Systems for Design and Implementation

2015-2016

Course 4

Contents

- RMI in Java
- Introduction to Spring Framework
- Spring Remoting

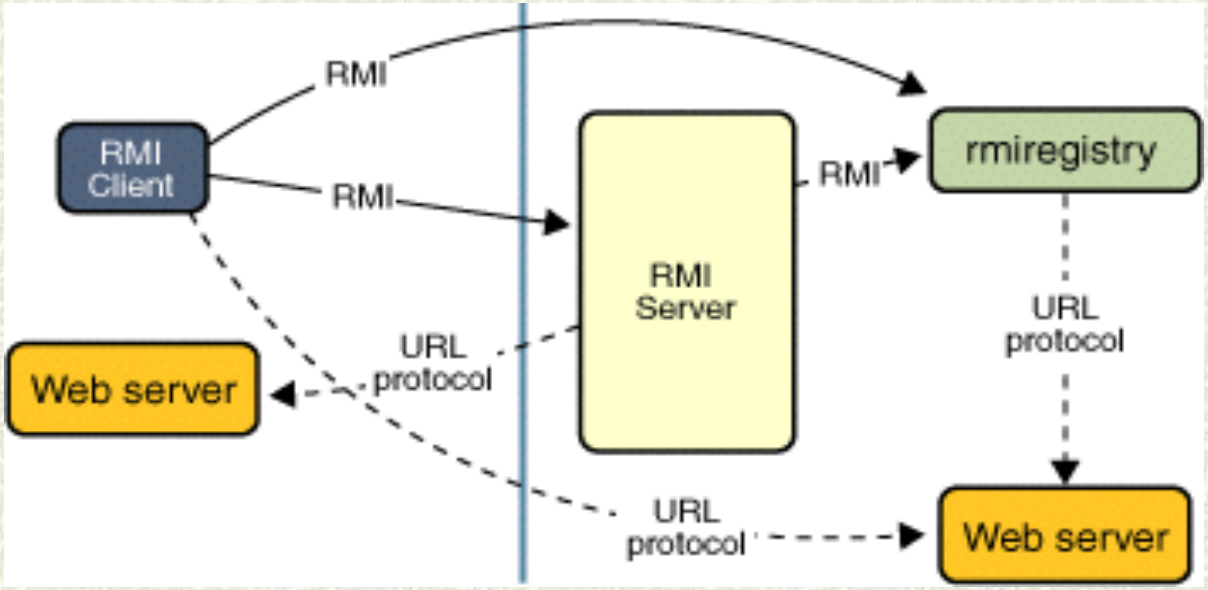
Java RMI

- The Java Remote Method Invocation (RMI) technology allows an object running in one Java virtual machine to invoke methods on an object running in another Java virtual machine.
- RMI applications often comprise two separate programs, a *server* and a *client*.
  - A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects.
  - A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them.
- RMI provides the mechanism by which the server and the client communicate and pass information back and forth.

