**Inversion of Control(Controlul Inversarii)**

**Proces de proiectare a construirii si administrarii obiectelor.**

**Acest proces prevede ca procesul de creare a obiectelor este dat unui Container, de aceea si e inversare, se inverseaza rolurile.**

Deci, aplicatia noastra va externaliza crearea si gestionarea obiectelor.

Aceasta externalizare va fi controlata de o fabrica de obiecte, asta si e rolul controlului inversarii.

De exemplu, unde avem nevoie de ea:

Fie ca avem o aplicatie de antrenament.



Aplicatie Antrenor(robot)



Noi spunem ceva aplicatiei, gen ce exercitii ar trebui sa fac saptamanal ca sa ajung un Hockeist bun si ea sa ne raspunda, insa problema e ca aplicatia trebuie cumva sa fie configurabila, si sa functioneze cu orice antrenor si treaca usor de la un antrenor la altul, caci mai pot fi si de football, de voleyball, inot si tot asa.

Sa zicem ca avem:

Interface Coach{

String getDialyWork();

}

Class RunningCoach implements Coach{

@Override

String getDialyWork(){

Return “Alearga 5km”;

}

}

Class SwimmingCoach implements Coach{

@Override

String getDialyWork(){

Return “Inoata 1km”;

}

}

In main, putem crea de exemplu:

Coach antrenor = new SwimmingCoach();

System.out.println(antrenor.getDialyWork());

Totusi, aplicatia noastra nu e configurabila, caci nu putem sa administram obiectele dintr-o sursa sau un config extern, ci trebuie mereu sa le cream si administram anume in metoda Main sau inca sunt create, si asta e foarte incomod.

**Anume aici ne ajuta Spring. Spring are un Container, si creaza o Configuration File, iar in Container va crea si stoca toate obiectele, fiind ca o fabrica de obiecte,unde le creaza,le pastreaza si noi le vom putea utiliza de acolo. Ele sunt create anume pe baza la Configuration File. Cand vom cere sa se creeze un obiect, Spring ne va da cea mai apropiata implementare.**

**Spring Container**

**Container este Core a lui Spring**

* Functii primare:
* Creaza si administreaza obiecte(**Inversion of Control)**
* Injecteaza dependentele obiectului(**Dependency Injection)**
* Metode de a configura Spring Container:
* Folosind XML configuration file(este **Configuration File** de origine, adica cu care a fost creat Spring)
* Java Annotations
* Java source code

**Odata ce rulam programul, Fabrica de obiecte din Spring Container va executa toate instructiunile din fisierul cofig XML cand se creaza obiectul context si va crea toate obiectele, va face injectarile etc. fara ca noi sa scoatem vreun bean! El pastreaza toate obiectele create, pe baza la XML in Container si asa nu trebuie sa tot trimitem obiectele prin argumente.**

**Spring Development Process**

1. Configurarea Spring Beans
2. Crearea unui Spring Container
3. Recuperam Beans din String Container

Spring Bean este un simplu obiect in Java. Cand obiectele java sunt create de Spring Container, Spring le trateaza sau se refera la ele ca Spring Beans. **Spring Beans sunt obiecte Java simple(ca orice obiect obisnuit) dar ce nu sunt create de noi, ci sunt administrate,instantiate si asamblate de Spring Container pe baza fisierului config. Beans sunt create anume de Container!**

* **Pasul 1:**

Fisierul cu Beans este applicationContext.xml

<beans>

<bean id=”ID” class=”Numele clasei in Java>

</bean>

</beans>

* Fiecare bean creat va fi de tip clasa din class. La class=”” se pune adresa clasei de tipul la care vrem sa cream obiecte. Nu adresa unui obiect, ci a unei clase, caci acest tagul <bean> e facut sa creeze obiecte, trebuie doar sa specificam ce tip de obiecte in class=””;
* ID e pentru a putea folosi anume acest tag, caci s-ar putea sa mai avem si alte taguri. Spring va crea un obiect si numelel lui va fi acel ID
* **Pasul 2- Crearea unui Spring Container pe baza la config File**

Spring Container e cunoscut ca applicationContext, dar poate avea si alt nume si putem avea oricate dorim.

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

Astfel, cream un obiect pentru un contextApplication, deja existent in Path, adica pentru un Configuration File de tip XML deja existent, si acesta va fi Containerul.

