Medii de proiectare și programare

2017-2018
Curs 3

Conținut curs 3

- Adnotări
- Java Beans
- Introducere în Spring

Adnotări (*Annotations*)

- Începând cu versiunea 1.5
- Adaugă informații unei părți de cod (clasă, metodă, pachet), dar nu fac parte din program. Adnotările nu au nici un efect direct asupra codului pe care îl marchează.
- Utilizări:
  - A furniza informații suplimentare compilatorului. Adnotările pot fi folosite de compilator pentru a detecta erori sau pentru a elimina atenționări.
  - Procesare automata din timpul compilării sau deploymentului. Instrumente soft specializate pot folosi adnotarile pentru a genera automat cod, fișiere XML, etc.
  - Procesare în timpul execuției. Unele adnotări sunt disponibile pentru a fi examinate în timpul execuției codului.

Definirea adnotărilor

```
[declaratii meta-adnotari]
public @interface NumeAdnotare {
   [declaratii elemente]
}
```

Meta-adnotările (pachetul `java.lang.annotation`)(adnotări pentru adnotări) pot fi:

- `@Target(ElementType):` specifică locul din codul sursă unde poate fi folosită adnotarea.
- `CONSTRUCTOR:` declararea unui constructor
- `FIELD:` declararea unui atribut (inclusiv constante enum)
- `LOCAL_VARIABLE:` declararea unei variabile locale
- `METHOD:`declararea unei metode
- `PACKAGE:` declararea unui pachet
- `PARAMETER:` declararea unui parametru
- `TYPE:` declararea unei noi clase, interfețe, adnotări sau enum.

Definirea adnotarilor

Meta-adnotările pot fi:

- `@Retention(RetentionPolicy):` specifică cât timp va fi păstrată adnotarea:
  - `SOURCE:` Adnotările nu sunt salvate la compilare.
  - `CLASS:` Adnotările sunt disponibile în fișierul .class, dar pot fi eliminate de mașina virtuală.
  - `RUNTIME:` Adnotările sunt păstrate de mașina virtuală în timpul execuției și pot fi citite folosind reflecție.
- `@Documented:` Adnotarea este inclusă în documentația Javadocs.
- `@Inherited:` Permit subclaselor să moștenească adnotările părinților.

Elementele unei adnotări

- Sintaxa:

```
Tip numeElement() [default valoare_implicita];
```

unde **`Tip`** poate fi:

  - orice tip primitiv (**`int`**, **`float`**, **`double`**, **`byte`**, etc.)
  - **`String`**
  - **`Class`**
  - Enumerări (**`enum`**)
  - Adnotări (**`annotation`**)
  - Tablouri de tipurile menționate mai sus.

Observații:

1. Dacă se folosește alt tip la declararea unui element, compilatorul va genera eroare.
2. Dacă o adnotare nu conține nici un element, adnotarea se numește de tip *marker*.

Constrângeri valori implicite

- Exista două constrângeri pentru valoarea unui element:

1. Nici un element nu poate avea o valoare nespecificată (fie se declară o valoare implicită, fie se atribuie o valoare pentru fiecare element în momentul folosirii adnotării).
2. Pentru elementele care nu sunt de tip primitiv, nu se acceptă valoarea **`null`** (în momentul folosirii sau ca și valoare implicită).

Adnotări - exemplu

```
import java.lang.annotation.*;


@Target(ElementType.CLASS)
@Retention(RetentionPolicy.RUNTIME)
public @interface ClassPreamble {
   String author();
   String date();
   int currentRevision() default 1;
   String lastModified() default "N/A";
   String lastModifiedBy() default "N/A";
   String[] reviewers();
}
```

Folosirea adnotărilor

Adnotarea apare prima, de obicei pe linie proprie, și poate conține elemente.

Observații:

1. Dacă adnotarea conține un singur element numit **`value`**, numele acestuia poate fi omis.
2. Dacă adnotarea nu conține nici un element, parantezele pot fi omise.

```
@ClassPreamble (
   author = “Popescu Vasile",
   date = "3/17/2008",
   currentRevision = 4,
   lastModified = "4/11/2011",
   lastModifiedBy = “Ionescu Matei"
   reviewers = {“Vasilescu Ana", “Marinescu Ion”, “Pop Ioana"}
)
public class A extends B{
//...
}
```

Exemplu adnotări

Declararea:

```
import java.lang.annotation.*;
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface UseCase {
    public int id();
    public String description() default "no description";
}
```

Folosirea

```
public class A{
  @UseCase(id=3, description="abcd")
  public void f(){
  }
}
```

Adnotări standard

- JSE conține 3 adnotari standard:
  - `@Override` pentru a indica că o metoda redefinește o metodă din clasa de baza. Dacă numele metodei sau signatura nu sunt corecte, compilatorul va genera o eroare.

```
class A{
  @Override
  public String toString(){...}
}
```