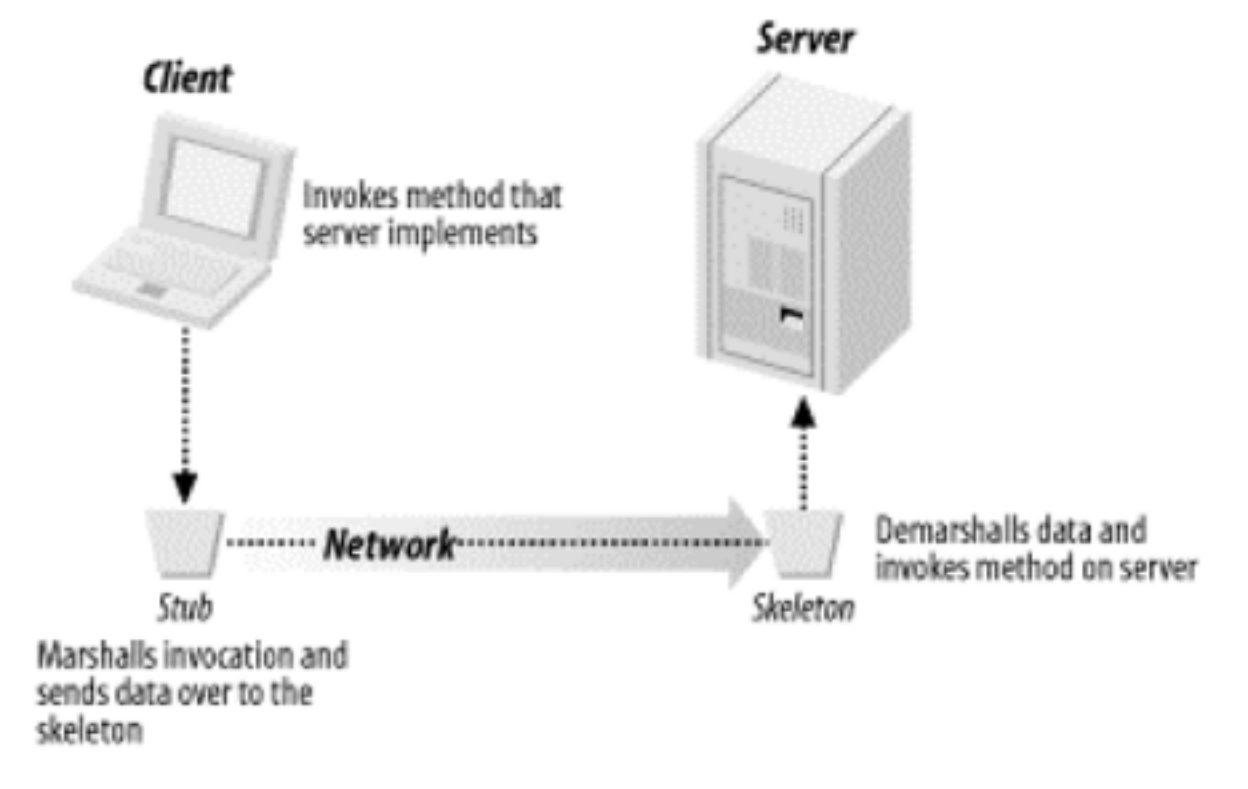
Java RMI

- *Locate remote objects.* Applications can use various mechanisms to obtain references to remote objects:
  - an application can register its remote objects with RMI's simple naming facility, the RMI registry.
  - an application can pass and return remote object references as part of other remote invocations.
- *Communicate with remote objects.* Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.
- *Load class definitions for objects that are passed around.* RMI enables objects to be passed back and forth, and it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

Java RMI


RMI
RMI Client
RMI
RMI Server
RMI
rmiregistry
URL protocol
URL protocol
Web server
URL protocol
Web server

Java RMI


Client
Server
Invokes method that server implements
Network
Stub
Skeleton
Demarshalls data and invokes method on server
Marshalls invocation and sends data over to the skeleton

RMI

- *Remote objects*: objects with methods that can be invoked across Java virtual machines. An object becomes remote by implementing a remote interface, that has the following characteristics:
  - A remote interface extends the interface **`java.rmi.Remote.`**
  - Each method of the interface declares **`java.rmi.RemoteException`** in its throws clause, in addition to any application-specific exceptions.

```
import java.rmi.*;
public interface IServer extends Remote{
    public void SomeObject method(Param1 p1, Param2 p2,…) throws
  RemoteException;
}
```

- *Locale objects* (non-remote objects) that are used in remote method calls (parameters or return value) must be serializable.

Architecture of a RMI based application

- *Shared Library (Remote interfaces).* A remote interface specifies the methods that can be invoked remotely by a client. The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods.
- *Server* (*Implementation of remote objects).* Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces and methods that are available only locally.
- *Client(s) (Implementation of clients).* Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

`java.rmi` package

- `Remote` interface: serves to identify interfaces whose methods may be invoked from a non-local virtual machine. Any object that is a remote object must directly or indirectly implement this interface. Only those methods specified in a *remote interface* are available remotely.
- `Naming` class: provides static methods for storing and obtaining references to remote objects in a remote object registry. Each method of the `Naming` class takes as one of its arguments a name that is a String in URL format of the form:

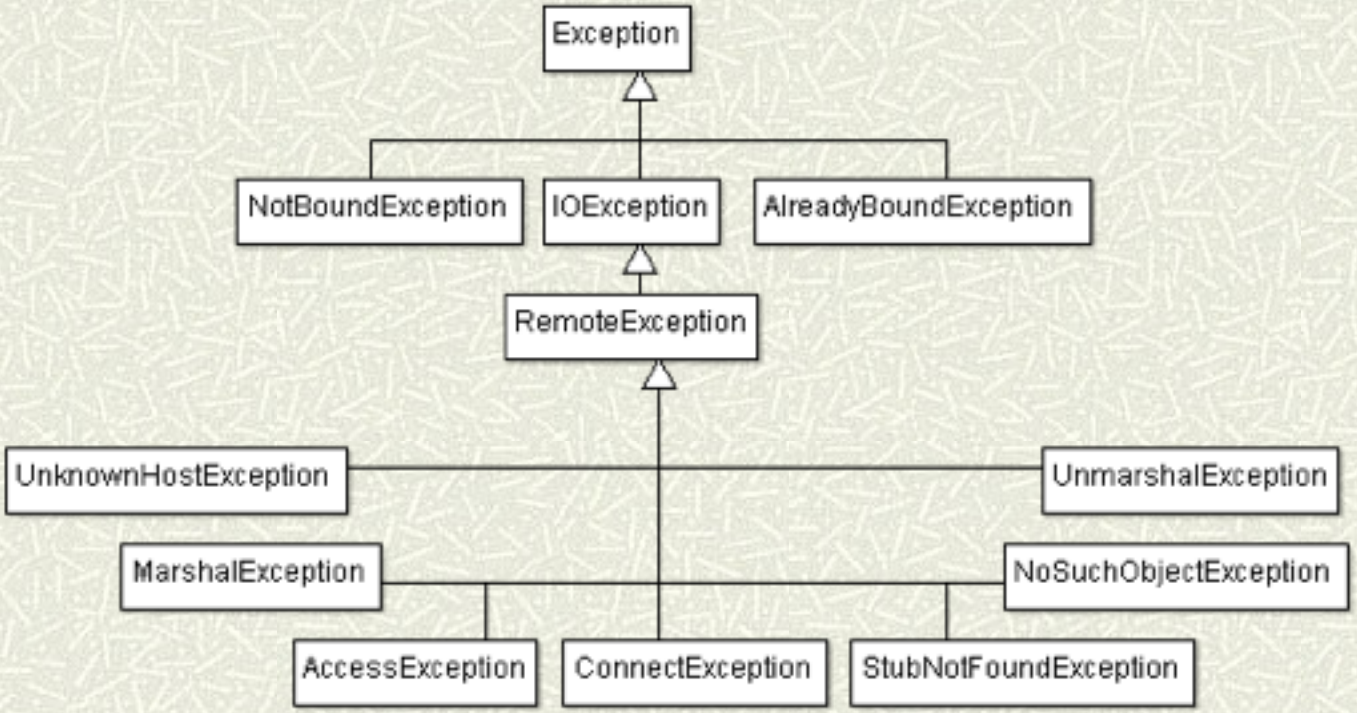
  `//host:port/name`

  - `bind(String name, Remote obj)` binds the specified name to a remote object.
  - `lookup(String name):Remote` returns a reference, a stub, for the remote object associated with the specified name.
  - `rebind(String name, Remote obj)` rebinds the specified name to a new remote object.
  - `unbind(String name)` destroys the binding for the specified name that is associated with a remote object.

java.rmi package

- **`RMISecurityManager`** class: A subclass of **`SecurityManager`** used by RMI applications that use downloaded code. RMI's class loader will not download any classes from remote locations if no security manager has been set.

```
System.setSecurityManager(new SecurityManager());
```



java.rmi.registry package

- A registry is a remote object that maps names to remote objects.
- A server registers its remote objects with the registry so that they can be looked up.
- When an object wants to invoke a method on a remote object, it must first lookup the remote object using its name. The registry returns to the calling object a reference to the remote object, on which a remote method can be invoked.
  - **`Registry`** interface: is a remote interface to a simple remote object registry that provides methods for storing and retrieving remote object references bound with arbitrary string names.
    - **`bind`**, **`lookup`**, **`rebind`**, **`unbind`**
  - **`LocateRegistry`** class is used to obtain a reference to a remote object registry on a particular host (including the local host), or to create a remote object registry that accepts calls on a specific port.

