**Spring / Hibernate**

**why Spring:**

**------------**

**\* Building JAVA enterprise applications**

**\* simple and lightweight alternative to J2EE**

**what is J2EE?**

**-------------**

**client side --> server side --> service business logic --> database**

**(contains JSP, servlets, EJB, JMS, web services, JSF, JAXB, JAX-WS, sockets)**

**Release Timeline:**

**-----------------**

**J2EE version 1.2 to 8**

**(1999)**

**Spring version 1.0 to 4.3**

**(2004-)**

**Goals of Spring core framework:**

**-------------------------------**

**\* Lightweight development with JAVA POJO (Plain old java objects)**

**\* Dependency injection to promote loose coupling**

**\* Declarative programming using AOP (Aspect oriented programming)**

**(adding application wide services to your objects)**

**\* Minimize java code.**

**Spring core container:**

**\* Beans**

**\* Core**

**-- factory for creating beans (BeanFactory) and manage bean dependencies**

**\* SpEL -- Spring Expression Language (used in config files to refer other beans)**

**\* Context -- holds the beans in the memory**

**Spring Infrastructure:**

**\* AOP -- Logging, security, transactions etc. through annotations or configs**

**\* Aspects**

**\* Instrumentation -- java agents to remotely monitor your app with JMX (java mgmt extn) we use agents provided by the spring team and web server.**

**\* Messaging**

**Spring Data Access Layer: (communicating with the database (RDBMS or NoSQL)**

* **JDBC -- contains JDBC helper class, reduces 50% of JDBC code**
* **ORM -- Object relational mapping (integration with Hibernate and JPA)**
* **Transactions -- add transaction support (make use of heavy AOP behind the scenes)**
* **OXM**
* **JMS -- Java messaging services (for sending async messages to a message broker (Queue)) Spring provides helper classes for JMS**

**Spring Web Layer: (spring MVC framework -- core / controller / view)**

* **Servlet**
* **WebSocket**
* **Web -- external client calls**
* **Portlet**

**Spring Test Layer: (supports TDD - test driven development)**

* **Unit**
* **Integration**
* **Mock -- MOCK objects for mocking out servlets and JNDI access**

**Spring Projects:**

**----------------**

**Spring modules built on top of the core module**

* **Spring Security**
* **Spring batch**
* **Spring boot**
* **Spring Cloud**
* **Spring webflow**

**First Spring Application:**

**-------------------------**

* **Create a project**
* **Download Spring – common logging jars**

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

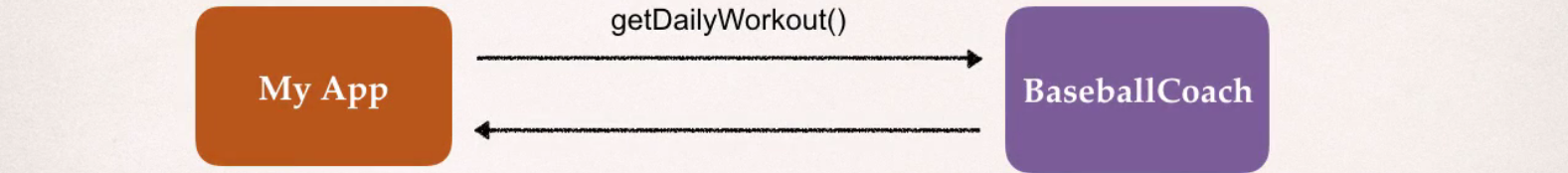
<https://commons.apache.org/proper/commons-logging/download_logging.cgi>

* **Build path 🡪 Add JARS 🡪 It gets downloaded into referenced libraries**

**Spring Inversion of Control (IoC)**

**---------------------------------**

* **Construction and management of objects.**



* **App should be configurable, easily change the coach for another sport**
* **Keys things to build this myApp:**
  + **MyApp.java -- main method**
  + **BaseballCoach.java**
  + **Coach.java -- interface (best S/W engg practices)**
  + **TrackCoach.java**

**MyApp.java**

Coach theCoach = **new** BaseballCoach();

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

Coach theCoach2 = **new** TrackCoach();

System.***out***.println(theCoach2.getDailyWorkout());

