Aspect Oriented Programming

2013-2014

Course 2

Course 2 Contents

- JDBC
- Introduction to XML
- Introduction to Spring
- Spring JDBC

JDBC

- The Java Database Connectivity (JDBC) API provides universal data access from the Java programming language.
- Virtually any data source can be accessed, from relational databases to spreadsheets and flat files.
- JDBC technology also provides a common base on which tools and alternate interfaces can be built.
- Packages:
  - `java.sql` provides the API for accessing and processing data stored in a data source (usually a relational database).
  - `javax.sql` - adds server-side capabilities

Establishing a Connection

- An application connects to a target data source using one of two mechanisms:
  - **`DriverManager`**: This fully implemented class requires an application to load a specific driver, using a hardcoded URL.
  - **`DataSource`**: This interface is preferred over **`DriverManager`** because it allows details about the underlying data source to be transparent to the application. A **`DataSource`** object's properties are set so that it represents a particular data source.
- Establishing a connection involves two steps:
  - Loading the driver

```
Class.forName(<DriverClassName>);
```

    - **`Class`**.**`forName`** automatically creates an instance of a driver and registers it with the **`DriverManager`**
    - You do not need to explicitly create an instance of the class.
  - Making the connection.

Making a connection

- DriverManager class:
  - The DriverManager class collaborates with the Driver interface to manage the set of drivers available to a JDBC client.
  - When the client requests a connection and provides a URL, the DriverManager is responsible for finding a driver that recognizes the URL and for using it to connect to the corresponding data source.
  - Connection URLs have the following form:

```
jdbc:subprotocol:<dbName>[propertyList]
Connection conn = DriverManager.getConnection("jdbc:derby:COFFEES");

String url = "jdbc:mysql:Fred";
Connection con = DriverManager.getConnection(url, <user>, <passwd>);
```

Making a connection

- DataSource:

```
InitialContext ic = new InitialContext()

 DataSource ds = ic.lookup("java:comp/env/jdbc/myDB");
 Connection con = ds.getConnection();

 DataSource ds = (DataSource)org.apache.derby.jdbc.ClientDataSource()
 ds.setPort(1527);
 ds.setHost("localhost");
 ds.setUser("APP")
 ds.setPassword("APP");

Connection con = ds.getConnection();
```

Connection

- Represents a session with a specific database.
- SQL statements are executed and results are returned within the context of a connection.
- Methods:
  - `close(), isClosed():boolean`
  - `createStatement(…):Statement  //overloaded`
  - `prepareCall(…):CallableStatement //overloaded`
  - `prepareStatement(…):PreparedStatement //overloaded`
  - `rollback()`
  - `setAutoCommit(boolean)`
  - `getAutoCommit():boolean`
  - `commit()`

Statement

- It is used for executing a static SQL statement and returning the results it produces.
- Methods:
  - `execute(sql:String, …):boolean`
    - `getResultSet():ResultSet`
    - `getUpdateCount():int`
  - `executeQuery(sql:String, …):ResultSet`
  - `executeUpdate(sql:String, …):int`
  - `cancel()`
  - `close()`

Statement example – table scheme

MS ACCESS database

	Field Name	Data Type
🔑	ID	AutoNumber
	title	Text
	authors	Text
	isbn	Text
	year	Number
▶		

Statement

```
Class.forName("sun.jdbc.odbc.jdbcOdbcDriver");
Connection conn=DriverManager.getConnection("jdbc:odbc:books");
//select
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from books");

//update
String upString="update books set isbn='tj234' where isbn='tj237'"
Statement stmt=conn.createStatement();
stmt.executeUpdate(upString);

//delete
String delString="delete from books where isbn='tj234'"
Statement stmt=conn.createStatement();
stmt.executeUpdate(delString);
```

ResultSet

- A table of data representing a database result set, which is usually generated by a executing a statement that queries the database.
- A ResultSet object maintains a cursor pointing to its current row of data.
- Initially the cursor is positioned before the first row.
- The **`next`** method moves the cursor to the next row. It returns false when there are no more rows in the ResultSet object.
- The next method is used to iterate through the rows.
- ResultSets can be scrollable.
- Its type is specified in the methods createStatement(…) or alike:

```
Statement stmt = con.createStatement(
                 ResultSet.TYPE_SCROLL_INSENSITIVE,
                 ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT name, address FROM users");
   // rs is scrollable, will not show changes made by others,
    // and it is updatable
```

ResultSet

- Methods:
  - `absolute(row:int)`
  - `relative(n:int)`
  - `afterLast(), beforeFirst() , first(), last(), next():boolean`
  - `getRow():int`
  - `getInt(columnIndex|columnLabel):int`
  - `getFloat(…), getString(…), getObject(…), etc`
  - `updateInt(columnIndex,columnLabel, newValue)`
  - `updateFloat(…), updateString(…), etc`
  - `updateRow()`
  - `refreshRow()`
  - `rowDeleted(), rowInserted(), rowUpdated()`
- Dy default, a `ResultSet` is one-directional, forward only cursor, and it is not updatable.

