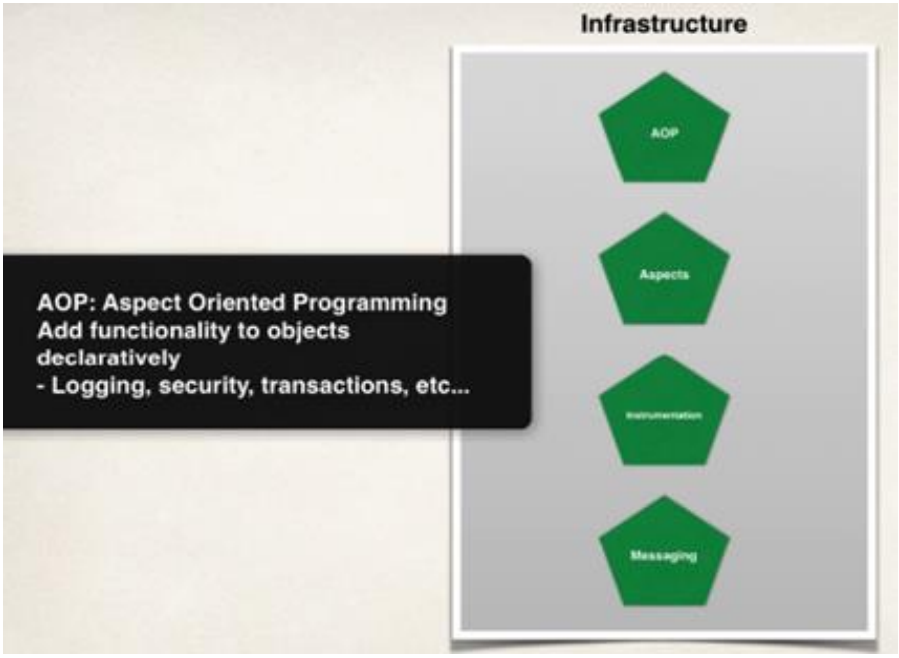
d) Minimize boilerplate Java code.
   [In early days of J2EE, there was a lot of code that you had to write, so the folks at Spring created a collection of helper classes to make it easier.]

Spring Core



> So, the core container is like heart of Spring framework.
> It basically manages how beans are created.
> It has a bean factory for creating the beans.
> It basically can read configuration files for setting properties in dependencies.
> Also, the Context here is really the spring container that holds the Beans in memory.
> Spring Expression Language (SpEL) is a language we can use within the configuration files to refer to other beans.

Spring AOP
- AOP, Aspects, Instrumentation, Messaging

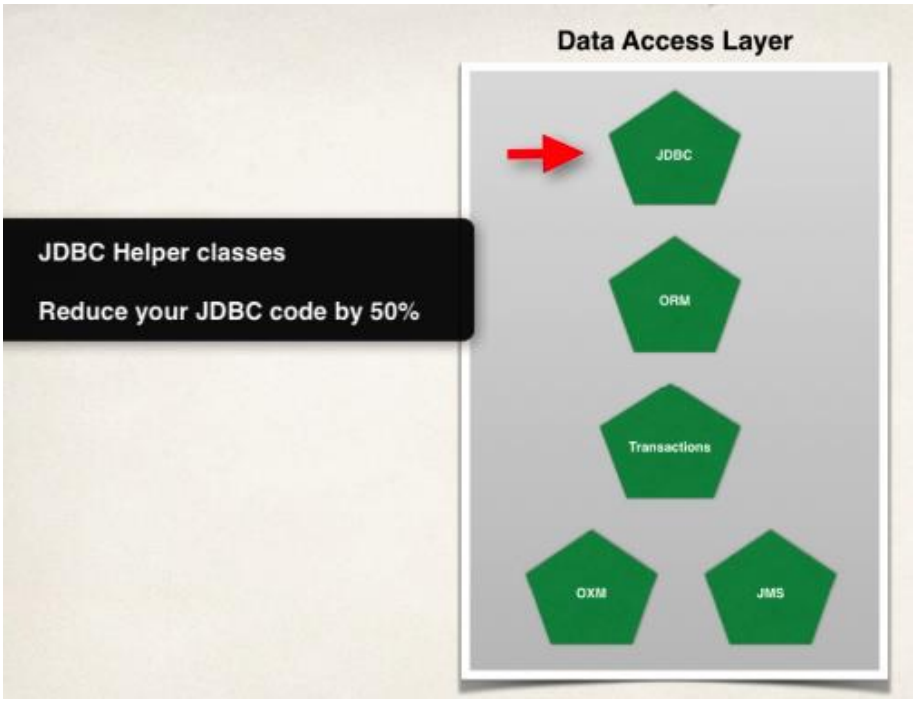


> **Spring AOP** allows you to create application wide services, like logging, security, transactions, instrumentation & then you can apply these services to your objects in a declarative fashion, so no need to modify your code to have support for this. You simply add a configuration in the configuration file or an annotation and that service will be applied to your application.

> **Spring Instrumentation** – Here you can actually make use of class loader implementations to work with different apps server. For e.g., it can be used to create a Java agent, so you can remotely monitor and instrument your application using JMX (Java Management Extension)

As a developer, you would not normally build an agent yourself. You would simply use the agents provided by the Spring team or your app server vendor [but behind the scenes, it's making use of some really cool technologies like AOP coding, bytecode manipulation etc.], so that's what you get in the instrumentational model.

Spring Data Access Layer



> **Spring Data access layer** is for communicating with the database, either a relational database or a NoSQL database and also making use of a message queue.

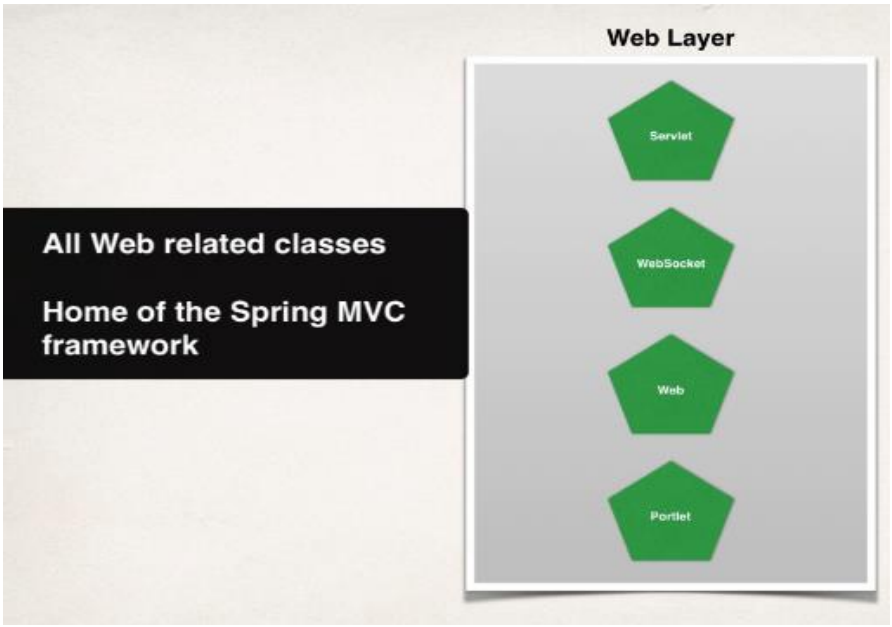
> Basically, Spring provides some helper classes to make it much easier to access a database, using JDBC & by using these Spring JDBC classes, you can actually reduce your source code by over 50%

> **ORM** (Object to Relational Mapping) is probably the most popular section of this module. Basically, it allows you to hook into Hibernate, or hook into JPAs.

> **JMS** (Java Message Service) allows you to send messages to a message queue (Message Broker) in an asynchronous fashion. That's a core part of JAVA EE. Here, they basically provide helper classes to allow you to make use of the JMS & again, you can reduce your code by over 50% by making use of spring's JMS integration.

> Spring has support for a **transaction manager** or supporting transactions & you can do this in a very lightweight fashion. So, you can make use of transactions on methods, on database calls, & pretty much anything you want. Transactions manager makes heavy use of AOP behind the scenes.

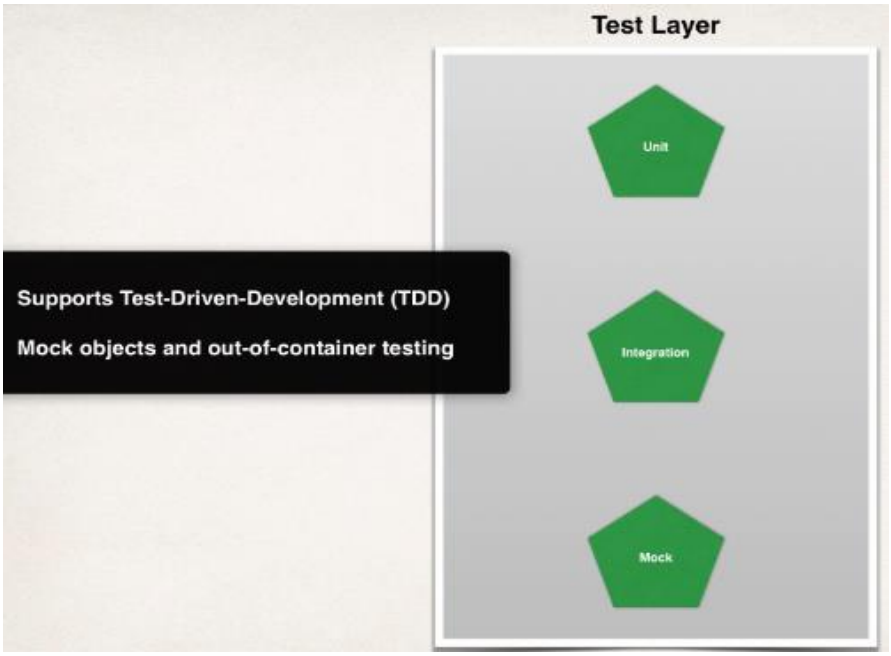
Spring Web Layer



> So, you can build web applications using the Spring core, & also making use of Spring Controllers & Spring View so you have a full MVC layout here.

> They also have support for Remoting here, so you can actually make use of web remoting, where you can have external clients make calls into the Spring container. [like a way of doing remote procedure calls (RPC) or also doing distributed computing]

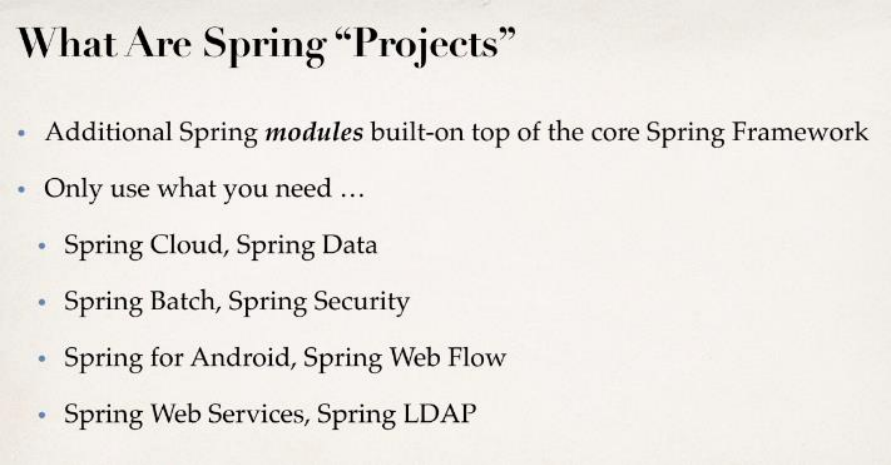
Spring Test Layer



> Spring has support for test – driven development, so the framework includes mock objects for mocking out servlets, JNDI access & so on and you can do all of this outside of the container.

> You can also make use of integration test by creating an application context & writing up your desired object. So testing is a first class citizen here, when making use of the Spring framework, so there's a lot of good support for it.

Spring Projects



> Spring projects are just additional Spring modules that are built on top of the core framework, so think of them as simply add – ons.
> You only use what you need so they have projects here for Spring cloud & Spring Data so
> Cloud for doing cloud development.
> Data for database integration.
> Spring batch for creating batch processes.
> Spring security for securing your application.
> Spring for Android for Android development
> Spring Web flow for doing web flow over x number of pages.
> Spring Web Services for doing restful & soap web services.
> Spring LDAP for accessing LDAP servers.

> Location to get information on Spring projects - https://spring.io/projects

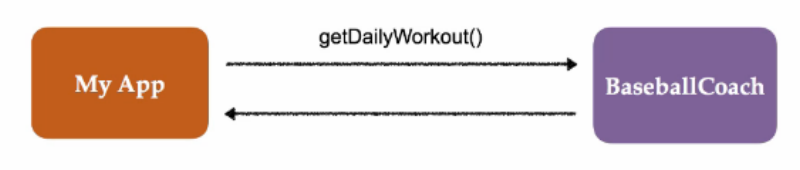
Spring Environment Setup

> You must have the Java Development Kit (JDK) installed.
> Spring 5 requires Java 8 or higher.
> Java Application server for web development like Glassfish, JBoss, WebLogic & so on. For simplicity, use Tomcat server.
> Java Integrated Development Environment (IDE) like Eclipse
> Installing Tomcat
Step 1: Go to website - https://tomcat.apache.org/
Step 2: Select version & download binary distribution for Core
Step 3: Install
> Installing Eclipse
Step 1: Go to website - https://www.eclipse.org/
Step 2: Download eclipse & install it.
> Connecting Tomcat to Eclipse
> Downloading Spring 5 JAR File & add JAR files to Eclipse project…Build Path [You can also use Maven]
> Spring Repo - https://repo.spring.io/release/org/springframework/spring/
> Download latest release dist file & unzip it & import it into the Java project lib
> Then go to project properties & find build path
> Add lib to build path libraries
> Once libraries will be built you will see "Referenced Libraries" in package Hierarchy.

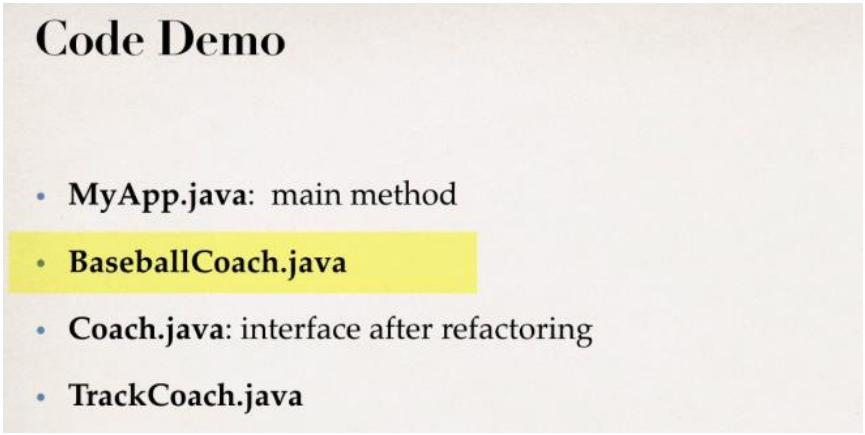
Spring Inversion of Control

> Inversion of Control (IoC) is simply the design process of externalizing the construction & management of objects.
> It basically says that your application's going to outsource the creation & management of the objects & that outsourcing will be handled by an object factory.

> **Coding Scenario**



Task 1: App should be configurable
Task 2: Easily change the coach for another sport [achieved by using Coach interface to hold different type of Coaches]



Code: By using Coach interface, we can easily change the coach for another sport

```
public interface Coach {

	public String getDailyWorkout();
}

public class BaseballCoach implements Coach {

	@Override
	public String getDailyWorkout() {
		return "Spend 30 minutes on batting practice.";
	}
}

public class TrackCoach implements Coach {

	@Override
	public String getDailyWorkout() {
		return "Run a hard 5k.";
	}

}
```

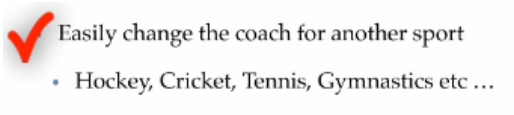
```
public class MyApp {

	public static void main(String[] args) {

		// create the object

		//BaseballCoach theCoach = new BaseballCoach();
		Coach theCoach1 = new BaseballCoach();
		Coach theCoach2 = new TrackCoach();

		// use the object
		System.out.println(theCoach1.getDailyWorkout());
		System.out.println(theCoach2.getDailyWorkout());
	}

}
```

Output:

```
Spend 30 minutes on batting practice.
Run a hard 5k.
```

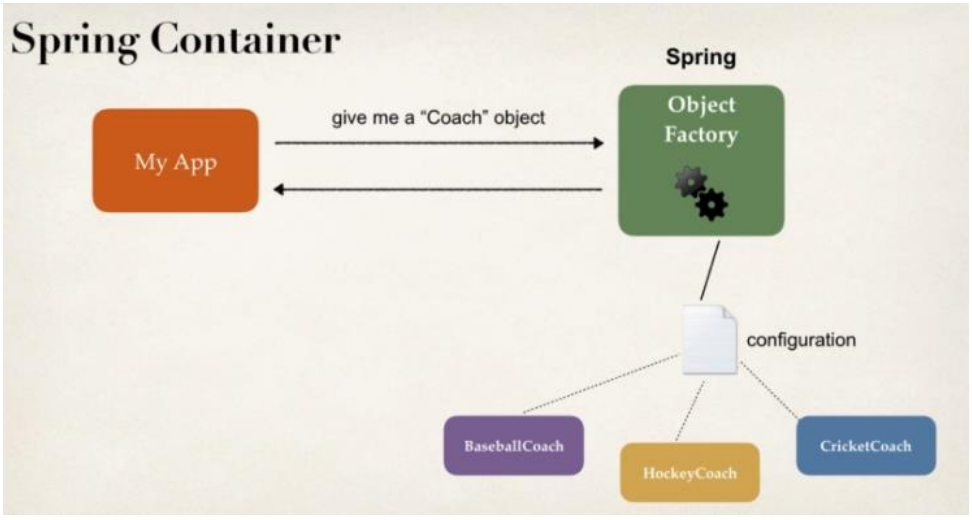
> **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. ***The determination of the method to be called is based on the object being referred to by the reference variable.***



Task 2 done

For **Task 1** to achieve i.e. App should be configurable, we need to make use of Spring Object factory



> ***Spring provides an object factory*** so we can have our application talk to spring, hey give me an object. Based on a configuration file or annotation, Spring will give the app the appropriate implementation. So now app is configurable.