**BaseballCoach.java:**

**public** **class** BaseballCoach **implements** Coach{

**public** String getDailyWorkout()

{

**return** "BaseballCoach: practice batting for 30 minutes....";

}

}

**Coach.java:**

**public** **interface** Coach {

**public** String getDailyWorkout();

}

**TrackCoach.java:**

**public** **class** TrackCoach **implements** Coach {

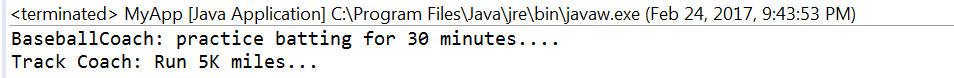
@Override

**public** String getDailyWorkout() {

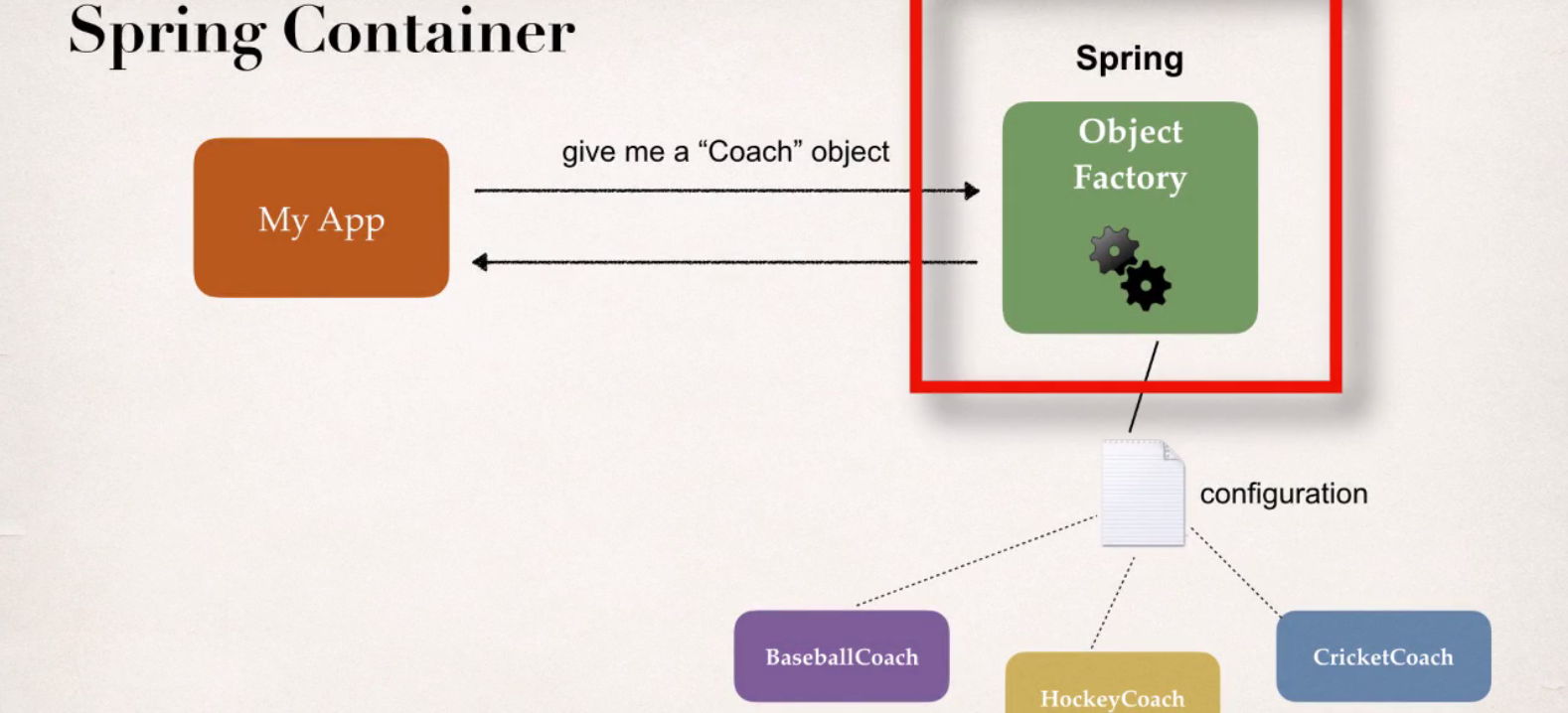
**return** "Track Coach: Run 5K miles...";

}

}



**// we are hard coding “BaseballCoach and TrackCoach” here, let’s bring Spring**



**Spring Container:**

* **IOC – creating and managing objects**
* **Dependency Injection – inject object dependencies**

**Configuring Spring Container:**

* **XML (legacy apps)**
* **JAVA annotations**
* **JAVA source code**

**Spring Development process:**

* **Configuring your spring beans**
* **Create a spring container**
* **Retrieve beans from spring container**

**Configuring your spring bean:**

**applicationContext.xml (place in src folder)**

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<bean id=*"baseballCoach"* class=*"com.test.spring.BaseballCoach"*>

</bean>

</beans>

**Create a spring container:**

* **Spring container = ApplicationContext**
  + **ClassPathXmlApplicationContext**
  + **AnnotationconfigApplicationContext**
  + **GenericWebApplicationContext….**

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

**Retrieve beans from spring container:**

Scanner scan = **new** Scanner(System.***in***);

System.***out***.println("Enter game name (baseballCoach or trackCoach): ");

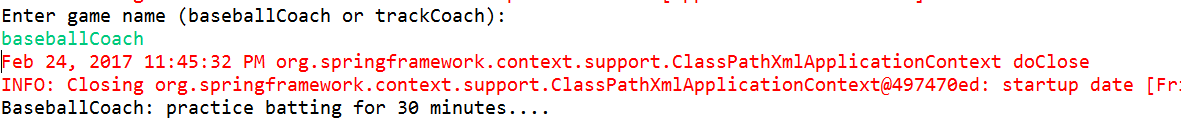
String name = scan.next();

Coach theCoach = context.getBean("baseballCoach", Coach.**class**);

// checks in configuration "baseballCoach" and interface of "baseball" class. (When we pass the interface to the method, behind the scenes Spring will cast the object for you.)