RMI Example

- Define the remote interface:

```
package rmi.services;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ITransformer extends Remote {
    String transform(String text) throws RemoteException;
}
```

RMI Example

- Implement the remote interface:

```
package rmi.server;

import rmi.services.ITransformer;
import java.rmi.RemoteException;
import java.util.Date;

public class TransformerImpl implements ITransformer{
    public String transform(String text) throws RemoteException {
        System.out.println("Method called "+text);
        return text.toUpperCase()+(new Date());
    }
}

//for dynamically stub creation
//public class TransformerImpl extends UnicastRemoteObject
                                    implements ITransformer
```

RMI Example

- Create the jar archive containing the classes and interfaces that must be available to both the server and the client:

```
jar cvf services.jar rmi/services/ITransformer.class
```

- Compile the remote interface implementation using the above created jar file
- Create the server stub, using `rmic:`

```
rmic -classpath services.jar rmi.server.TransformerImpl
```

A file called `rmi.server.TransformerImpl_stub.class` is created.

- Create a server policy file:

```
grant {
    permission java.security.AllPermission;
    permission java.net.SocketPermission "127.0.0.1:1024-",
   "connect,resolve";
};
```

RMI Example

- Create the class that starts the server:

```
public class StartServer {
    public static void main(String[] args) {
       if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Transformer";
            ITransformer engine = new TransformerImpl();
            ITransformer stub =(ITransformer)
  UnicastRemoteObject.exportObject(engine, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(name, stub);
            System.out.println("Transformer bound");
        } catch (Exception e) {
            System.err.println("Transformer exception:"+e);
        }
  }
}
```

RMI Example

- Starting the server:
  - Start the RMI registry:
    - **`Start rmiregistry`** (Windows)
    - **`Rmiregistry`** (Linux, Mac OS)
  - Copy the (stub) class, and the shared jar in a publicly available place:
    - http://scs.ubbcluj.ro/~xyz/rmi
    - c:/temp/rmi/transformer/server/
  - Start the server with the following command:

```
java -cp .;<path to>services.jar
-Djava.rmi.server.codebase=file:/c:/temp/rmi/transformer/server/
-Djava.rmi.server.hostname=localhost
-Djava.security.policy=server.policy
 StartServer
```

RMI Example

- Create the client

```
public class StartClient {
  public static void main(String args[]) {
    if (System.getSecurityManager() == null)
            System.setSecurityManager(new SecurityManager());
    try {
        String name = "Transformer";
        Registry registry = LocateRegistry.getRegistry("localhost");
        ITransformer comp = (ITransformer) registry.lookup(name);
        String text="Ana are mere.";
        String resp=comp.transform(text);
        System.out.println("Result: "+resp);
    } catch (RemoteException e) {…}
     catch (NotBoundException e) {…}
     }
}
```

RMI Example

- Create the client policy file:

```
grant {
    permission java.security.AllPermission;
    permission java.net.SocketPermission "127.0.0.1:1024-",
   "connect,resolve";
};
```

- Start the client

```
java -cp .;<path to>services.jar
-Djava.security.policy=client.policy StartClient
```

Introduction to Spring - Motivation

- Any nontrivial application is made up of two or more classes that collaborate with each other to perform some business logic. Traditionally, each object is responsible for obtaining its own references to the objects it collaborates with (its dependencies). This can lead to highly coupled and hard-to-test code.
- The traditional approach to creating associations between application objects (via construction or lookup) leads to complicated code that is difficult to reuse and unit test.

//version 1

```
class Contest{
   private ParticipantsRepositoryMock repo;
  public Contest(){
     repo=new ParticipantsRepositoryMock();
  }
  //...
}.
```

Introduction to Spring

//version 2

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

//version 2a

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

//version 3

```
class Contest{
   private ParticipantsRepositoryJdbc repo;
  public Contest(){
     Properties props=...
     repo=new ParticipantsRepositoryJdbc(props);
  }
}
```