> **Primary functions of Spring Container**

a) Create & manage objects (***Inversion of Control***)
b) Inject object's dependencies (***Dependency Injection***)

> **Configuring Spring Container**

1. XML configuration file (legacy, but most legacy apps still use this)
2. Java Annotation (modern)
3. Java Source Code (modern)

> **Spring Development Process**

Step 1: Configure your Spring Beans.
Step 2: Create a Spring Container (**Spring container** is also known as **ApplcationContext**).
Step 3: Retrieve Beans from Spring Container.

Step 1:

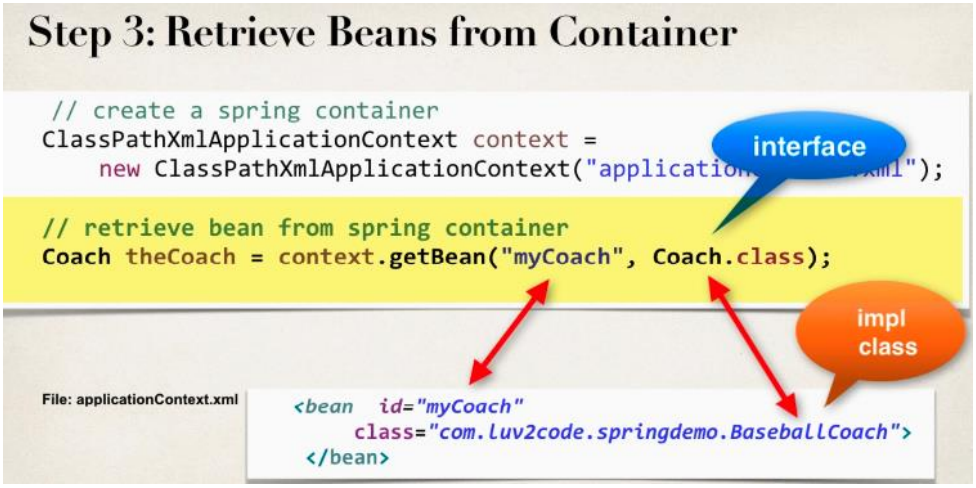
> **myCoach** is the bean **id** & **class** is the fully qualified name of implementation class.
> This bean id is used by java application to retrieve a bean from Spring container.

Step 2:


```
ClassPathXmlApplicationContext context =
    new ClassPathXmlApplicationContext("applicationContext.xml");
```

> ***applicationContext.xml*** is the name of configuration file.
> You can keep any name of configuration file as long as it is same in step 1 & step 2.

Step 3:



> This step is retrieving the beans from the container.
> So your application is simply going to talk to the spring container & say give me a coach object & based on the information in configuration file, it'll give you an implementation of that given interface.

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.

> In Spring, the objects that form the backbone of your application & that are managed by the Spring IoC container are called beans.
> ***A bean is an object that is instantiated, assembled, & otherwise managed by a Spring IoC container.***
> Otherwise, a bean is simply one of many objects in your application. Beans, & the dependencies among them, are reflected in the configuration metadata used by a container.

Code:

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

       <!-- Define your beans here -->
       <bean id="myCoach" class="com.srvcode.springdemo.TrackCoach">
       </bean>
</beans>
```

```
public class TrackCoach implements Coach {

	@Override
	public String getDailyWorkout() {
		return "Run a hard 5k.";
	}

}


public class HelloSpringApp {

	public static void main(String[] args) {

		// load the spring configuration file
		ClassPathXmlApplicationContext context =
			new
ClassPathXmlApplicationContext("applicationContext.xml");

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

		// call methods on the bean
		System.out.println(theCoach.getDailyWorkout());

		// close the context
		context.close();
	}

}
```

OUTPUT

```
Run a hard 5k.
```

Q. Why do we specify the Coach interface in getBean ()?

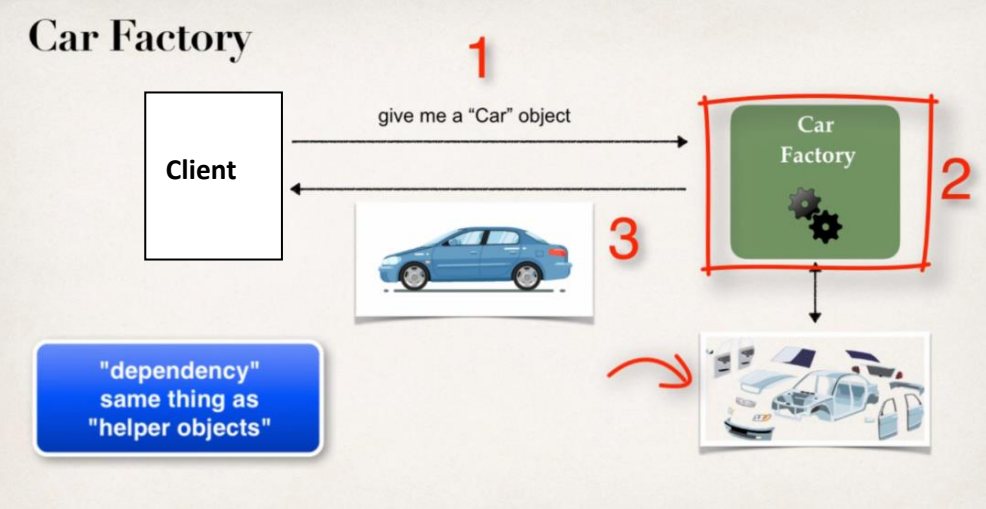
Ans: - e.g. `Coach theCoach = context.getBean ("myCoach", Coach.class);`

When we pass the interface to the method, behind the scenes Spring will cast the object for you. However there are some slight differences than Normal casting i.e. behaves the same as *getBean (String),* but provides a measure of type safety by throwing a *BeanNotOfRequiredTypeException* if the beans not of the required type i.e. *ClassCastException* can't be thrown on casting the result correctly, as can happen with *getBean (String).*

Spring Dependency Injection

> The client delegates to calls to another object the responsibility of providing its dependencies.

e.g.



Explanation: -

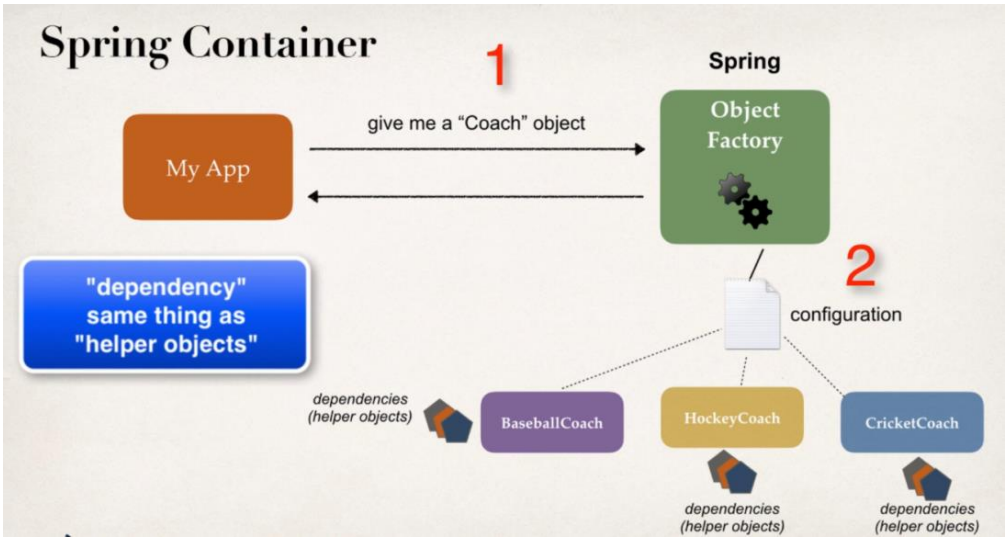
> Client is going to buy a car & this car is built at the factory on demand. So you have to actually talk to factory & put in a request & they will build for you.

> So, at the factory you have all the different parts of the car like car chassis, engine, seats etc. & the mechanic actually assemble the car for you & deliver to you the final car.

> So, you don't have to actually build the car. The car's already built for you at the factory i.e. they actually inject all of the dependencies for the car.

> So that's basically what you have here with dependency injection

i.e. ***You simply outsource the construction & injection of your object to an external entity like car factory***

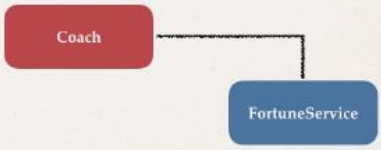


> ***Spring has an object factory. So when you retrieve an object like a Coach object, this coach object may have some additional dependencies. So these dependencies are really just helper objects, other objects that it needs to perform its operation. So instead of you have to manually build the coach object & all of its dependencies. The Spring framework of the Spring factory will actually do this work for you. So just like you get a car that's ready to drive, here you'll get an object that's ready to use.***

Demo Example

dependency = helper

- Our **Coach** already provides daily workouts
- Now will also provide daily fortunes
  - New helper: **FortuneService**
  - This is a ***dependency***



Injection Types

- There are many types of injection with Spring
- We will cover the two most common
  - Constructor Injection
  - Setter Injection
- Will talk about "auto-wiring" in the Annotations section later

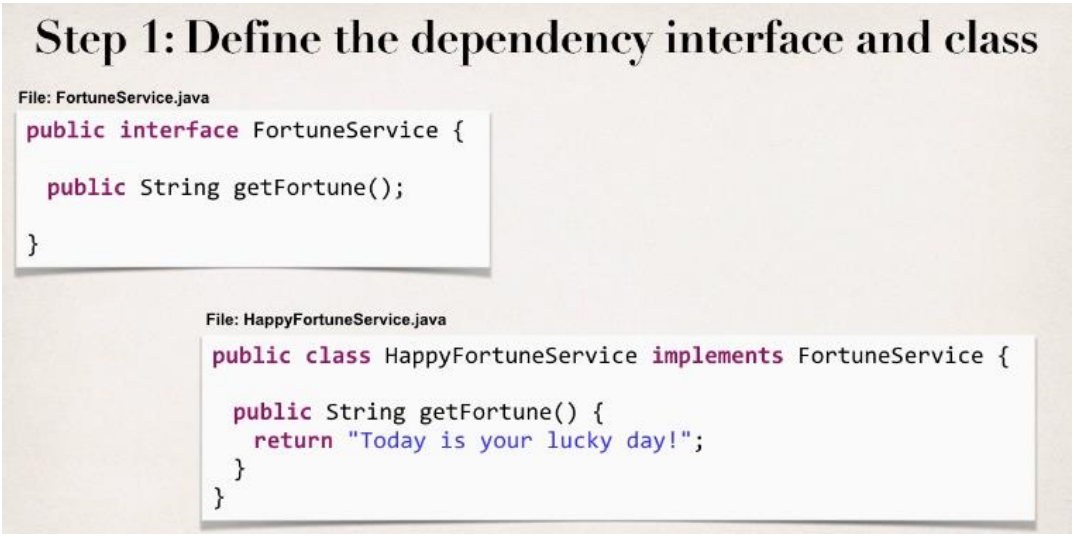
1. Constructor Injection

Development Process - Constructor Injection

Step-By-Step

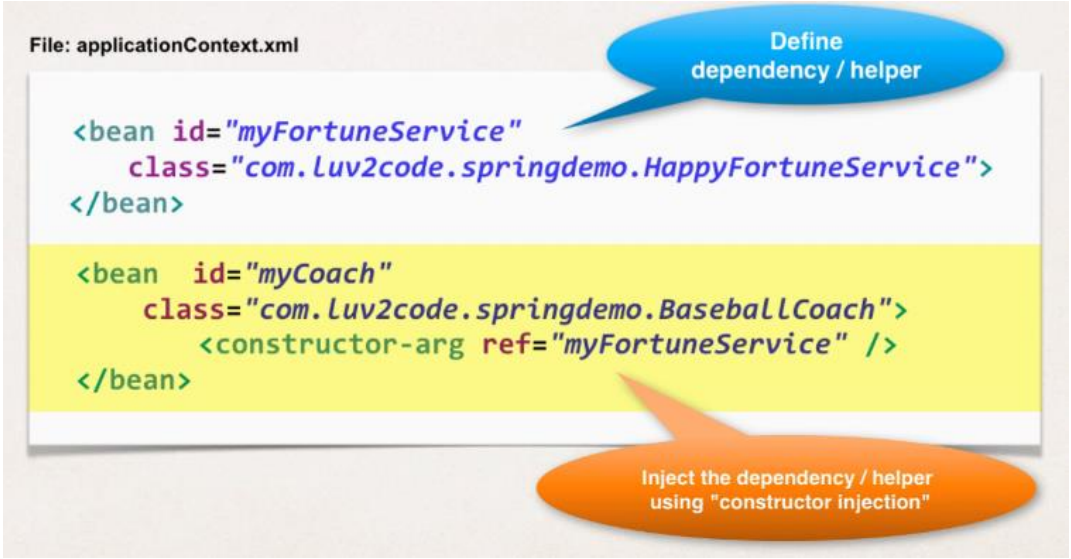
1. Define the dependency interface and class
2. Create a constructor in your class for injections
3. Configure the dependency injection in Spring config file

Step 1: Define the dependency interface & class

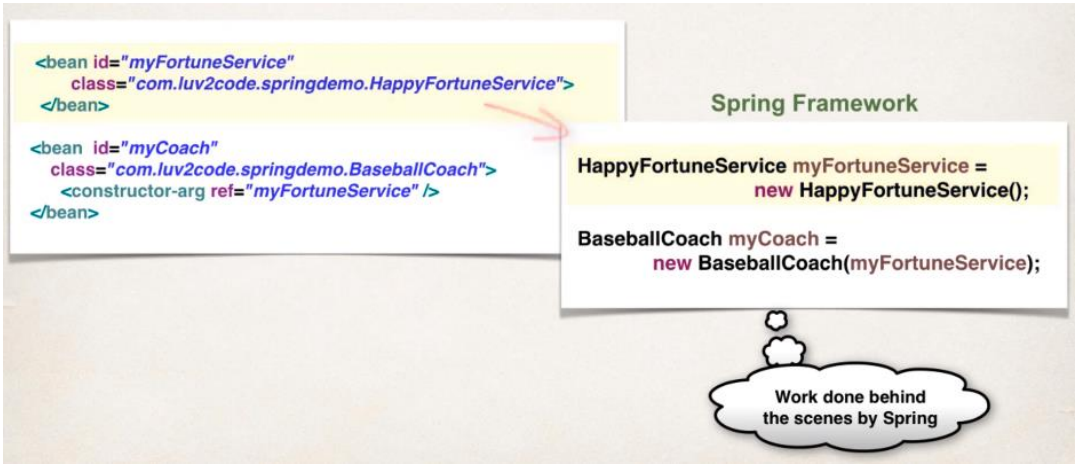


Step 2: Create a constructor in your class for injection

Step 3: Configure the dependency injection in Spring configuration file



How Spring Processes, the configuration File



Code:

Step 1: Define the dependency interface & class

```
public interface FortuneService {

	public String getFortune();
}

public class HappyFortuneService implements FortuneService {

	@Override
	public String getFortune() {
		return "Today is your lucky day!";
	}

}

public interface Coach {

	public String getDailyWorkout();

	public String getDailyFortune();
}
```

Step 2: Create a constructor in your class for injection

```
public class BaseballCoach implements Coach {

	// define a private field for the dependency
	private FortuneService fortuneService;

	// define a constructor for dependency injection
	public BaseballCoach(FortuneService theFortuneService) {
		fortuneService = theFortuneService;
	}

	public BaseballCoach() {
	}
```

```
	@Override
	public String getDailyWorkout() {
		return "Spend 30 minutes on batting practice.";
	}

	@Override
	public String getDailyFortune() {

		// use my fortuneService to get a fortune
		return fortuneService.getFortune();
	}
}
```

Step 3: Configure the dependency injection in Spring configuration file

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans…>

	<!-- define the dependency -->
	<bean id="myFortuneService" class="com.srvcode.springdemo.HappyFortuneService">
	</bean>
	<bean id="myCoach" class="com.srvcode.springdemo.BaseballCoach">
		<!-- set up constructor injection -->
		<constructor-arg ref="myFortuneService" />
	</bean>
</beans>
```

Step 4: Main class

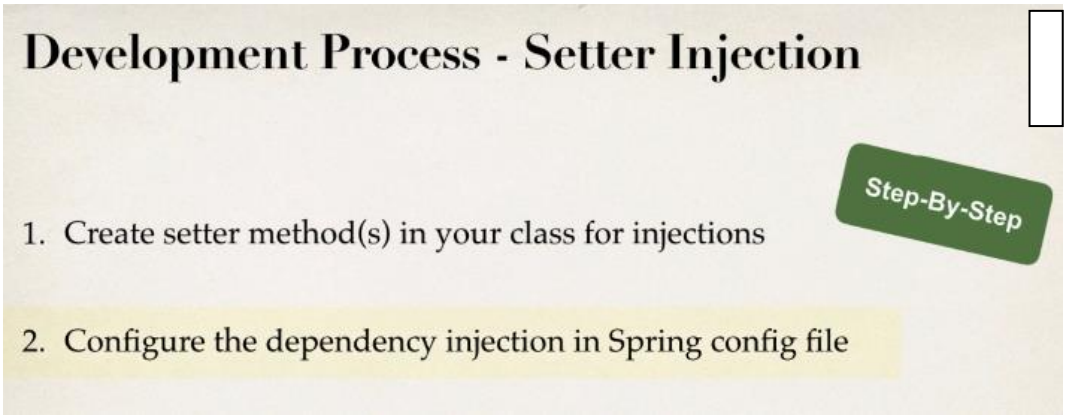
```
public class HelloSpringApp {

	public static void main(String[] args) {

		// load the spring configuration file
		ClassPathXmlApplicationContext context =
				new ClassPathXmlApplicationContext("applicationContext.xml");

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

		// call methods on the bean
		System.out.println(theCoach.getDailyWorkout());

		// let's call our new method for fortunes
		System.out.println(theCoach.getDailyFortune());

		// close the context
		context.close();
	}

}
```

Output

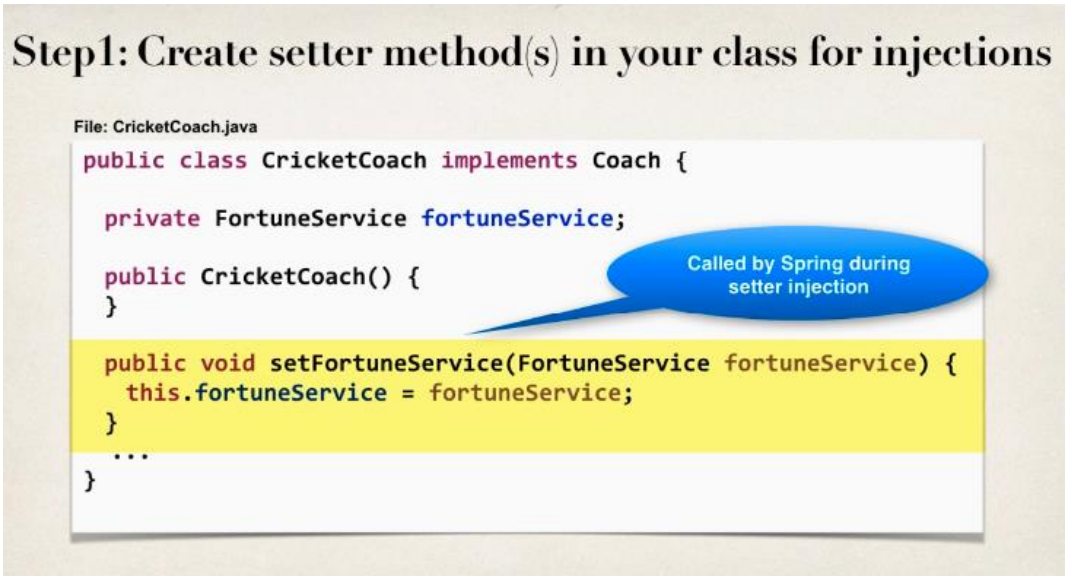
```
Spend 30 minutes on batting practice.
Today is your lucky day!
```

Q. What is the purpose for the no arg constructor?
Ans: - When you don't define any constructor in your class, compiler defines default one for you, however when you declare any constructor (in your example you have already defined a parameterized constructor), compiler doesn't do it for you.