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

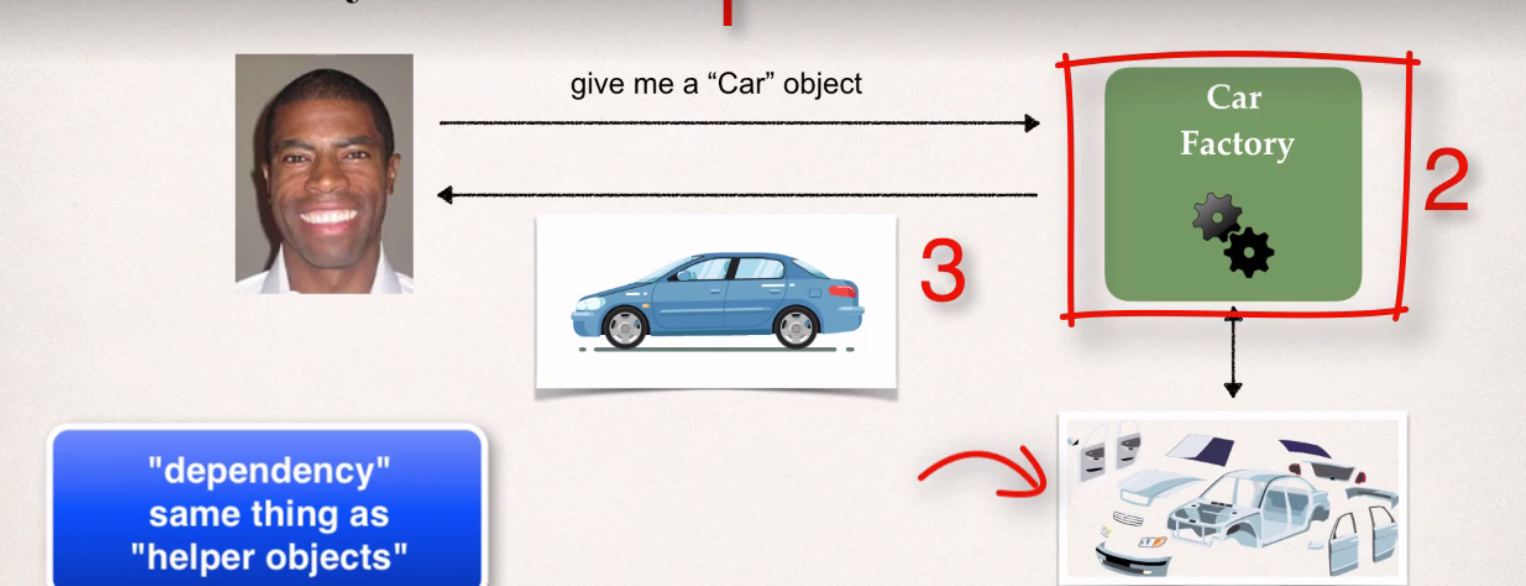
context.close();



