
Spring Framework

Reda Bendraou

Reda.Bendraou@lip6.fr

<http://pagesperso-systeme.lip6.fr/Reda.Bendraou/>

This course is inspired by the readings/sources listed in the last slide

Spring Framework

Introduction

Introduction

Java/Java EE Development requirements:

- ✓ Separation of concerns
- ✓ Development productivity
- ✓ Platform independence
- ✓ Tests (costly and hard to realize in JEE environments)

JEE Development Issues

Standardization Processes take too long!

Before:

- ✓ In case of problems: development of “Home Made” solutions

Currently:

- ✓ **A problem => A Framework**

Issues:

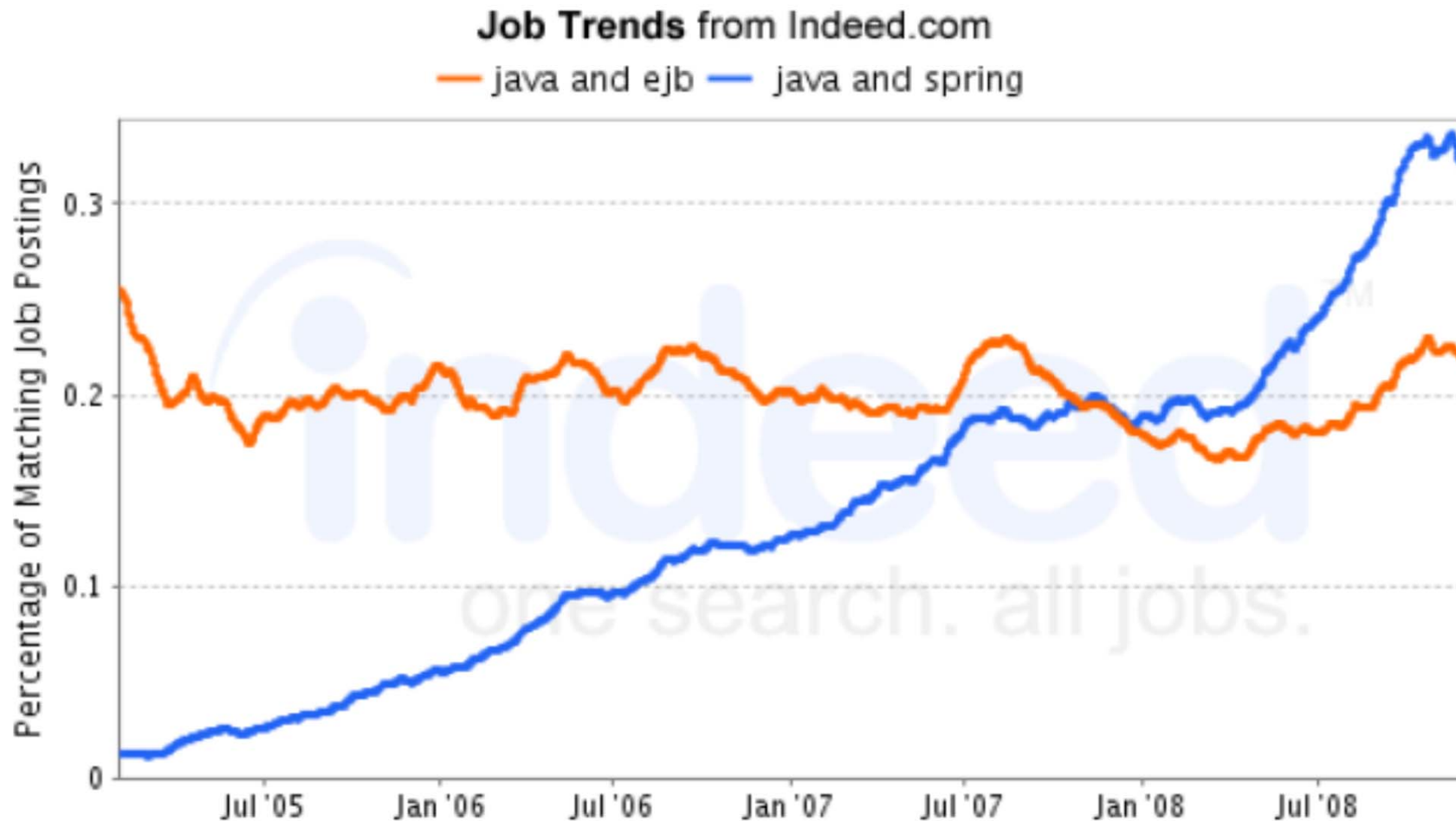
- ✓ Proliferation of frameworks
- ✓ Requires integration efforts

Spring Response

- ✓ Notion of *Lightweight Container*
- ✓ **AOP** (Aspect Oriented Programming) Support
- ✓ **Integration with other frameworks** (Struts, Hibernate, etc.)

Use of Spring in the Industry wrt to EJBs

- Claims to compile data from most major jobs sites
 - Data through 12/2008



Notion of Lightweight Container

Management of Application's Components

- ✓ Configuration via XML
- ✓ Management of components life cycles
- ✓ Management of dependencies between components

No specific infrastructure requirements

- ✓ Just a JVM

AOP Support

Focuses on the resolution of transversal problems, sometimes hard to realize with traditional OO programming

Allows offering services similar to EJB without the complexity of using EJBs and EJB containers

Advantages :

- ✓ Separation of Concerns
- ✓ Less code duplication

AOP Support

Spring uses massively AOP for its internal machinery

Spring provides a set of predefined ready-to-use aspects
(performance monitoring, transaction management, etc.)

Spring provides an infrastructure for developing its own aspects
using either Spring AOP or AspectJ

