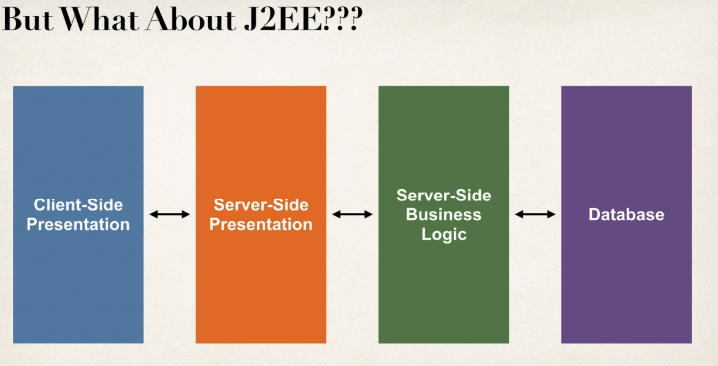
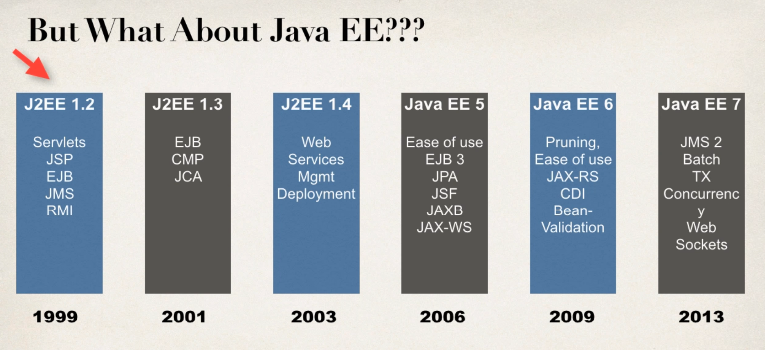
**Spring Framework**

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

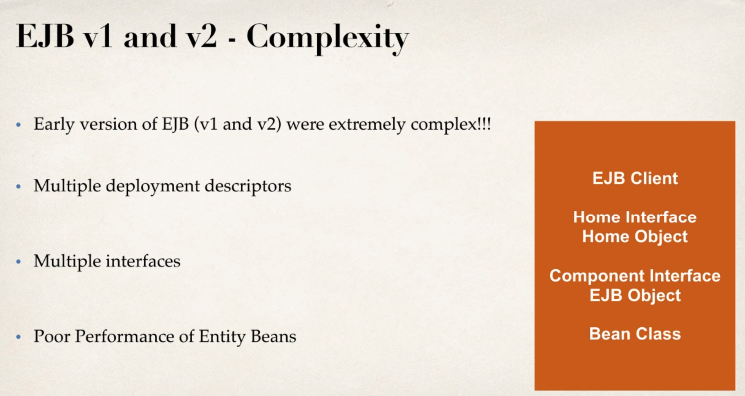
JSF – Java Server Faces

JAXB – Java API for XML Binding

JAX-WS – Java Web Services (SOAP)

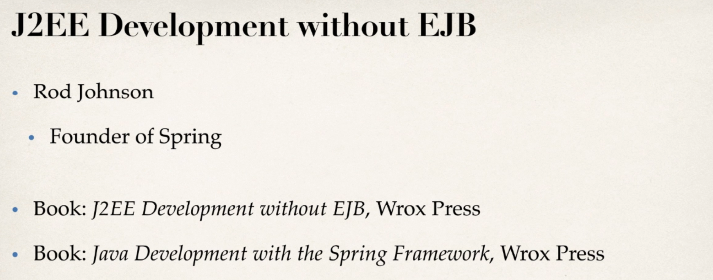
JAX-RS – Java Web Services (REST)

CDI – Context Dependency Injection (IoC)

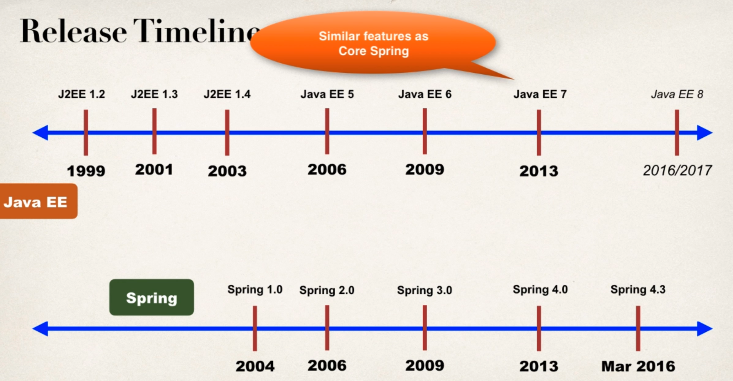


> Entity Beans basically mapping between java class & Database table are just awful slow. Even on one deployment, we actually had to pull our code back out of the production & actually remove the EJB functionality because it slowed everything down.