  - `@Deprecated` pentru a genera o atenționare la compilare când se folosește elementul adnotat (clasă, metodă, etc.)
  - `@SuppressWarnings` spune compilatorului să nu furnizeze anumite atenționări: `unchecked, deprecated`

```
@SuppressWarnings("unchecked", "deprecated")
void metodaA() { }
```

Beans

- Orice clasă Java este un POJO (eng. *Plain Old Java Object*).
- JavaBeans: este o clasă Java specială. Reguli:
  - Trebuie să aibă un constructor implicit (public și fără nici un parametru). Alte instrumente specializate vor folosi acest constructor pentru a instanția un obiect.
  - Atributele trebuie să poată fi accesate folosind metode de tip **`getXyz`**, **`setXyz`** și **`isXyz`** (pentru atribute de tip boolean). Atributele pentru care sunt definite aceste metode se numesc proprietăți, numele proprietății fiind **`xyz`**. Când se modifică sau se dorește valoarea unei proprietăți se apelează una dintre metodele corespunzătoare.
  - Clasa trebuie sa fie serializabilă. Acest lucru permite instrumentelor specializate să salveze și să refacă starea unui JavaBean.
  - Exemplu: Componentele GUI
- Enterprise Java Beans (EJBs): pentru aplicații complexe (tranzacții, securitate, acces la baze de date)

Exemplu Java Beans

```
public class Student implements java.io.Serializable {
    private String nume;
    private int grupa;
    private boolean licentiat;
    private int note[];
    public Student() { }
    public Student(String nume, int grupa, boolean licentiat){...}
    public String getName() { return nume; }
    public void setName(String name) { nume = name;  }
    public int getGrupa() {return grupa;}
    public void setGrupa(int g){grupa=g;}
    public void setLicentiat(boolean l){licentiat=l;}
    public boolean isLicentiat(){ return licentiat;}
    public void setNote(int[] n){ note=n;}
    public int[] getNote(){return note;}
}
```

Introducere în Spring - Motivație

- Orice aplicație medie sau complexă este compusă dintr-o mulțime de obiecte care colaborează pentru atingerea unui scop. Aceste obiecte știu despre celelalte obiecte (asocierile) și comunică prin transmiterea de mesaje.
- Abordarea tradițională pentru crearea asocierilor dintre obiecte (prin instanțiere sau căutare) generează cod complicat care este dificil de reutilizat și testat (folosind unit testing).

//varianta 1

```
class Concurs{
    private ParticipantiRepositoryMock repo;
  public Concurs(){
      repo=new ParticipantiRepositoryMock();
  }
  //...
}
```

Introducere în Spring

//varianta 2

```
class Concurs{
   private ParticipantiRepositoryFile repo;
  public Concurs(){
     repo=new ParticipantiRepositoryFile("Participanti.txt");
  }
}
```

//varianta 2a

```
 public Concurs(){
     repo=new ParticipantiRepositoryFile("Participanti2.txt",
                                          new ParticipantValidator());
  }
```

//varianta 3

```
class Concurs{
   private ParticipantiRepositoryJdbc repo;
  public Concurs(){
     Properties props=...
     repo=new ParticipantiRepositoryJdbc(props);
  }
}
```

Introducere în Spring

- Spring este un framework open-source, creat inițial de Rod Johnson și descris în cartea sa, *Expert One-on-One: J2EE Design and Development*.
- Frameworkul Spring a fost creat pentru a facilita dezvoltarea aplicațiilor complexe și foarte mari.
- În Spring se pot folosi obiecte simple Java (*POJO*), pentru a crea aplicații care anterior erau posibile doar folosind EJB.
- Un bean Spring este orice clasă Java (nu respectă regulile Java Beans).
- Spring promovează cuplarea slaba prin “injectarea” asocierilor și folosirea interfețelor.
- Spring folosește principiul IoC pentru “injectarea” asocierilor/ dependențelor.