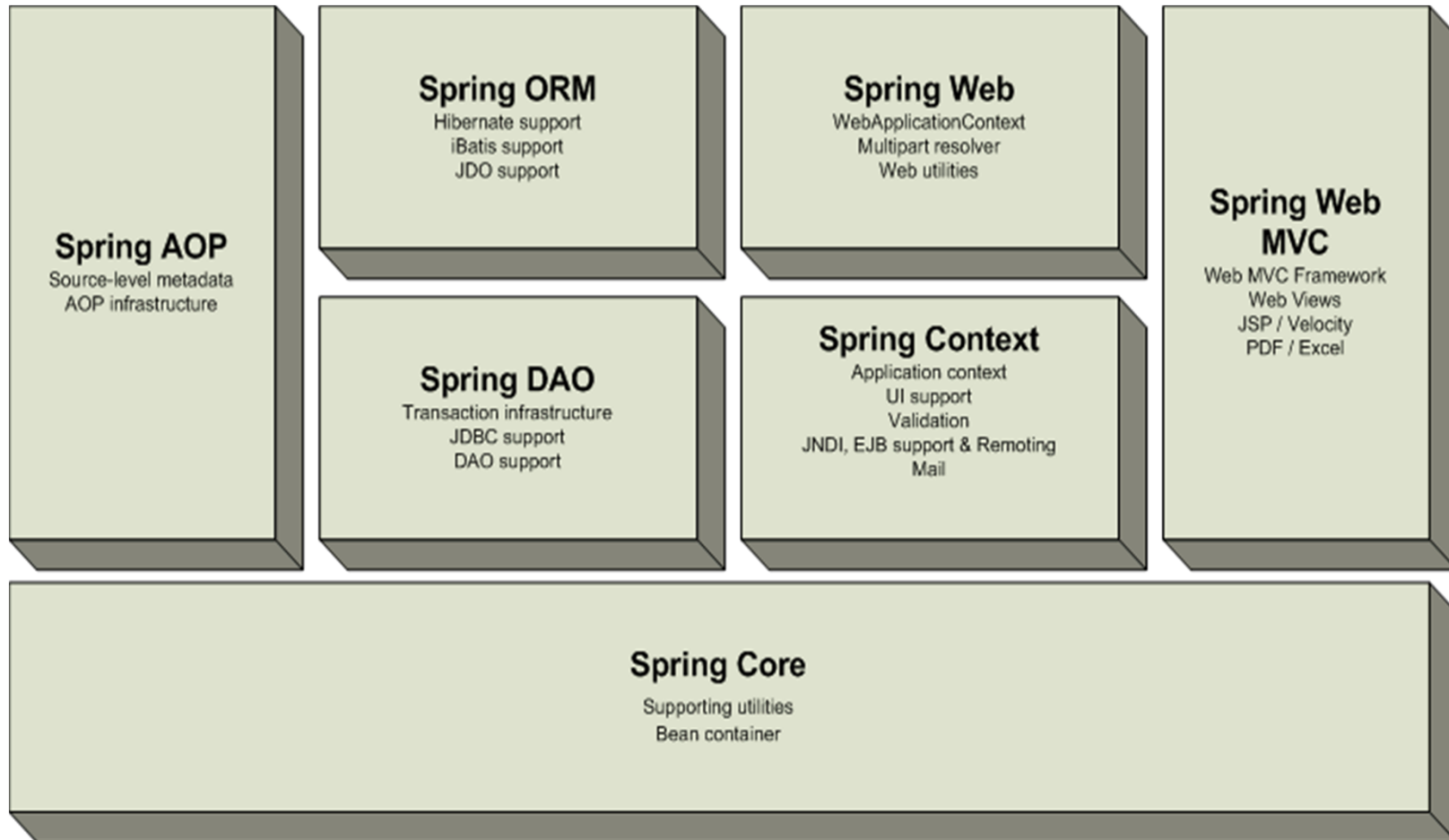
Integration with other frameworks

Eases the use of specific technologies while taking in charge repetitive code (example : JdbcTemplate, HibernateTemplate, JmsTemplate)

Exporting POJOs (rmi, webservices, jmx...) without modifying the POJO code.

Integration with other frameworks JUnit, Struts, Struts 2, DWR (Direct Web Remoting), apache Wicket

Spring Global Architecture



Installing the Development Environment

Eclipse

- ✓ Eclipse JEE IDE
- ✓ Spring IDE for Eclipse (optional)
- ✓ Plugin Maven (optional)

Tomcat

- ✓ Web Container

Framework Spring

- ✓ Mainly .Jars!

tiers Framework that can be integrated to Spring

- ✓ Struts, Hibernate, JUnit, ...

In this Course

Spring covers many aspects!!

In this course we will focus on:

- IoC (Inversion of Control) and Dependency Injection
- Spring AOP
- Spring Integration with Hibernate

Spring Framework, the foundations

- **IoC (Inversion of Control)**
 - **Dependency Injection**
 - Lightweight Containers

The dependency issue in nowadays applications

The naïve approach

The Service Locator approach

The Inversion of Control approach

The Naïve Approach

Example

```
public class Foo {  
  
    private IBar bar;  
    private IBaz baz;  
  
    public Foo() {  
        bar = new SomeBar();  
        baz = new SomeBaz();  
    }  
}
```

+

- Easy to understand/realize

-

- **Strong Coupling (dependency)**

- Requires the knowledge of how to assemble dependent components (and their own dependencies)

- Need to have access to the code in order to modify the application behavior

- Hard to test => need to test also all the dependencies. Impossibility to use Mocks.

Service Locator: Approach

Example

```
public class Foo {  
  
    private IBar bar;  
    private IBaz baz;  
    private IServiceLocator locator;  
  
    public Foo(IServiceLocator locator_) {  
        locator = locator_;  
        bar = (IBar)locator.Get(ServiceNames.BAR);  
        baz = (IBaz)locator.Get(ServicesNames.BAZ);  
    }  
}
```

+

- Easy to understand/realize
- **Testable, Flexible, Extensible**
- Forces the separation between the interface and the implementation

-

- **Still a dependency to the Service Locator**
- Need to get an instance of the Service Locator (in a static class for instance)
- **Less coupling but still a coupling !!**

Questions

How can we break this dependency?

How can we ensure that **Foo** works **with any implementation of IBar or IBaz**? Even if we don't know them in advance (the implementations)

How to ensure that instances of the given implementations are **injected/reported inside** the Foo class?

But in this case, who is deciding/controlling the execution of the application?
It is an external application => **Inversion of Control** by **Dependency Injection**

The IoC Approach

Example

```
public class Foo {  
    private IBar bar;  
    private IBaz baz;  
    public Foo(IBar bar_, IBaz baz_) {  
        bar = bar_;    baz = baz_;  
    }  
}
```

+

- The code is easy to understand/realize

- Testable, Flexible, Extensible**

- Forces the separation between the interface and the implementation

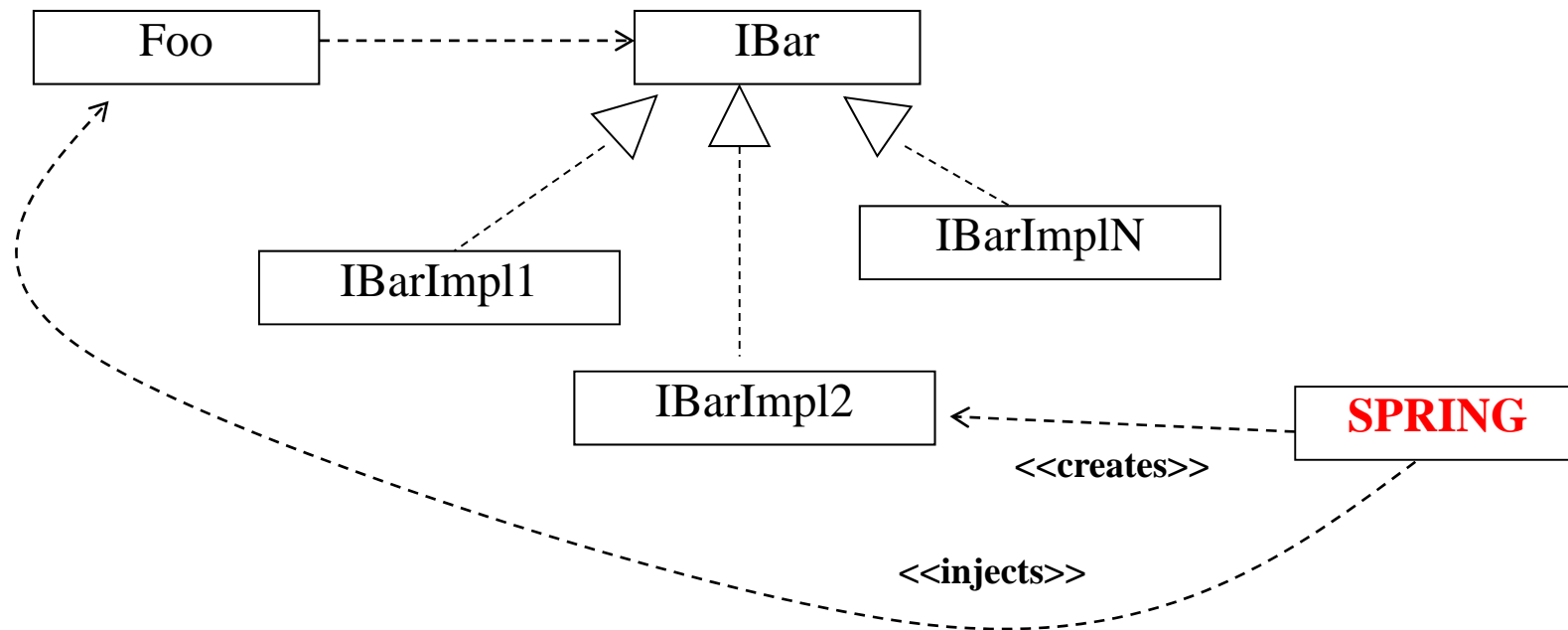
-

- We still need to create the dependencies but outside the application

This where SPRING comes into the play!!