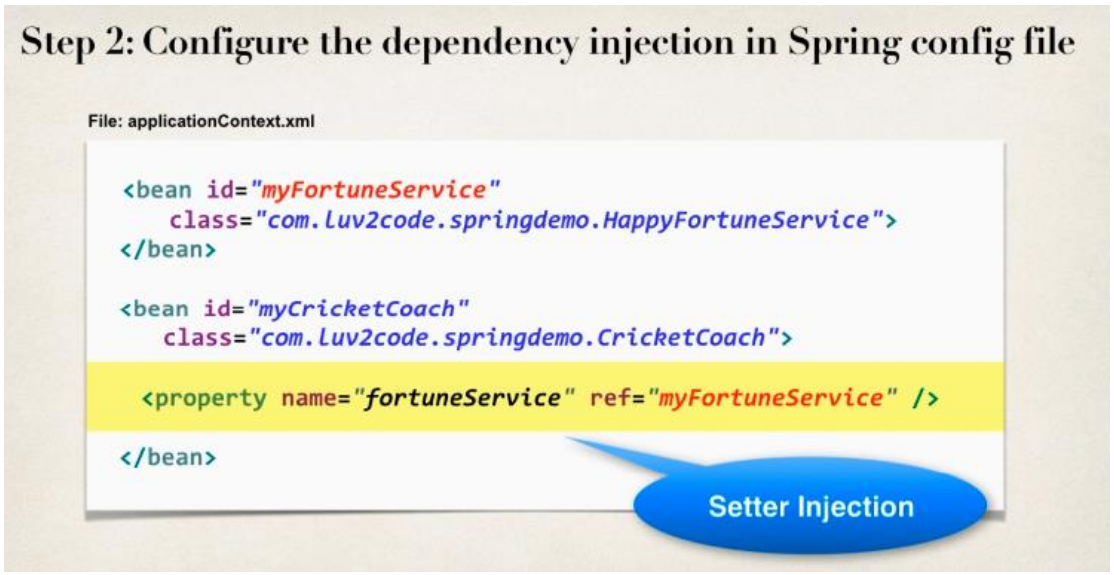
2. Setter Injection



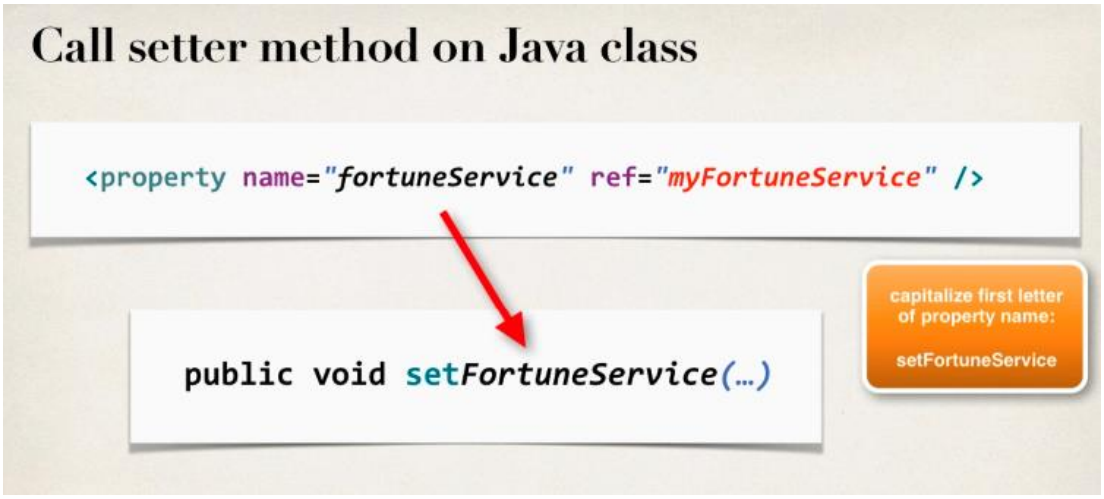
Step 1: Create Setter method(s) in the class for injections



Step 2: Configure the dependency injection in Spring configuration file



> What Spring will actually do in background



> How Spring processes your configuration file

Code:

Step 1: Create Setter method(s) in the class for injections

```
public class CricketCoach implements Coach {

	private FortuneService fortuneService;

	// create a no - arg constructor
	public CricketCoach() {
		System.out.println("CricketCoach: inside no-arg construtor");
	}

	// our setter method
	public void setFortuneService(FortuneService fortuneService) {
		System.out.println("CricketCoach: inside setter method - setFortuneService");
		this.fortuneService = fortuneService;
	}

	@Override
	public String getDailyWorkout() {
		return "Practice fast bowling for 15 minutes";
	}

	@Override
	public String getDailyFortune() {
		return fortuneService.getFortune();
	}
}
```

Step 2: Configure the dependency injection in Spring configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans…>

	<!-- define the dependency -->
	<bean id="myFortuneService" class="com.srvcode.springdemo.HappyFortuneService">
	</bean>
	<bean id="myCoach" class="com.srvcode.springdemo.BaseballCoach">
		<!-- set up constructor injection -->
		<constructor-arg ref="myFortuneService" />
	</bean>
</beans>
```

Main Class

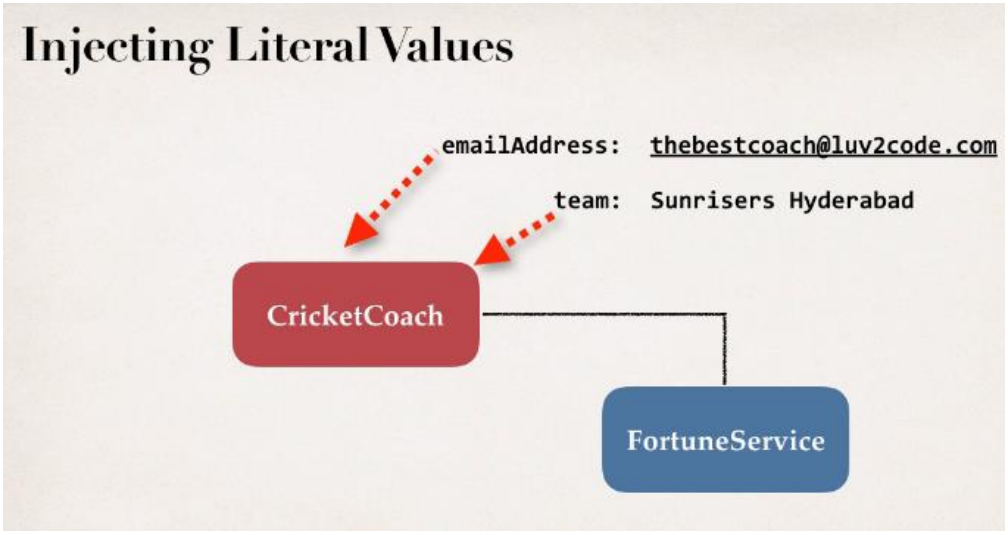
```
public class SetterDemoApp {

	public static void main(String[] args) {

		// load the spring configuration file
		ClassPathXmlApplicationContext context =
				new ClassPathXmlApplicationContext("applicationContext.xml");

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

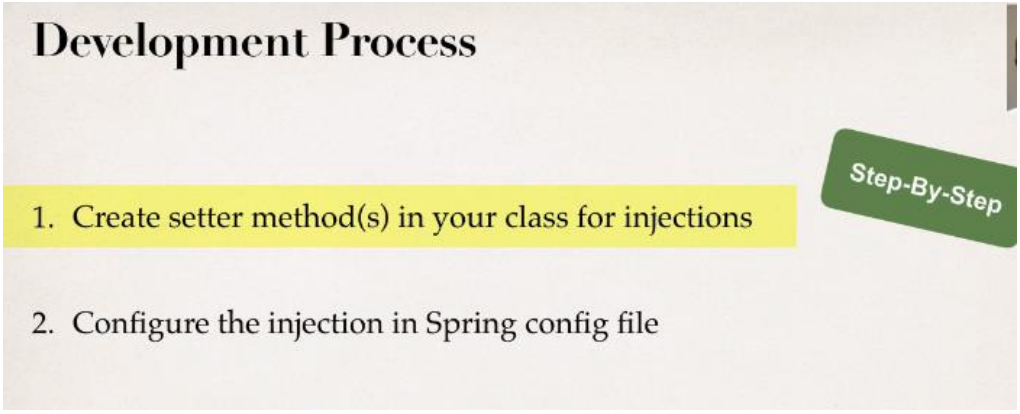
```
                // call methods on the bean
                System.out.println(theCoach.getDailyWorkout());

                System.out.println(theCoach.getDailyFortune());

                // close the context
                context.close();
        }

}
```

Output

```
CricketCoach: inside no-arg construtor
CricketCoach: inside setter method - setFortuneService
Practice fast bowling for 15 minutes
Today is your lucky day!
```

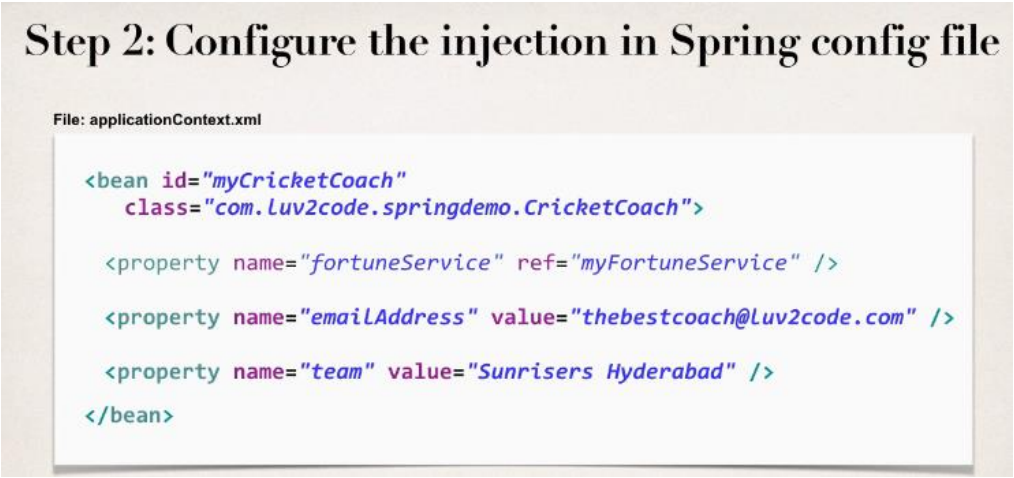
Injecting Literal Values





Step 1: Create setter method(s) in your class for injections

Step 2: Configure the injection in spring configuration file



Code:

Step 1: Create setter method(s) in your class for injections

```
public class CricketCoach implements Coach {

	private FortuneService fortuneService;

	// add new fields for emailAddress & team
	private String emailAddress;
	private String team;

	// create a no - arg constructor
	public CricketCoach() {
		System.out.println("CricketCoach: inside no-arg construtor");
	}

	public String getEmailAddress() {
		return emailAddress;
	}
```

```
	public void setEmailAddress(String emailAddress) {
		System.out.println("CricketCoach: inside setter method - setEmailAddress");
		this.emailAddress = emailAddress;
	}

	public String getTeam() {
		return team;
	}

	public void setTeam(String team) {
		System.out.println("CricketCoach: inside setter method - setTeam");
		this.team = team;
	}

	// our setter method
	public void setFortuneService(FortuneService fortuneService) {
		System.out.println("CricketCoach: inside setter method - setFortuneService");
		this.fortuneService = fortuneService;
	}

	@Override
	public String getDailyWorkout() {
		return "Practice fast bowling for 15 minutes";
	}

	@Override
	public String getDailyFortune() {
		return fortuneService.getFortune();
	}
}
```

Step 2: Configure the injection in spring configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans..>

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

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

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

		<!-- inject hard coded literal values -->
		<property name="emailAddress" value="srvcode@gmail.com"/>
		<property name="team" value="Sunrisers Hyderabad"/>
	</bean>
</beans>
```

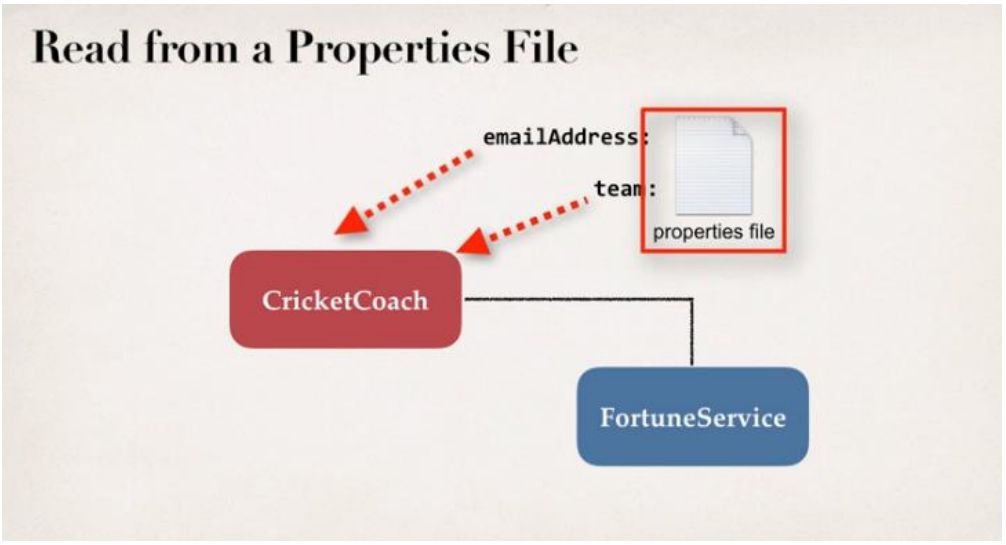
Main Class

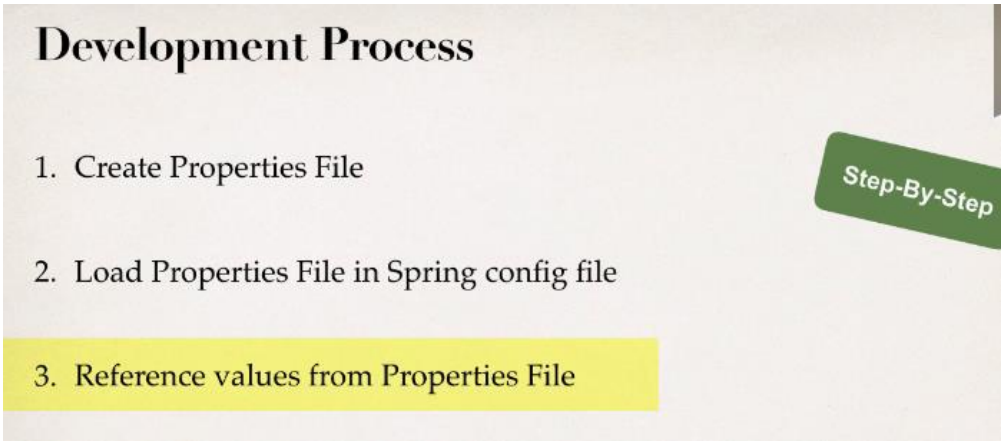
```
public class SetterDemoApp {

	public static void main(String[] args) {

		// load the spring configuration file
		ClassPathXmlApplicationContext context =
				new ClassPathXmlApplicationContext("applicationContext.xml");

		// retrieve bean from spring container
		CricketCoach theCoach = context.getBean("myCricketCoach", CricketCoach.class);

		// call methods on the bean
		System.out.println(theCoach.getDailyWorkout());

		System.out.println(theCoach.getDailyFortune());

		// call our new methods to get the literal values
		System.out.println(theCoach.getEmailAddress());

		System.out.println(theCoach.getTeam());