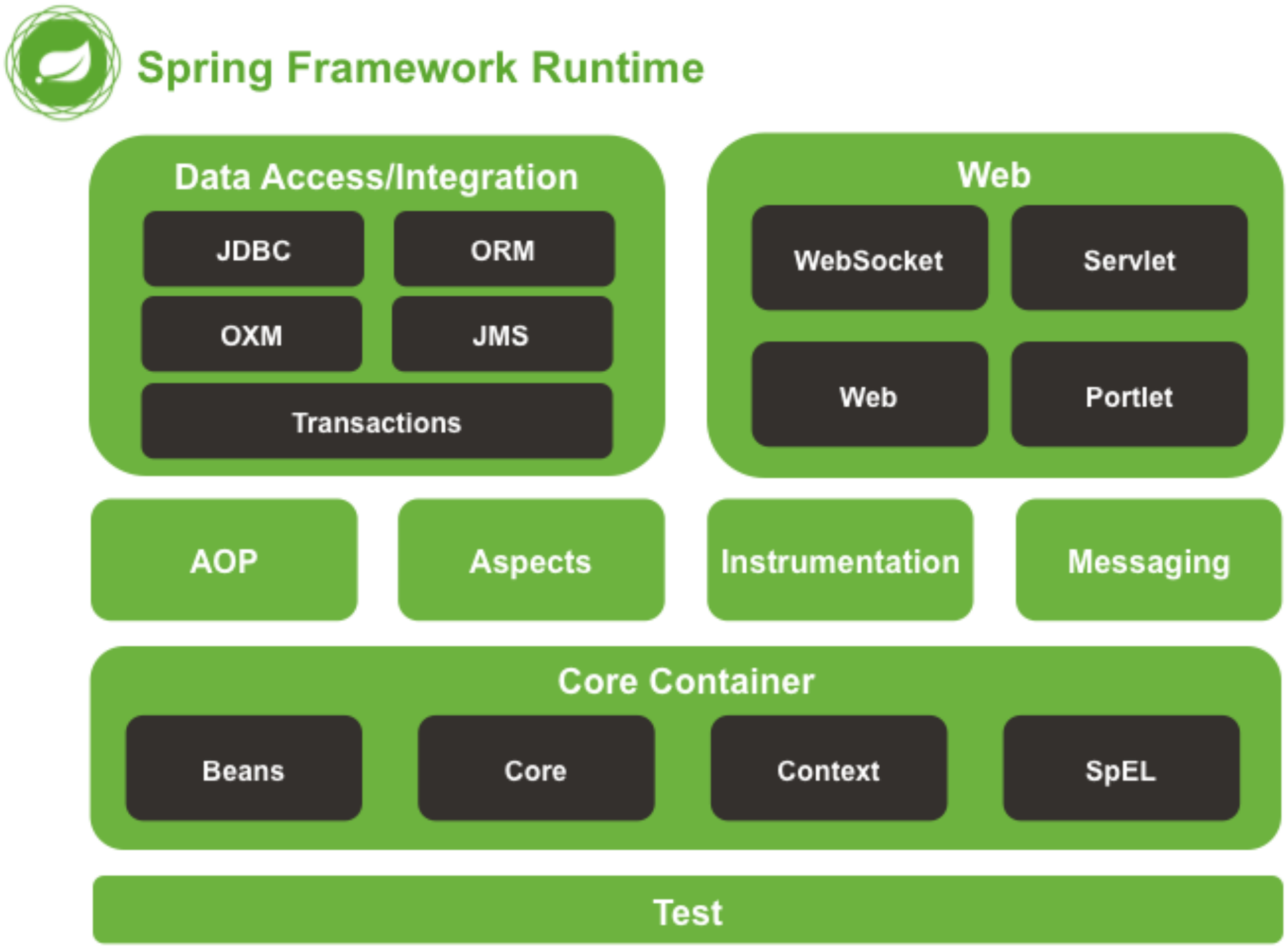
Introducere în Spring

```
public interface ParticipantRepository{...}
class Concurs{
   private ParticipantiRepository repo;
  public Concurs(ParticipantiRepository r){
     repo=r;
  }
  ...
}
//sau
 class Concurs{
  private ParticipantiRepository repo;
  public Concurs(){... }
  public void setParticipanti(ParticipantiRepository r){repo=r;}
}
public class ParticipantiRepositoryFile implements
                                          ParticipantiRepository{...}
public class ParticipantiRepositoryJdbc implements
                                          ParticipantiRepository{...}
```