**Anume httpServlet, daca are configurata ca applicationContext acest config, se va asigura de crearea Containerului si a beanurilor din el. Deci, putem sa nu scriem absolut nimic cod, oricum beanurile sunt deja create cand ruleaza applicatia. Daca nu exista httpServlet, se vor crea cand acest applicationContext va fi creat ca un object de tip ClassPatXmlApplicationContext.**

* **Pasul 3**

Cerem lui Spring sa ne dea un obiect din Container. Obiectul dat e bazat pe configuration Files. Pentru a extrage un obiect din Container, facem asa:

Coach obiect = context.getBean(“ID”,Coach.class);

Coach.class – returneaza tipul si numele, adica va returna **interface Coach.**

**getBean returneaza un obiect de tip Object, si al doilea parametru ne arata in ce va fi facut downcastigul. Adica, codul de sus e echivalent cu**

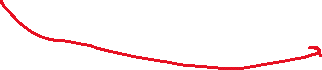
Coach obiect = (Coach)context.getBean(“ID”);

Doar ca metoda getBean poate arunca exceptii in caz ca castingul facut e Ea nu merge bine.

**Cum cream un bean si il folosim**

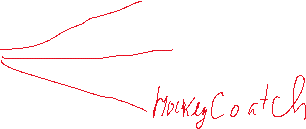
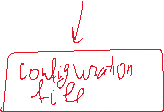
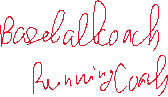
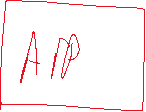
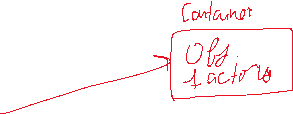
* Cu Containerul putem crea obiecte. Dupa ce am creat obiectul ce se refera la contextApplication.xml, il folosim ca sa retragem beanuri:

Coach coach = context.getBean(“ID”,Coach.class);



Dupa asta, se recupereaza un obiect de tip RunningCoach, si de crearea lui s-a ocupat contaierul, de asta obiectul e bean.

* Folosim context.close() pentru a inchide Containerul.



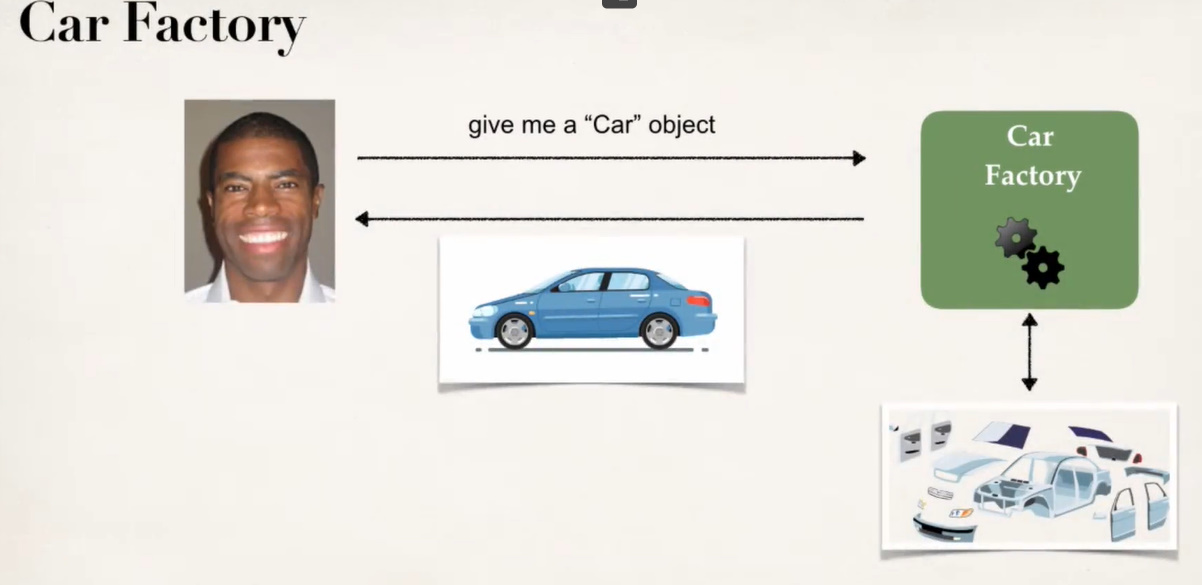
Un mare avantaj al acestui sistem din Spring e ca putem avea obiecte ce sunt create si gestionate de Container. Noi am scris context.getBean, si pe baza la id trimis ca prim parametru, Spring va decide ce obiect sa returneze din cele create. Ideea e ca putem modifica acest lucru din contextApplication, si desi in Main,sursa, nu modificam nimic, se va crea un alt obiect, in dependenta de ce scrie la class=”” in <bean>. Asa modificam doar configuration File(XML)

**Dependency Injection**

**The dependency inversion principle –** obiectele de care un alt obiect depinde(cele ce sunt membrii sai) sunt creati undeva in afara lui si trimisi apoi la el prin diferite metode.