The IoC Approach

Dependency Injection



Principle: instantiation of the appropriate components, management of the dependencies and the injections

Inversion of Control within Lightweight Containers

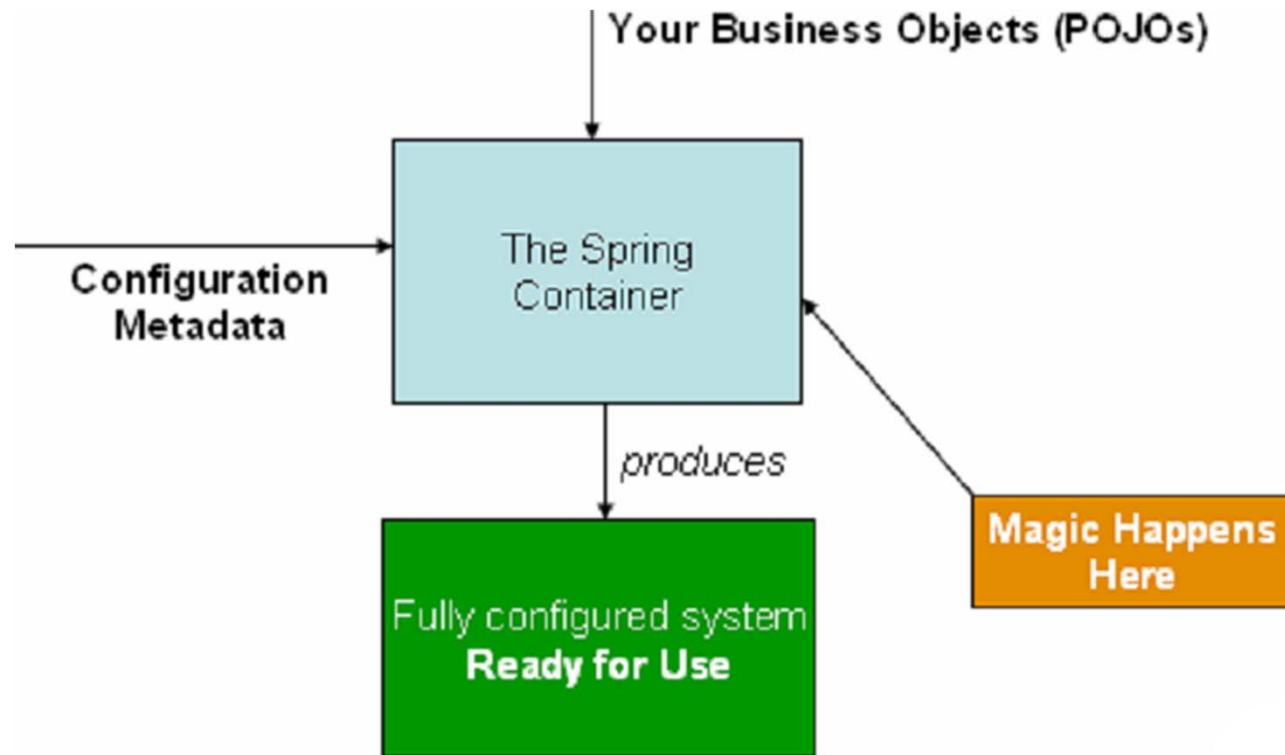
Spring as a lightweight container takes in charge IoC with
Dependency injection

Spring :

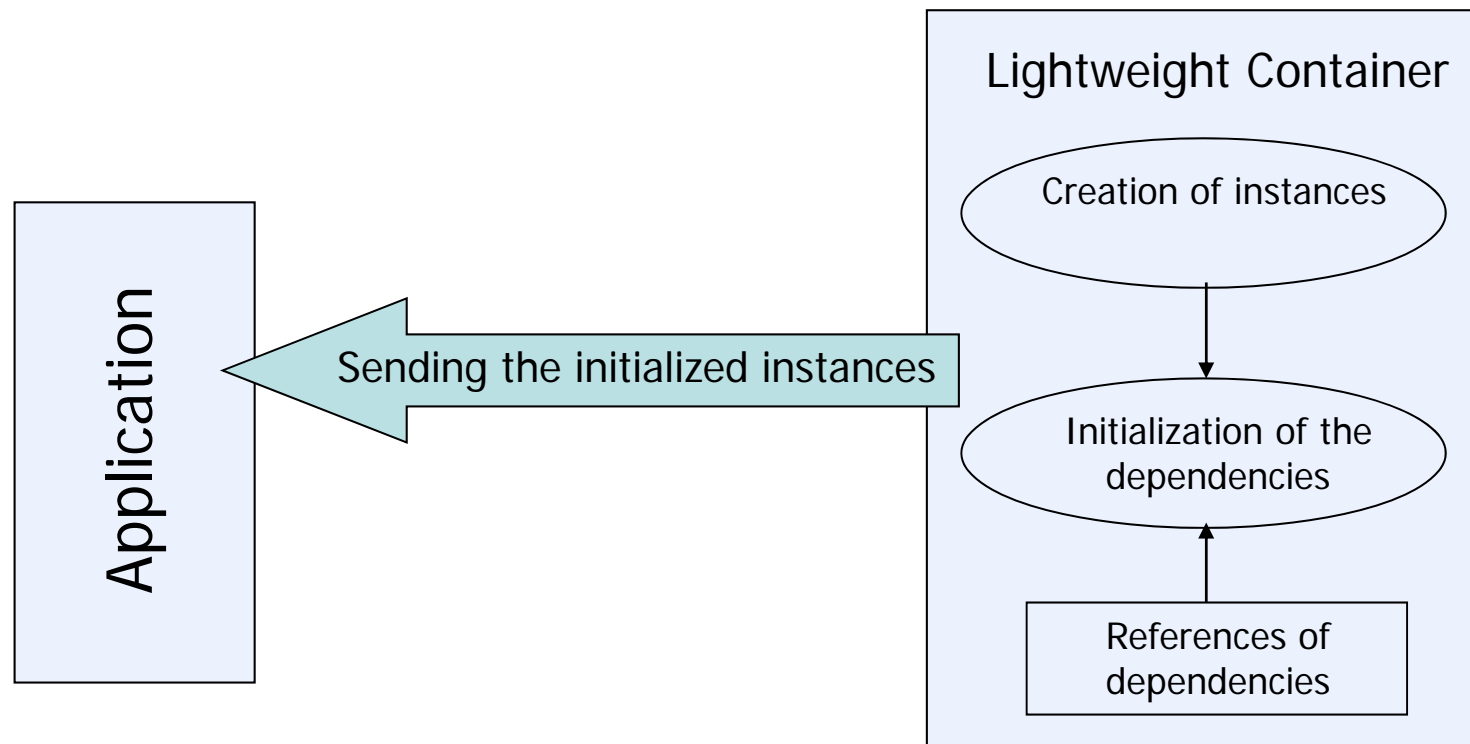
- Manages complex inter dependencies between components
- Manages component life cycles: use of the singleton or prototype instantiation patterns

Inversion of Control within Lightweight Containers

IoC within the Spring container



Spring: Dependency Injection



Spring Framework, the foundations

- IoC (Inversion of Control)
 - Dependency Injection
 - **Lightweight Containers**

Spring: Lightweight Container

Bean Definition

- ✓ Basic data
- ✓ Injection methods
- ✓ Injection of properties
- ✓ Injection of collaborators

Bean factory & Application context

Interactions with the container

- ✓ Access to the *Bean Factory* or to the *Application Context*
- ✓ Post-processors

Bean Definition

Class name => Bean's implementation class

- ✓ If the bean is built from a **factory's class static method**, give the name of this factory

Configuration of the bean's behavior within the container

- ✓ (i.e. ***prototype*** or ***singleton***, *autowiring mode*, *dependency checking mode*, initialization & destruction methods)

Constructor arguments of the bean as well as its properties

Other beans required by a given bean to execute

- ✓ *i.e. collaborators => dependencies*

Bean Definition: Example

```
<beans>
  <bean id="beanFoo1" class="some.package.IBarImpl1"/>
  <bean id="beanFoo2" class="some.package.IBarImpl2" scope="prototype"/>
</beans>

<!-- in case of using a class's static method-->
<bean id="exampleBean" class="examples.ExampleBean2" factory-method="createInstance"/>

<!-- in case of using a factory's class static method-->
<bean id="exampleBean" factory-bean="myFactoryBean" factory-method="createInstance"/>
```