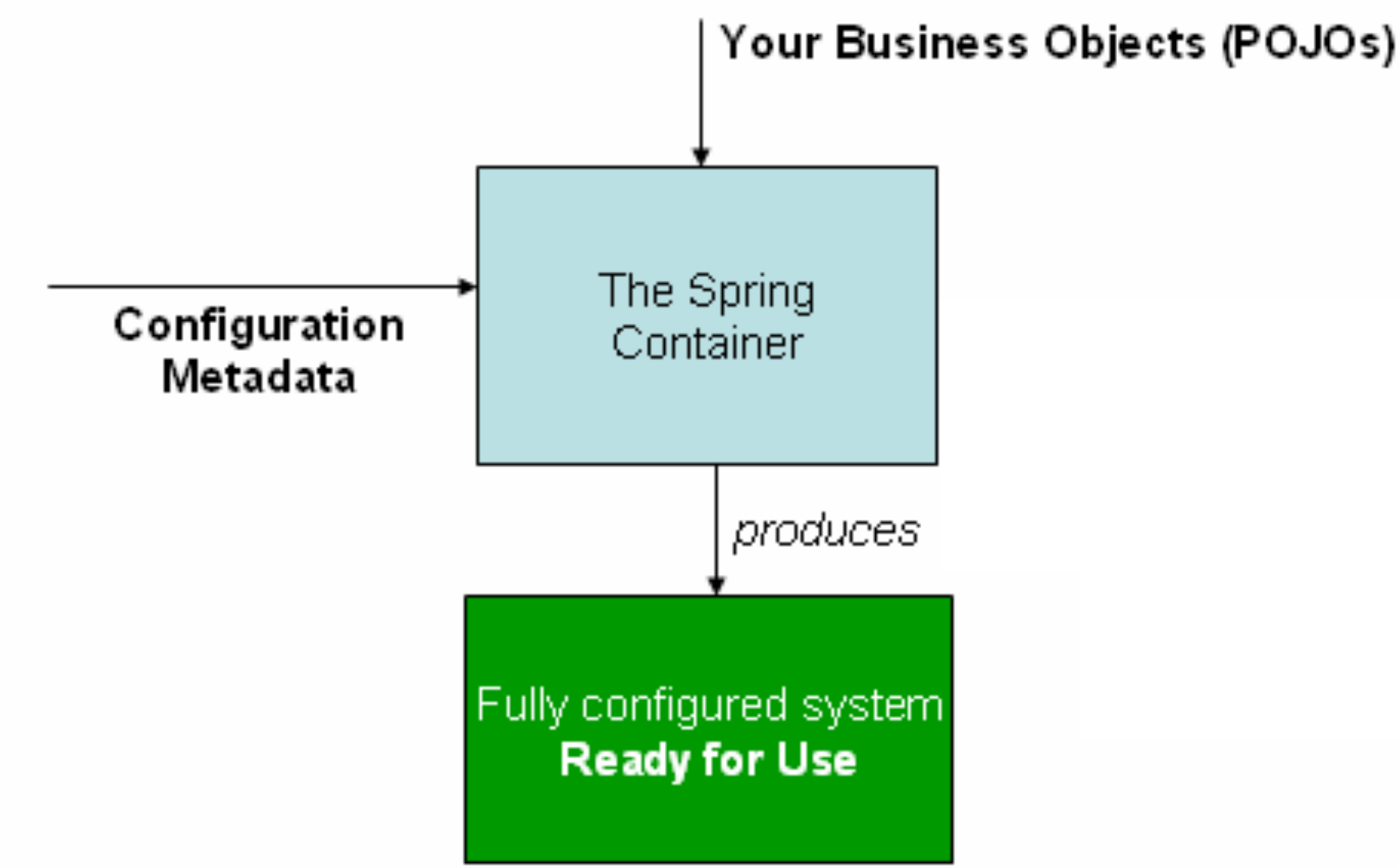
IoC, Dependecy Injection

- Principiul *Inversion of Control* (IoC) este cunoscut și ca *dependency injection* (DI).
- DI este procesul prin care obiectele își definesc asocierile (dependențele) fie prin parametrii constructorilor, fie prin argumentele unei metode de tip factory sau prin proprietăți de tip set, care trebuie apelate imediat după crearea obiectului.
- Un container "injectează" aceste dependențe când creează obiectul. Acest proces este invers celui tradițional, în care obiectul este responsabil de instanțierea sau localizarea dependențelor sale.
- În Spring, obiectele care formează un sistem (aplicație) soft sunt gestionate de containerul bazat pe IoC și sunt numite *bean*-uri.
- Un *bean Spring* este un obiect Java obișnuit care este instanțiat, asamblat și gestionat de containerul Spring IoC.
- Bean-urile și asocierile dintre ele sunt descrise în datele de configurare folosite de container.
- Două variante de a descrie bean-urile: folosind fișiere de configurare în format XML sau cod Java (fișiere de configurare sau autowire).

Arhitectura framework-ului Spring



Containerul Spring



Crearea containerului Spring

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;


public class StartApp{
  public static void main(String[] args){
    ApplicationContext factory = new
    ClassPathXmlApplicationContext("classpath:spring-concurs.xml");
//obtinerea referintei catre un bean din container
Concurs concurs= factory.getBean(Concurs.class);
  }
}
```

Fișierul de configurare XML

- Când se declară bean-urile folosind fișiere XML, elementul rădăcină a fișierului de configurare este **`<beans>.`**
- Un șablon simplu pentru fisierul de configurare este:

```
<?xml version="1.0"encoding="UTF-8"?>
<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">


<!-- Declararea bean-urilor-->


</beans>
```

- În interiorul elementului **`<beans>,`** sunt descrise toate configurările specifice containerului Spring (dacă există) și toate declarațiile bean-urilor.

Declararea unui bean simplu

```
package pizzax.validation;
import pizzax.model.Pizza;
public class DefaultPizzaValidator implements Validator<Pizza> {
    public void validate(Pizza pizza) {
     //...
    }
}
```

```
//spring-pizza.xml
<beans ...>
  <bean id="pizzaValidator"
    class="pizzax.validation.DefaultPizzaValidator"/>
</beans>
```

- Elementul **<bean>** este elementul de bază dintr-un fișier de configurare XML. El spune containerului Spring să creeze un obiect.
- Atributul **id** specifică numele prin care obiectul va fi referit în container.
- Când containerul Spring va încărca bean-urile, el va instanția bean-ul **"pizzaValidator"** folosind constructorul implicit.

DI - Constructori

```
package pizzax.repository.file;
import pizzax.repository;
public class PizzaRepositoryFile implements PizzaRepository {
    private String numefis;
    public PizzaRepositoryFile(String numefis){
    ...
    }
    //implementarea metodelor
    ...
}
//spring-pizza.xml
 ...
<bean id="pizzaRepository"
    class="pizzax.repository.file.PizzaRepositoryFile">
    <constructor-arg value="Pizza.txt" />
</bean>
```

DI - Constructori (2)