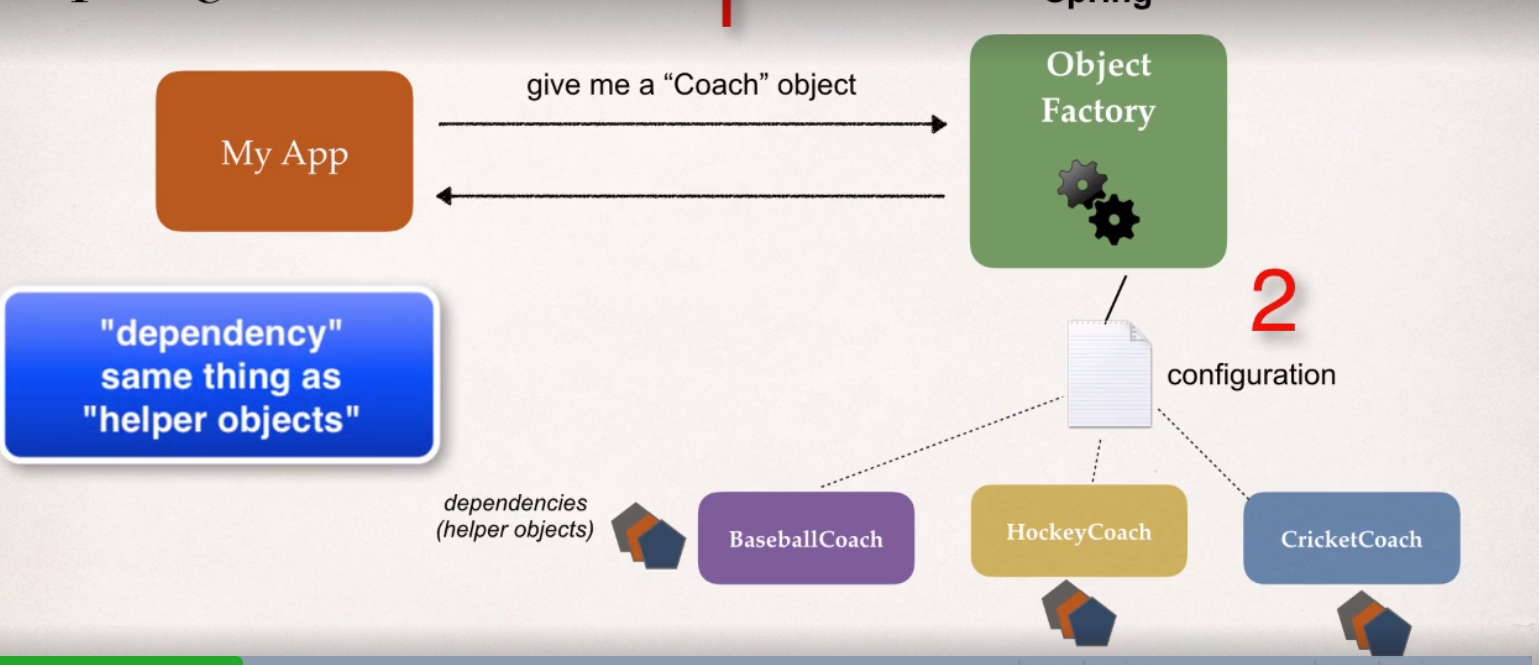
**Spring Dependency injection:**

* **Car factory 🡪 build a car 🡪 tires, doors etc. are been injected**

**This is called dependency injection.**



* **These coach object might have some dependencies.**



**Coding: Extra feature added to coach 🡪 provide Fortune Service. (dependency now)**

**Injection Types:**

* **Constructor injection**
* **Setter injection**

**Constructor injection (Development process):**

* **Define the dependency interface and class**
* **Create a constructor in your class for injections**
* **Configure the dependency injection in spring config file**

**Define the dependency interface and class:**

**public** **interface** Fortune {

**public** String getFortune();

}

**public** **class** FortuneService **implements** Fortune{

@Override

**public** String getFortune() {

**return** "Today you will enjoy in work...."; // read from file system, DB, web service

}

}

**Create a constructor in your class for injections:**

* **Injecting a dependency by calling in constructor**

**private** Fortune fortune;

**public** BaseballCoach(Fortune fortune)

{

**this**.fortune=fortune;

}

**public** String getDailyFortune() //dependency

{

**return** fortune.getFortune();

}

**Configure the dependency injection in spring config file:**

**public** **interface** Coach {

**public** String getDailyWorkout();

**public** String getDailyFortune();

}

<bean id=*"baseballCoach"* class=*"com.test.spring.constructorinjection.BaseballCoach"*>

<constructor-arg ref=*"fortune"* /> <!-- inject the dependency "fortune" -->

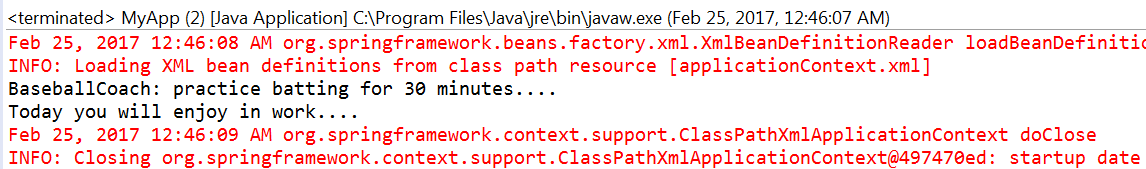
</bean>

<bean id=*"fortune"* class=*"com.test.spring.constructorinjection.FortuneService"* />

Coach theCoach = context.getBean("baseballCoach", Coach.**class**);

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

context.close();



**Setter injection:**

* **Inject dependencies by calling setter method on your class**

**Steps:**

* **Creating a setter method in your class for injections**
* **Configure the dependency injection in spring config file**

**Creating a setter method in your class for injections:**

**private** Fortune fortune;

**public** **void** setFortune(Fortune fortune) {

**this**.fortune = fortune;

}

@Override

**public** String getFortuneDetails() {

**return** fortune.getFortune();

}

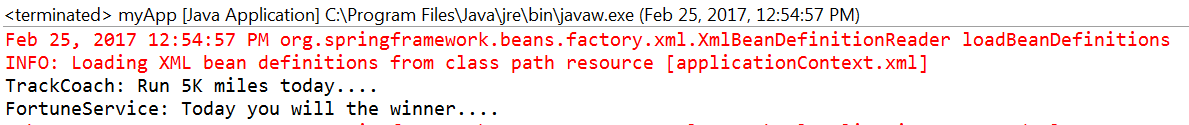
**Configure the dependency injection in spring config file:**