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

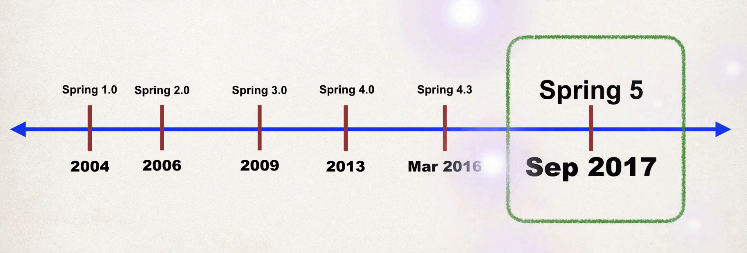


EJB 3.1 in Java EE 6 was much easier to use i.e. in J2EE 6, added CDI Context Dependency Injection IoC.

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.]

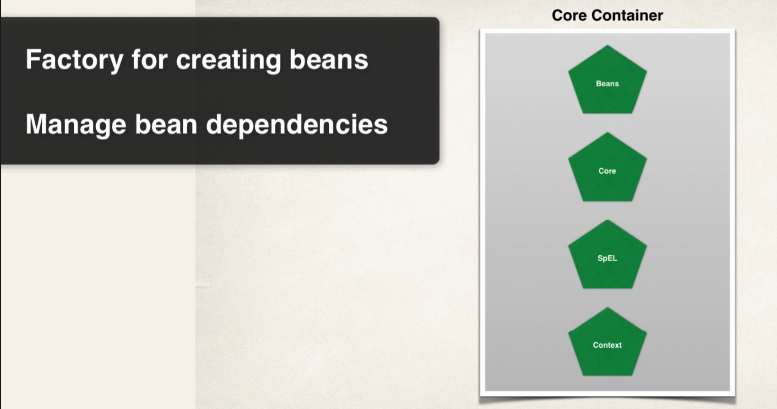
c) Declarative programming with Aspect – Oriented – Programming (AOP)

[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.

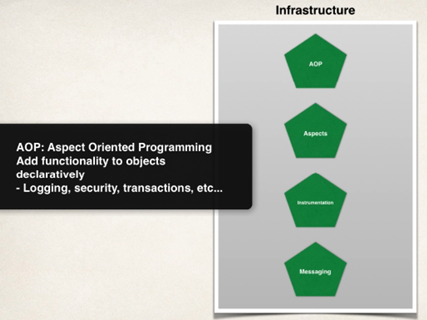
> It basically can read configuration files for setting properties in dependencies.

> 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.

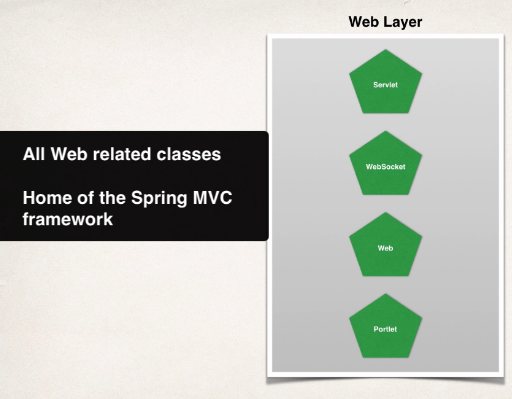
> Basically, Spring provides some helper classes to make it much easier to access a database, using JDBC & by using these Spring JDBC classes, you can actually reduce your source code by over 50%

> **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.

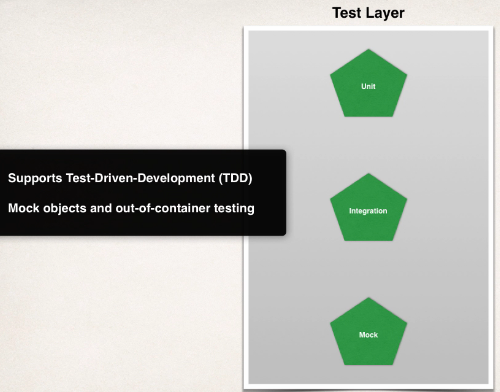
**Spring Web Layer**



> 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]