ResultSet

```
Statement stmt=conn.createStatement();
ResultSet rs=stmt.executeQuery("select * from books");
while(rs.next()){
  System.out.println("Book "+rs.getString("title")+'
   '+rs.getString("author")+' '+rs.getInt("year"));
}
rs.close();
stmt.close();
```

PreparedStatement

- A PreparedStatement, unlike a Statement object, receives an SQL statement when it is created.
- The SQL statement is sent to the DBMS right away, where it is compiled.
- When the PreparedStatement is executed, the DBMS can just run the PreparedStatement SQL statement without having to compile it first.
- It can have parameters. They are marked with '?'.

```
PreparedStatement preStmt = con.prepareStatement(
    "select * from books WHERE year=?");
```

- The value of a parameter is set using `setXYZ` method

```
preStmt.setInt(1, 2008);
ResultSet rs=preStmt.executeQuery();
```

Transactions

- By default, each individual SQL statement is treated as a transaction and is automatically committed right after it is executed.
- The default behaviour can be changed using `setAutoCommit(false)` from `Connection`.
- Methods:
  - `commit`
  - `rollback`
  - `setSavePoint`

Transactions

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Black");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
    "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Black");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

Properties

- The class **`java.util.Properties`** is used to store key-value pairs. The pairs can be read from a stream or saved to a stream (i.e. text file). The keys and the values are of type String, and the keys must be unique.

```
//example.properties
studentsFile=students.txt
resultsFile=results.txt
inputDir=input\\app
outputDir=output\\app

//Reading the properties file
Properties props=new Properties();
try {
    props.load(new FileInputStream("example.properties"));
} catch (IOException e) {
    System.out.println("Error: "+e);
}
```

Properties

- Methods:
  - `getProperty(key:String):String`
  - `setProperty(k:String, v:String):Object`
  - `list(PrintWriter)`
  - `load(Reader)`
  - `store(w:Writer, comments:String)`

```
Properties props=new Properties();
try {
    props.load(new FileInputStream("example.properties"));
} catch (IOException e) {
    System.out.println("Error: "+e);
}
String students=props.getProperty("studentsFile");
if (students==null)    //the property was not found
   System.out.println("Incorrect file");
...
```

System +Properties

- Methods from System class:
  - setProperties(Properties)
  - setProperty(k:String, v:String):String
  - getProperty(String):String
  - ...

```
Properties serverProps=new Properties(System.getProperties());
try {
     serverProps.load(new FileReader("example.properties"));
     System.setProperties(serverProps);
     System.getProperties().list(System.out);
} catch (IOException e) {
      System.out.println("Error "+e);
}
String students=System.getProperty("studentsFile");
```

Database configuration example

```
//bd.properties
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost/agenti
jdbc.user=test
jdbc.pass=test

Connection getNewConnection(){
        String driver=System.getProperty("jdbc.driver");
        String url=System.getProperty("jdbc.url");
        String user=System.getProperty("jdbc.user");
        String pass=System.getProperty("jdbc.pass");
        Connection con=null;
        try {
            Class.forName(driver);
             con= DriverManager.getConnection(url,user,pass);
        } catch (ClassNotFoundException e) {
            System.out.println("Error loading driver "+e);
        } catch (SQLException e) {
            System.out.println("Error establishing connection "+e);
        }
        return con;
 }
```

Extensible Markup Language (XML)

- It is a text-based markup language.
- It is cross-platform, extensible.
- It is used for representing data.
- It is not used in order to specify how to display the data (HTML).
- The data is identified using *tags*.
- XML tags identify the data.
- It is sometimes described as a mechanism for specifying the semantics (meaning) of the data.