Clientul delegheaza un alt obiect  **cu** responsabilitatea deasigura dependentele sale. De ex:

**Inject** – proces de a trimite obiectul creat catre obiectul ce depinde de el.

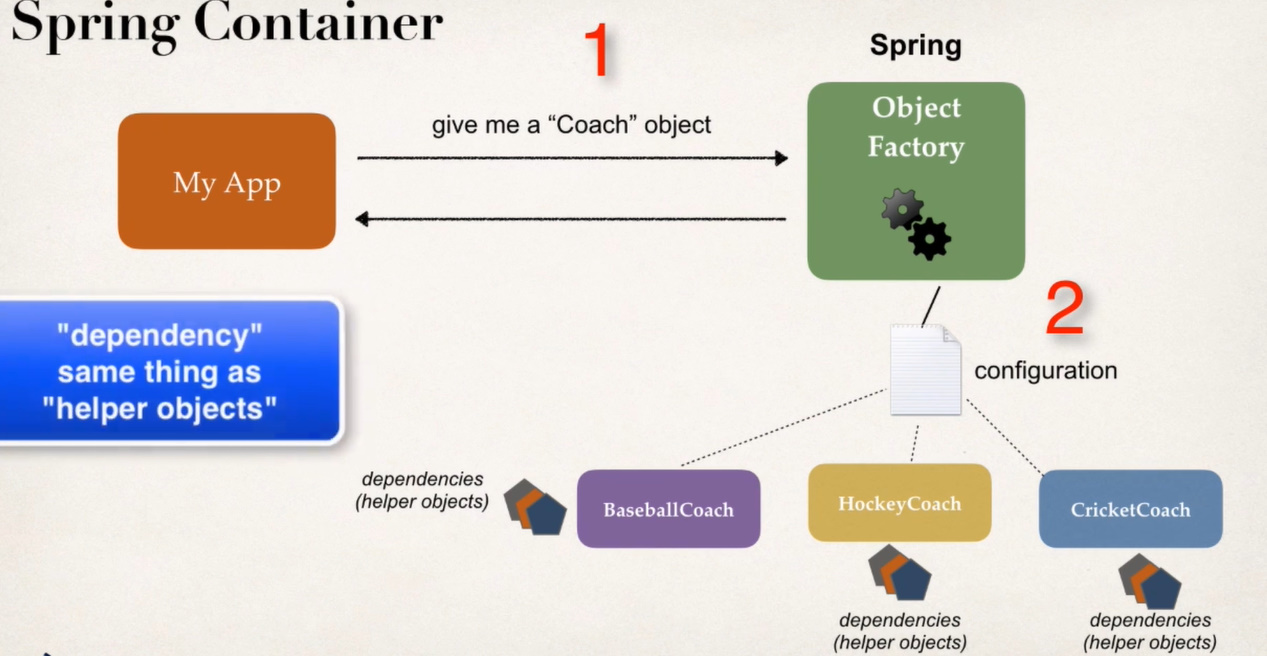


Clientul vrea sa cumpere o masina si aceasta e produsul Fabricii de masini,care creaza o masina.Fabrica va avea deci o comanda. La fabrica se gasesc toate piesele necesre pentru constructie. Fabrica va fi cea care va crea masina din bucati/piese si respectiv o va si asambla, dupa o va livra la client.Clientul nu trebuie sa o construiasca, ci Fabrica o face. **Fabrica injecteaza toate dependentele. Dependenta e acelasi lucru ca si Obiecte de ajutor sau Helper Objects. Ei injecteaza motorul, sacunele, anvelopele etc.**

**Deci, procesul de creare a motorului, usii, rotilor etc. de care depind masina, asta si e Dependency Injection. Procesul de livrare a acestor parti masinii e Injection.**

Asta si inseamna injectarea dependentei – pur si simplu externalizam constructia si injectia obiectului unei alte entitati.

**Cum functioneaza in Spring**



In Containerul Spring se gaseste o fabrica de obiecte. Cand extragem de la el un obiect de tip Coach, cu getBean, el poate avea unele dependente(obiecte) aditionale, ce nu ar fi in mod normal cand cream noi obiectul cu new. Aceste dependente sau obiecte pe care el le mai are sunt obiecte ajutatoare, obiecte de care are nevoie pentru a-si desfasura activitatea. Deci nu trebuie sa cream obiectul Coach si toate obiectele sau dependentele singuri, ci Spring o face pentru noi.

Obiectul coach deja ofera un daily workouts. Sa zicem ca el va oferi si un Fortune Service care va oferi daily fortune, sau un fel de recompense. Avem:

* New helper: Fortune Service
* Aceasta e o dependency - deoarece coach depinde acum de un Fortune Service

**Helperi sunt obiectele membri din Clasa**

**Helper = dependency**

**Injection Type**

Cele mai cunoscute sunt:

* Constructor Injection
* Setter Injection
* field injection**(nu e posibil in XML)**

**Constructor Injection:**

1. Cream interfata si clasa de dependenta
2. Cream un constructor pentru dependenta
3. Configuram dependency injection in Spring config file.