<bean id=*"trackCoach"* class=*"com.test.spring.TrackCoach"*>

<property name=*"fortune"* ref=*"fortune"*></property> <!-- setter injection -->

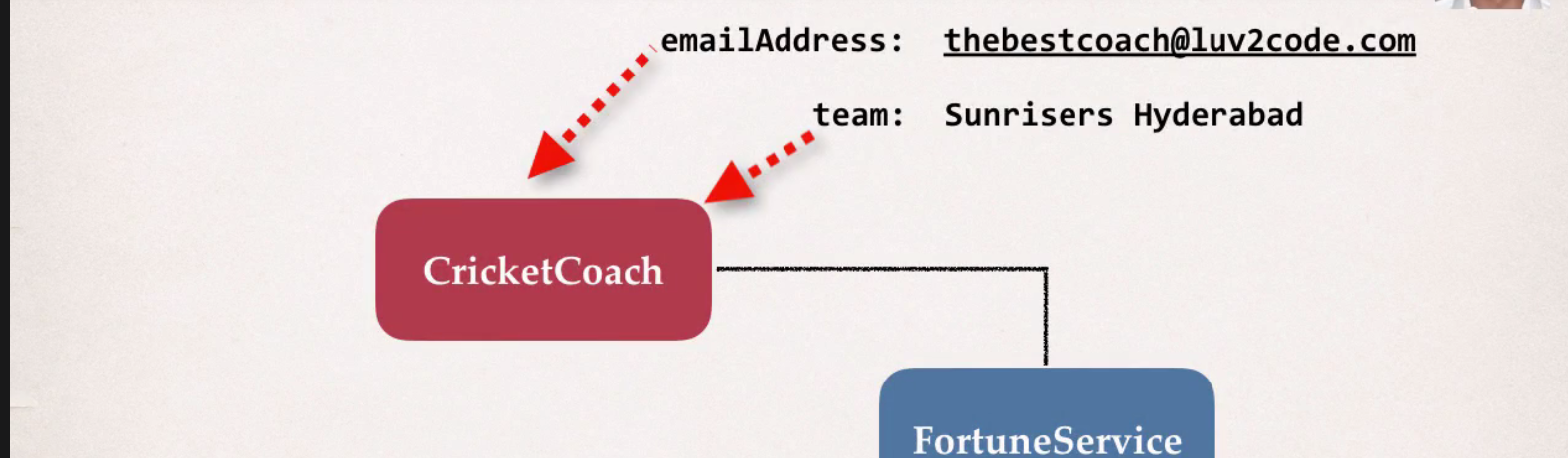
</bean>

<bean id=*"fortune"* class=*"com.test.spring.FortuneService"* />



**Injecting Literal Values:**

* **Adding some values into our spring objects.**



**Create a setter method for those private fields:**

**public** **class** BaseballCoach **implements** Coach {

**private** String email;

**private** String team;

**public** **void** setEmail(String email) {

**this**.email = email;

}

**public** **void** setTeam(String team) {

**this**.team = team;

}

**public** String getEmail() {

**return** email;

}

**public** String getTeam() {

**return** team;

}

**Configure those fields in the spring config file:**

<bean id=*"baseballCoach"* class=*"com.test.spring.BaseballCoach"*>

<property name=*"email"* value=*"yunus@gmail.com"* /> <!-- literal value injection -->

<property name=*"team"* value=*"Chennai super kings"* />

</bean>

**Injecting values from properties file: (XML)**

* **This can help in avoiding hardcoded values in config file**
* **Read values from properties file**
* **Steps:**
  + **Create a properties file**
  + **Load properties file in spring config file**
  + **Reference values from properties file**

**Create a properties file: (src folder)**

coach.email=yunusirshad@yahoo.com

coach.team=Chennai Super kings

**Load properties file in spring config file:**

<context:property-placeholder location=*"classpath:coach.properties"*/>

**Reference values from properties file:**

<bean id=*"baseballCoach"* class=*"com.test.spring.BaseballCoach"*>

<property name=*"email"* value=*"${coach.email}"* /> <!-- ${prop name} -->

<property name=*"team"* value=*"${coach.team}"* />

</bean>

**Bean Scopes:**

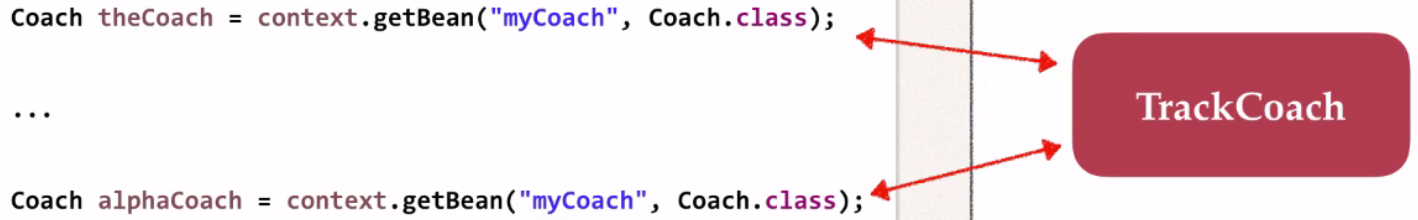
* **Scopes = lifecycle of a bean**
  + **How long it will live?**
  + **How many instances are created?**
  + **How is the bean shared?**