XML Tags

- A tag is an identifier enclosed in angle brackets: <...>.
- Collectively, the tags are known as markup.
- The data between the tag and its matching end tag defines an element of the XML data.
- A tag may contain other tag(s).
- The data represented in an XML document have hierarchical structure.

```
<message>
  <sender> ana </sender>
  <receiver> mihai </receiver>
  <text> Hello! </text>
</message>
```

XML Attributes

- A tag can also contain attributes (additional information included as part of the tag itself, within the tag's angle brackets) .

```
<message sender="ana" receiver="mihai" >
        <text> Hello! </text>
</message>
```

- The attribute name is followed by an equal sign and the attribute value, and multiple attributes are separated by spaces (not by commas).

XML Empty tags, comments

- If a tag does not contain other tags, it is called an empty tag.

```
<message> </message>
<message />
<message type="X"/>
```

- XML comments

```
<message sender="ana" receiver="mihai">
    <!-- The text of the message -->
    <text> Hello! </text>
</message>
```

XML Prolog

- An XML file should always start with a prolog.
- The minimal prolog contains a declaration that identifies the document as an XML document:

```
<?xml version="1.0"?>
```

- The declaration may also contain additional information:

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

- The XML declaration may contain the following attributes:
  - **`version`**: Identifies the version of the XML markup language used in the data. This attribute is not optional.
  - **`encoding`**: Identifies the character set used to encode the data. The default is 8-bit Unicode: UTF-8.
  - **`standalone`**: Tells whether or not this document references an external entity or an external data type specification. If there are no external references, then "**`yes`**" is appropriate.
- Everything that comes after the XML prolog constitutes the document's content.

XML Special Characters

Character	Name	Reference
&	ampersand	&amp;
<	less than	&lt;
>	greater than	&gt;
"	quote	&quot;
'	apostrophe	&apos;

```
<message sender="ana" receiver="mihai">
        <!-- A < B -->
        <text> A &lt; B </text>
</message>
```

XML CDATA

- Text in a CDATA section is not parsed: the whitespaces are significant, and characters in it are not interpreted as XML.
- A CDATA section starts with <![CDATA[and ends with]]>.

```
<message sender="ana" receiver="mihai">
     <!-- The text of the message -->
     <text> <![CDATA[
     a+b >c
     "Ana are mere in panere!"
     >> <<
     ]]>
 </text>
</message>
```

Well formed XML documents

- Non-empty elements are delimited by both a start-tag and an end-tag.

```
<message> … </message>
<text> …</text->
```

- Empty elements may be marked with an empty-element (self-closing) tag, such as `<text />`. This is equal to `<text></text>`.
- All attribute values are quoted with either single (') or double (") quotes. Single quotes close a single quote and double quotes close a double quote.
- Tags may be nested but must not overlap. Each non-root element must be completely contained in another element.

```
<message> <text> </message> </text>
```

- The document complies with its declared character encoding.
- Element names are case-sensitive.

```
<message> ... </message>
<Message> ... </message>
<SENDER> ... </senDer>
```

XML Documents Validation

```
<message sender="ana" receiver="mihai">
        <text> Hello! </text>
</message>

<message>
        <text> Hello! </text>
        <sender> ana </sender>
        <receiver> mihai </receiver>
</message>
<message>
        <sender> ana </sender>
        <receiver> mihai </receiver>
        <text> Hello! </text>
</message>
```

DTD, XML Schema, RELAX NG

Document Type Definition (DTD)

- It is the oldest schema format for XML.
- A DTD specifies the kinds of tags that can be included in an XML document, and their order.

```
//message.dtd.
<?xml version="1.0"?>
<!ELEMENT message (sender, receiver, text)>
<!ELEMENT sender (#PCDATA)>
<!ELEMENT receiver (#PCDATA)>
<!ELEMENT text (#PCDATA)>
<message>
        <text> Hello! </text>
        <sender> ana </sender>
        <receiver> mihai </receiver>
</message>
```

XML Schema

- XML Schema is an XML-based language used to create XML-based languages and data models.
- An XML schema defines element and attribute names for a class of XML documents, the structure of the XML documents, and the type of content that each element can hold.
- XML documents that attempt to adhere to an XML schema are said to be instances of that schema.

```
<?xml version="1.0" encoding="utf-16"?>
<xsd:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
    version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="message">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="sender" type="xsd:string" />
        <xsd:element name="receiver" type="xsd:string" />
        <xsd:element name="text" type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

XML Parsing

- SAX, the Simple API for XML, is an event-driven parser.
  - It reads the XML document sequentially moving from one element to the next.
  - It cannot go back to already parsed elements.
  - It is a light-weight parser.
  - It cannot be used to modify the XML document.
- DOM Document Object Model
  - It describes an XML document as a tree-like structure, with every XML element being a node in the tree.
  - A DOM-based parser reads the entire document, and forms the corresponding document tree in memory.
  - It can be used to construct new XML documents or to modify existing XML documents.

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 - 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);
  }
}
```

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.