Primul pas

interface FortuneService{  
 String getFortune();  
}  
class HappyFortuneService implements FortuneService{  
 @Override  
 public String getFortune(){  
 return "Today is your lucky day!";  
 }  
}

Pasul2

class SwimmingCoach implements Coach{  
 private FortuneService fortuneService;  
  
 public SwimmingCoach(FortuneService fortuneService){  
 this.fortuneService = fortuneService;  
 }  
 @Override  
 public String getDailyWork() {  
 return "Swim daily";  
 }  
}

Acesta e un constructor pentru Injection. Anume in acest constructor injectam dependente.

Pasul3

<bean id="myFortuneService" class="com.example.spring.HappyFortuneService"/>  
<bean id="Swim" class="com.example.spring.SwimmingCoach">  
 <constructor-arg ref="myFortuneService"/>  
</bean>

Aici adaugaum o noua bean, si anume clasa HappyFortuneService si apoi specificam ca Swim coach va avea un constructor care injecteaza dependenta in clasa, si o va face cu un obiect bean cu id myFortuneService. De asta si are numele constructor-arg, spunem ca constructorul va avea drept argument un bean cu id in config file myFortuneService si **va crea un obiect de acest tip dupa ce el va fi trimis constructorului.**

Spring va returna obiectul ce il dam noi in constructor ca parametru, si apoi il va injecta in Coach.

* Daca avem un constructor cu mai multe argumente, pentru fiecare argument, in oridnea aparitie lor, scriem <constructor-arg ref=””/>
* Putem adauga si index=”” pentru a arata ordinea parametrilor. index=”0” e primul parametru
* <bean id="teacher" class="com.java.project.Teacher">  
    
   <constructor-arg index="0" value=""/>  
   <constructor-arg index="1" value=""/>  
  </bean>

**Ce face Spring**

In mod normal, daca am vrea sa cream un obiect de tip SwimmingCoach, ar trebui sa facem asa:

HappyFortuneService fortuneService = new HappyFortuneService();  
SwimmingCoach swimmingCoach = new SwimmingCoach(fortuneService);

Insa, Spring va face tot acest lucru pentru noi:

1. Spring va crea obiectul fortuneService ce trebuie sa fie trecut in parametru. Numele sau va fi ID din bean
2. Va crea obiectul SwimmingCoach, cu numele sau ca ID din bean si va trimite direct in constructor obiectul fortuneService creat de el,cu numele sau tot din ID, si il va injecta in acest obiect, apoi ne trimite obiectul creat.

Deci, desi avem doar un singur constructor, care cere ca parametru un obiect ce trebuie sa fie setat in obiectul sau ajutator, nu va trebui noi sa cream unul si sa il trimitem constructorului. Cand scriem

Coach coach = context.getBean("Swim",Coach.class);

Containerul ne returneaza un obiect, dar inainte de asta a creat acel obiect Helper si apoi cand a creat obiectul de tip SwimmingCoach, a trimis automat acel obiect lui in constructor.

**One constructor only**

Putem sa avem si un singur constructor si acela sa fie cu un argument. Deci, nu e obligatoriu sa avem un constructor default(fara argumente), si daca e doar unul,cu oricate argumente, va trebuie sa specificam parametrii lui, adica id la beanuri ce sunt Helperi. Daca nu, Spring pur si simplu va cauta singur prin Container un bean care sa corespunda ca parametru, adica sa fie injectat.

De ex:

Fie clasa Student cu 2 constructors, unul default si unul nu

Fie clasa Teacher ce are un membru Student, dar numai un singur constructor, si acela are un argument de tip Student

private Student student;  
  
public Teacher(Student student) {  
 this.student = student;  
}

In app context putem face asa ceva:

<bean id="student" class="org.java.app.Student">  
 <constructor-arg value="1"/>  
 <constructor-arg value="Mititiuc"/>  
 <constructor-arg value="Eduard"/>  
 </bean>  
<bean id="teacher" class="org.java.app.Teacher"/>

Deci, beanului teacher nu i-am setat constructorul folosit si nici referinta la “student”, dar Spring oricum va lua acel constructor, care e unicul, ca e clar ca doar pe el il poate folosi, si va cauta beanuri pentru a le injecta. Iata de ce, totul va fi bine.