The Bean Definition File

- Within a file named **applicationContext.xml** (standard) or name defined by the user (beanDefinition.xml for instance)

IMPORTANT:

- **In the context of Java Desktop application, the file should be placed at the root of the project**
- **In the context of a web application (eg. JSP/Servlet) the file should be placed in the WEB-INF folder (i.e. WEB-INF/applicationContext.xml)**

In the context of a web application

We need to define Listeners

ContextLoaderListener

- This listener runs when the app is first started. It instantiates the `ApplicationContext` (from **WEB-INF/applicationContext.xml**) and places a reference to it in the `ServletContext`
- You can retrieve this reference with the static **`getRequiredWebApplicationContext`** method of **`WebApplicationContextUtils`**

• **RequestContextListener**

- This listener is needed if you declare any of your beans to be request-scoped or session-scoped (i.e., Web scopes eg. *session* instead of the usual Spring scopes of *singleton* or *prototype*)

In the context of a web application

These Listeners are configured in the Web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  <listener>
    <listener-class>
      org.springframework.web.context.request.RequestContextListener
    </listener-class>
  </listener>
  ...
</web-app>
```

Bean Definition: Instantiation

Two modes:

- ✓ *Singleton* (a unique instance). By default
- ✓ *Prototype* a new instance for each new request

```
<bean id="exampleBean" class="examples.ExampleBean" singleton="false"/>
```

```
<bean name="yetAnotherExample" class="examples.ExampleBeanTwo" singleton="true"/>
```

Bean Definition: other options

Attributes of the 'bean' element:

lazy : lazy loading / instant loading

parent : the 'parent' of the bean from whom we can reuse injections

name : alias

autowire : automatic resolution of dependencies (by type, by name).

init-method : method called automatically at bean's creation time

destroy-method : method called automatically at bean's destruction time

Spring's Dependency Injection

Two Ways:

Injection using Setters (i.e. setXXX();)

Injection using the class's constructor

Dependency Injection: the *Setters* way

```
<bean id="exampleBean" class="examples.ExampleBean">  
    <property name="beanOne"><ref bean="anotherExampleBean"/></property>  
    <property name="beanTwo"><ref bean="yetAnotherBean"/></property>  
    <property name="integerProperty"><value>1</value></property>  
</bean>  
<bean id="anotherExampleBean" class="examples.AnotherBean"/>  
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

a reference to another bean

```
public class ExampleBean {    // the JAVA class  
    private AnotherBean beanOne;  
    private YetAnotherBean beanTwo;  
    private int i;  
  
    public void setBeanOne(AnotherBean beanOne) {this.beanOne = beanOne; }  
    public void setBeanTwo(YetAnotherBean beanTwo) { this.beanTwo = beanTwo; }  
    public void setIntegerProperty(int i) { this.i = i; }  
}
```

Dependency Injection: the *Setters* way

Injection of values 'constants' :

```
<beans>
  <bean id="beanFoo" class="some.package.Foo">
    <property name="someString" value="theValue"/>
    <property name="someBoolean" value="true"/>
    <property name="someFloat" value="10.5"/>
    <property name="someStringArray"
      value="value1,value2"/>
  </bean>
</beans>
```

Dependency Injection: the *Constructor* way

- The Java Class

```
public class ExampleBean {  
    private AnotherBean beanOne;  
    private YetAnotherBean beanTwo;  
    private int i;  
  
    public ExampleBean(AnotherBean anotherBean,  
        YetAnotherBean yetAnotherBean, int i) {  
        this.beanOne = anotherBean;  
        this.beanTwo = yetAnotherBean;  
        this.i = i;  
    }  
}
```

Dependency Injection: the *Constructor* way

Bean Definition

```
<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg><ref bean="anotherExampleBean" /></constructor-arg>
  <constructor-arg><ref bean="yetAnotherBean" /></constructor-arg>
  <constructor-arg type="int"><value>1</value></constructor-arg>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean" />
<bean id="yetAnotherBean" class="examples.YetAnotherBean" />
```

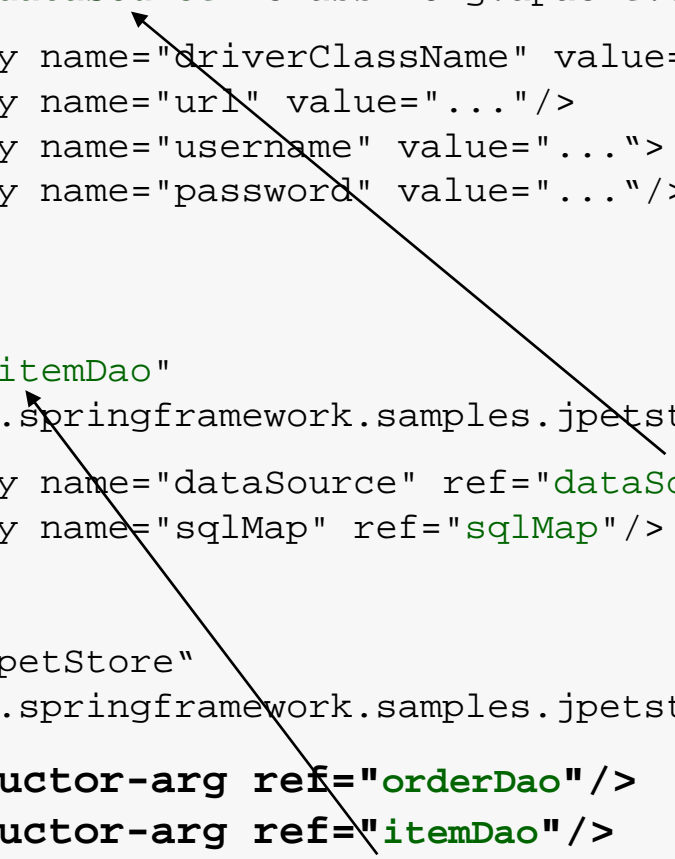
Dependency Injection: the *Constructor* way

An Example

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="... " />
  <property name="url" value="..." />
  <property name="username" value="...">
  <property name="password" value="..." />
</bean>

<bean id="itemDao"
class="org.springframework.samples.jpetsy.dao.ibatis.SqlMapItemDao">
  <property name="dataSource" ref="dataSource" />
  <property name="sqlMap" ref="sqlMap" />
</bean>

<bean id="petStore"
class="org.springframework.samples.jpetsy.domain.logic.PetStoreImpl">
  <constructor-arg ref="orderDao" />
  <constructor-arg ref="itemDao" />
</bean>
```

A diagram consisting of two arrows. The first arrow originates from the `dataSource` property value in the `itemDao` bean definition and points to the `dataSource` property value in the `dataSource` bean definition. The second arrow originates from the `itemDao` bean definition and points to the `itemDao` constructor argument value in the `petStore` bean definition.

Bean's Life Cycle : Initialization

Bean Initialization

- ✓ After initializing bean's properties using either the Setter or Constructor way, there is a mean to initialize the behavior/properties of the bean

- ✓ **Two ways:**

- a) with « init-method »

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>
```

//Java Class

```
public class ExampleBean {  
    public void init() { // do some initialization work } }
```

- b) By implementing the «InitializingBean» interface

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements InitializingBean {  
    public void afterPropertiesSet() { // do some initialization work } }
```

Bean's Life Cycle : Finalization

Finalisation du Bean

- ✓ Possibilité d'exécuter un comportement

- ✓ Two ways:

- a) using the « destroy-method »

```
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>
```

//Java Class

```
public class ExampleBean {  
    public void cleanup() { // close connection } }
```

- b) By implementing the « DisposableBean » interface

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements DisposableBean {  
    public void destroy() { // eg.close connection } }
```


Property Definitions

For a property it is possible to inject :

A constant value : **value**

A reference to another bean : **ref**

A list : **list** (with multiple values or references to other beans)