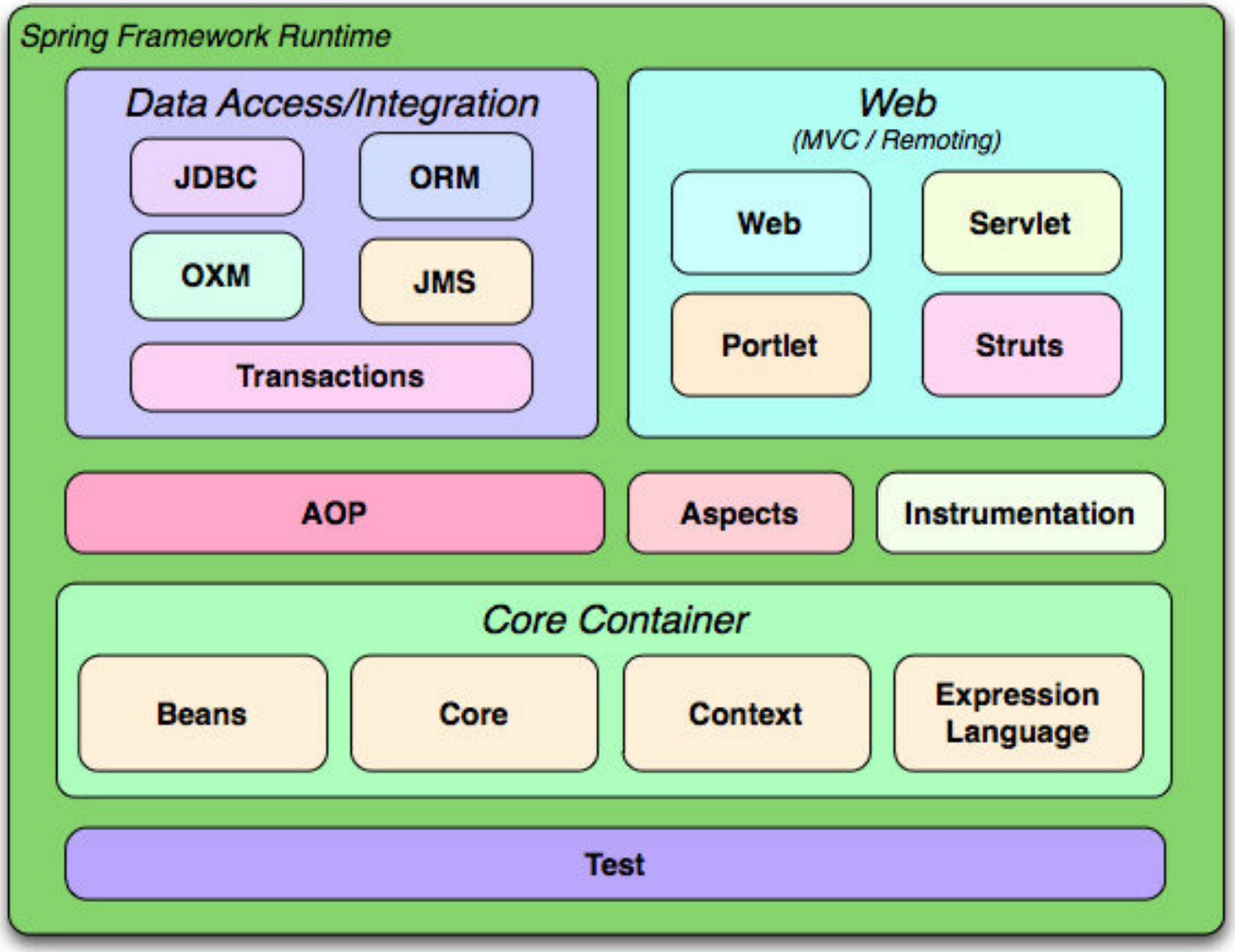
Introduction to Spring

```
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{...}
```