		// close the context
		context.close();
	}

}
```

Output

```
CricketCoach: inside no-arg construtor
CricketCoach: inside setter method - setFortuneService
CricketCoach: inside setter method - setEmailAddress
CricketCoach: inside setter method - setTeam
Practice fast bowling for 15 minutes
Today is your lucky day!
srvcode@gmail.com
Sunrisers Hyderabad
```

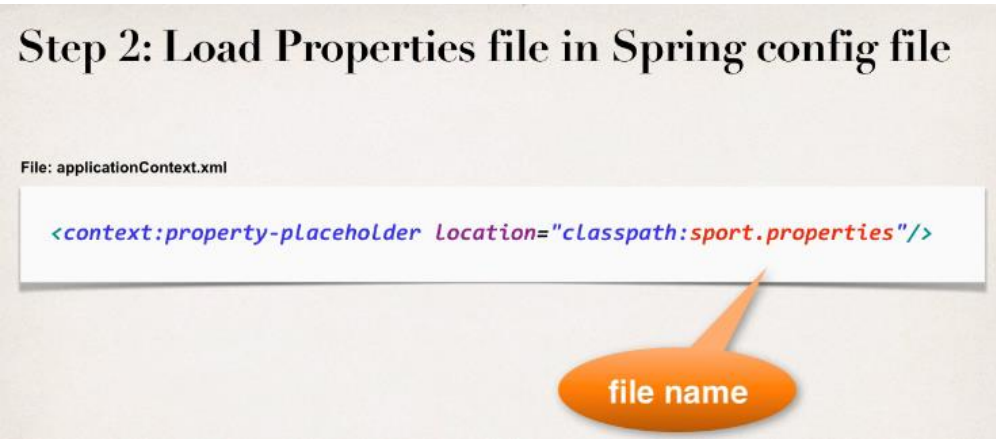
Injecting values from Properties File



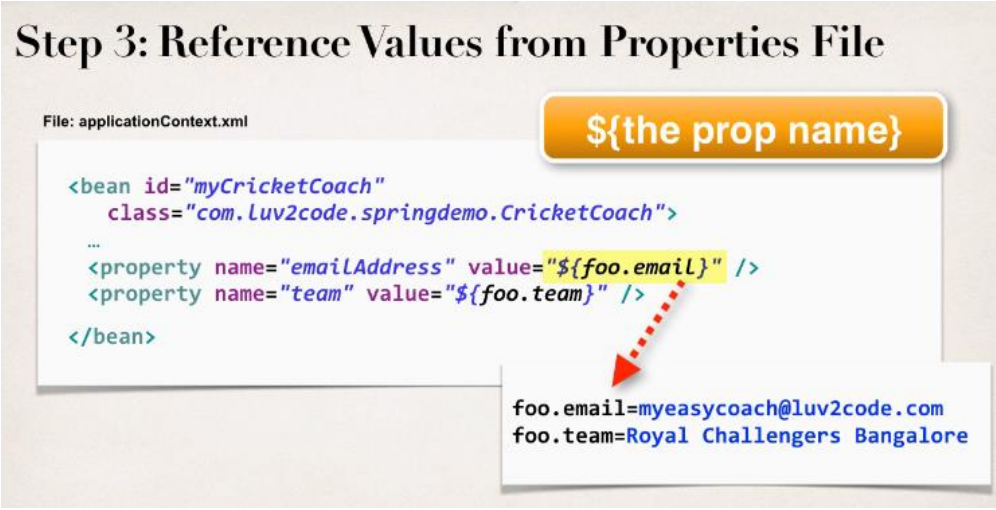


Step 1: Create properties file

Step 2: Load properties file in Spring configuration file



Step 3: Reference values from Properties file



Code:

Step 1: Create properties file

```
foo.email=myeasycoach@gmail.com
foo.team=Royal Challengers Bangalore
```

Step 2: Load properties file in Spring configuration file

Step 3: Reference values from Properties file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans..>

	<context:property-placeholder location="classpath:sport.properties"/> // Step 2

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

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

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

		<!-- inject configuration literal values -->                    // Step 3
		<property name="emailAddress" value="${foo.email}"/>
		<property name="team" value="${foo.team}"/>

	</bean>
</beans>
```

Main Class

```
public class SetterDemoApp {

	public static void main(String[] args) {

		// load the spring configuration file
		ClassPathXmlApplicationContext context =
				new ClassPathXmlApplicationContext("applicationContext.xml");

		// retrieve bean from spring container
		CricketCoach theCoach = context.getBean("myCricketCoach", CricketCoach.class);

		// call methods on the bean
		System.out.println(theCoach.getDailyWorkout());

		System.out.println(theCoach.getDailyFortune());

		// call our new methods to get the literal values
		System.out.println(theCoach.getEmailAddress());

		System.out.println(theCoach.getTeam());

		// close the context
		context.close();
	}

}
```

Output

```
CricketCoach: inside setter method - setFortuneService
CricketCoach: inside setter method - setEmailAddress
CricketCoach: inside setter method - setTeam
Practice fast bowling for 15 minutes
Today is your lucky day!
myeasycoach@gmail.com
Royal Challengers Bangalore
```

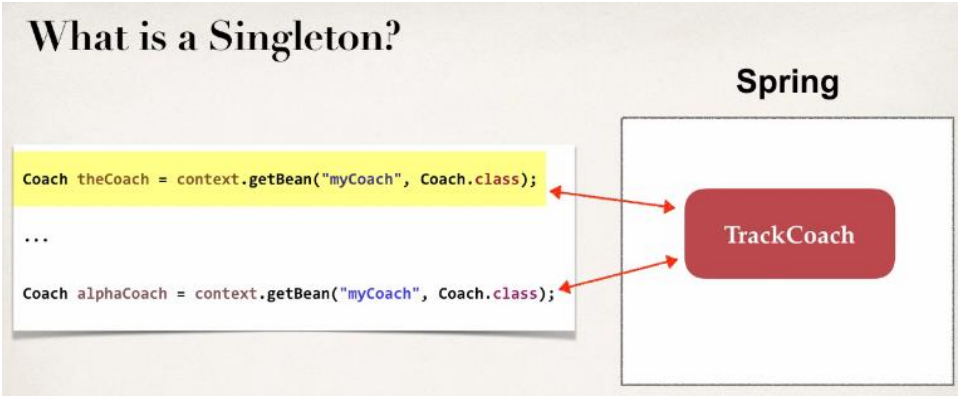
Bean Scopes

> Scope refers to the lifecycle of a bean like
    a) How long does the bean live?
    b) How many instances are created?
    c) How is the bean shared?

> Default Scope for all beans is Singleton

Singleton Scope

> 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.



> The best use case for Singleton scope is for a stateless bean where you don't need to maintain any state.

Explicitly Specify Bean Scope

```
<beans … >

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

</beans>
```

Additional Spring Bean Scopes

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.

Prototype Scope

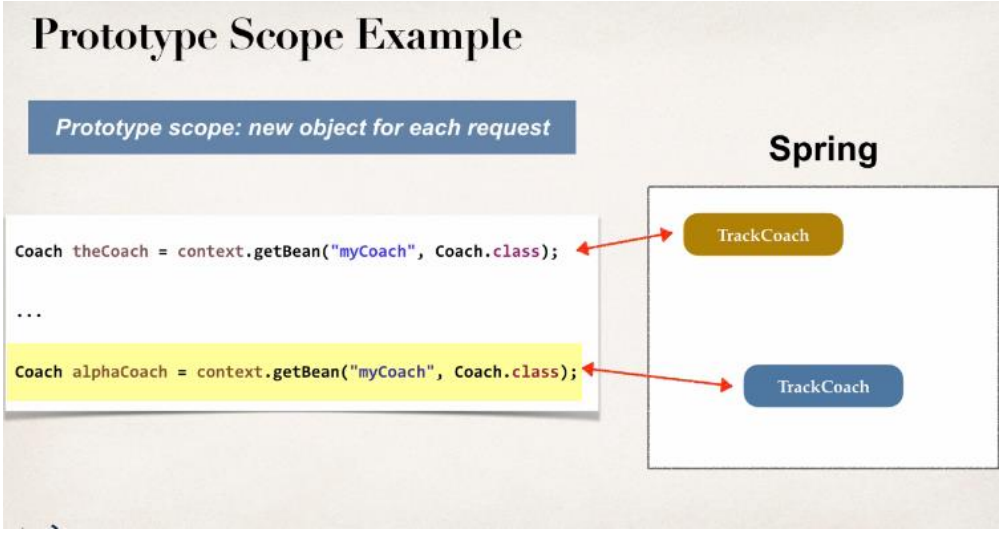
> In prototype scope, new object is created for each request.

Prototype Scope Example

Prototype scope: new object for each request

```
<beans … >

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

</beans>
```

Set bean scope



> **Prototype scope** is good for keeping track of stateful data, so whenever you see prototype, just think of the new keyword, it's gonna create a new bean for each request for that component or that object.

Code: [Singleton Scope]

beanScope-applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans..>
	<!-- define the dependency -->
	<bean id="myFortuneService" class="com.srvcode.springdemo.HappyFortuneService">
	</bean>

	<bean id="myCoach" class="com.srvcode.springdemo.BaseballCoach"
		scope="singleton">

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

Main Class

```
public class BeanScopeDemoApp {

	public static void main(String[] args) {

		// load the spring configuration file
		ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("beanScope-applicationContext.xml");

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

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

		// check if they are the same
		boolean result = (theCoach == alphaCoach);
```

```
            // print out the results
             System.out.println("\nPointing to the same object: " + result);

             System.out.println("\nMemory location for theCoach: " + theCoach);

             System.out.println("\nMemory location for alphaCoach: " + alphaCoach + "\n");

             // close the context
             context.close();
      }
}
```

Output

```
Pointing to the same object: true

Memory location for theCoach: com.srvcode.springdemo.BaseballCoach@1e1a0406

Memory location for alphaCoach: com.srvcode.springdemo.BaseballCoach@1e1a0406
```

Code: [Prototype Scope]

beanScope-applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

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

       <bean id="myCoach" class="com.srvcode.springdemo.BaseballCoach"
             scope="prototype">

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

Main Class

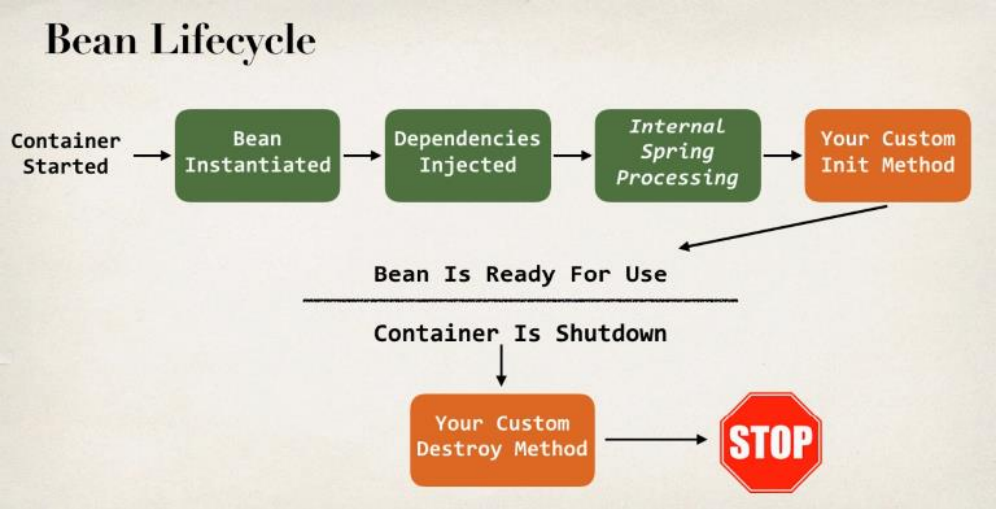
```
public class BeanScopeDemoApp {

       public static void main(String[] args) {

             // load the spring configuration file
             ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("beanScope-applicationContext.xml");
```

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

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

            // check if they are the same
            boolean result = (theCoach == alphaCoach);

            // print out the results
            System.out.println("\nPointing to the same object: " + result);

            System.out.println("\nMemory location for theCoach: " + theCoach);

            System.out.println("\nMemory location for alphaCoach: " + alphaCoach + "\n");

            // close the context
            context.close();
      }
}
```

Output

```
Pointing to the same object: false

Memory location for theCoach: com.srvcode.springdemo.BaseballCoach@3cebbb30

Memory location for alphaCoach: com.srvcode.springdemo.BaseballCoach@12aba8be
```

Bean Lifecycle



Bean Lifecycle Methods / Hooks

> You can add custom code during bean initialization
    a) Calling custom business logic methods.
    b) Setting up handles to resources (db, sockets, file etc)

> You can add custom code during bean destruction
    a) Calling custom business logic method
    b) Clean up handles to resources (db, sockets, files etc)

> During the bean lifecycle, Spring allows you to call some of your custom code, & these are what we call hooks, where you can actually hook in codes to execute during bean initialization or bean destruction.

> Init method

> Destroy method

Special Note: Defining init & destroy methods – Method Signatures

> **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." as you will not be able to capture the return value.
> **Method name** – The method can have any method name.
> **Arguments** – The method can not accept any arguments. The method should be no – arg.

Code:

```
public class TrackCoach implements Coach {

	private FortuneService fortuneService;

	public TrackCoach() {

	}

	public TrackCoach(FortuneService fortuneService) {
		this.fortuneService = fortuneService;
	}

	@Override
	public String getDailyWorkout() {
		return "Run a hard 5k.";
	}

	@Override
	public String getDailyFortune() {
		return "Just Do It: " + fortuneService.getFortune();
	}

	// add an init method
	public void doMyStartupStuff() {
		System.out.println("TrackCoach: inside method doMyStartupStuff");
	}
```

```
	// add a destroy
	public void doMyCleanupStuffYoYo() {
		System.out.println("TrackCoach: inside method doMyCleanupStuffYoYo");
	}
}
```

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans..>
	<!-- define the dependency -->
	<bean id="myFortuneService" class="com.srvcode.springdemo.HappyFortuneService">
	</bean>

	<bean id="myCoach" class="com.srvcode.springdemo.TrackCoach"
		init-method="doMyStartupStuff" destroy-method="doMyCleanupStuffYoYo">

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

</beans>
```

Main Class

```
public class BeanLifeCycleDemoApp {

	public static void main(String[] args) {

		// load the spring configuration file
		ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("beanLifeCycle-applicationContext.xml");

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

		System.out.println(theCoach.getDailyWorkout());

		// close the context
		context.close();

	}

}
```

Output

```
TrackCoach: inside method doMyStartupStuff
Run a hard 5k.
TrackCoach: inside method doMyCleanupStuffYoYo
```

Special Note about Destroy Lifecycle & Prototype Scope

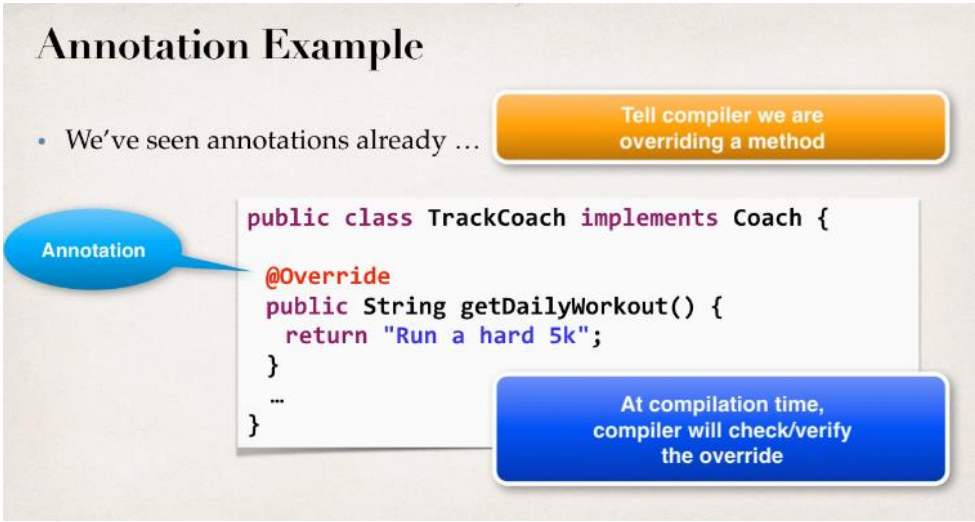
> *For "prototype" scoped beans, Spring does not call the destroy method*.
> In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean: the container instantiates, configures, & otherwise assembles a prototype object, & hands it to the client, with no further 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 & release expensive resources that the prototype bean(s) are holding.

> **Do refer code for calling the destroy method on prototype scope beans**

Spring Configuration with Java Annotations – Inversion of Control

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.

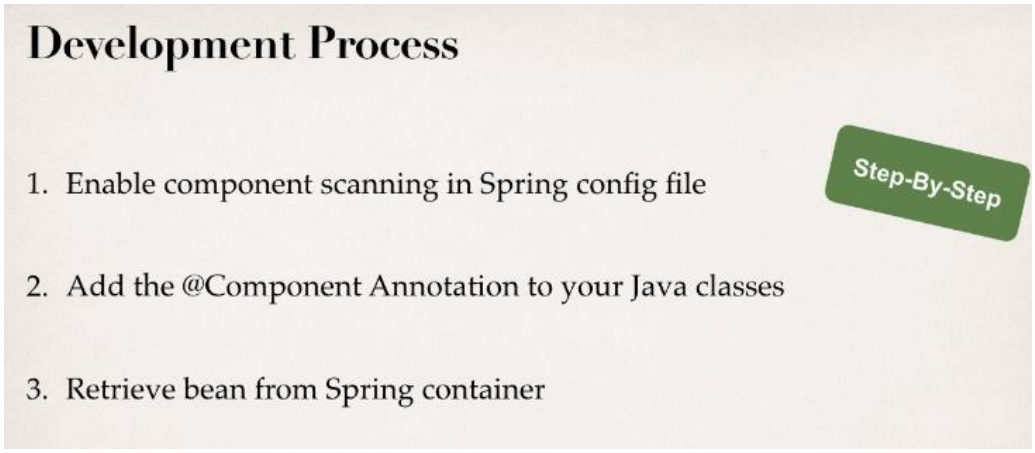


Why Spring Configuration with Annotations?

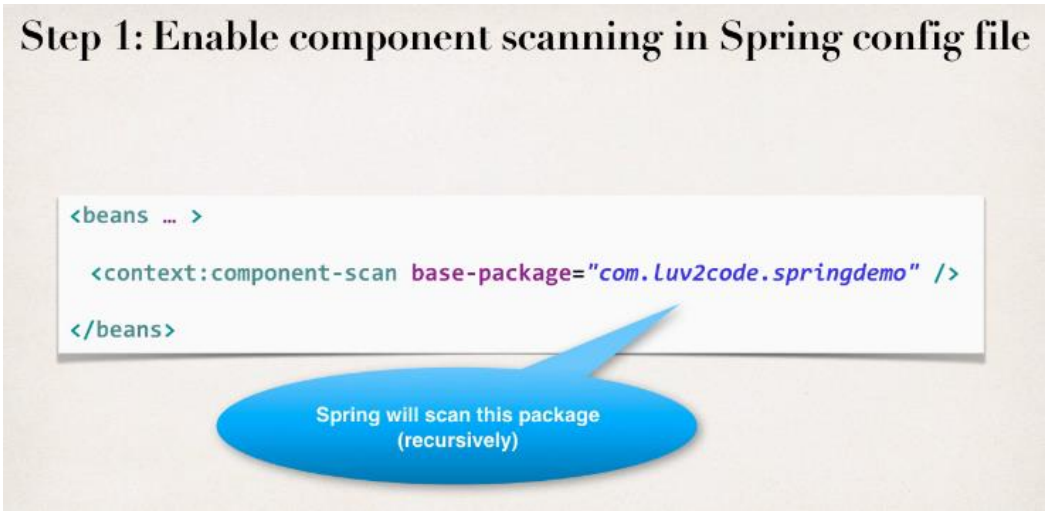
> Annotations minimizes the XML configuration.