A set : **set** (with multiple values or references to other beans)

A map : **map** (with entries like : value/reference to another bean)

Properties : **props**

The name of the bean : **idref**

Property Definitions: Example

```
<beans>

  <bean id="beanFoo" class="some.package.Foo">

    <property name="someValue" value="someHardCodedValue"/>
    <property name="someRef" ref="beanBar"/>
    <property name="someList">
      <list>
        <ref bean="bean1"/>
        <ref bean="bean2"/>
      </list>
    </property>
    <property name="someMap">
      <map>
        <entry key="key1" value-ref="bean1"/>
        <entry key="key2" value-ref="bean2"/>
      </map>
    </property>
  </bean>

</beans>
```

Collaborator Definitions

```
<bean id="beanFoo" class="some.package.Foo">
    <property name="someDependence" ref="dependenceBeanInstance"/>
</bean>
```

```
<bean id="beanFoo" class="some.package.Foo">
    <property name="someDependence">
        <bean class="some.package.FooDependence" />
    </property>
</bean>
```

Inheritance

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="..." />
  <property name="url" value="..." />
  <property name="username" value="..." />
  <property name="password" value="..." />
</bean>

<bean id="parentDao" abstract="true">
  <property name="dataSource" ref="dataSource" />
</property>

<bean id="itemDao" parent="parentDao"
class="org.springframework.samples.jpetstore.dao.ibatis.SqlMapItemDao">
</bean>

<bean id="orderDao" parent="parentDao"
class="org.springframework.samples.jpetstore.dao.ibatis.SqlMapOrderDao">
</bean>
```

Bean Factory & Application Context

The *Bean Factory* defines the interface used by the application in order to interact with the Lightweight Container (Spring in our case)

The *Application Context* is also an interface, which extends the Bean Factory with some additional functionalities

Bean Factory & Application Context

Application Context adds the following functionalities wrt the Bean Factory

- ✓ Support of messages and internationalization
- ✓ Advanced support of resources loading
- ✓ Support of events publishing
- ✓ A possibility to define a hierarchy of contexts

The context can be accessed through the `ApplicationContext` interface and uses the XML configuration files

Bean Factory & Application Context

Instantiation of Spring lightweight container :

```
ApplicationContext context = new ClassPathXmlApplicationContext(new  
    String[]{"applicationContext.xml",  
    "applicationContext-part2.xml"});  
IFoo foo = (IFoo)context.getBean("foo");
```

Id of the bean defined in the XML file



Bean Factory & Application Context

Instantiation of Spring lightweight container in the context of a WEB Application :

```
ApplicationContext context =  
WebApplicationContextUtils.getWebApplicationContext(getServletContext());  
  
IFoo foo =(IFoo)context.getBean("foo");
```

Id of the bean defined in the XML file



Interactions with the Container

Writing post-processors

Accessing the *Bean Factory* or the *Application Context* and
modifying the context in a programmatic way

Interactions with the Container

Post processing

- ✓ Post-processor of beans : **BeanPostProcessor**
 - Allows the inclusion of behavior before and after property initializations
- ✓ Post-processor of Bean Factories : **BeanFactoryPostProcessor**
 - Allows modifying the bean factory configuration after its creation

Interactions avec le conteneur

Advanced option : manipulation of the context

```
ConfigurableApplicationContext ctx = new ClassPathXmlApplicationContext("appCtx.xml");
```

```
DefaultListableBeanFactory factory = ((DefaultListableBeanFactory) ctx.getBeanFactory());
```

```
// Creation of a bean in a programmatic way
```

```
BeanDefinitionBuilder builder = BeanDefinitionBuilder.rootBeanDefinition(String.class);
```

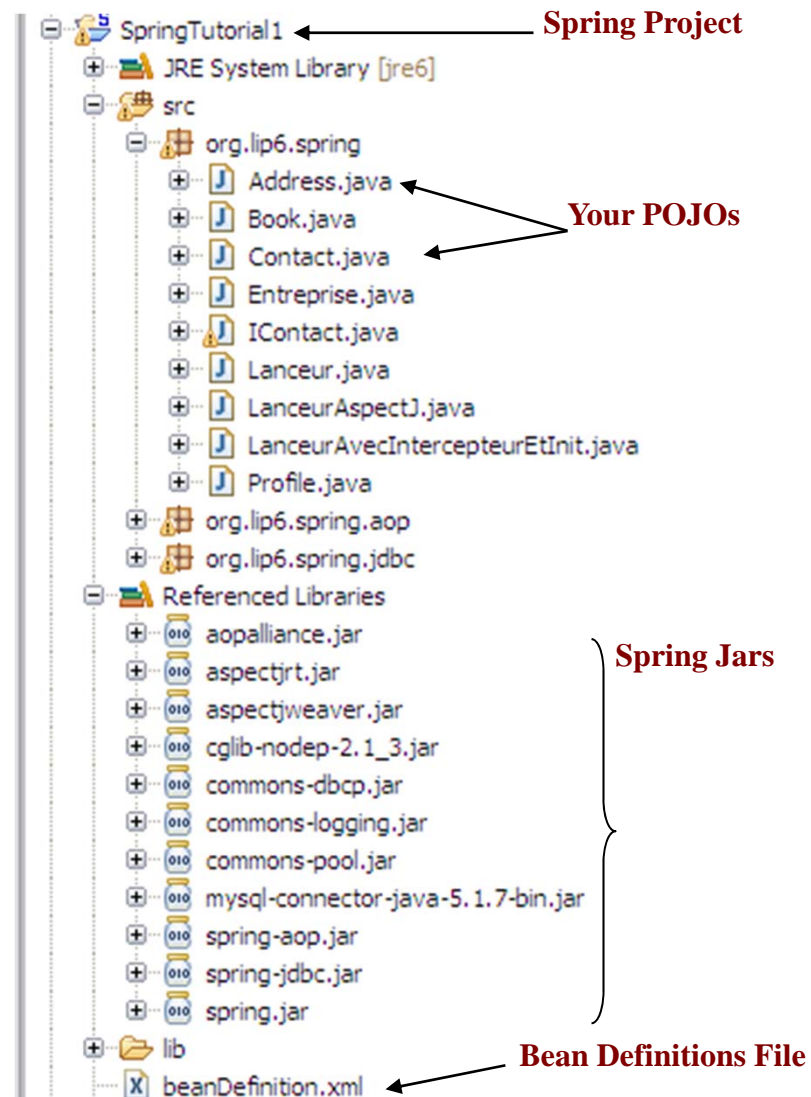
```
builder.addConstructorArg("Hello world");
```