**Types of Bean Scopes:**

* **Singleton (default)**
* **Prototype -- creates a new bean instance for every container request.**
* **Request -- scoped to a HTTP request. (web apps)**
* **Session -- scoped to a HTTP session. (web apps) shopping cart.**
* **global-session –- scoped to a global HTTP web session. (web apps)**

**Singleton Bean Scope: (stateless bean)**

* **Spring container creates only one instance of the bean**
* **It is cached in main memory**
* **All requests of the bean will return shared reference to same bean. (like alphaCoach)**



<bean id=*"coachBean"* class=*"com.test.spring.BaseballCoach"* scope=*"singleton"*/>

Coach theCoach = context.getBean("coachBean", BaseballCoach.**class**);

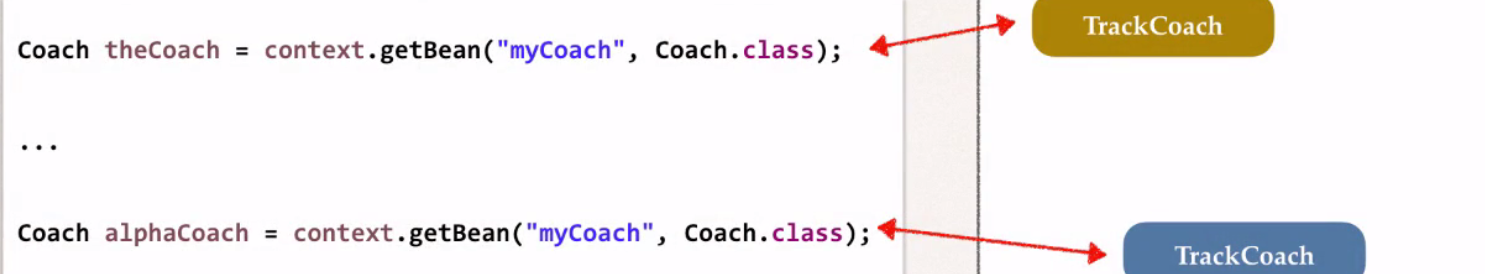
Coach theCoach2 = context.getBean("coachBean", BaseballCoach.**class**);

System.***out***.println(theCoach.hashCode()); //1620303253

System.***out***.println(theCoach2.hashCode()); //1620303253

**Prototype Bean Scope:**

* **New object for each request.**



<bean id=*"coachBean"* class=*"com.test.spring.BaseballCoach"* scope=*"prototype"*/>

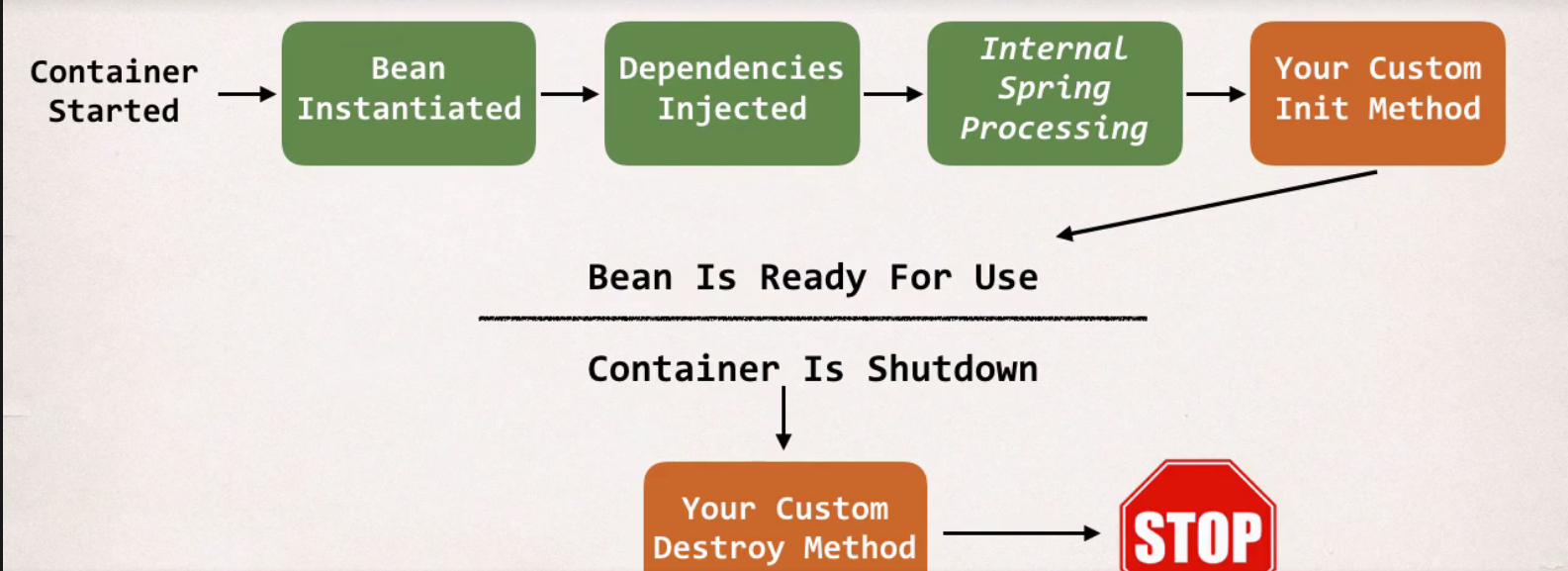
System.***out***.println(theCoach.hashCode()); //1547425104