**Acum, ca asta sa functioneze, vom adauga <context:component-scan base-package=”...”/>. Daca nu, Spring nu va cauta prin Container beanuri potrivite pentru constructor, si vom avea eroare ca un constructor default nu exista.**

Daca avem mai multi constructori, trebuie sa specificam care e folosit fie in xml, fie sa punem la el @Autowired, caci desi @Autowired e automat pus, aici trebuie pus manual ca sa stie care constructor automat e folosit.

private Student student;  
  
@Autowired  
public Teacher(Student student) {  
 this.student = student;  
}  
  
public Teacher(int id, String firstName, String lastName, Student student) {  
 this.id = id;  
 this.firstName = firstName;  
 this.lastName = lastName;  
 this.student = student;  
}

Insa, daca nu vom pune @Autowired, Spring mereu va lua constructorul default. Daca un constructor default nu exista, dar mai sunt constructori, si nu specificam care il folosim, Spring da eroare, caci nu va sti pe care sa il foloseasca.

**SETTER Injection**

Spring injecteaza dependentele prin metodele setter din clasa.

Pasi:

1.Cream metoda setter

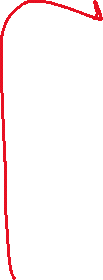
2.Configuram fisierul Containerului

1.Sa zicem ca avem o noua clasa, tot cu dependente, dar fara constructor

class CricketCoach implements Coach{  
 private FortuneService fortuneService;  
 @Override  
 public String getDailyWork() {  
 return null;  
 }  
 @Override  
 public String getDailyFortune() {  
 return null;  
 }  
 public void setFortuneService(FortuneService fortuneService){  
 this.fortuneService = fortuneService;  
 }  
}



2.In fisier la Container, adaugam bean ca deobicei, dar acum folosim tagul <property name=”nume” ref=”clasa”>

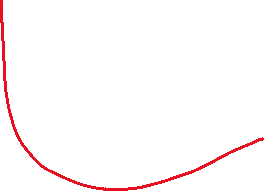


Numele metodei setter mereu are forma: setCe, adica set+Cuvant ce incepe cu Caps

Name=”nume metodata fara set”, adica aici scriem numele metodei setter, dar fara cuvantul set! Spring automat va cauta o metoda publica,cu void, ce incepe cu set + name din tag.

Nu e neaparat ca metoda scrisa fara set in Name=”” sa inceapa cu litera mare, oricum Spring capitalzieaza prima litera, insa metoda setter trebuie neaparat sa aiba dupa set o litera mare

<bean id="myFortune" class="com.example.udemy.HappyFortuneService"/>



<bean id="Cricket" class="com.example.udemy.CricketCoach">  
 <property name="fortuneService" ref="myFortune"/>  
</bean>



**Cricket.setFortuneservice(myFortune);**

Spring creaza un obiect de tip CricketCoach cu numele identic cu ID al bean:

CricketCoach Cricket = new CricketCoach();

Spring va folosi mai sus Constructorul default, daca exista, si cel default default daca cel default nu exista si altele nu sunt tot.

Spring creaza un obiect de injectat, iar cu numele ca la ID al beanului

HappyFortuneService myFortune = new HappyFortuneService();

Apoi apeleaza metoda setter:

Cricket.setFortuneService(myFortune);

**Dupa ce returnam bean cu getBean, nu apelam nicio metoda setter. Spring a facut automat asta si pentru toate metodele setter!**

**ID, Name, ref, class,value in tag**

* ID – va fi un nume al unui obiect. De aceea incepe mereu cu litera mica, ca si conventia de a defini variabile ce incep cu litere lower.
* Name – va fi o metoda cautata sau altceva cautat. Si bean poate avea name, si acesta va fi un nume secundar pentru el.
* Ref – e numele obiectului ce va fi trimis. De aceea corespunde cu ID de la alt bean
* Class – numele clasei folosite pentru a crea obiecte
* Value – este o valoare care o trimitem unui parametru de ex, in loc sa trimitem un alt obiect deja creat

**Injectarea valorilor literale**

S-ar putea sa avem nevoie sa injectam obiecte care au deja definite anumite valori in ele. De exemplu:

class CricketCoach implements Coach{  
 private String emailAdress;  
 private String team;

public void setEmailAdress(String emailAdress){  
 this.emailAdress = emailAdress;  
 }  
 public void setTeam(String team){  
 this.team=team;  
 }  
}

Aici vrem sa injectam niste stringuri ce au deja definite in ele valori. Am putea crea beanuri pentru aceste obiecte String si Spring le va trimite la setteri, dar ele vor avea valori nule.