```
// saving the new bean in the factory context
```

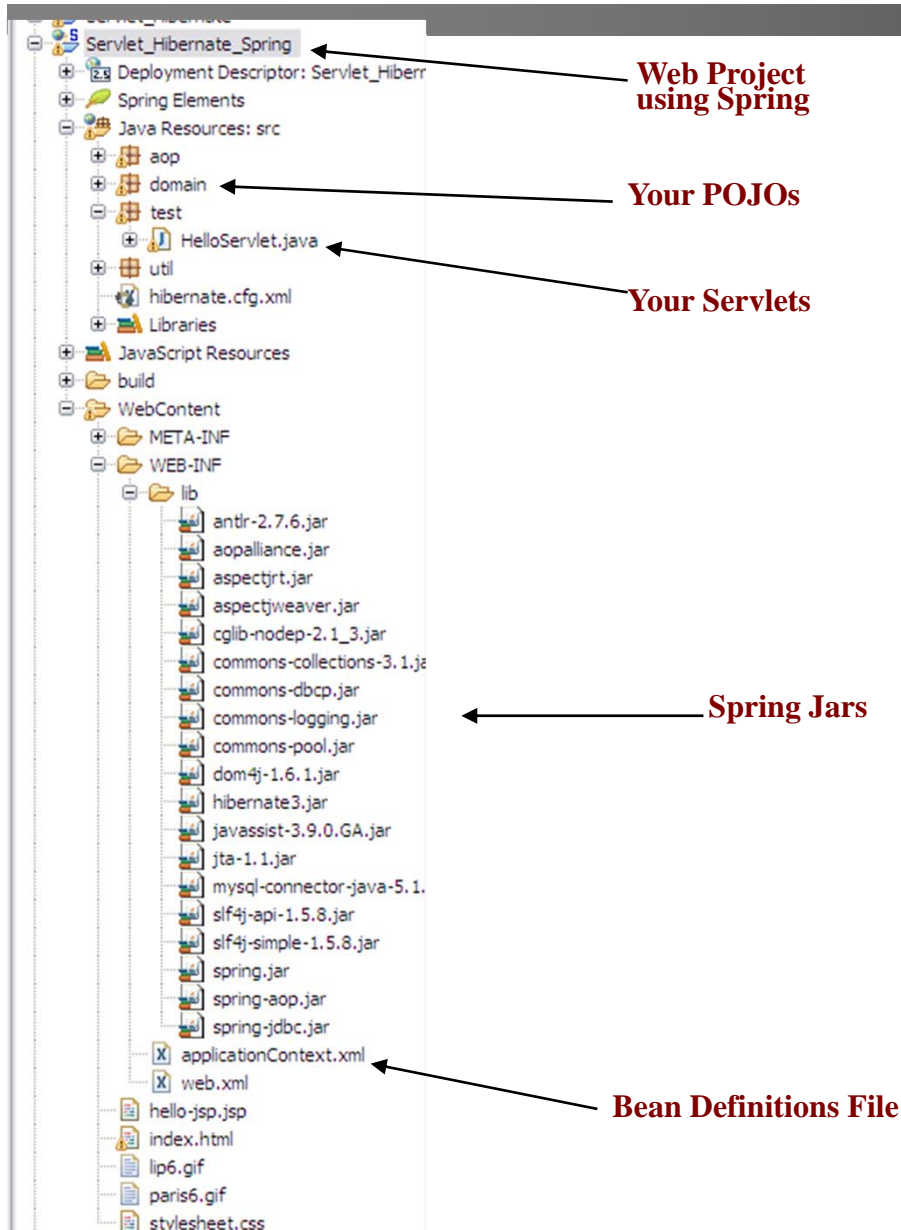
```
factory.registerBeanDefinition("bean1", builder.getBeanDefinition());
```

```
System.out.println(ctx.getBean("bean1"));
```

Example of a Spring project: Packaging



Example of a Web Project using Spring : Packaging



Spring Framework

AOP (Aspect Oriented Programming)

AOP

Introduction to AOP

AspectJ AOP

Spring AOP

Integration of Spring with AspectJ

AOP: Introduction

Cross-cutting issues

- ✓ Functionalities which their implementation cross-cuts different modules
- ✓ Multiple examples: traces and logging, transaction management, security, caching, exception handling, performance monitoring

AOP

- ✓ A way of programming in order to handle cross-cutting concerns.

AOP: Basic Concepts

Joinpoint

- ✓ An Identifiable point in the execution of a program (Constructor, a method invocation, etc.). In Spring AOP it always represents the execution of a method

Pointcut

- ✓ A programmatic expression that selects the *Joinpoints* where to apply the aspect

Advice

- ✓ Code to be executed before/after/around a *Joinpoint* expressed by a *Pointcut*

Target

- ✓ The main class for which the aspects have to be woven. The caller does not see the difference between the target class and the proxy class (the proxy class implements the same interfaces as the target class)

Coupling of Spring and AspectJ

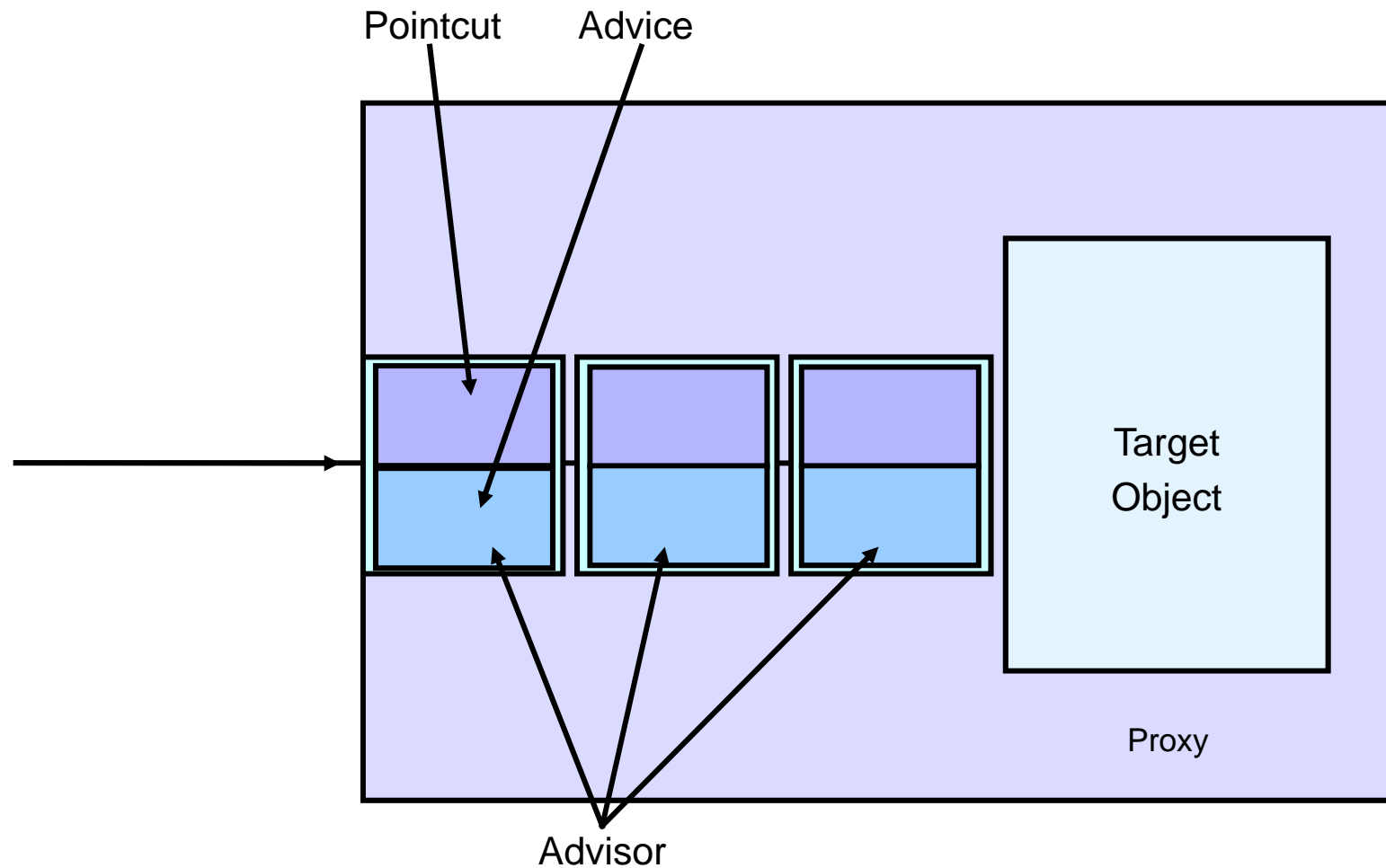
2 possible approaches :

- ✓ Declarative weaving of aspects: in the XML file
- ✓ Aspect weaving using annotations.

Aspects are applied :

- ✓ At compilation time : aspects + source -> class
- ✓ Over the binary files : aspects + binaries -> binaries
- ✓ At loading time : modification of classes when they are loaded in the JVM

Spring AOP



Weaving of Aspects: the Declarative Way