System.***out***.println(theCoach2.hashCode()); //152134087

**boolean** result = (theCoach == theCoach2);

System.***out***.println(result); // false memory location

**Bean Lifecycle methods:**



**Container is Shutdown = context.close();**

**Bean initialization:**

* **You can add custom code during bean initialization.**
* **Setting up handles to resources (db, sockets, file etc)**
* **This method name must be “public void” method and contains NO argument**

**Bean destruction:**

* **You can add custom code during bean destruction**
* **Cleaning up handles to resources (db, sockets, file etc)**
* **This method name must be “public void” method and contains NO argument.**
* **In scope “prototype”, destruction are not called.**

**Steps:**

* **Define your methods for init and destroy**

**public** **void** insertCustomLogic()

{

System.***out***.println("BaseballCoach: Insert some custom logic through init()method....");

}

**public** **void** destructCustomLogic()

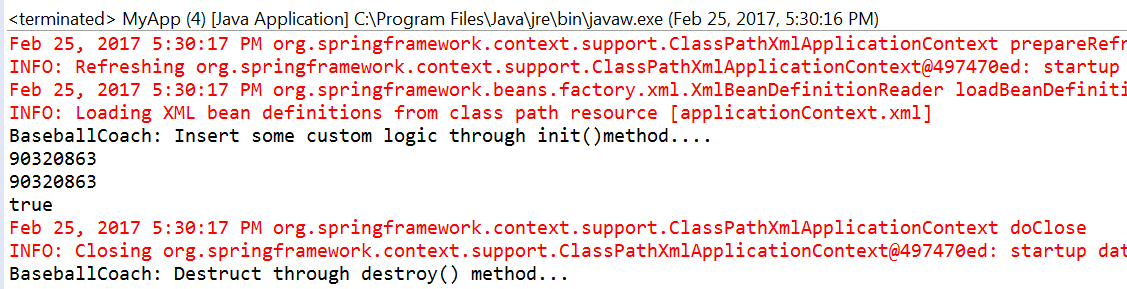
{

System.***out***.println("BaseballCoach: Destruct through destroy() method...");

}

* **Configure the methods into spring config file.**

<bean id=*"coachBean"* class=*"com.test.spring.BaseballCoach"* init-method=*"insertCustomLogic"* destroy-method=*"destructCustomLogic"* scope=*"singleton"*/>



**Spring JAVA Annotations:**

* **Special labels added to JAVA classes**
* **Provide meta-data about the class (eg. Shoe box contains meta-data “size… color… model.”)**
* **Processed at runtime or compile time for special processing.**

**@Override = telling compiler that we are overriding a method from parent class or interface.**

**Why Spring needs annotations?**

* **XML configuration is verbose (if we have 100 beans like that…)**
* **Configure your spring beans with annotations**
* **Annotations minimizes XML configuration**

**Scanning of Component classes:**

* **Spring will scan all JAVA classes in search of annotations.**
* **If find, then automatically register into spring container.**

**Development process:**

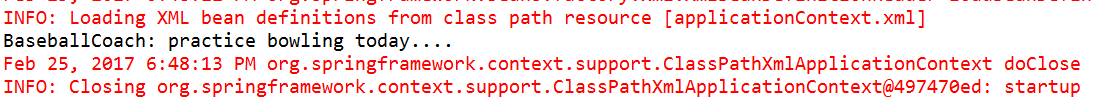
* **Enable component scanning in config file. (Spring scan recursively)**
* **Add @Component annotation in JAVA class (declare spring bean)**
* **Retrieve your bean from spring container.**

<context:component-scan base-package=*"com.test.spring"* />

@Component("coachBean")

**public** **class** BaseballCoach **implements** Coach {

Coach theCoach = context.getBean("coachBean",BaseballCoach.**class**);



**Default component names:**

* **It is good practice to use default component names**

@Component // it will take default bean id ("baseballCoach")

**public** **class** BaseballCoach **implements** Coach {

* **special case of when BOTH the first and second characters of the class name are upper case, then the name is NOT converted.**