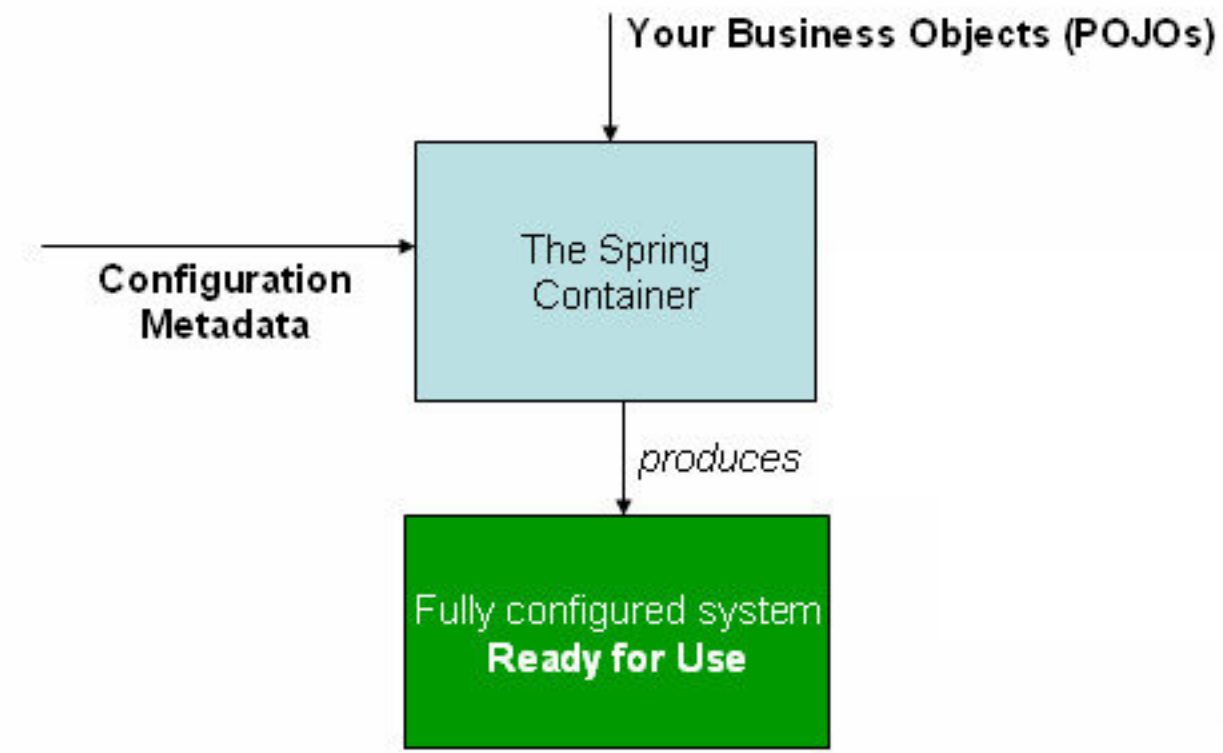
IoC, Dependecy Injection

- 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:

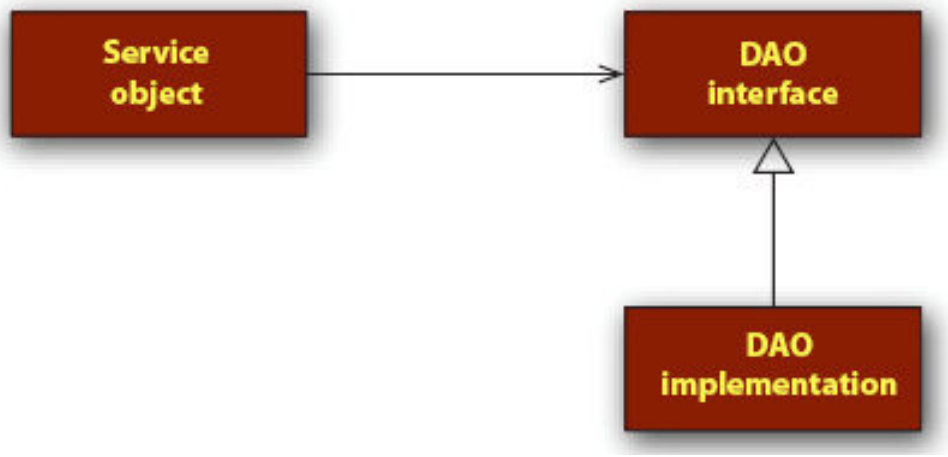
```
<property name="propertyName"> <null/></property>
```