```
public class PizzaRepositoryFile implements PizzaRepository {
   private String numefis;
   private Validator<Pizza> valid;
   public PizzaRepositoryFile(String numefis, Validator<Pizza> valid){ ... }
   ...
}
//spring-pizza.xml
 ...
<bean id="pizzaValidator" class="pizzax.validation.DefaultPizzaValidator"/>
<bean id="pizzaRepository"
    class="pizzax.repository.file.PizzaRepositoryFile">
    <constructor-arg value="Pizza.txt" />
    <constructor-arg ref="pizzaValidator" />
</bean>


Validator<Pizza> pizzaValidator=new DefaultPizzaValidator();
PizzaRepository pizzaRepository=new PizzaRepositoryFile("Pizza.txt",
    pizzaValidator);
```

DI - Constructori (3)

```
public class Produs {
    private String denumire="";
    private double pret=0;
    public Produs(String denumire, double pret) {
        this.pret = pret;
        this.denumire = denumire;
    }
    //...
}
//spring-exemplu.xml
 <bean id="mere" class="Produs">
        <constructor-arg index="0" value="Mere" />
        <constructor-arg index="1" value="3.14"/>
</bean>
<!–sau →
 <bean id="mere" class="Produs">
        <constructor-arg type="java.lang.String" value="Mere" />
        <constructor-arg type="double" value="3.14"/>
  </bean>
```

DI - Metode factory

```
public class A {
   private static A instance;
   private A(){...};
   public static A getInstance(){ ...}
   ...
}
```

```
//spring-exemplu.xml
 ...
<bean id="instanta" class="A"  factory-method="getInstance"/ >

//echivalent cu
A objA=A.getInstance();
```

Scopul

Implicit toate bean-urile sunt *singleton* (se creează o singură instanță, indiferent de câte ori un bean este folosit la configurare, sau folosind metoda **`getBean()`** din clasa **`ApplicationContext`**).

Pentru a schimba scopul implicit, se folosește atributul "**`scope`**" al tag-ului **`<bean>`**

```
<bean id="bilet" class="xyz.Bilet" scope="prototype"/>
```

- Valorile posibile pentru atributul "**`scope`**" sunt:
  - **`singleton`**: o singură instanță pentru un container Spring.
  - **`prototype:`** pentru fiecare utilizare se creează un nou bean.
  - **`request, session, global-session`**: se utilizează pentru aplicații Web.

```
<bean id="beanA" class="test.B" scope="prototype"/>
<bean id="beanB" class="test.B"/>
<bean id="beanC" class="test.C">
  <constructor-arg ref="beanA"/> <!--se creeaza o noua instanta -->
</bean>
<bean id="beanD" class="test.D">
  <constructor-arg ref="beanA"/> <!--se creeaza o noua instanta  -->
</bean>
```

DI folosind proprietăți

```
package pizzax.repository.file;
import pizzax.repository;
public class PizzaRepositoryFile implements PizzaRepository {
   private String numefis;
   public PizzaRepositoryFile() {  ...   }
   public void setNumeFisier(String numefis}{...}
   //implementarea metodelor
   ...
}
```

```
//spring-pizza.xml
 ...
<bean id="pizzaRepository"
    class="pizzax.repository.file.PizzaRepositoryFile">
    <property name="numeFisier" value="Pizza.txt"/>
</bean>
```

DI folosind proprietăți

```
package pizzax.repository.file;
import pizzax.repository;
public class PizzaRepositoryMock implements PizzaRepository {
   private Validator<Pizza> valid;
   public PizzaRepositoryMock() {  ...   }
   public void setValidator(Validator<Pizza> v){valid=v;}
   //implementarea metodelor
   ...
}
```

```
//spring-pizza.xml
 ...
<bean id="pizzaRepository"
    class="pizzax.repository.file.PizzaRepositoryMock">
    <property name="validator" ref="pizzaValidator"/>
</bean>
```

DI Constructor + proprietăți

```
package pizzax.repository.file;
import pizzax.repository;
public class PizzaRepositoryFile implements PizzaRepository {
   private Validator<Pizza> valid;
   private String numefis;
   public PizzaRepositoryFile(String numefis) {  ...   }
   public void setValidator(Validator<Pizza> v){valid=v;}
   //implementarea metodelor
   ...
}

//spring-pizza.xml
 ...
<bean id="pizzaRepository"
    class="pizzax.repository.file.PizzaRepositoryMock">
    <constructor-arg value="Pizza.txt"/>
    <property name="validator" ref="pizzaValidator"/>
</bean>
```

Bean-uri inner

```
package pizzax.repository.file;
import pizzax.repository;
public class PizzaRepositoryMock implements PizzaRepository {
   private Validator<Pizza> valid;
   public PizzaRepositoryMock() {  ...   }
   public void setValidator(Validator<Pizza> v){valid=v;}
   //implementarea metodelor
   ...
}