Pentru a face acest lucru, cream un <property> tag, cu name=”” dar fara ref=””! Nu avem nevoie ca Spring sa ia vreun obiect concret pentru acest String, caci noi nici nu setam vreun bean penru un obiect String. Putem direct sa trimitem valoarea parametrului din functia setter prin **value=”Textul nostru”;**

De ex:

<bean id="Cricket" class="com.example.udemy.CricketCoach">  
 <property name="fortuneService" ref="myFortune"/>  
 <property name="emailAddress" value="edikutsu2002@mail.ru"/>  
 <property name="team" value="Team1"/>  
  
</bean>

E echivalent cu

Cricket.setEmailAdress(“[edikutsu2002@mail.ru](mailto:edikutsu2002@mail.ru)”);

Cricket.setTeam(“Tem1”);

**Injectarea Valorilor Literale prin Properties File**

E destul de incomod sa trimitem valorile din applicationContext, adica din Config File. De asta, putem crea un Properties File, unde sa stocam aceste valori.

Proces de dezvoltare a unui Properties File:

1. Cream Properties File
2. Il configuram in Spring config file
3. Referentiem valorile din properties file
4. Cream un fisier Nume.properties

Pentru a inregistra valori aici, facem asa:

atribut1 = valoare;

atribut2 = valoare;

Alegem ce nume vrem si atribute.

Putem sa nu punem “” la valorile String

1. Adaugam la inceput in tagurile <beans> </beans>:

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

1. Facem referinta la ele la value prin @Value(value=“${numeProprietate})”

Name=”foo.email”;

**Putem seta si valori integer. Chiar daca punem value=”${}”, oricum Spring va decide daca valoarea e un string sau nu, caci aceste “” nu sunt neaparat String. Spring face conversiile singur.**

**Cel mai bine e sa punem applicationContext si properties file in folderul resources creat de spring.**

* Putem folosi “${NumeProp}” si in XML la value=”${Prop}”:
* <context:component-scan base-package="org.test.app"/>  
  <context:property-placeholder location="classpath:values.properties"/>  
  <bean id="student" class="org.test.app.Student">  
  <property name="age" value="${age}"/>  
   <property name="firstName" value="${firstName}"/>  
   <property name="lastName" value="${lastName}"/>  
  </bean>

**Beans Scope**

* Beans scope se refera la ciclul de existenta a unui Bean.
* Scope se refera la ciclul de existenta, cat traieste un bean?Cate instante sunt create?Cum sunt ele shared?
* Default Scope pentru un bean este **Singleton**:

<bean id="myFortune" class="com.example.udemy.HappyFortuneService">

**Sigleton** inseamna,in software engeneering, ca o clasa poate avea o singura instanta sau un singur obiect la un anumit moment.In Spring asta inseamna ca Spring va crea un singur bean dupa modelul din XML care il va pastra in memorie si il va oferi oricui il cere. El se afla intr-un singur scope, sau namespace, si nu se creaza nicaieri altundeva! Spring doar da referinte la el cand se cere, nu il recreaza.

* Spring creaza doar o instanta a unui bean, care e pastrata in memorie, si orice cerere de a accesa acest bean va primi o refernta la el. Toata aplicatia va folosi anume acest bean, daca folosim anume numele lui din ID.

CricketCoach obj1 = context.getBean("Cricket",CricketCoach.class);  
CricketCoach obj2 = context.getBean("Cricket",CricketCoach.class);

Obj1 si obj2 vor primi referinta la acelasi obiect, adica Cricket, creat deja de spring Container pe baza la config file si pastrat in Container tot.

* La bean mai putem pune

scope="singleton"

dar e inutil, asa cum e deja default definit.

**Additional Spring Bean Scopes**

* **Singleton –** creaza o singura instanta a unui bean. E default.
* **Prototype –** creaza cate o instanta noua a unui bean la fiecare cerere.
* **Request –** are scope la o cerere HTTP.Deci, se creaza pentru ficare request. Pentru webapps
* **Session –** are scope la o HTTP web session. Se creaza cand o sesiune incepe la un user, si cand sesiunea e termina, e sters.Pentru webapps
* **Global-session –** are scop la sesiune HTTP globala. Pentru webapps

Pentru a modifica scopul,utilizam in <bean> atributul scope=””