Scanning for Component Classes

> Spring will scan your java classes for special annotations
> Automatically register the beans in the Spring container



Step 1: Enable component scanning in Spring configuration file



Step 2: Add the @Component Annotation to your java classes

Step 3: Retrieve bean from Spring container

Code:

Step 1: Enable component scanning in Spring configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

       <!-- add entry to enable component scanning -->

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

</beans>
```

Step 2: Add the @Component Annotation to your java classes

```
public interface Coach {

       public String getDailyWorkout();
}

@Component("thatSillyCoach")
public class TennisCoach implements Coach {

       @Override
       public String getDailyWorkout() {
              return "Practice your backhand volley";
       }

}
```

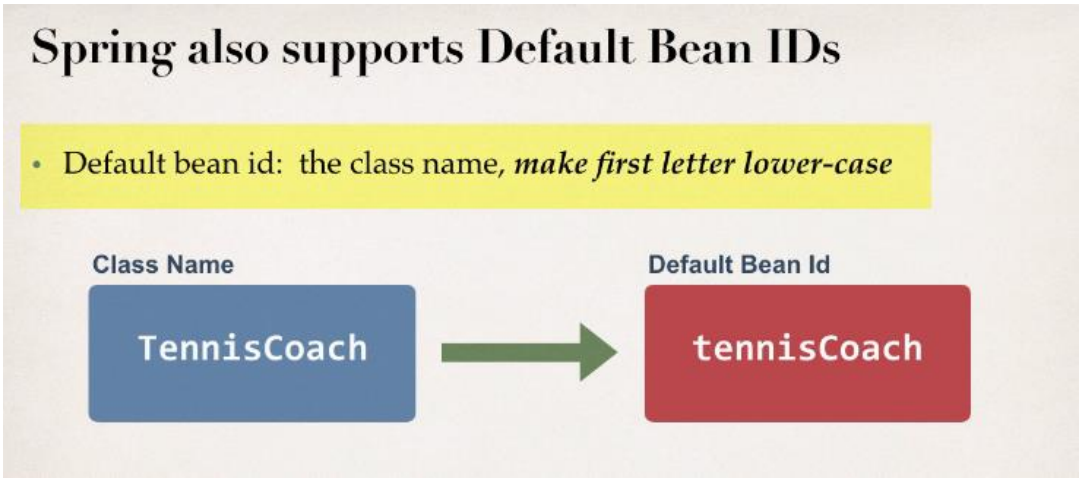
Step 3: Retrieve bean from Spring container

```
public class AnnotationDemoApp {

	public static void main(String[] args) {

		// read spring config file
		ClassPathXmlApplicationContext context =
					new ClassPathXmlApplicationContext("applicationContext.xml");

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

		// call a method on the bean
		System.out.println(theCoach.getDailyWorkout());

		// close the context
		context.close();
	}
}
```

Output

```
Practice your backhand volley
```

Default Component Names



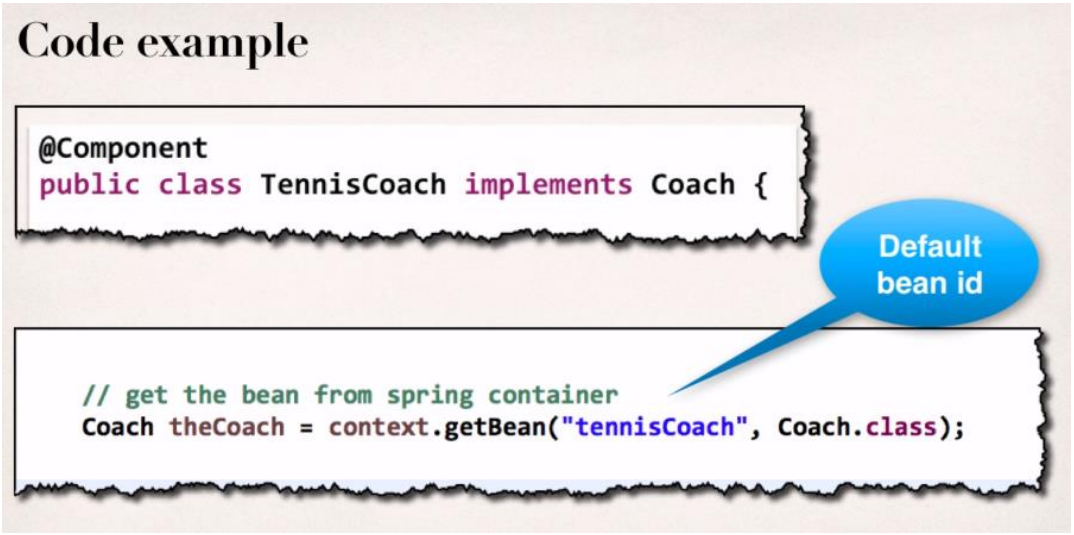
> In general, when using Annotations, for the default bean name, Spring uses the following rule:

a) 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).

e.g. *HappyFortuneService* → *happyFortuneService*

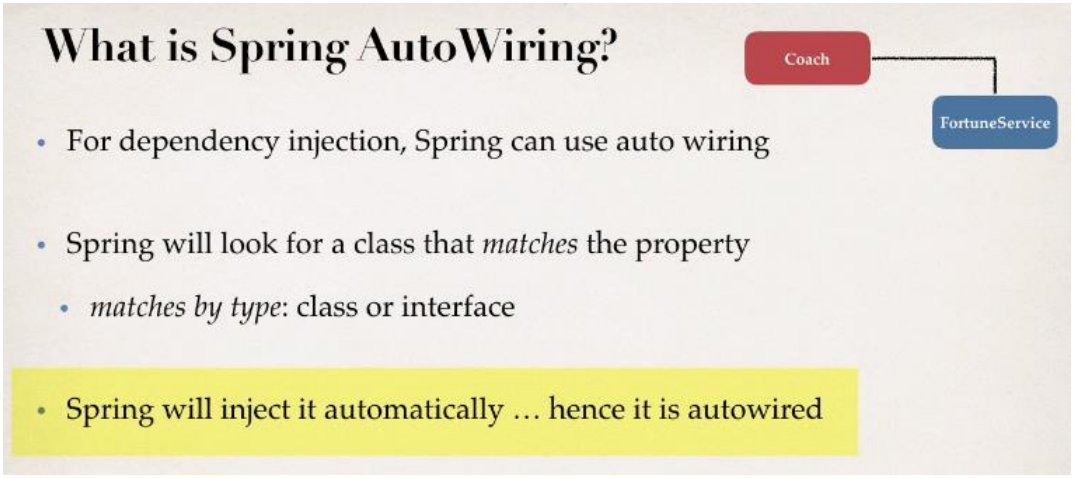
b) However, for the special case of when BOTH the first & second characters of the class name are upper case, then the name is NOT converted.

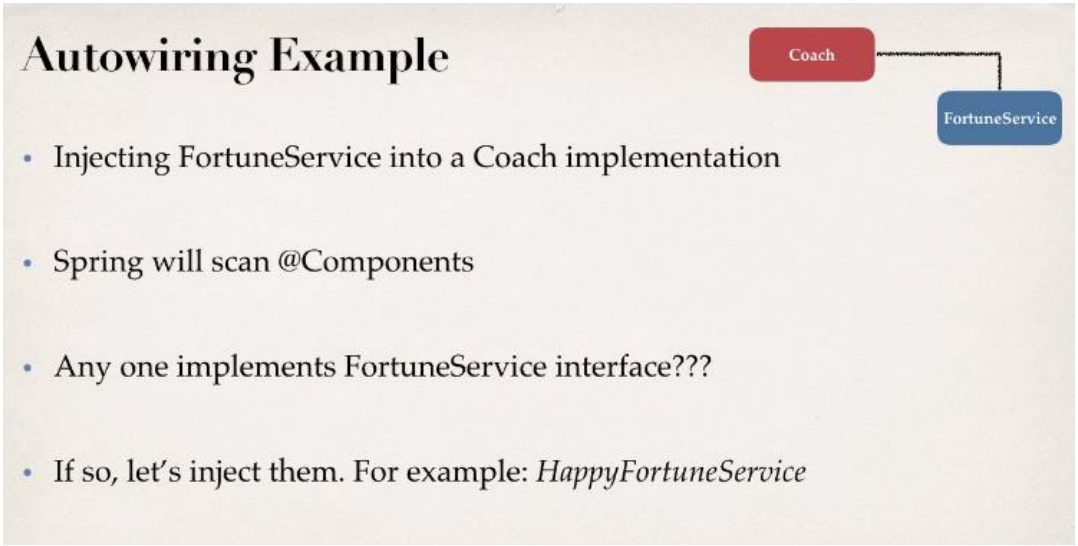
e.g. *RESTFortuneService* → *RESTFortuneService*



> Behind the scenes, Spring uses the **Java Beans introspector** to generate the default bean name.

Spring Dependency Injection with Annotations & Autowiring





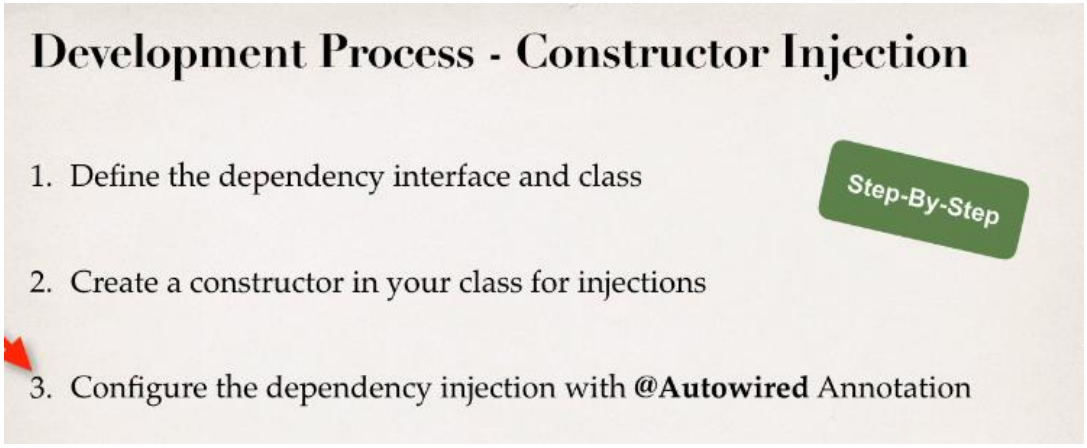
Autowiring Injection Types

1. Constructor Injection

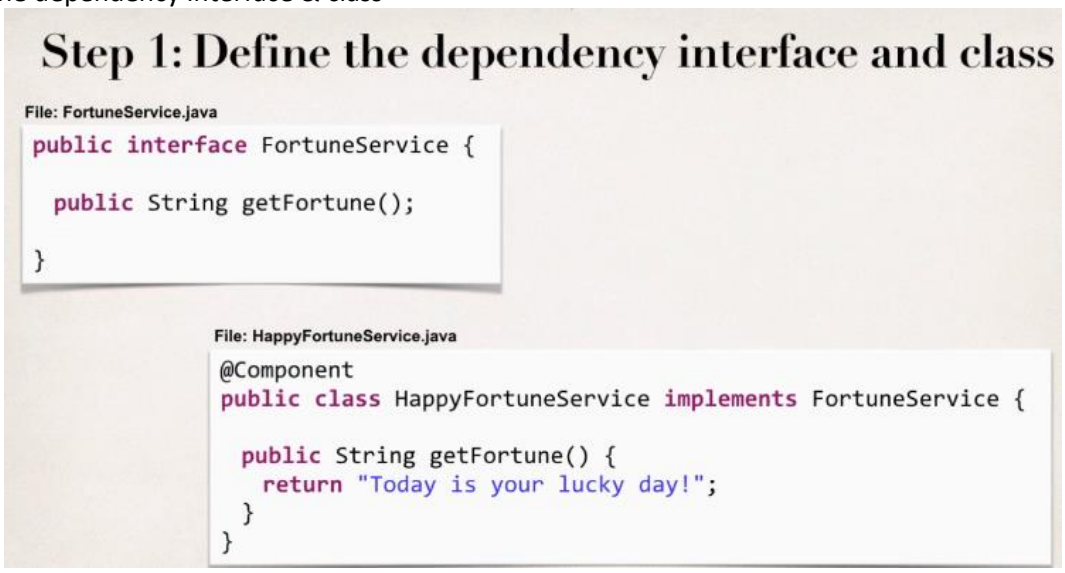
2. Setter Injection

3. Field Injection

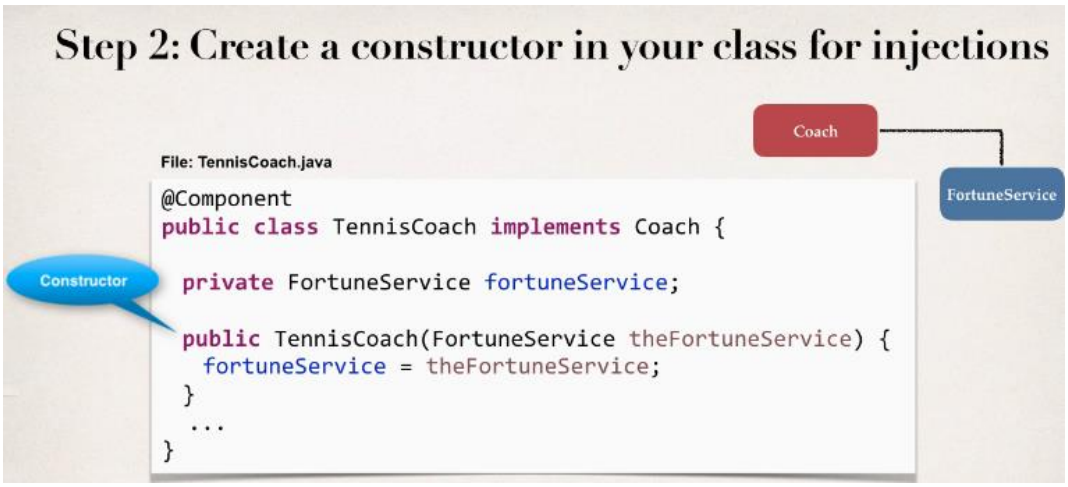
1. Constructor Injection



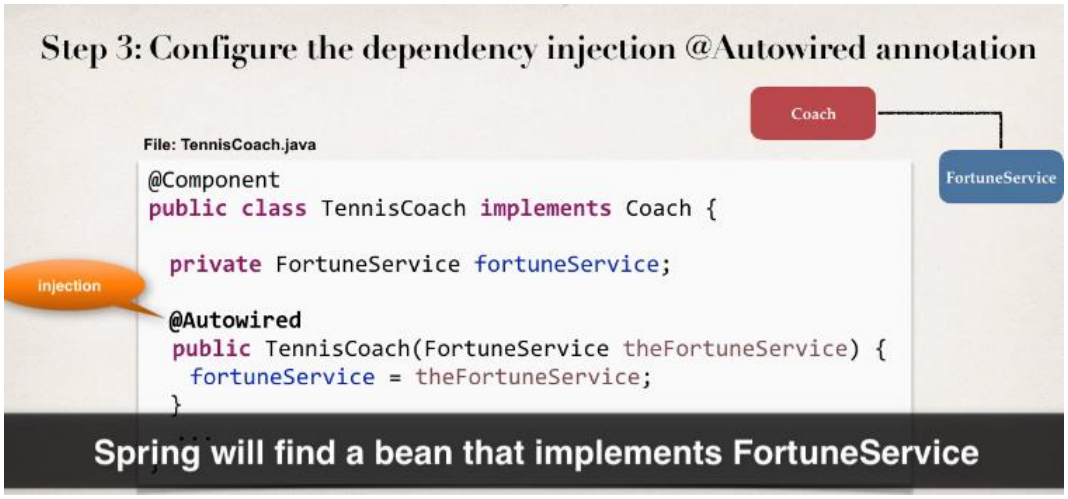
Step 1: Define the dependency interface & class



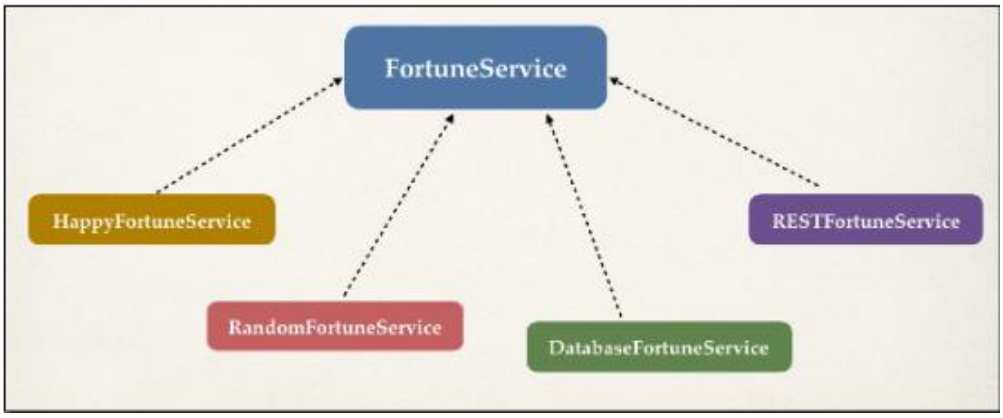
Step 2: Create a constructor in your class for injections



Step 3: Configure the dependency injection @Autowired annotation



Q. What if there are multiple FortuneService implementations?



Ans: - 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.

Code:

Step 1: Define the dependency interface & class

```
public interface FortuneService {

	public String getFortune();

}

public interface Coach {

	public String getDailyWorkout();

	public String getDailyFortune();

}

@Component
public class HappyFortuneService implements FortuneService {

	@Override
	public String getFortune() {
		return "Today is your lucky day!";
	}

}
```

Step 2: Create a constructor in your class for injections

Step 3: Configure the dependency injection @Autowired annotation (Spring will scan for a component that implements FortuneService interface)

```
@Component
public class TennisCoach implements Coach {

	private FortuneService fortuneService;

	@Autowired
	public TennisCoach(FortuneService theFortuneService) {
		fortuneService = theFortuneService;
	}

	@Override
	public String getDailyWorkout() {
		return "Practice your backhand volley";
	}

	@Override
	public String getDailyFortune() {
		return fortuneService.getFortune();
	}