//spring-pizza.xml
 ...
<bean id="pizzaRepository"
    class="pizzax.repository.file.PizzaRepositoryMock">
    <property name="validator">
      <bean class="pizzax.validation.DefaultPizzaValidator"/>
    </property>
</bean>
```

Bean-uri inner

Observații:

- Bean-urile inner nu necesită specificarea atributului `id`. Se poate declara o valoare pentru `id`, dar nu este folosită de container.
- Acest tip de bean-uri nu pot fi refolosite. Sunt folosite pentru “injectare” o singură dată și nu pot fi referite de alte bean-uri.

Proprietăți de tip colecție

- Există situații când o proprietate/parametru-constructor este de tip container (colecție, mulțime, dicționar, tablou, etc...).
- Pentru a inițializa acest tip de proprietăți Spring a definit 4 elemente de configurare:
  - `<list>:` o listă de valori ce poate conține duplicate
  - `<set>:` o lista de valori ce nu conține duplicate
  - `<map>:` o mulțime de perechi cheie-valoare (dicționar)
  - `<props>:` o mulțime de perechi cheie-valoare, unde atât cheia cât și valoarea sunt de tip `String` (clasa `java.util.Properties`)

Proprietăți de tip colecție

- Liste, mulțimi, tablouri:

```
class Produs{
   private String denumire;
  private double pret;
  public Produs(){...}
  public void setDenumire(String d){...}
  public void setPret(double d){…}

  //metode get si set

}
```

```
class Depozit{
  //...
  public void setProduse(java.util.List<Produs> lp){...}
  //sau
  public void setProduse(java.util.Collection<Produs> lp){...}
  //sau
  public void setProduse(Produs[] lp){...}
}
```

Proprietăți de tip colecție

- Liste, tablouri:

```
//spring-exemplu.xml
<bean id="mere" class="Produs">
    <property name="denumire" value="Mere"/>
    <property name="pret" value="2.3"/>
</bean>
<bean id="pere" class="Produs"> ...</bean>
<bean id="prune" class="Produs"> ...</bean>
<bean id="depozit" class="Depozit">
  <property name="produse">
    <list>
        <ref bean="mere"/>
        <ref bean="pere"/>
        <ref bean="prune"/>
    </list>
  </property>
</bean>
```

Proprietăți de tip colecție

- Mulțimi:

```
//spring-exemplu.xml
<bean id="mere" class="Produs">
    <property name="denumire" value="Mere"/>
    <property name="pret" value="2.3"/>
</bean>
<bean id="pere" class="Produs"> ...</bean>
<bean id="prune" class="Produs"> ...</bean>
<bean id="depozit" class="Depozit">
  <property name="produse">
    <set>
        <ref bean="mere"/>
        <ref bean="pere"/>
        <ref bean="prune"/>
        <ref bean="prune"/>
    </set>
  </property>
</bean>
```

Proprietăți de tip colecție

- Dicționare:

```
class Depozit{
  //...
  public void setProduse(java.util.Map<String, Produs> lp){...}


}


//spring-exemplu.xml
<bean id="mere" class="Produs">...</bean>
<bean id="pere" class="Produs"> ...</bean>
<bean id="prune" class="Produs"> ...</bean>
<bean id="depozit" class="Depozit">
  <property name="produse">
    <map>
        <entry key="pMere" value-ref="mere"/>
        <entry key="pPere" value-ref="pere"/>
        <entry key="pPrune" value-ref="prune"/>
    </map>
  </property>
</bean>
```

Proprietăți de tip colecție

- Dicționare: elementul **<entry>** are următoarele atribute:
  - **key**: specifică cheia ca și string;
  - **key-ref:** specifică cheia ca și referință la alt bean din container;
  - **value:** specifică valoarea ca și string;
  - **value-ref:** specifică valoarea ca și referință la un alt bean din container.
- Proprietăți: elementele **props** și **prop**

```
class Depozit{
    public void setProprietati(Properties p){...}
   }

 <bean id="depozit" class="Depozit">
<property name="proprietati">
  <props>
     <prop key="prop1"> A1 </prop>
     <prop key="prop2"> B C2 </prop>
  </props>
</property>
</bean>
```

Bean-uri tip container

- Există situații când trebuie creat un bean de tip container (colecție, mulțime, dicționar, tablou, etc...).
- Pentru a crea un bean de tip container:
  - **<util:list>, <util:set>, <util:map>, <util:props>**

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util.xsd”> … </beans>

<util:properties id="jdbcProps">
    <prop key="tasks.jdbc.driver">org.sqlite.JDBC</prop>
    <prop key="tasks.jdbc.url">jdbc:sqlite:database.db</prop>
</util:properties>
sau
<util:properties id="jdbcProps" location="classpath:bd.config"/>
```

Valori null

- E posibil ca valoarea unei proprietăți să fie setată la null. Pentru aceasta se folosește elementul `<null/>`

```
<property name="numeProprietate"> <null/></property>
```