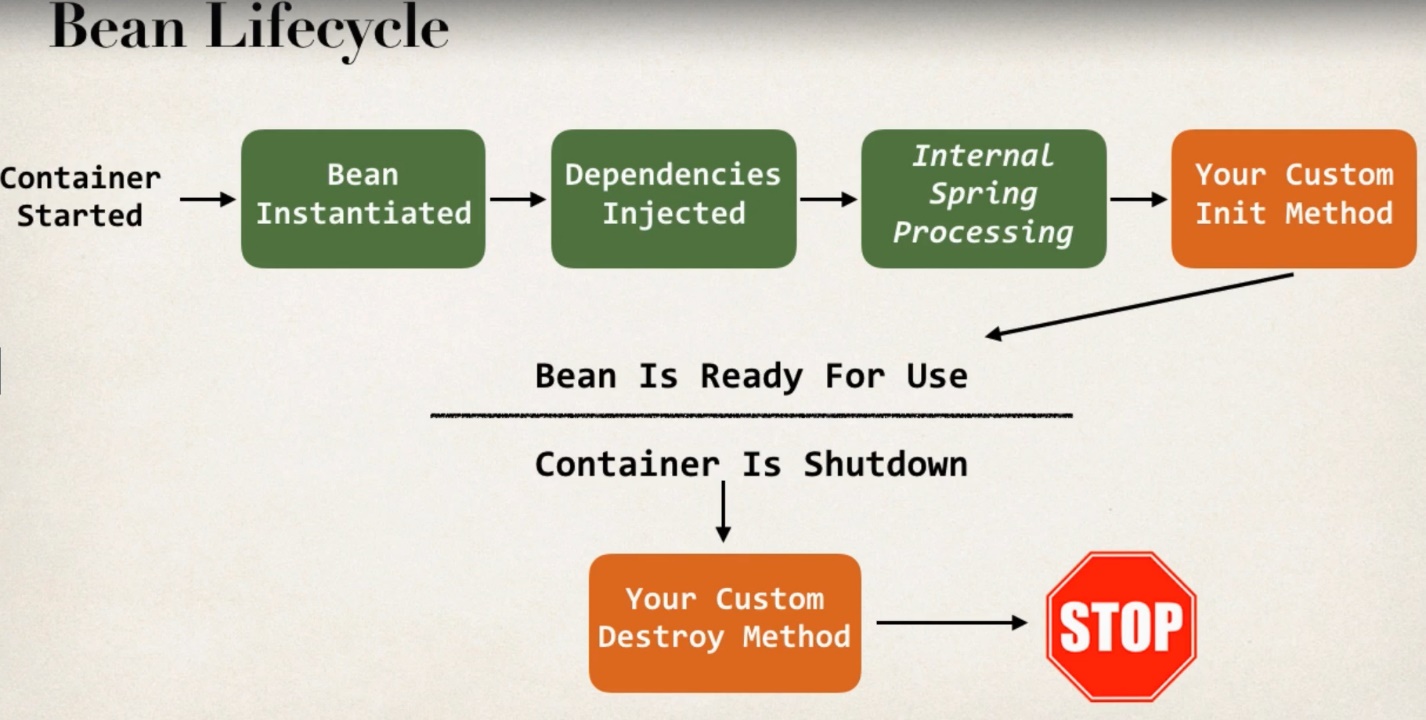
<bean id="myFortune" class="com.example.udemy.HappyFortuneService" scope="prototype">

CricketCoach obj1 = context.getBean("Cricket",CricketCoach.class);  
CricketCoach obj2 = context.getBean("Cricket",CricketCoach.class);

Obj1 si obj2 au referinta la obiecte diferite!

Daca beanul are ca scope prototype, Spring nu mai creaza din start un bean(obiect) de acest tip! El creaza chiar la rularea aplicatiei toate beanurile cu scope singleton, si le pastreaza in Container, dar nu va crea singur nici un obiect cu scope prototype, caci nu stie cate sa creze. Va crea apoi cate un obiect pentru fecare cerere de a returna un bean.

**Bean LifeCycle methods**



* Putem sa adaugam custom code care sa fie executat la momentul Bean Instantieted, adica metode care sa fie executate cand se creaza beanurile chiar.
* Putem sa adaugam Custom code care sa fie executat cand bean e distrus.

Custom code pe care Spring il executa cand bean e creat sau distrus--> Hooks

* Pentru a-i spune lui Spring ca trebuie sa execute o metoda la CREAREA unui bean, folosim atributul **init-method=”numeMetoda”**
* Pentru a-i spune lui Spring ca trebuie sa execute o metoda la DISTRUGEREA unui bean, folosim atributul **destroy-method=”numeMetoda”**
* Pot fi accesate orice tip de metode, public,protected,default sau si private
* Metoda poate avea orice return type, dar void e mai des folosit, asa cum oricum nu vom putea folosi ceea ce e returnat
* Metoda trebuie sa fie fara argumente!

**Metodele noastre custom sunt executate dupa crearea si injectarea!!!**

**Destroy-method nu e executat daca scope e prototype! Spring nu adminsitreaza complet ciclul de viata al unui bean cu scope prototype, deoarece posibil noi sa avem nevoie ca ele sa fie pastrate chiar si la incheierea executiei programului.**

* **Atentie! DestroyMethod e executata doar dupa ce folosim context.close(). Atunci Spring va sti ca containerul isi termina ciclul de existenta, sau cand dam Stop la aplicatie**

**Executarea unei metode la incheierea ciclului de existenta a unui bean de tip prototype scope**

Exista totusi o metoda prin care putem executa o anumita metoda la incheierea ciclului de existenta a unui bean.