}
```

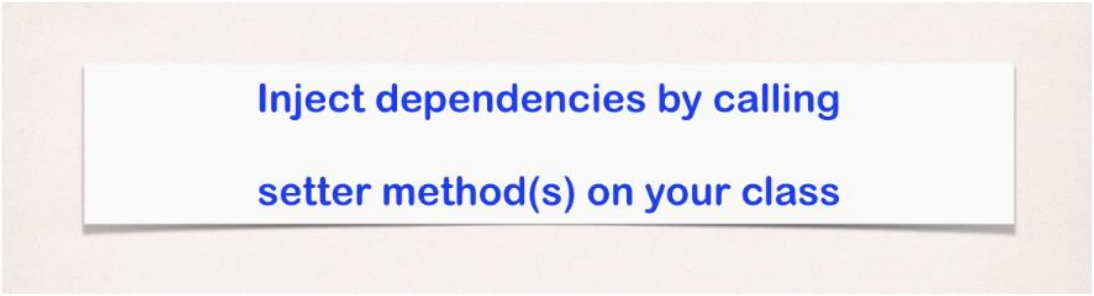
Main Application

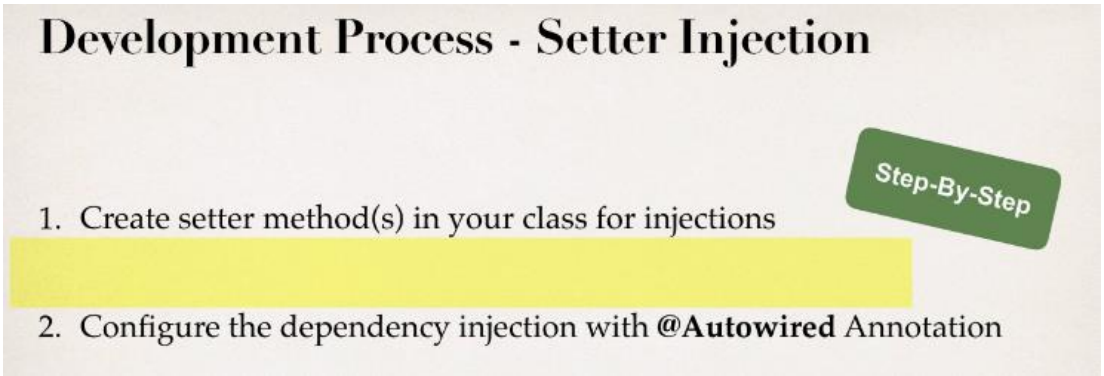
```
public class AnnotationDemoApp {

	public static void main(String[] args) {

		// read spring config file
		ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

		// get the bean from spring container
		Coach theCoach = context.getBean("tennisCoach", Coach.class);

		// call a method on the bean
		System.out.println(theCoach.getDailyWorkout());

		System.out.println(theCoach.getDailyFortune());

		// close the context
		context.close();
	}
}
```

Output

```
FINEST: Returning cached instance of singleton bean 'tennisCoach'
Practice your backhand volley
Today is your lucky day!
```

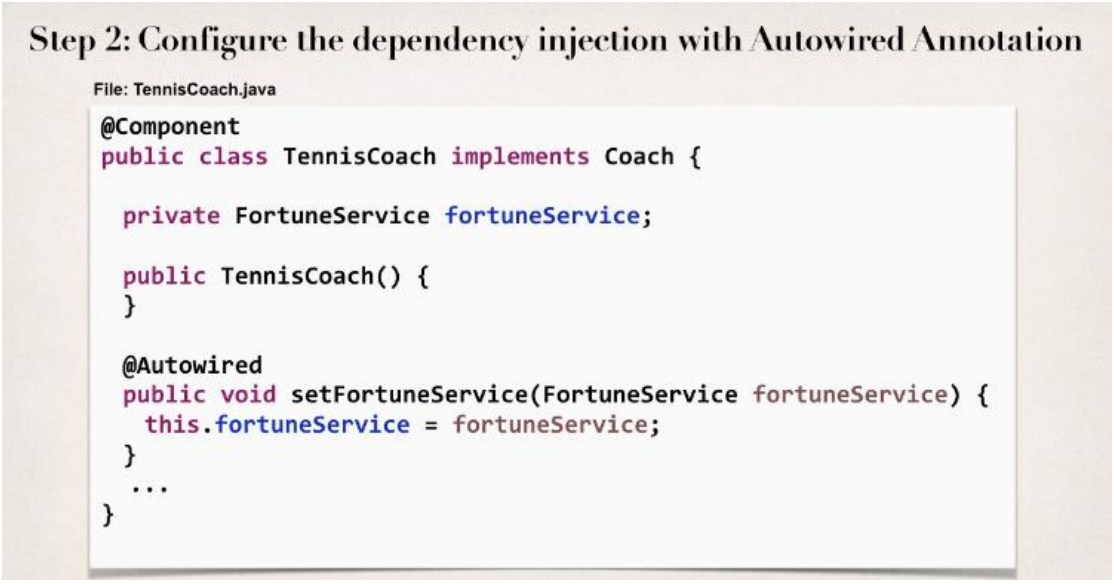
2. Setter Injection



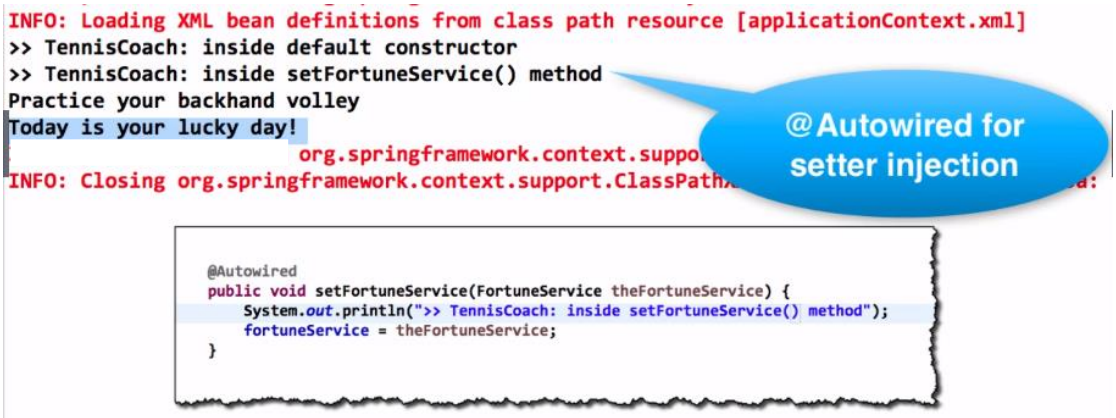


Step 1: Create setter method(s) in your class for injections

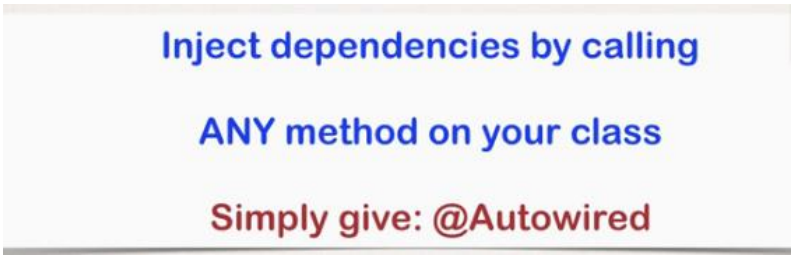
Step 2: Configure the dependency injection with @Autowired Annotation



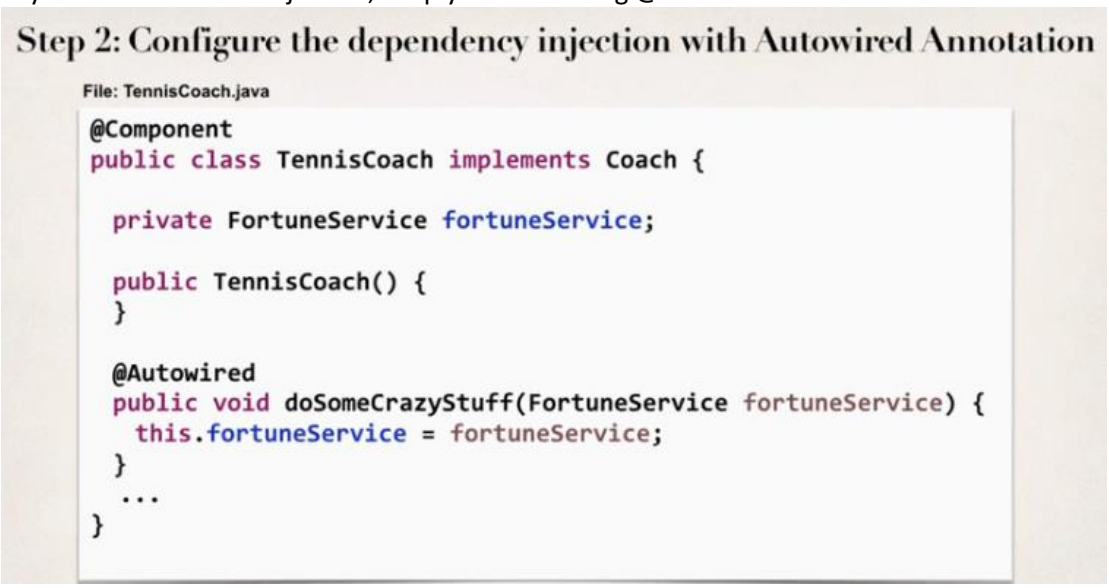
Output:



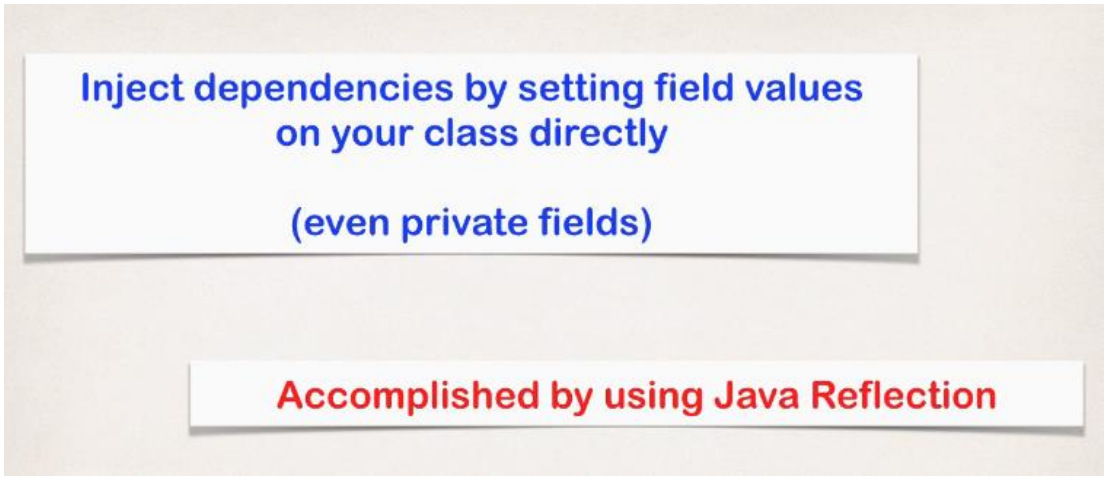
Method Injection

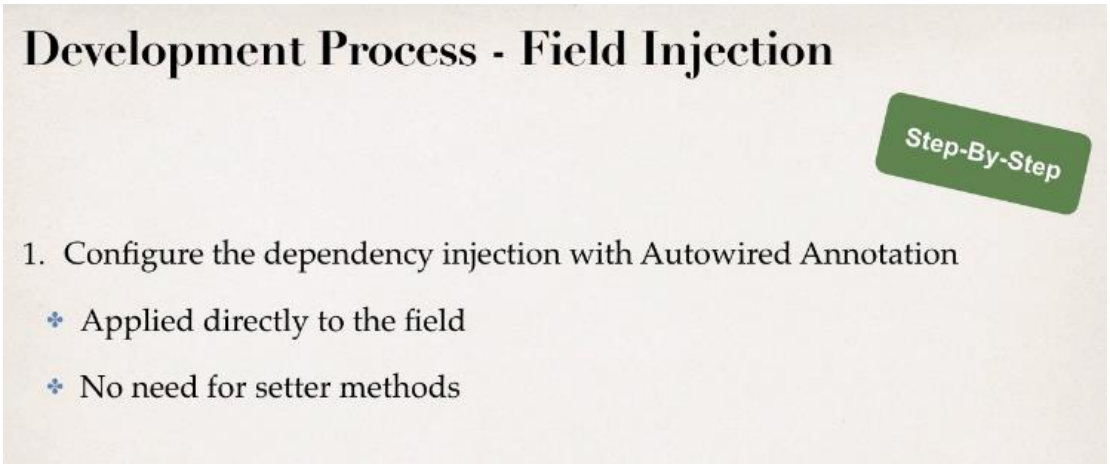


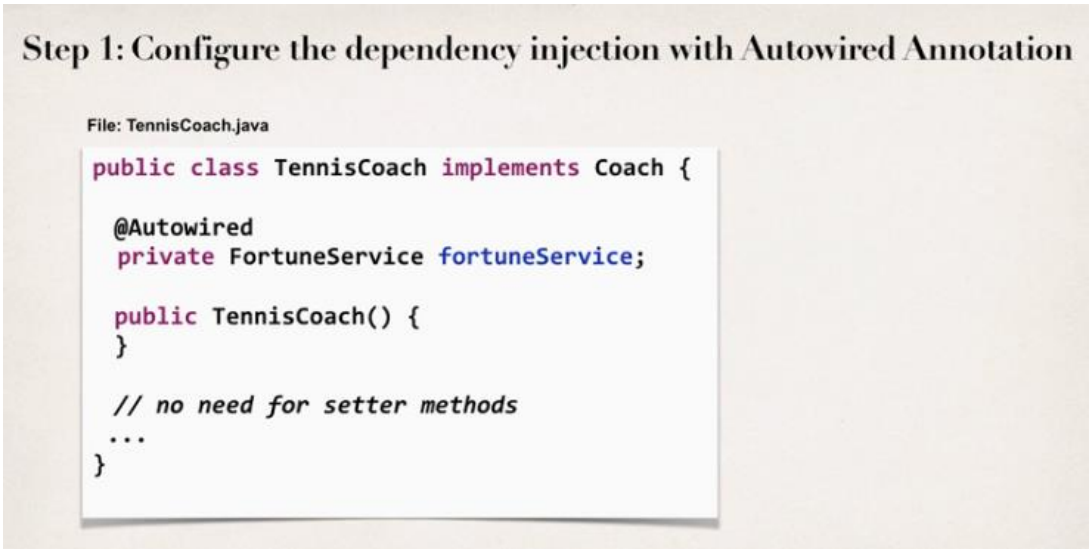
> We can use any method name for injection, simply list it as being @Autowired.



3. Field Injection



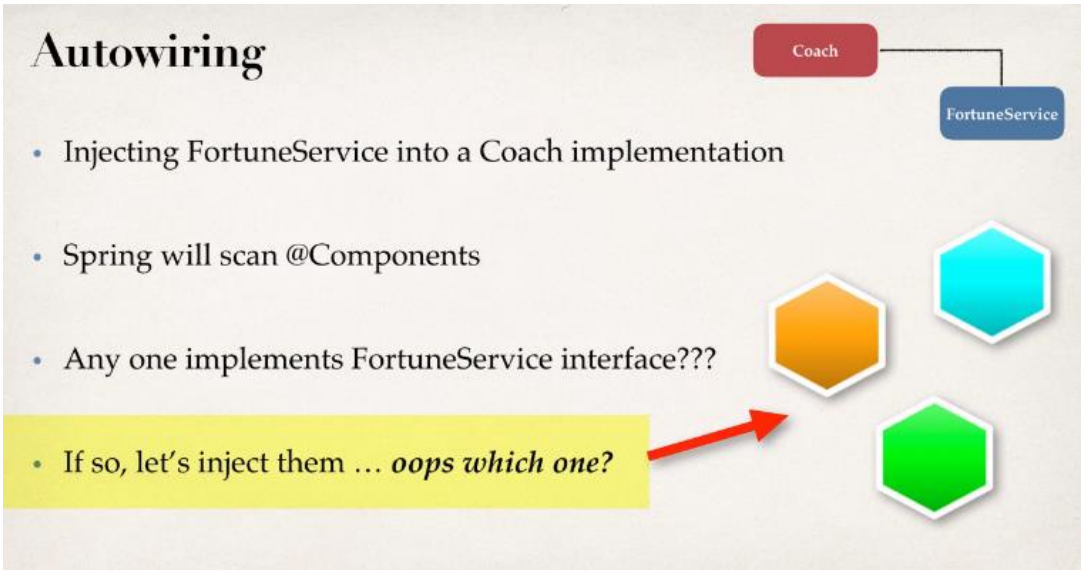


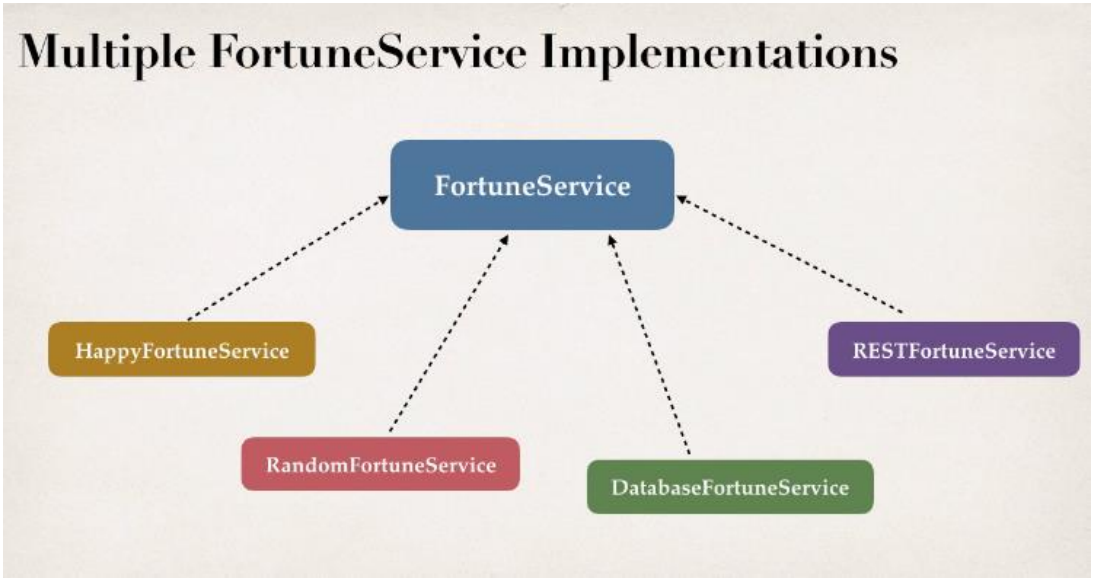


Note:
Choose a spring injection Type & stay consistent in your project.

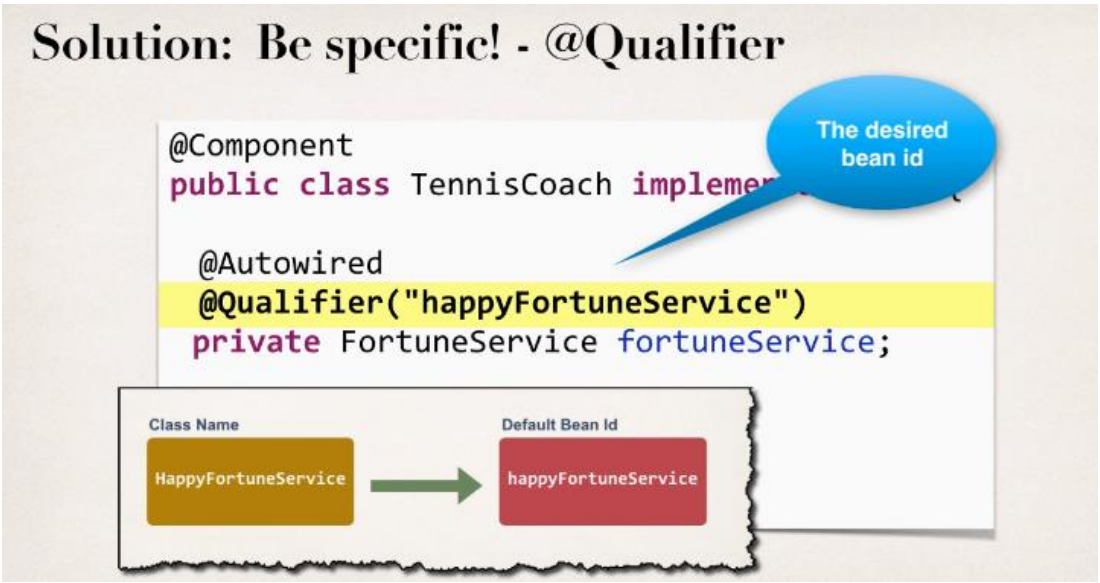
Qualifiers for Dependency Injection

> Problem when multiple Service implementation





> Solution (@Qualifier can be applied to all type of injections)



Note:
> @Qualifier is a nice feature but it's tricky when used with Constructors.
> ***You have to place the @Quantifier annotation inside of the constructor arguments.***

Code:

```
@Component
public class TennisCoach implements Coach {

	private FortuneService fortuneService;

	// define a default constructor
	public TennisCoach() {
		System.out.println(">> TennisCoach: inside default constructor");
	}


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

	@Override
	public String getDailyWorkout() {
		return "Practice your backhand volley";
	}

	@Override
	public String getDailyFortune() {
		return fortuneService.getFortune();
	}

}
```

Inject properties file using Java Annotations

Step1: Create a properties file to hold your properties. It will be a name value pair.

New text file: src/sport.properties

```
foo.email=srvcode@gmail.com
foo.team=Silly Java Coders
```

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

Step2: Load the properties file in the XML configuration file.

File: applicationContext.xml

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

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

> This should appear just after the <context:component – scan…/> line.

Step 3: Inject the properties value into our SwimCoach

```
@Value("${foo.email}")
private String email;

@Value("${foo.team}")
private String team;
```

Bean Scopes & Lifecycle Methods

> Scope refers to the lifecycle of a bean like how long does the bean live, how many instances are created & how is the bean shared?

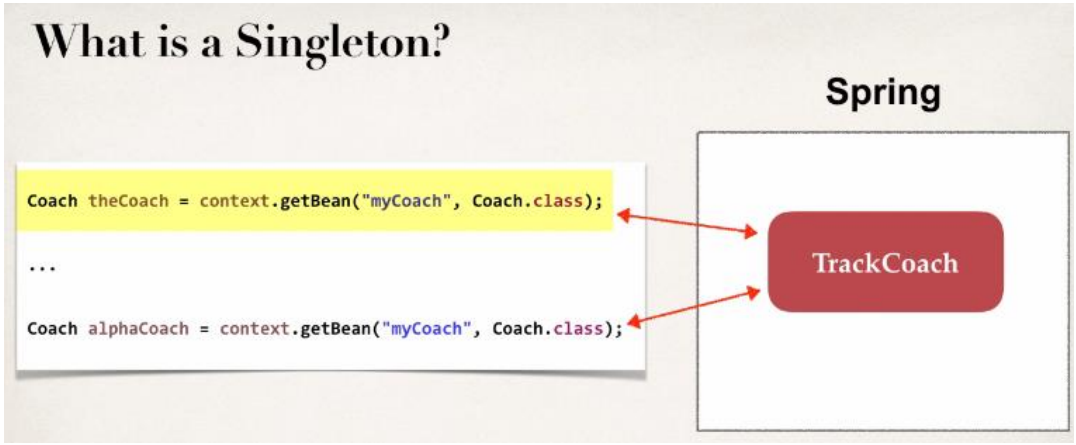
> Default scope is singleton

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.



Explicitly Specify Bean Scope

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

...

}
```

Prototype Scope Example

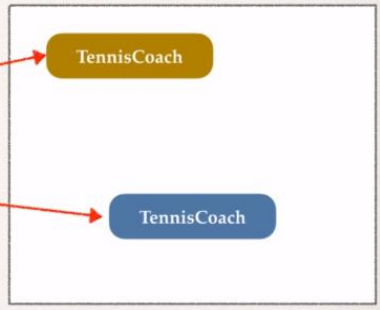
Prototype scope: new object for each request

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

...

}
```

Prototype Scope Example

Prototype scope: new object for each request

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

...

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



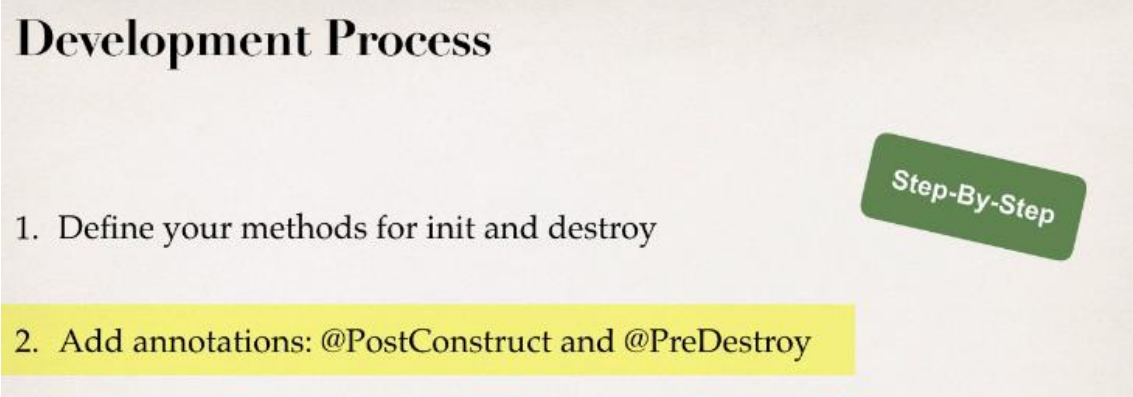
Bean Lifecycle Methods / Hooks

> You can add custom code during **bean initialization**

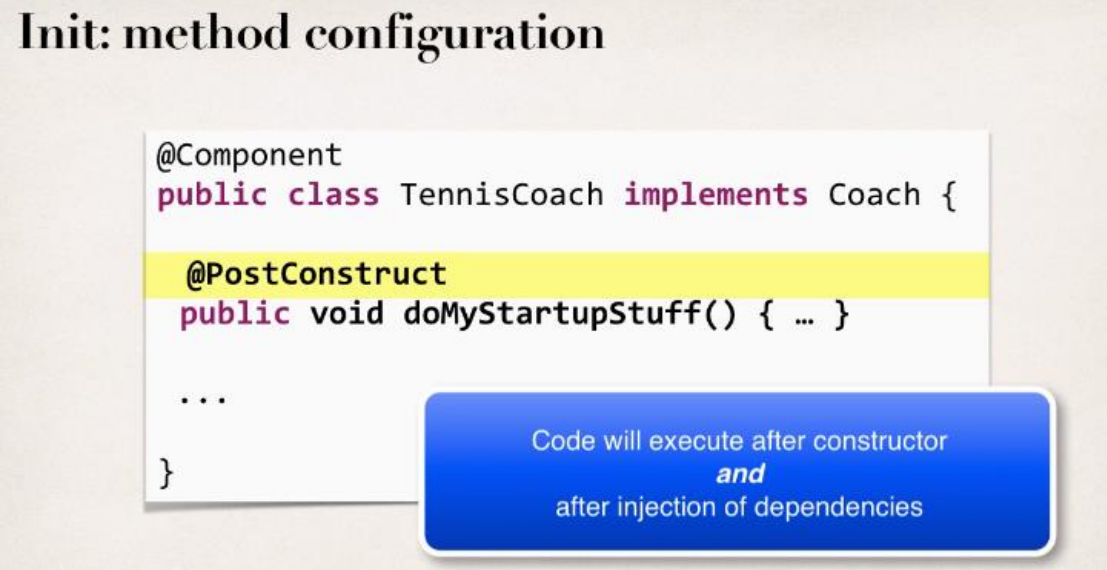
- Calling custom business logic methods
- Setting up handles to resources (db, sockets, file etc)

> You can add custom code during **bean destruction**

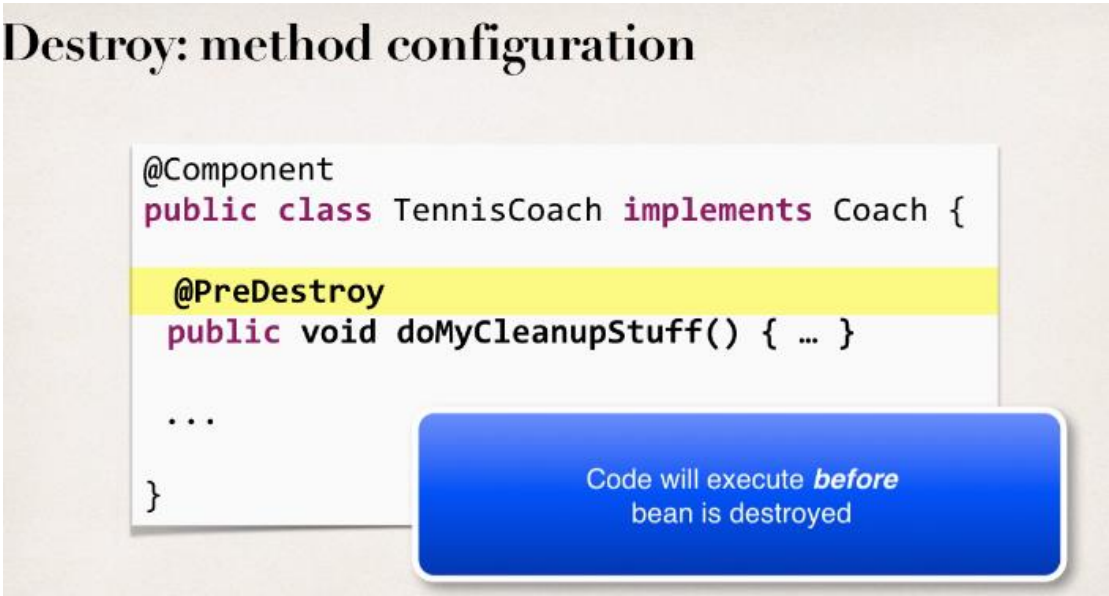
- Calling custom business logic methods
- Clean up handles to resources (db, sockets, file etc)



> Init: method configuration



> Destroy: method configuration



@PostConstruct & @PreDestroy Method Signatures

> **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 as if you give a return type just note that you will not be able to capture the return value.

> **Method name**: - The method can have any method name.

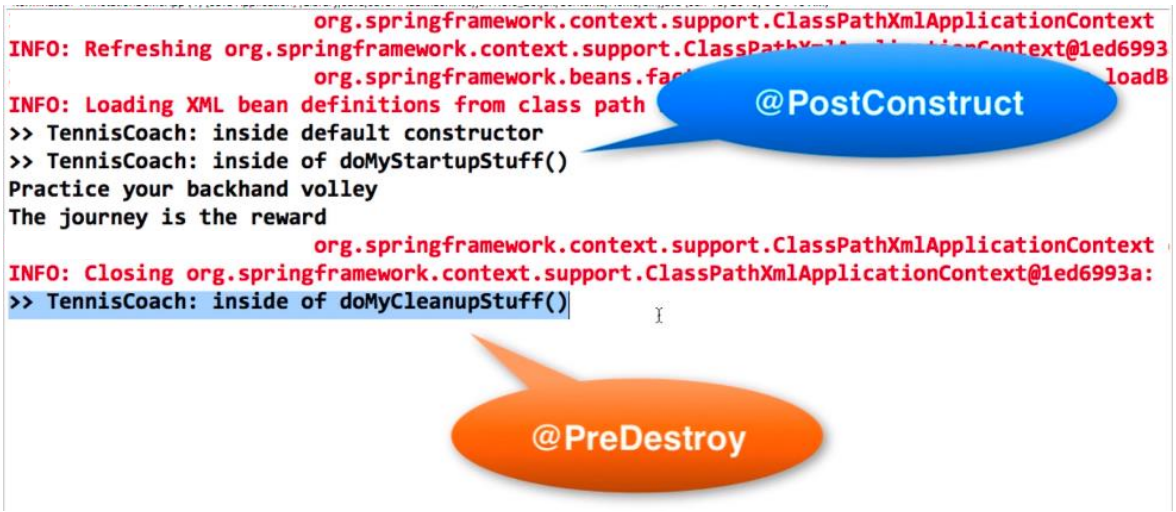
> **Arguments**: - The method can't accept any arguments. The method should be no – arg.

Code:

```
@Component
public class TennisCoach implements Coach {

	@Autowired
	@Qualifier("happyFortuneService")
	private FortuneService fortuneService;

	// define a default constructor
	public TennisCoach() {
		System.out.println(">> TennisCoach: inside default constructor");
	}


	// define my init method
	@PostConstruct
	public void doMyStartupStuff() {
		System.out.println(">> TenisCoach: inside of doMyStartupStuff()");
	}

	// define my destroy method
	@PreDestroy
	public void doMyCleanupStuff() {
		System.out.println(">> TennisCoach: inside of doMyCleanupStuff");
	}
```

```
	@Override
	public String getDailyWorkout() {
		return "Practice your backhand volley";
	}

	@Override
	public String getDailyFortune() {
		return fortuneService.getFortune();
	}