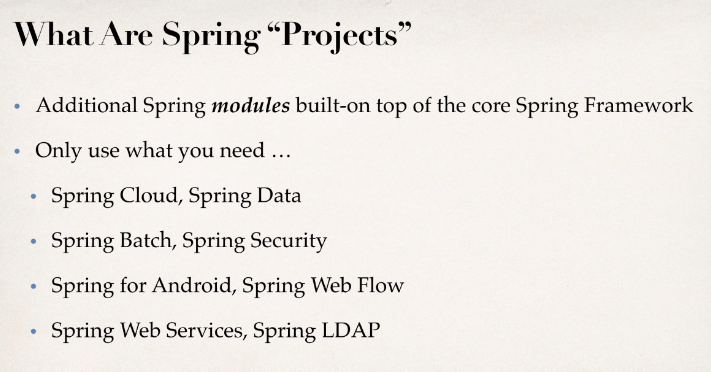
**Spring Test Layer**

****

> 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

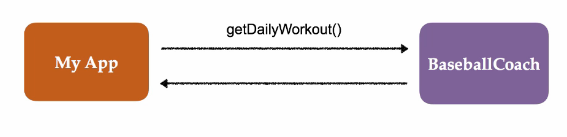
> Once libraries will be built you will see “Referenced Libraries” in package Hierarchy.

**Spring Inversion of Control**

> Inversion of Control (IoC) is simply the design process of externalizing the construction & management of objects.

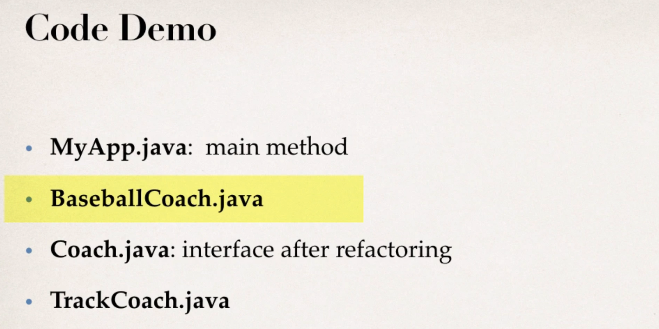
> It basically says that your application’s going to outsource the creation & management of the objects & that outsourcing will be handled by an object factory.

> **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]

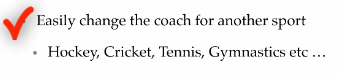


**Code**: By using Coach interface, we can easily change the coach for another sport

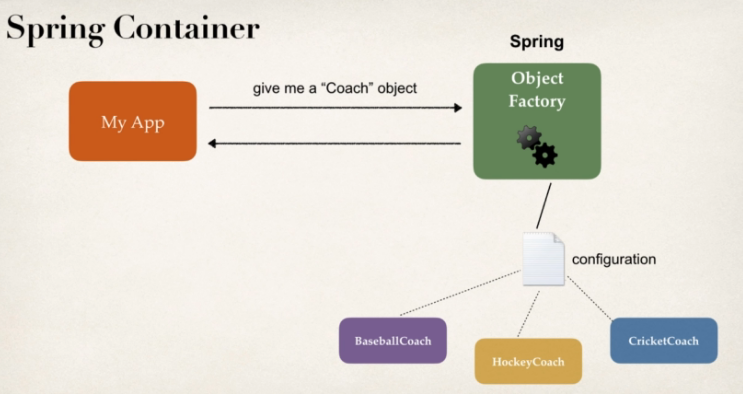
|  |
| --- |
| **public** **interface** Coach {  **public** String getDailyWorkout();  }  **public** **class** BaseballCoach **implements** Coach {  @Override  **public** String getDailyWorkout() {  **return** "Spend 30 minutes on batting practice.";  }  }  **public** **class** TrackCoach **implements** 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.

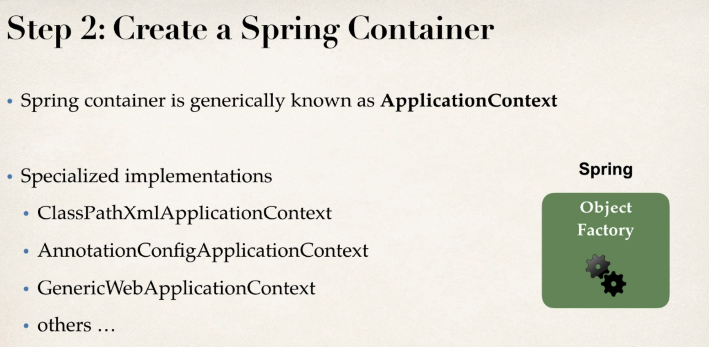
**Step 1:**



> **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:**

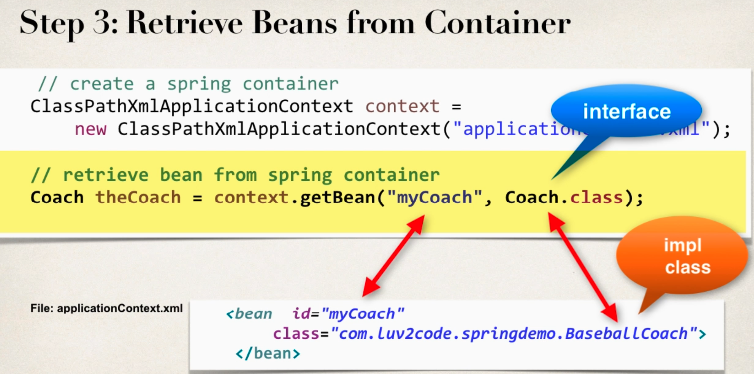




> ***applicationContext.xml*** is the name of configuration file.