1. Cream o clasa a carei instanta sa se ocupe de colectarea tuturor beanurilor noastre ce stim ca au prototype scope, si cand va fi stearsa, sa aiba o metoda ce va fi executata la final de lifecycle ce se va ocupa de apelarea tuturor metodelor destinate sa fie executate la stergerea beanurilor ce au prototype scope.

class Delete implements DisposableBean{  
 private LinkedList<Coach> lista = new LinkedList<>();  
  
 public void addBean(Coach bean){  
 lista.add(bean);  
 }  
 @Override  
 public void destroy() throws Exception {  
 for(Coach obj : lista){  
 System.*out*.println("In destroy");  
 obj.destroy();  
 }  
 }  
}

1. Cream un bean al acestei clase, si nu e necesar sa punem artributul destroy-method, caci Spring are grija sa execute metoda asta la tot beanurile a caror clasa implementeaza interfata DisposableBean

<bean id="delete" class="com.example.udemy.Delete" />

1. Adaugam beanurile ce stim ca au prototype scope in lista ce o oare obiectul pentru stergerea lor.

Coach theCoach1 = context.getBean("myCoach",Coach.class);  
Coach theCoach2 = context.getBean("myCoach",Coach.class);  
  
Delete delete = context.getBean("delete",Delete.class);  
delete.addBean(theCoach1);  
delete.addBean(theCoach2);

Beanurile care au scope singleton si implementeaza interfata DisposableBean, si respectiv suprascriu metoda destroy() pe care o are interfata, Spring se va asigura sa execute automat metoda destroy pentru ele, fara a mai pune atributul destroy-method. Iata de ce, putem si sa cream o lista List<DisposableBean> si sa adaugam beanurile noastre ce sunt prototype, si sa apelam metoda destroy() pentru fiecare. Evident, fiecare bean trebuie sa implemeneteze DisposableBean

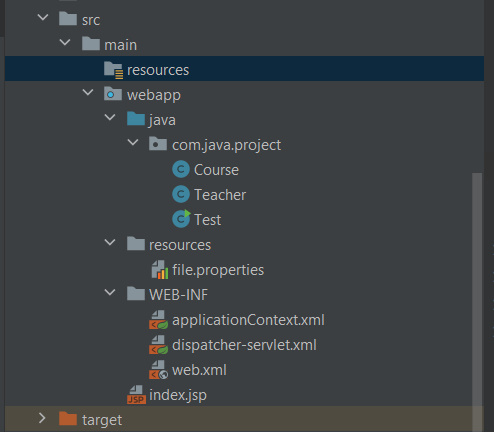
Putem sa nu oferim niciun nume pentru bean cand il extragem, caci daca exista doar unul de acel tip, Spring il va returna automat, dar daca exista mai multe, va da erroare. Scriem intre () doar tip.class

**Injectarea beanurilor cu scope prototype**

Spring poate injecta si beanuri ce au scope prototype. El pur si simplu va crea cate un nou obiect ce va fi injectat in bean, si atat. Cand vom scrie ID la bean ce vrem sa fie injectat, spring va crea un bean si il va injecta in Beanul sursa, apoi va pierde legatura cu acel bean injectat, asa cum face cu orice bean cu scope prototype.

**Resources Folder**

Daca avem o asa structura a proiectului:

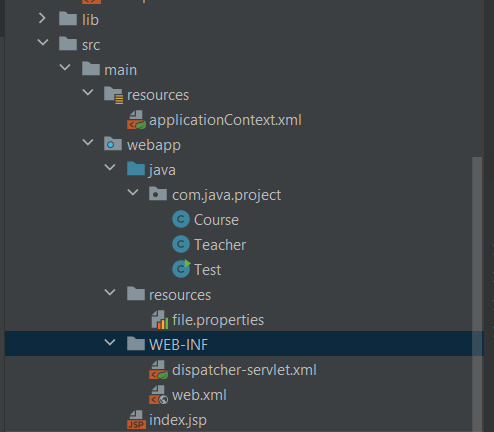


Unde applicationContext.xml se gaseste in WEB-INF, vom avea o problema. Fisierele Spring au acces la tot folderul webapp, insa clasele din source foolder nu. Ele au acces doar la package a lor. Deci, chiar de am scrie asa ceva la crearea obiectului:

ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("WEB-INF/applicationContext.xml");

am avea eroare, caci clasele nu pot cauta inafara la package a lor.

* Insa, fisierul resources creat de maven este vazut de orice clasa din indiferent ce package din webapp.Deci daca facem asa:

****



Asa deja,

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

nu va da nicio eroare.

* **Anotatiile ca @PropertySource pot accesa liber ce e si in webapp!**