}
```

Output



Destroy Lifecycle & Prototype Scope

> For “prototype” scoped beans, Spring does not call the @PreDestroy method.

> 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 & 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.

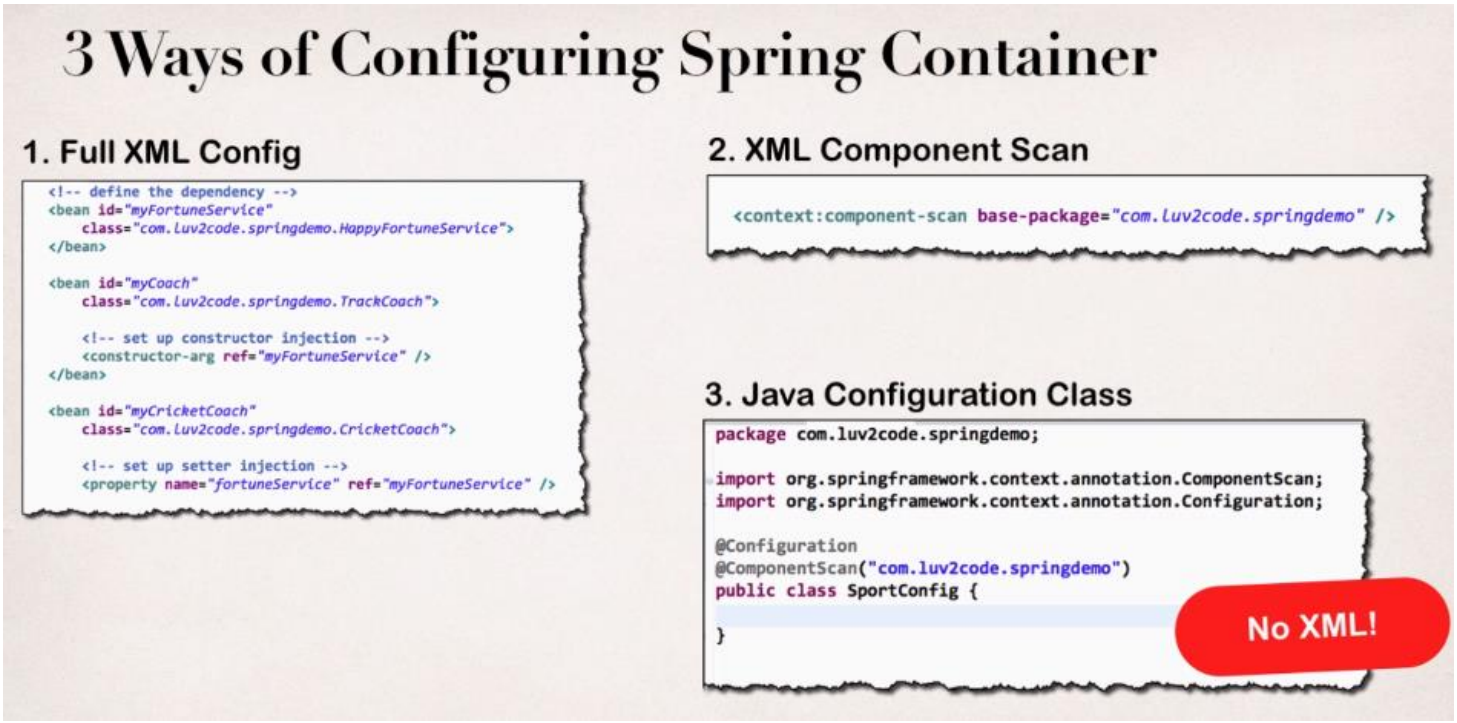
Spring Configuration with Java Code

> How to configure a Spring container using java code

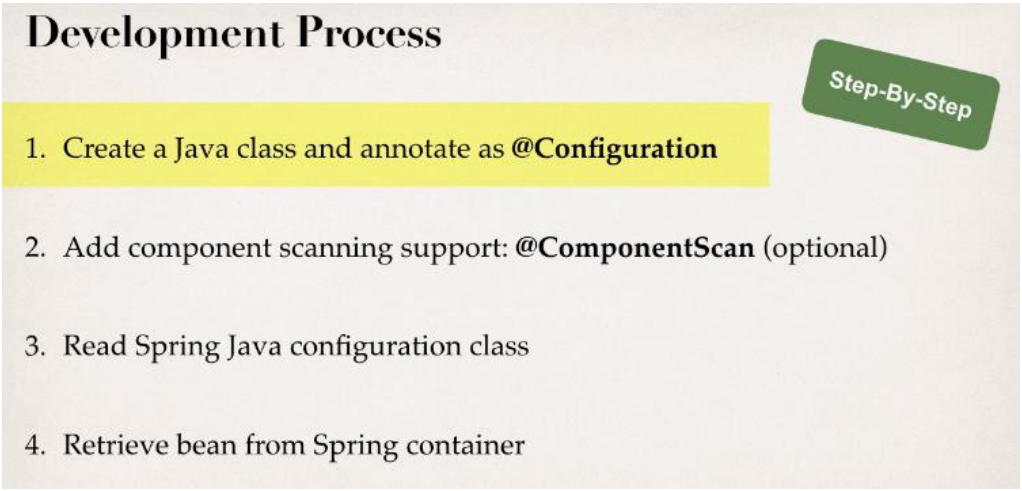
> Instead of configuring Spring container using XML, we can configure the Spring container with Java code.

> There are 3 ways of configuring Spring Container

1. Full XML Configuration
2. XML Component Scan
3. Java Configuration Class



3. Java Configuration Class

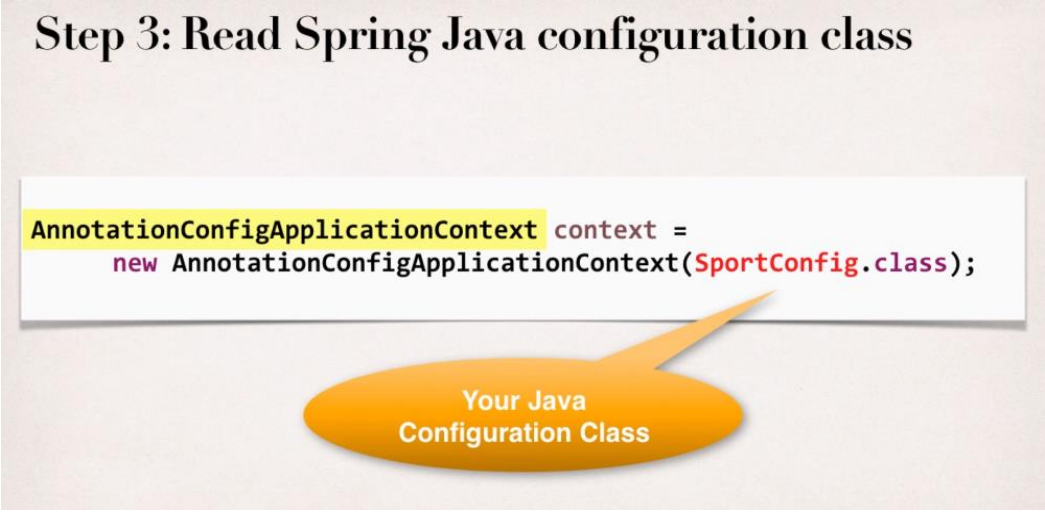


Step 1: Create a Java class & annotate as **@Configuration**

Step 2: Add component scanning support: **@ComponentScan** (simply tell Spring to scan a given package) – we're simply scanning for classes that support at component, & it'll automatically register those with the container.

> Works exactly like XML component scanning

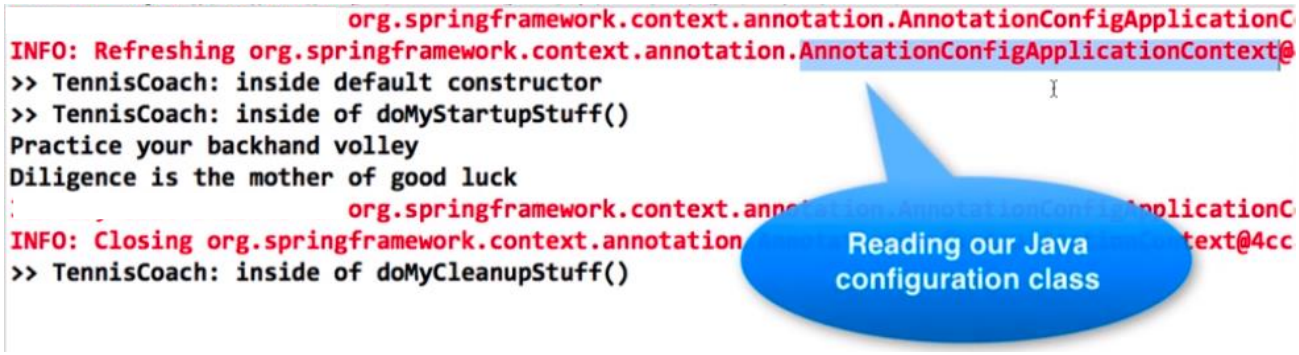
Step 3: Read Spring Java Configuration class



> It'll actually create the Spring container based on that Java source configuration & make the context available for us.

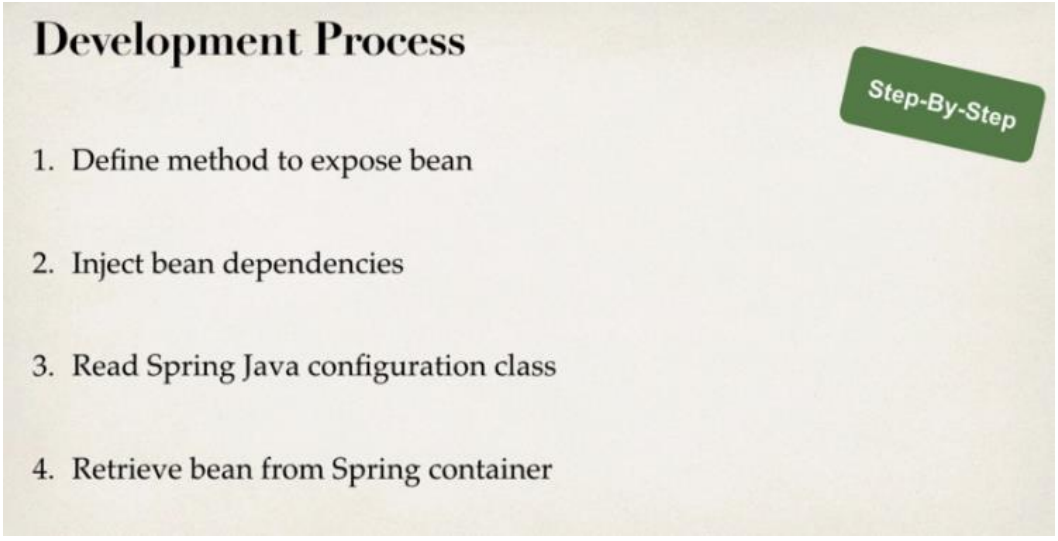
Step 4: Retrieve bean from Spring container (once your context is created, you simply retrieve a bean)

Output



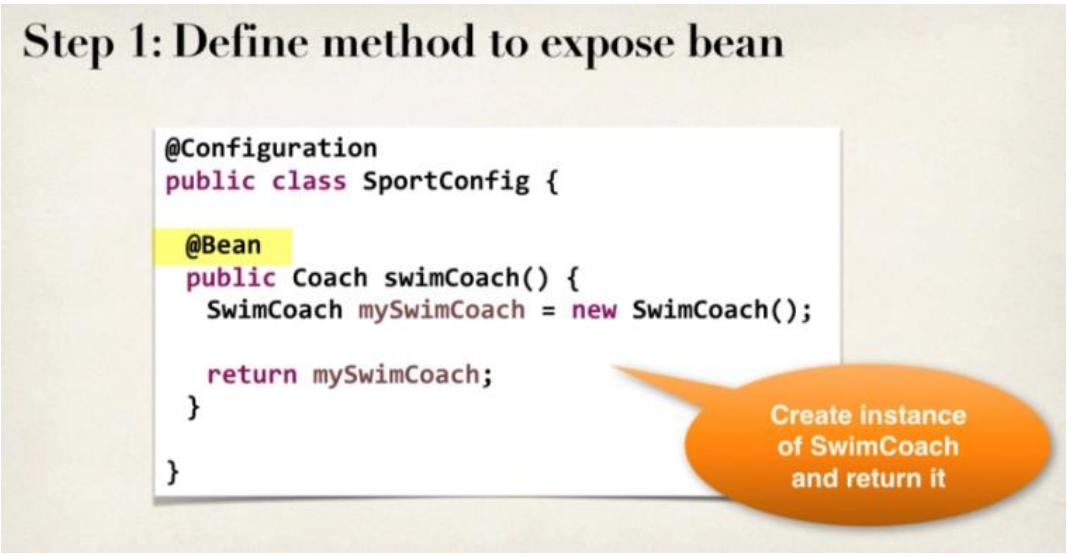
Defining Spring Bean with Java Code (no XML)

> Earlier we learned how to define beans using XML, so we would list each bean out separately in the XML file. Well now, we're making use of these Java configuration classes.



Step 1: Define method to expose bean (Define each bean individually in this configuration class)

> No component scan



Step 2: Inject bean dependencies

Step 3: Read Spring Java Configuration class

Step 4: Retrieve bean from Spring container

Code:

```
public class SadFortuneService implements FortuneService {

	@Override
	public String getFortune() {
		return "Today is a sad day";
	}

}

public class SwimCoach implements Coach {

	private FortuneService fortuneService;

	public SwimCoach(FortuneService theFortuneService) {
		fortuneService = theFortuneService;
	}

	@Override
	public String getDailyWorkout() {
		return "Swim 1000 meters as a warm up.";
	}

	@Override
	public String getDailyFortune() {
		return fortuneService.getFortune();
	}
}
```

Step 1: Define method to expose bean (Define each bean individually in this configuration class)

Step 2: Inject bean dependencies

```java
@Configuration
public class SportConfig {


	// define bean for our sad fortune service
	@Bean
	public FortuneService sadFortuneService() {
		return new SadFortuneService();
	}

	// define bean for our swim coach AND inject dependency

	@Bean
	public Coach swimCoach() {
		return new SwimCoach(sadFortuneService());
	}
}
```

Step 3: Read Spring Java Configuration class

Step 4: Retrieve bean from Spring container

```java
public class SwimJavaConfigDemoApp {

	public static void main(String[] args) {

		// read spring config java class
		AnnotationConfigApplicationContext context =
					new AnnotationConfigApplicationContext(SportConfig.class);

		// get the bean from spring container
		Coach theCoach = context.getBean("swimCoach", Coach.class);

		// call a method on the bean
		System.out.println(theCoach.getDailyWorkout());

		System.out.println(theCoach.getDailyFortune());

		// close the context
		context.close();
	}
}
```

Output:

```
Swim 1000 meters as a warm up.
Today is a sad day
```

Q. How @Bean works behind the scenes?

Ans: - This is an advanced concept. Let's 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.

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

```
public Coach swimCoach() {
```

> This specifies that the bean will have bean 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 & it will check in memory of the Spring container (applicationContext) & see if this given bean has already been created. 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.

> If this is the first time the bean has been created then @Bean will execute the method as normal. It will also register the bean in the application context. Effectively setting a flag.

> If the bean has already been created then it will immediately return the instance from memory. It will not execute the code inside the method. Hence this is a singleton bean.

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

These lines won't be executed if bean has already been created.

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

> This code will create a new instance of the SwimCoach.

```
return mySwimCoach;
```

> This code returns an instance of the swimCoach.

```
return new SwimCoach(sadFortuneService())
```

> This code returns an instance of SwimCoach. The call to the method sadForturnService(), the @Bean will intercept & 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)

Note:

Real – time use case of @Bean: - You can use @Bean to make an existing third – party class available to your Spring framework application context.

e.g., Our Spring application needed to integrate with AWS S3 (Amazon Simple Storage Service) & 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.

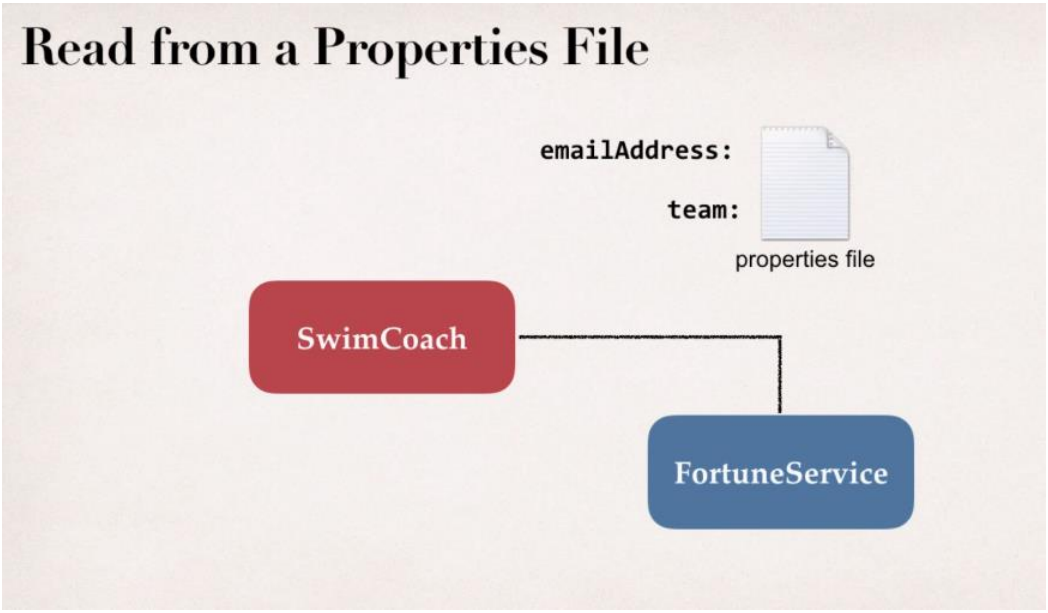
Since the S3Client is part of the AWS framework, we can't modify the source code for the S3Client source code. As a result, we need an alternative solution.

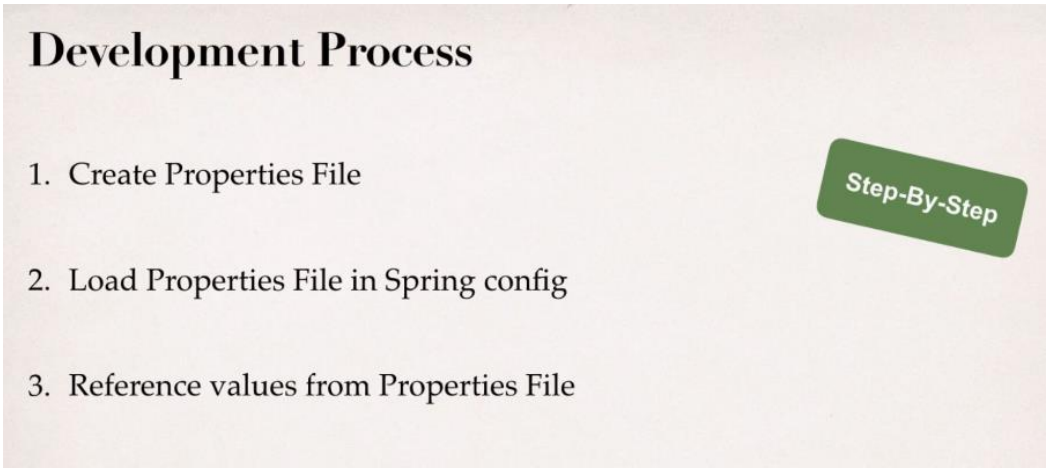
But no problem, by using the @Bean annotation, you can wrap this third – party class, S3Client, as a Spring bean. And then once it is wrapped using @Bean, it is as a singleton object & available in our Spring framework application context.

You can now easily share this bean in your app using dependency injection & @Autowired. So, think of the @Bean annotation as a wrapper/ adapter for third – party classes.

```
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.      }
```

Injecting Values from Properties File

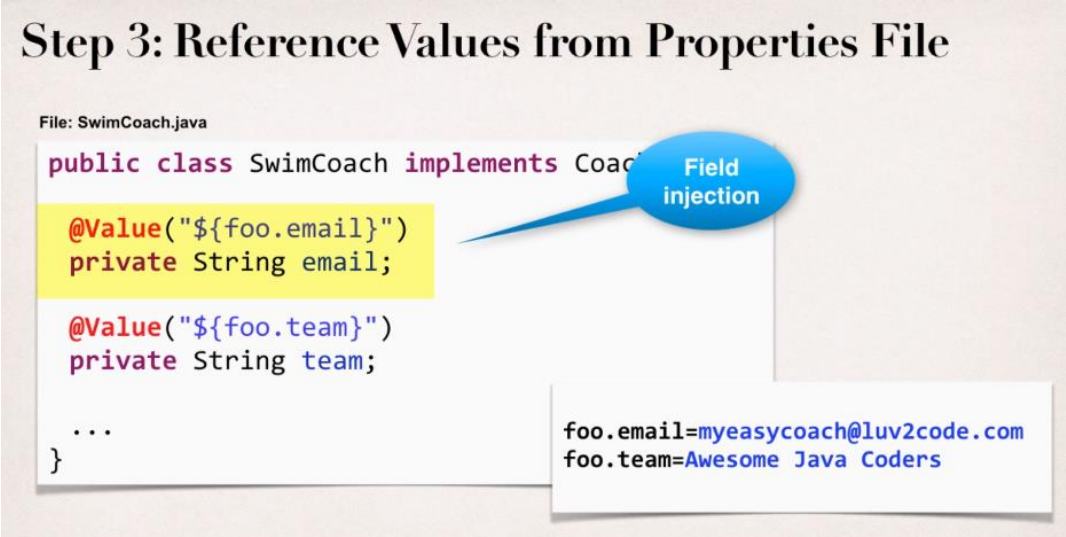




Step 1: Create Properties File

Step 2: Load Properties file in Spring configuration

Step 3: Reference Values from Properties File



Q. I am running the code for Java Configuration and injecting values from props file. However, I'm getting:
${foo.email}
${foo.team}
Instead of the actual property values. How can I resolve this?

Ans: -

This is an issue with Spring versions.

If you are using Spring 4.2 and lower, you will need to add the code in bold.

```
package com.luv2code.springdemo;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;
@Configuration
// @ComponentScan("com.luv2code.springdemo")
@PropertySource("classpath:sport.properties")
public class SportConfig {

   // add support to resolve ${...} properties
   @Bean
   public static PropertySourcesPlaceholderConfigurer
            propertySourcesPlaceHolderConfigurer() {

      return new PropertySourcesPlaceholderConfigurer();
   }

   // define bean for our sad fortune service
   @Bean
   public FortuneService sadFortuneService() {
      return new SadFortuneService();
   }

   // define bean for our swim coach AND inject dependency
   @Bean
   public Coach swimCoach() {
      SwimCoach mySwimCoach = new SwimCoach(sadFortuneService());

      return mySwimCoach;
   }

}
```

In Spring 4.3 and higher, they removed this requirement. As a result, you don't need this code.