> 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.

e.g.

|  |
| --- |
| **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 ()?

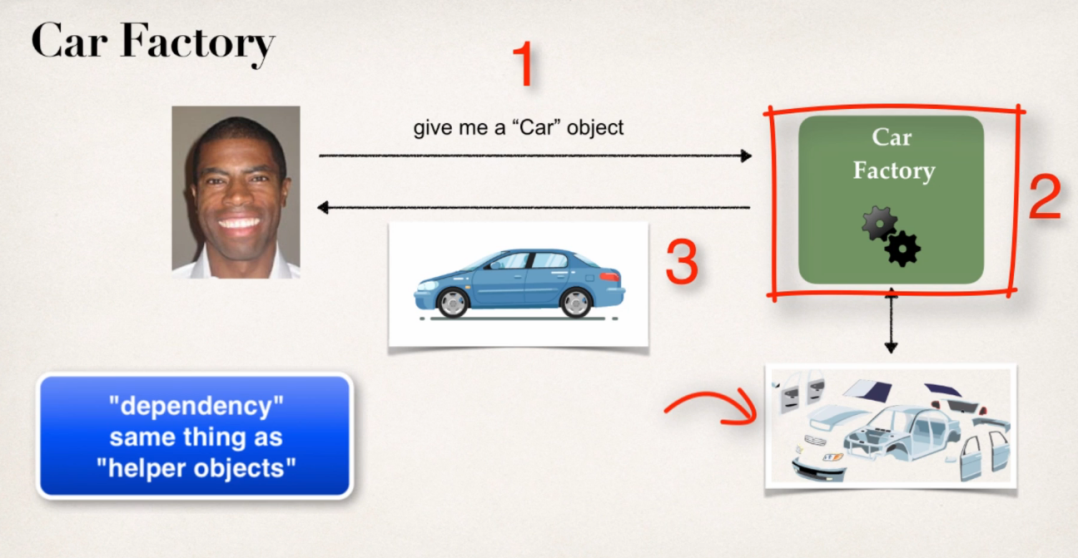
Ans: - e.g. Coach theCoach = context.getBean (“myCoach”, Coach.class);

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.



**Client**

Explanation: -

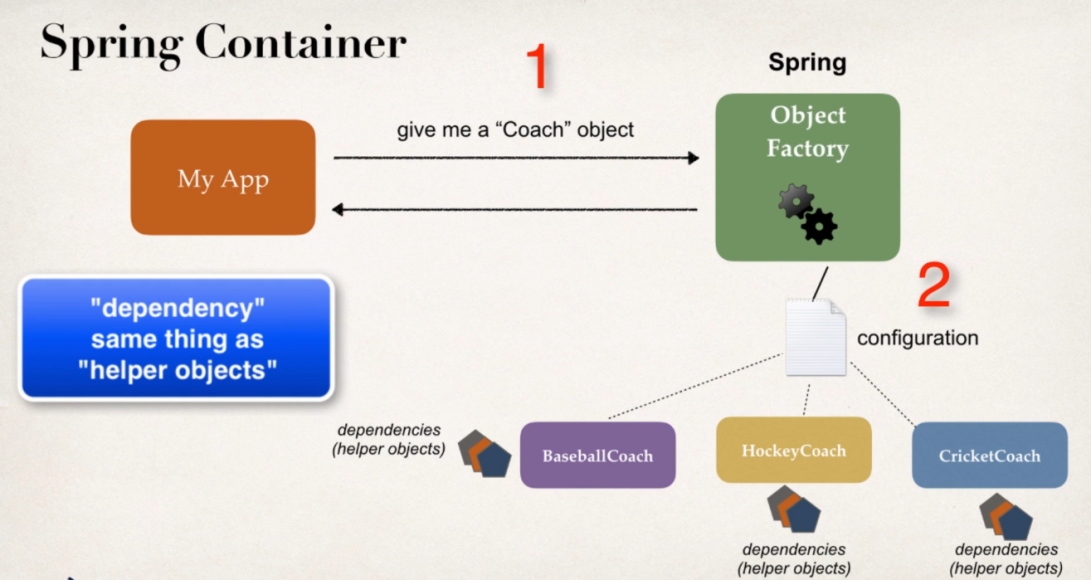
> Client is going to buy a car & this car is built at the factory on demand. So you have to actually talk to factory & put in a request & they will build for you.

> 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.***