- 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}"/>
```

Data Access Object (DAO)

- The Data Access Object (DAO) support in Spring is aimed at making it easy to work with data access technologies like JDBC, Hibernate, JPA or JDO in a consistent way.
- DAO stands for data access object, which describes a DAO's role in an application. They provide a means to read and write data to the database.
- They should expose this functionality through an interface by which the rest of the application will access them.

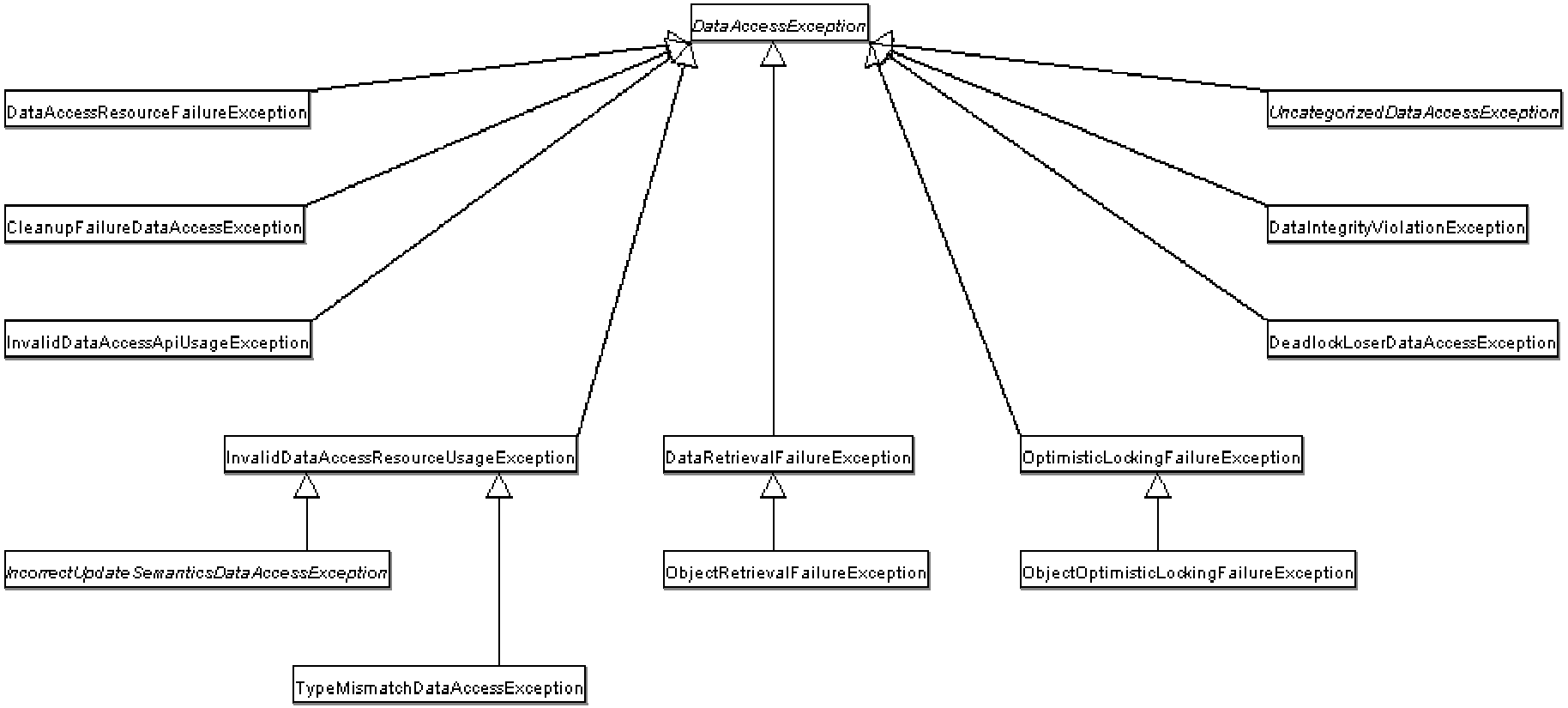


- A **DAO** object is also known as a **repository** (data store).

JDBC - SQLException

- SQLException means that something went wrong while trying to access a database. There is little information about that exception that indicates what went wrong or how to deal with it.
- Some common problems that might cause an SQLException to be thrown include:
  - The application is unable to connect to the database.
  - The query being performed has errors in its syntax.
  - The tables and/or columns referred to in the query don't exist.
  - An attempt was made to insert or update values that violate a database constraint.
- Spring provides a convenient translation from technology-specific exceptions like SQLException to its own exception class hierarchy with the DataAccessException as the root exception.
- These exceptions wrap the original exception in order to keep the original information.

Spring JDBC - Exceptions



JDBC Example - Insert