**RESTFortuneService --> RESTFortuneService**

* **No conversion since the first two characters are upper case.**
* **Behind the scenes, Spring uses the Java Beans Introspector to generate the default bean name.**
* **In this case, we can give explicit name (“restFortuneService”)**

**Spring Dependency injection (Autowiring using Annotations)**

**Auto Wiring:**

* **Spring will match up the objects together**
* **Spring will look for a class that matches the property**
  + **Matches by type: class or interface**
* **Spring will inject it automatically…. Hence it is autowired.**
* **Spring supports multiple @autowire methods**

**Steps:**

* + **Create a dependency interface and class.**
  + **Inject the dependency into a class**
  + **Spring will scan components**
  + **Any one implements?**
  + **Once Spring finds it, then automatically inject into our class.**

**Auto Wiring Implementation Types:**

* **Constructor**
* **Setter**
* **Field**

**Auto Wiring (Constructor Dependency Injection):**

**Steps:**

* **Define dependency interface and class**
* **Create a constructor in your class for injections**
* **Configure dependency injection with @Autowired annotation.**

@Component

**public** **class** FortuneService **implements** Fortune {

**public** String getFortune() {

**return** "FortuneService: Today you will the winner....";

}

@Component

**public** **class** TrackCoach **implements** Coach {

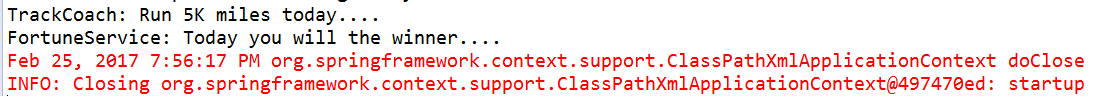
**private** FortuneService FS;

@Autowired

**public** TrackCoach(FortuneService FS) {

**this**.FS=FS;

}



**Auto Wiring: (Setter Injection)**

**Steps:**

* **Create setter method in your class for injections.**
* **Configure the dependency injection with @Autowired annotation.**

@Component

**public** **class** TrackCoach **implements** Coach {

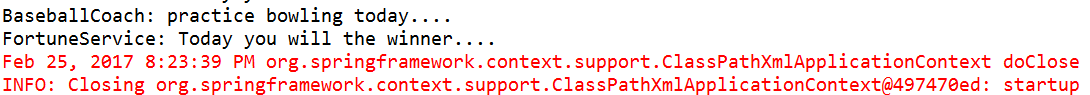
**private** FortuneService FS;

@Autowired

**public** TrackCoach(FortuneService FS) {

**this**.FS=FS;

}



**Auto Wiring Method Injection:**

* **You can inject any method.**
* **Some sort of input we need to get, so connection is required.**

@Component

**public** **class** BaseballCoach **implements** Coach {

**private** FortuneService FS;

/\* @Autowired

public void setFS(FortuneService fS) {

FS = fS;

}\*/

@Autowired

**public** **void** methodInjectionFS(FortuneService FS) {

**this**.FS=FS;

}

**Auto Wiring (Field Injection):**

* **Using this you can directly inject dependencies on the fields even though it is private.**
* **It is accomplished by using JAVA reflection**
* **No need of constructor or setter injection.**

@Autowired

**private** FortuneService FS;

**Annotation Auto Wiring and Qualifiers:**

* **Anyone implements FortuneService interface?**
  + **Multiple implementations from one interface**
* **Which one to pick?**
  + **We will get NoUniqueBeanDefinitionException**

**Logs 🡪 we expected single match but we found 4.**

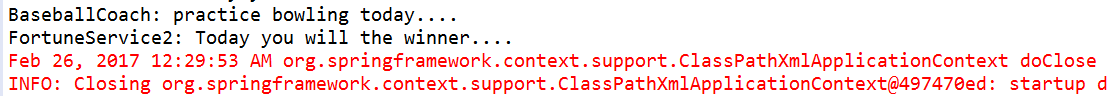
@Component

**public** **class** BaseballCoach **implements** Coach {

@Autowired

@Qualifier("fortuneService2") // declare bean id

**private** Fortune FS; // use interface to declare a field



**Constructor injection:**

@Autowired

**public** TrackCoach(@Qualifier("fortuneService") Fortune FS) { //constructor injection

**this**.FS=FS;

}

**Annotations for bean initialization and destruction:**

// define my init method

@PostConstruct

**public** **void** doMyStartupStuff() {

System.***out***.println(">> TrackCoach: inside of doMyStartupStuff()");

}

// define my destroy method

@PreDestroy

**public** **void** doMyCleanupStuff() {

System.***out***.println(">> TrackCoach: inside of doMyCleanupStuff()");

}

**Injecting values from properties file: (Annotation)**

* **Configure spring config file for loading properties file**
* **Declare @Value on top of the field.**

<context:property-placeholder location=*"classpath:coach.properties"*/>

</beans>

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

**private** String team;

**public** String getTeam() {

**return** team;

}

**public** **void** setTeam(String team) {

**this**.team = team;

}