- SpEL (Spring Expression Language)
  - a fost introdus începând cu versiunea 3.0
  - permite calcularea/ determinarea valorilor unor proprietăți în timpul execuției:

```
<property name="numarAleator" value="#{T(java.lang.Math).random()}"/>

<property name="song" value="#{songSelector.selectSong().toUpperCase()}"/>

<property name="fullName"

   value="#{person.firstName + ' ' + person.lastName}"/>
```

DI constructor vs. DI proprietăți

Recomandări:

- DI constructor - pentru dependențe obligatorii
- DI proprietăți - pentru dependențe opționale
- Situații speciale:
  - constructori cu prea mulți parametrii
  - Verificarea dependențelor opționale că sunt nenule

Configurare folosind JavaConfig

- Specificarea bean-urilor care trebuie create se face într-o altă clasă, adnotată cu **`@Configuration`**:

```
@Configuration
public class PizzerieConfig {
    //…
}
```

- Adnotarea marchează clasa ca și o clasă de configurare
- Containerul Spring consideră că această clasă conține detalii despre bean-urile ce trebuie create și cum trebuie create.

Configurare folosind JavaConfig

- Declararea unui bean se face cu adnotarea **`@Bean`**:

```
import pizzax.validation.DefaultPizzaValidator;
@Configuration
public class PizzerieConfig {
    @Bean
     public Validator<Pizza> validator(){
       return new DefaultPizzaValidator();
     }
}
```

- Id-ul implicit al bean-ului este numele metodei (ex. **`validator`**)
- Setarea unui id explicit se face folosind elementul "**`name`**".

```
@Bean(name="pizzaVal")
public Validator<Pizza> validator(){
    return new DefaultPizzaValidator();
}
```

Configurare folosind JavaConfig

- Injectarea dependențelor:

```
@Configuration
public class PizzerieConfig {
     @Bean
      public Validator<Pizza> validator(){
         return new DefaultPizzaValidator();
       }
    //varianta a - apelul metodei care creeaza bean-ul
      @Bean
    public PizzaRepositoryMock pizzaRepo(){
         return new PizzaRepositoryMock(validator());
    }

    @Bean
    public PizzaRepositoryFile pizzaFileRepo(){
         return new PizzaRepositoryFile(validator(),”Pizza.txt”);
     }
}
```

- Un singur bean de tip Validator<Pizza> este creat!

Configurare folosind JavaConfig

- Injectarea dependențelor:

```
@Configuration
public class PizzerieConfig {
     @Bean
      public Validator<Pizza> validator(){
        return new DefaultPizzaValidator();
       }
   //varianta b - transmiterea bean-ului ca si parametru
     @Bean
   public PizzaRepositoryMock pizzaRepo(Validator<Pizza> val){
        return new PizzaRepositoryMock(val);
   }

   @Bean
   public PizzaRepositoryFile pizzaFileRepo(Validator<Pizza> val){
        return new PizzaRepositoryFile(val,”Pizza.txt”);
    }
}
```

Configurare folosind JavaConfig

- În interiorul unei metode adnotate cu @Bean se poate folosi orice cod Java necesar creării bean-ului:

```
@Configuration
public class PizzerieConfig {
    @Bean
    public PizzaRepositoryFile pizzaFileRepo(Validator<Pizza> val){
        PizzaRepositoryFile repo=new new PizzaRepositoryFile(”Pizza.txt”);
        repo.setValidator(val);
        return repo;
    }
    @Bean
    public PizzaRepositoryJdbc pizzaJdbcRepo(){
      Properties jdbcProps=new Properties();
      try {
          jdbcProps.load(new FileReader("bd.config"));
      } catch (IOException e) {
          System.out.println(“No properties were set. Cannot find bd.config “+e);
      }
      return new PizzaRepositoryJdbc(jdbcProps);
    }
}
```

Crearea containerului Spring JavaConfig

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AnnotationConfigApplicationContext;


public class StartApp{
  public static void main(String[] args){
    ApplicationContext context=new
          AnnotationConfigApplicationContext(PizzerieConfig.class);
    //obtinerea referintei catre un bean din container
    PizzaService repo= factory.getBean(PizzaService.class);
  }
}
```

Configurare automată folosind Java

- Configurarea automată (implicită) este o metodă prin care containerul Spring descoperă automat bean-rile care trebuie create, dependențele dintre acestea și încearcă să creeze aceste bean-uri.
- @ComponentScan specifică containerului opțiunea de configurare automată

```
package pizzerie.config;
@Configuration
@ComponentScan
public class PizzerieAutowireConfig {

}
```

- Implicit, containerul va încerca să descopere bean-urile începând cu pachetul clasei de configurare.