```
Connection con=null
PreparedStatement stmt=null;
try {
   con=DriverManager.getConnection(url);
   stmt=con.prepareStatement("insert into rezervari (idCursa,
    nrLocuri,numeClient) values  (?,?,?)");
   stmt.setInt(1, rezervare.getCursa().getId());
   stmt.setInt(2, rezervare.getNrLocuri());
   stmt.setString(3, rezervare.getNumeClient());
   int ok=stmt.executeUpdate();
   //...
} catch (SQLException e) {
    throw new RepositoryException("Eroare "+e);
}finally {
   if (stmt!=null) {try{ stmt.close();}catch(SQLException e){...}}
   if (con!=null) { try{ con.close();}catch(SQLException e){...}}
 }
```

JDBC Example - Select

```
Connection con=null;
PreparedStatement stmt=null;
ResultSet result=null;
try {
    con=DriverManager.getConnection(url);
    stmt=con.prepareStatement("select name from Angajati where username=? and
    password=?");
    stmt.setString(1,id);
    stmt.setString(2,passwd);
    result=stmt.executeQuery();
    Angajat angajat=null;
    if (result.next())
        angajat=new Angajat(id, result.getString("name"), passwd);
} catch (SQLException e) {...}
 finally {
   if (result!=null) {try{ result.close(); } catch (SQLException e) {...}}
   if (stmt!=null) {try{ stmt.close(); } catch (SQLException e) {...}}
   if (con!=null) {try{ con.close(); } catch (SQLException e) {...}}
}
```

Spring JDBC

Action	Spring	Programmer
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, statement and resultset.	X	

JDBC Support

Classes:

- **`JdbcTemplate`** (**`org.springframework.jdbc.core`**) is the classic Spring JDBC approach and the most popular. The methods are updated with Java 5 support such as generics and varargs
- **`NamedParameterJdbcTemplate`**

  (**`org.springframework.jdbc.core.namedparam`**) wraps a **`JdbcTemplate`** to provide named parameters instead of the traditional JDBC "?" placeholders. This approach provides better documentation and ease of use when you have multiple parameters for an SQL statement.
- **`SimpleJdbcInsert`** and **`SimpleJdbcCall`**

  (**`org.springframework.jdbc.core.simple`**) optimize database metadata to limit the amount of necessary configuration.

Data Source Configuration

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
   destroy-method="close">
    <property name="driverClassName" value="${jdbc.driver}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.user}"/>
    <property name="password" value="${jdbc.pass}"/>
</bean>

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

//jdbc.properties
jdbc.driver=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://localhost
jdbc.user=sa
jdbc.pass=
```

JdbcTemplate

- Instances of the `JdbcTemplate` class are threadsafe once configured.
- You can configure a single instance of a `JdbcTemplate` and then safely inject this shared reference into multiple DAOs (or repositories).
- Methods:
  - `queryForList(sql:String, elementType:Class<T>):List<T>`
  - `queryForList(sql:String, elemType:Class<T>, args:Object...):List<T>`
  - `queryForObject(sql:String, elemType:Class<T>): T`
  - `queryForObject(sql:String, args:Object[], requiredType:Class<T>):T`
  - `queryForObject(sql:String, rowMapper:RowMapper<T>):T`
  - `queryForObject(sql:String, rowMapper:RowMapper<T>, args:Object...):T`
  - `query(sql:String, rowMapper:RowMapper<T>): List<T>`
  - `query(sql:String, rowMapper:RowMapper<T>, args:Object...): List<T>`
  - `...`

RowMapper

- **`RowMapper<T>:`** is an interface used by **`JdbcTemplate`** to map the rows from **`ResultSet`** to the corresponding objects (line by line).
  - For every row that results from the query, **`JdbcTemplate`** will call the **`mapRow()`** method of the **`RowMapper`**:

```
T mapRow(ResultSet rs, int rowNum)  throws SQLException
```