Beans

- Any Java class is a POJO (Plain Old Java Object).
- JavaBeans: is a special Java class. Requirements:
  - It must have a public default constructor. Other tools will use this constructor to instantiate an object.
  - Its attributes must be accesable using methods called `getXyz`, `setXyz` and `isXyz` (for boolean attributes). Attributes that have these kind of methods are called properties, and the name of the property is `xyz`. When you want to modify/obtain the value of a property, you call one of the corresponding methods (get/set).
  - The class must be serializable (it implements java.io.Serializable interface). Other tools can save/restore the state of a bean between executions.
  - Example: GUI components
- Enterprise Java Beans (EJBs): for complex applications (transactions, security, database access)

Java Bean Example

```
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;}

}
```

Introduction to Spring

- Spring is an open-source framework initially created by Rod Johnson and presented in his book, *Expert One-on-One: J2EE Design and Development*.
- It was designed to ease the development of complex and very large applications.
- Any simple Java objects (POJOs) can be used with Spring to build systems that previously could be built only using EJB.
- A Spring bean is any Java class.
- Spring promotes low coupling through dependency injection and programming to interfaces.
- Spring uses Inversion of Control (IoC) principle to "inject" object dependencies.

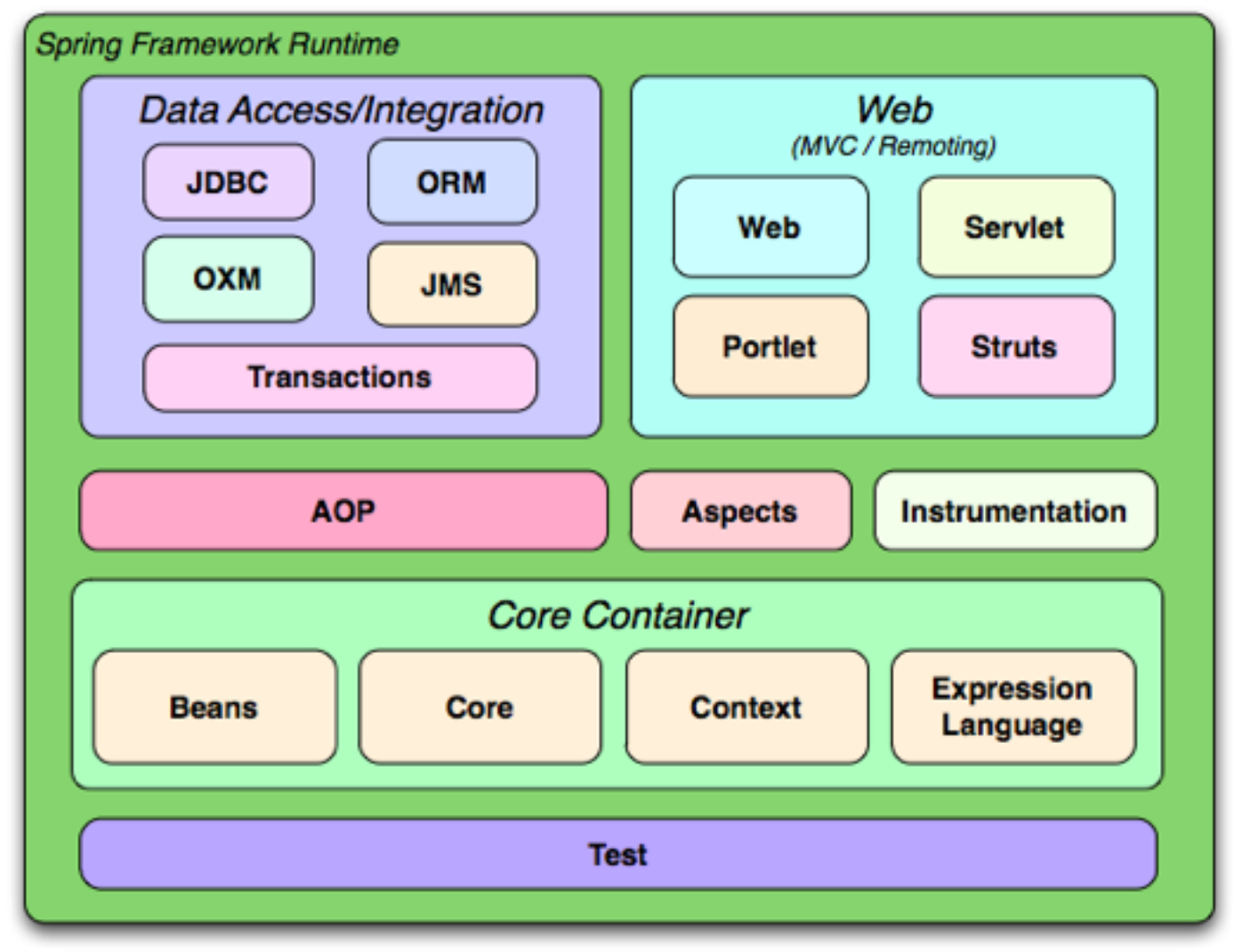
Introduction to Spring

```
public interface ParticipantsRepository{...}
class Contest{
   private ParticipantsRepository repo;
  public Contest(ParticipantsRepository r){
     repo=r;
  }
  ...
}
//or
 class Contest{
  private ParticipantsRepository repo;
  public Contest(){... }
  public void setParticipants(ParticipantsRepository r){repo=r;}
}
public class ParticipantsRepositoryFile implements
    ParticipantsRepository{...}
public class ParticipantsRepositoryJdbc implements
    ParticipantsRepository{...}
```