```
@ComponentScan(“pizzerie”)
@ComponentScan(basePackages={“pizzerie”,”cofetarie”})
@ComponentScan(basePackageClasses={C.class, D.class})
```

Configurare automată folosind Java

- Declararea bean-urilor: @Component

```
@Component
public class DefaultPizzaValidator implements Validator<Pizza> {
    public void validate(Pizza pizza) {
     //...
    }
}
```

- Implicit, id-ul bean-ului este numele clasei cu prima literă transformată în literă mică.
- Bean cu id explicit:

```
@Component("pizzaVal")
public class DefaultPizzaValidator implements Validator<Pizza> {
    public void validate(Pizza pizza) {
     //...
    }
}
```

Configurare automată folosind Java

- Marcarea dependențelor: @Autowired
- Constructori, atribute, metode (set, etc.)

```
@Component
public class PizzaInMemoryRepository implements PizzaRepository {
   private Validator<Pizza> valid;
   @Autowired
   public PizzaInMemoryRepository(Validator<Pizza> valid){ ... }
   ...
}
@Component
public class PizzaInMemoryRepository implements PizzaRepository {
   private Validator<Pizza> valid;
   public PizzaInMemoryRepository(){ ... }
   @Autowired
   public void setValidator(Validator<Pizza> val){…}
}
```

Configurare automată folosind Java

- Marcarea dependențelor: @Autowired

```
@Component
public class PizzaInMemoryRepository implements PizzaRepository {
   private Validator<Pizza> validator;
   public PizzaInMemoryRepository(){ ... }
   @Autowired(required=false)
   public void setValidator(Validator<Pizza> val){…}
}
```

- Dacă nu există nici un bean care să satisfacă dependență, proprietatea va rămâne neinițializată.

```
public void save(Pizza p){
  if (validator!=null){
     //…
  }
}
```

Configurare automată folosind Java

- @Component, @Autowired: adnotări specifice frameworkului Spring
- Dependența codului de frameworkul Spring
- @Named, @Inject : adnotări din specificația *Java Dependency Injection*
- Pachetul **`javax.inject`**

```
import javax.inject.Inject;
import javax.inject.Named;
```

```
@Named
public class PizzaRepositoryMock implements PizzaRepository {
   private Validator<Pizza> valid;
   @Inject
   public PizzaRepositoryMock(Validator<Pizza> val){ ... }
 //…
 }
```

- In majoritatea cazurilor sunt interschimbabile.

Configurare automată folosind Java

- @Scope: specificarea scopului (implicit singleton, prototype, request, session)
- @Component, @Bean

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class ABean { ... }
```

```
@Bean
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public ABean abean() {
        return new ABean();
}
```

Crearea containerului Spring Java Autowire

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AnnotationConfigApplicationContext;


public class StartApp{
  public static void main(String[] args){
    ApplicationContext context=new
    AnnotationConfigApplicationContext(PizzerieAutowireConfig.class);
    //obtinerea referintei catre un bean din container
    PizzaService repo= factory.getBean(PizzaService.class);
  }
}
```

Configurare automată folosind Java

- Ambiguități.

```
@Autowired
public void setDessert(Dessert dessert) {
            this.dessert = dessert;
}

@Component
public class Cake implements Dessert { ... }
@Component
public class Cookies implements Dessert { ... }
@Component
public class IceCream implements Dessert { ... }
```

- Care bean satisface dependența?

```
NoUniqueBeanDefinitionException: nested exception is
  org.springframework.beans.factory.NoUniqueBeanDefinitionException:
No qualifying bean of type [com.desserteater.Dessert] is defined:
  expected single matching bean but found 3: cake,cookies,iceCream
```

Configurare automată folosind Java

- Ambiguități - Soluția 1 - @Primary.

```
@Autowired
public void setDessert(Dessert dessert) {
            this.dessert = dessert;
}

@Component
@Primary
public class Cake implements Dessert { ... }
@Component
public class Cookies implements Dessert { ... }
@Component
public class IceCream implements Dessert { ... }
```

- Soluția 2 - @Qualifier
- Soluția 3 - Adnotare proprie

Configurare XML vs. JavaConfig vs Autowire

XML	JavaConfig	Autowire
• Nu necesită modificarea codului sursă • Nu necesită recompilarea când apar modificări • Necesită învățarea unui nou limbaj (XML) • Se poate folosi când nu avem acces la tot codul sursă al aplicației • Nu se pot verifica tipurile bean-urilor și dependențele la compilare • Nu apar ambiguități	• Necesită recompilare când apar modificări • Nu necesită învățarea unui nou limbaj • Se verifică static tipurile • Se poate folosi când nu avem acces la tot codul sursă al aplicației	• Necesită recompilare când apar modificări • Nu necesită învățarea unui nou limbaj • Se verifică static tipurile • NU se poate folosi când nu avem acces la tot codul sursă al aplicației • Dependența codului sursă de Spring

Configurare folosind Gradle

- Fișierul **`build.gradle`**

```
dependencies {
    compile 'org.springframework:spring-context:5.0.4.RELEASE'
    runtime group: 'org.xerial', name: 'sqlite-jdbc', version: ‘3.16.1’
    testCompile group: 'junit', name: 'junit', version: '4.11'
}
```

Referințe Spring

- Documentația frameworkului Spring
  http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/index.html

- Craig Walls, *Spring in Action*, Fourth Edition, Ed. Manning, 2015