```
<beans>
  <aop:config>
    <aop:aspect id="emailLoggerBean" ref="emailerLogger">
      <aop:before
        pointcut="execution(* send(String, *, *, String))
                  and args(address, *, *, zipcode)"
        method="log" arg-names="zipcode, address"/>
    </aop:aspect>
  </aop:config>
  <bean id="emailerLogger" class="example.EmailLogger"/>
</beans>
```

Explanation: The *log* method is invoked *before* all the calls to methods called “*send*” and having 4 parameters with the first parameter and the forth one are Strings. The first and the forth parameters are forwarded to the *log* method

Weaving of Aspects: the Declarative Way

```
public class EmailLogger {  
    public void log(String zipcode, String address) {  
        System.out.println("hello");  
    }  
}
```

Weaving of Aspects: the Declarative Way

```
<beans>
  <aop:config>
    <aop:aspect id="emailLoggerBean" ref="emailerLogger">
      <aop:around
        pointcut="execution(* send(String, *, *, String))
                  and args(address, *, *, zipcode)"
        method="log" arg-names="pjp, zipcode, address"/>
    </aop:aspect>
  </aop:config>
  <bean id="emailerLogger" class="example.EmailLogger"/>
</beans>
```

Explanation: the *log* method is called *in place* of all the calls to methods called '*send*' and having 4 parameters with the first parameter and the forth one are Strings. The first and the forth parameters are forwarded to the *log* method with an additional parameter, the *pjp* (ProceedingJoinPoint) which represents the execution context

Weaving of Aspects: the Declarative Way

```
public class EmailLogger {  
    public void log(ProceedingJoinPoint pjp, String zipcode, String address) throws  
        Throwable {  
        if(Calendar.getInstance().get(Calendar.HOUR_OF_DAY)>12)  
        {  
            //Continue the normal execution  
            pjp.proceed();  
        }  
        else  
        {  
            // do nothing  
        }  
    }  
}
```

Definition of a Pointcut

When a method is executed :

execution(void Point.setX(int))

here the code of the class that contains the method is advised by an aspect

When a method is called :

call(void Point.setX(int))

here the code of the client that call the method is advised by an aspect

When the executing object ('this') is of a certain type

this(SomeType)

When the target of an object is of a certain type :

target(SomeType)

When the code being executed belongs to a class :

within(MyClass)

Weaving of Aspects: using annotations

@ AspectJ

```
package example;
```

```
@Aspect
```

```
public class EmailLogger {  
    @Before("execution(* send(String, *, *,String)) && args(address, *,*,zipcode)")  
    public void log(String zipcode, String address)  
    {  
    }  
}
```

The pointcut is written similarly as in the declarative weaving way

Tissage par annotations : @AspectJ

```
<beans ...>
```

```
  <aop:aspectj-autoproxy/>
```

```
  <bean id="emailerLogger"  
        class="example.EmailLogger"/>
```

```
  ... Other beans ...
```

```
</beans>
```

You must add in the XML file the line above even if you are using annotations

Spring AOP or AspectJ?

Use Spring AOP when:

- ✓ Method Interceptions is enough
- ✓ We don't want to use a specific compiler
- ✓ The pointcuts relate only to objects defined within the Spring context

Use AspectJ:

- ✓ For all the rest

Conclusions

Spring AOP is a simple yet very powerful mean to deal with cross-cutting concerns

AspectJ integration within Spring brings all the advantages of AOP in the same environment

The choice of using AOP or not can be taken at any time: not a strategic decision

Coupling Spring and Hibernate

The Hibernate framework

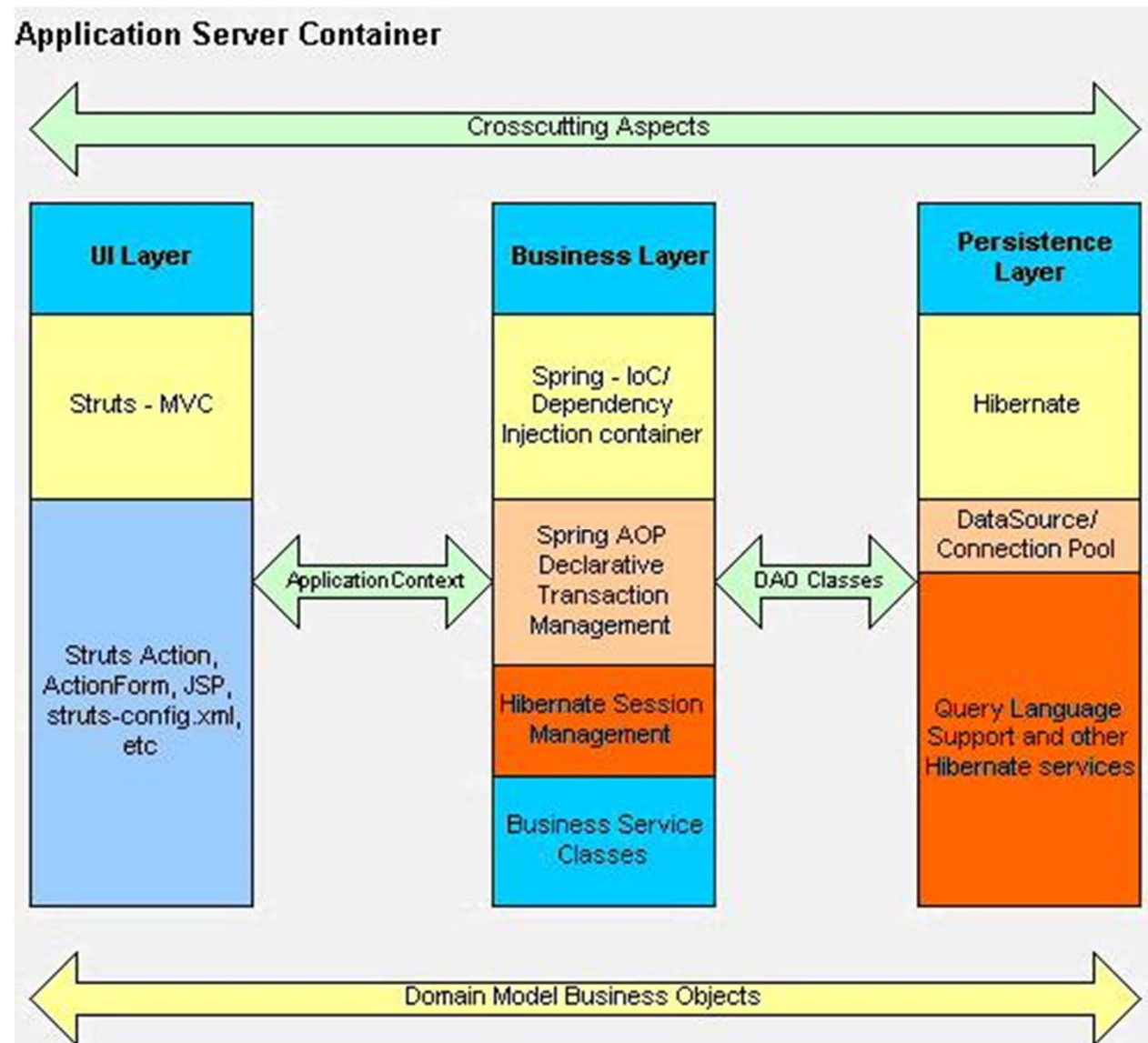
The Hibernate framework

- ✓ See the Hibernate course

Integration with Spring

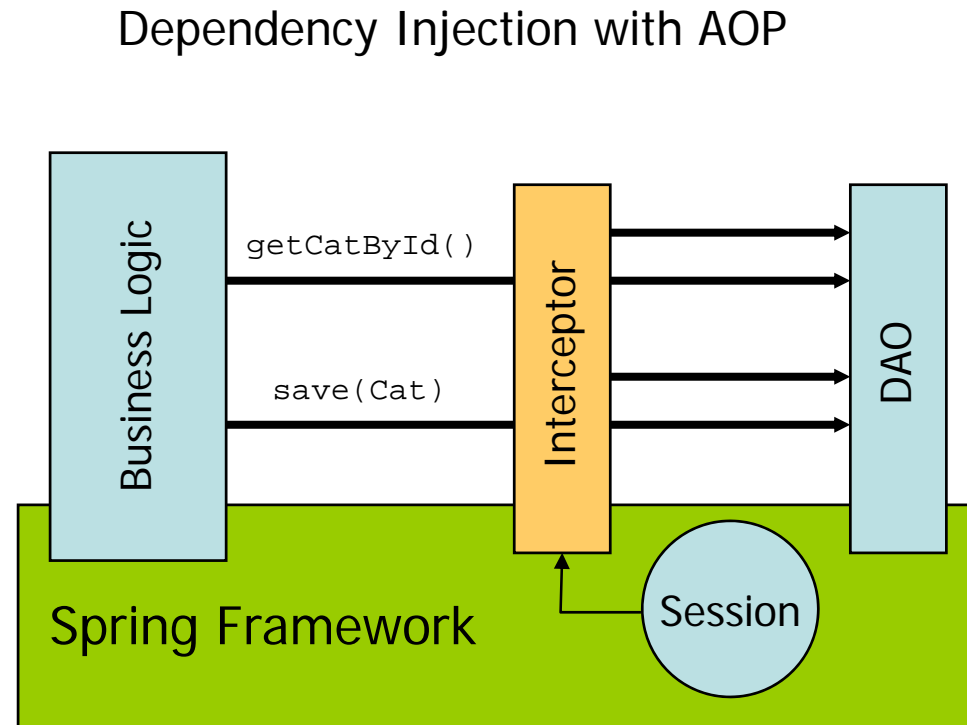
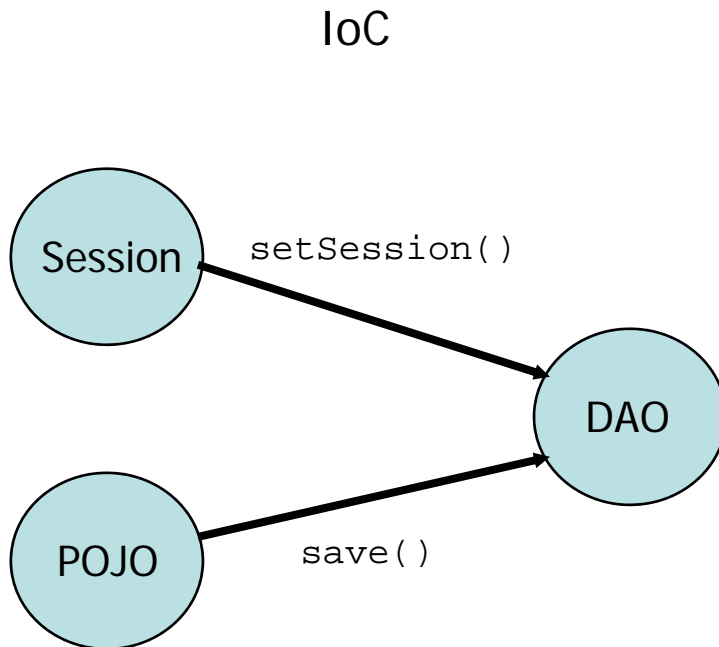
- ✓ Advantages
 - Resource management
 - Session management via IoC and AOP
 - Extended transaction management JTA and/or JDBC
 - Use of an Hibernate Template provided by Spring

Spring Integration with Hibernate



Spring Integration with Hibernate

Session management with IoC and POA



Spring Integration with Hibernate

Use of the Hibernate template

with Hibernate only