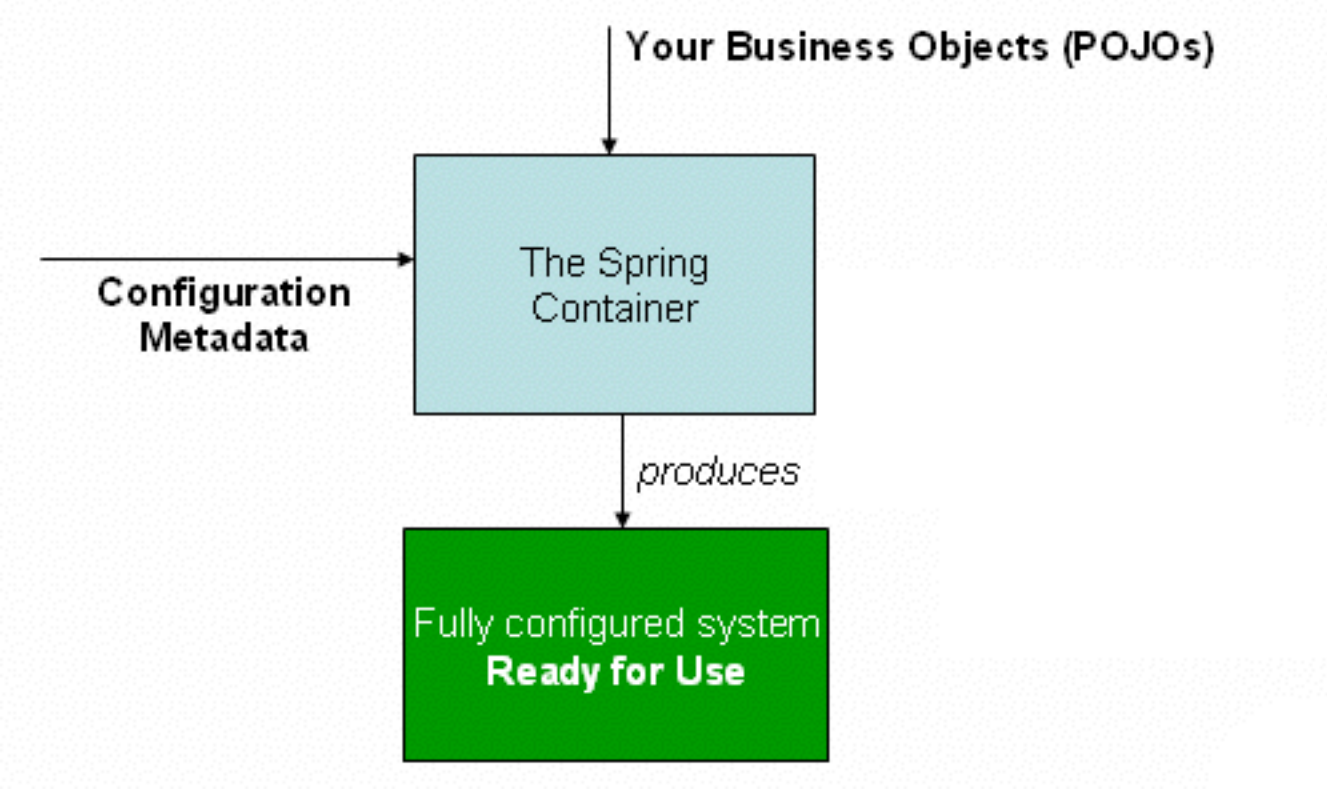
IoC, Dependecy Injection

- The *Inversion of Control* (IoC) principle is also known as *dependency injection* (DI).
- It is a process where objects define their dependencies (the other objects they work with) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method.
- The container injects those dependencies when it creates the bean.
- This process is fundamentally the inverse (the name Inversion of Control-IoC) of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes.
- In Spring, the objects that form the backbone of the application and that are managed by the Spring IoC container are called *beans*.
- A *bean* is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.
- Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.
- There are two ways to configure beans in the Spring container: using XML files or Java-based configuration.

Overview of the Spring Framework



The Spring IoC container



Instantiating a Spring container

```
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-contest.xml");

//obtaining a reference to a bean from the container
Contest contest= (Contest)factory.getBean("contest");
  }
}
```

XML Configuration File

- When declaring beans in XML, the root element of the Spring configuration file is the `<beans>` element from Spring's beans schema.
- A typical Spring configuration XML file looks like this

```
<?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-3.0.xsd">

<!- Beans declaration-->

</beans>
```

- Within the `<beans>` you can place all of your Spring configuration, including `<bean>` declarations.

Declaring a simple bean

```
package pizzax.validation;
import pizzax.model.Pizza;
public class DefaultPizzaValidator implements Validator<Pizza> {
    public void validate(Pizza pizza, Errors errors) {
      //...
    }
}
//spring-pizza.xml
<beans ...>
  <bean id="pizzaValidator"
    class="pizzax.validation.DefaultPizzaValidator"/>
</beans>
```

- The `<bean>` element is the most basic configuration unit in Spring. It tells Spring to create an object.
- The `id` attribute gives the bean a name by which it will be referred to in the Spring container. When the Spring container loads its beans, it will instantiate the `"pizzaValidator"` bean using the default constructor.

DI using constructors

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

DI using constructors(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 using constructors(3)

```
public class Product {
    private String name="";
    private double price=0;
    public Product(String name, double price) {
        this.price = price;
        this.name = name;
    }
    //...
}
//spring-exemplu.xml
 <bean id="mere" class="Product">
        <constructor-arg index="0" value="Mere" />
        <constructor-arg index="1" value="3.14"/>
</bean>
<!--or -->
 <bean id="mere" class="Product">
        <constructor-arg type="java.lang.String" value="Mere" />
        <constructor-arg type="double" value="3.14"/>
  </bean>
```

DI – Factory Methods

```
public class A {
   private static A instance;
   private A(){...};
   public static A getInstance(){ ...}
   ...
}
//spring-exemplu.xml
 ...
<bean id="instanta" class="A" factory-method="getInstance"/ >
//same as
A objA=A.getInstance();
```

Scope

- By default, all Spring beans are *singletons* (only one instance of a bean is created, independent of the number of times it is used in the configuration file or using `getBean()` method from `ApplicationContext` class).
- You declare the scope under which beans are created using the `scope` attribute.

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

- Possible values for `scope` attribute:
  - `singleton`: a single instance per Spring container.
  - `prototype`: Allows a bean to be instantiated any number of times (once per use).
  - `request, session, global-session`: for Web applications .

```
<bean id="beanA" class="test.B" scope="prototype"/>
<bean id="beanB" class="test.B"/>
<bean id="beanC" class="test.C">
  <constructor-arg ref="beanA"/> <!–new instance --></bean>
<bean id="beanD" class="test.D">
  <constructor-arg ref="beanA"/> <!–new instance --> </bean>
```

DI using properties

```
package pizzax.repository.file;
import pizzax.repository;
public class PizzaRepositoryFile implements PizzaRepository {
   private String numefis;
   public PizzaRepositoryFile() {  ...   }
   public void setFileName(String numefis}{...}
   //methods definition
   ...
}
//spring-pizza.xml
 ...
<bean id="pizzaRepository"
    class="pizzax.repository.file.PizzaRepositoryFile">
    <property name="fileName" value="Pizza.txt"/>
</bean>
```

DI using properties

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

DI constructors + properties

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

Inner Beans

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

Inner Beans

Remarks:

- Inner beans do not have an `id` attribute set. Though it is allowed to declare an ID for an inner bean, it is not necessary because you will never refer to the inner bean by name.
- They cannot be reused. Inner beans are only useful for injection once and cannot be referred to by other beans.