```
public User getUserByEmailAddress(final String email) {  
    try {  
        Session session = sessionFactory.openSession();  
        List list = session.find(  
            "from User user where user.email=?",  
            email,  
            Hibernate.STRING);  
  
        return (User) list.get(0);  
    }  
    finally {  
        session.close();  
    }  
}
```

```
public User getUserByEmailAddress(final String email) {  
    List list = getHibernateTemplate().find(  
        "from User user where user.email=?",  
        email,  
        Hibernate.STRING);  
    return (User) list.get(0);  
}
```

Using the
Hibernate
template within
Spring

Spring Integration with Hibernate

Setting up a datasource

```
<beans>
  <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:hsqldb://localhost:9001"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
  </bean>
</beans>
```

Spring Integration with Hibernate

Setting up a sessionFactory

```
<bean id="mySessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource" />
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop
key="hibernate.dialect">org.hibernate.dialect.HSQLDialect</prop>
        <props>
        </property>
      </props>
    </property>
  </bean>
```

Spring Integration with Hibernate

Setting up a transactionManager

```
<bean id="transactionManager"  
      class="org.springframework.orm.hibernate3.  
      HibernateTransactionManager">  
    <property name="sessionFactory" ref="mySessionFactory"/>  
  </bean>
```

Spring Integration with Hibernate

Your DAO class must extends the `HibernateDaoSupport` class

- ✓ Contains the methods: `get/setSessionFactory`
- ✓ Then, you can use the *`getHibernateTemplate()`* method within your DAO class in order to bring Hibernate facilities to you class/application
- ✓ The 'HibernateTemplate' allows you to access all Hibernate functions (find, delete, save, update, get...) without manipulating sessions, transactions, etc.

```
<beans>
  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>
</beans>
```

Spring Integration with Hibernate

Using the `HibernateTemplate`

```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao{  
    public Collection loadProductsByCategory(String category) throws  
                                   DataAccessException{  
        return this.getHibernateTemplate().find("from test.Product product  
        where product.category = ?",category);  
    }  
}
```

Spring Integration with Hibernate

Using the HibernateTemplate

```
public class ProductDaoImpl implements ProductDao{
    private HibernateTemplate hibernateTemplate;
    public void setHibernateTemplate(SessionFactory sessionFactory){
        this.hibernateTemplate = new HibernateTemplate(sessionFactory);
    }
    public Collection loadProductsByCategory(final String category) {
        return (Collection)this.hibernateTemplate.execute(new HibernateCallback(){
            public Object doInHibernate(Session session) throws
                HibernateException{
                Query query = session.createQuery("from test.Product product where
                                                product.category=?");

                query.setString(0,category);
                query.setFirstResult(0).setMaxResults(10);
                return query.list();
            }
        });
    }
}
```

Declarative management of transactions

Handling transactions by Spring supposes that you declare a bean 'transactionManager'. Many types are provided :

HibernateTransactionManager

JpaTransactionManager

JdoTransactionManager

TopLinkTransactionManager

JtaTransactionManager

Declarative management of transactions

Using an aspect

Setting up transaction management using AOP requires 2 steps:

- Definition of a transactional strategy (tx:advice)
- Application of this strategy to a set of joinpoints

```
<tx:advice id="txAdvice">
  <tx:attributes>
    <tx:method name="get*" read-only="true" timeout="-1" />
    <tx:method name="sav*" propagation="REQUIRED" />
    <tx:method name="find*" read-only="true" />
    <tx:method name="*" propagation="REQUIRED" timeout="1"/>
  </tx:attributes>
</tx:advice>

<aop:config>
  <aop:advisor pointcut="execution(* eg.domain.service.*(..))" advice-
    ref="txAdvice"/>
</aop:config>
```

Important: this is a standard block, just use it as it is. Just modify the eg.domain.service line to point the package where you have your DAOs

Declarative management of transactions

Using annotation

Setting up transaction management using annotations is ensured by adding the annotation `@Transactional` over every method that requires a transactional context

Advantage : readable code

Inconvenient : dependency of your classes towards spring, becomes a burden in case of large number of methods.

To-Do list for Hibernate Integration with Spring

Prepare a datasource bean

Prepare a sessionFactory (or entityManagerFactory in case of JPA) bean that references the datasource bean

Prepare a transactionManager bean that references the sessionFactory bean (or entityManagerFactory)

Declare the AOP part that handles transactions within the XML file or by using annotations inside your code i.e. @Transactional

Readings/Sources

- The Spring specification : <https://www.springsource.org/>
- Book: Spring par la pratique: J. Dubois, et al., Eyrolles, 2007
- Book: Spring in Action, Craig Walls, 2008, Manning publications
- H. Ouahidi Courses, UniConsulting (slides in French not provided online)