Wiring Collections

- There are situations when a property is a (data structure) container (collection, set, dictionary, array, etc...).
- Spring offers four types of collection configuration elements that are useful when configuring collections of values.
  - `<list>`: Wiring a list of values, allowing duplicates
  - `<set>`: Wiring a set of values, ensuring no duplicates
  - `<map>`: Wiring a collection of name-value pairs where name and value can be of any type
  - `<props>`: Wiring a collection of name-value pairs where the name and value are both Strings

Wiring Collections

- Lists, sets, arrays:

```
class Product{
   private String name;
  private double price;
  public Product(){...}
  public void setName(String d){...}
  public void setPrice(double d){...}

  //get and set methods

}
class Warehouse{
  //...
  public void setProducts(java.util.List<Product> lp){...}
  //or
  public void setProducts(java.util.Collection<Product> lp){...}
  //or
  public void setProducts(Product[] lp){...}
}
```

Wiring Collections

- Lists, arrays:

```
//spring-exemplu.xml
<bean id="mere" class="Product">
    <property name="name" value="Mere"/>
    <property name="price" value="2.3"/>
</bean>
<bean id="pere" class="Product"> ...</bean>
<bean id="prune" class="Product"> ...</bean>
<bean id="depozit" class="Warehouse">
  <property name="products">
    <list>
        <ref bean="mere"/>
        <ref bean="pere"/>
        <ref bean="prune"/>
    </list>
  </property>
</bean>
```

Wiring Collections

- Sets:

```
//spring-exemplu.xml
<bean id="mere" class="Product">
    <property name="name" value="Mere"/>
    <property name="price" value="2.3"/>
</bean>
<bean id="pere" class="Product"> ...</bean>
<bean id="prune" class="Product"> ...</bean>
<bean id="depozit" class="Warehouse">
  <property name="products">
    <set>
        <ref bean="mere"/>
        <ref bean="pere"/>
        <ref bean="prune"/>
        <ref bean="prune"/>
    </set>
  </property>
</bean>
```

Wiring Collections

- Map(Dictionary):

```
class Warehouse{
  //...
  public void setProducts(java.util.Map<String, Product> lp){...}
}
//spring-exemplu.xml
<bean id="mere" class="Product">...</bean>
<bean id="pere" class="Product"> ...</bean>
<bean id="prune" class="Product"> ...</bean>
<bean id="depozit" class="Warehouse">
  <property name="products">
    <map>
        <entry key="pMere" value-ref="mere"/>
        <entry key="pPere" value-ref="pere"/>
        <entry key="pPrune" value-ref="prune"/>
    </map>
  </property>
</bean>
```

Wiring Collections

- Map: the `<entry>` element has the following attributes:
  - `key`: Specifies the key of the map entry as a String;
  - `key-ref`: Specifies the key of the map entry as a reference to a bean in the Spring context;
  - `value`: Specifies the value of the map entry as a String;
  - `value-ref`: Specifies the value of the map entry as a reference to a bean in the Spring context.
- Properties: `props` and `prop` elements

```
class Warehouse{
    public void setProperties(Properties p){...}
}
<bean id="depozit" class="Warehouse">
  <property name="properties">
    <props>
      <prop key="prop1"> A1 </prop>
      <prop key="prop2"> B C2 </prop>
    </props>
  </property>
</bean>
```

Null values

- It is possible to set the value of a property to `null`, by using the `<null/>` element:

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

- SpEL (Spring Expression Language)
  - It was introduced starting from version 3.0
  - It allows computing and obtaining the value of some properties dynamically during container initialization.

```
<property name="randomNumber" value="#{T(java.lang.Math).random()}"/>
<property name="song" value="#{songSelector.selectSong().toUpperCase()}"/>
<property name="fullName"
     value="#{person.firstName + ' ' + person.lastName}"/>
```