JdbcTemplate- Examples

- Example `queryForInt`

```
int totalElevi = jdbcTemplate.queryForInt("select count(*) from
   elevi");
```

- Example `queryForInt`

```
int nrEleviClasa = jdbcTemplate.queryForInt(
        "select count(*) from elevi where clasa = ?", "12B");
```

- Example `queryForObject`

```
String nume = jdbcTemplate.queryForObject(
        "select nume from elevi where id = ?",
        new Object[]{1212L}, String.class);
```

JdbcTemplate- Examples

- Example `queryForObject`

```
Elev elev = jdbcTemplate.queryForObject(
        "select nume, prenume from elevi where id = ?",
        new RowMapper<Elev>() {
 public Elev mapRow(ResultSet rs, int rowNum) throws SQLException{
      Elev elev = new Elev();
      elev.setNume(rs.getString("nume"));
      elev.setPrenume(rs.getString("prenume"))
       return elev;
  }}, 1212L);
```

JdbcTemplate- Examples

- Example **query**

```
List<Elev> listaElevi = jdbcTemplate.query(
        "select nume, prenume from elevi where clasa = ?",
        new RowMapper<Elev>() {
 public Elev mapRow(ResultSet rs, int rowNum) throws SQLException{
      Elev elev = new Elev();
      elev.setNume(rs.getString("nume"));
      elev.setPrenume(rs.getString("prenume"))
       return elev;
  }}, "12B");
```

- Remark: The implementation of **RowMapper** interface may be a nested class. The same implementation is used for two different queries.

```
private static class ElevMapper implements RowMapper<Elev>{...}
//...
 jdbcTemplate.query(SQL_COMMAND, new ElevMapper());
```

JdbcTemplate- Examples

- Example `update (for INSERT/UPDATE/DELETE)`

```
jdbcTemplate.update("insert into elevi (nume, prenume) values (?,
   ?)", "Pop", "Vasile");

jdbcTemplate.update("update elevi set nume= ? where id = ?",
        "Popescu", 5276L);

jdbcTemplate.update("delete from elevi where id = ?",
        Long.valueOf(elevId));
```

- Example `execute`

```
jdbcTemplate.execute("create table participanti (id integer, nume
   varchar(100))");
```

DAO Configuration

```
public interface EleviDAO{
  // add, delete, ... operations
}
public class JdbcEleviDAO implements EleviDAO{
  private JdbcTemplate jdbcTemplate;
  public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
  //methods definition using jdbcTemplate
}
public interface ClaseDAO{
   //...
}
public class JdbcClaseDAO implements ClaseDAO{
  private JdbcTemplate jdbcTemplate;
  public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
  //methods definition using jdbcTemplate
}
```

DAO Configuration

```
//configuration file
<?xml version="1.0" encoding="UTF-8"?>
<beans...>
    <bean id="eleviDAO" class="JdbcEleviDao">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="claseDAO" class="JdbcClaseDao">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
    <context:property-placeholder location="jdbc.properties"/>
</beans>
```

NamedParameterJdbcTemplate

- The **`NamedParameterJdbcTemplate`** class adds support for programming JDBC statements using named parameters, as opposed to using only classic **`'?'`** .
- The **`NamedParameterJdbcTemplate`** class wraps a **`JdbcTemplate`**, and delegates to the wrapped **`JdbcTemplate`** to do much of its work.
- Methods:
  - **`query(sql:String, paramMap:Map<String,?>, rowMapper:RowMapper<T>) :List<T>`**
  - **`queryForInt(sql:String, paramMap:Map<String,?>):int`**
  - **`queryForInt(String sql, SqlParameterSource paramSource):int`**
  - **`getJdbcOperations(): JdbcOperations`**
- Remarks:
  1. The **`NamedParameterJdbcTemplate`** class does not offer all the operations exposed by **`JdbcTemplate`**. If you need to use a method from the JdbcTemplate that is not defined on the **`NamedParameterJdbcTemplate,`** you can access the underlying **`JdbcTemplate`** by calling the **`getJdbcOperations()`** method on the **`NamedParameterJdbcTemplate.`**
  2. For associating a named parameter with a value you can use a Map (**`Map<String, ?>)`** or **`SqlParameterSource`** interface (and its implementation **`MapSqlParameterSource`**).

NamedParameterJdbcTemplate -Examples

```
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
public void setDataSource(DataSource dataSource) {
    namedParameterJdbcTemplate = new
    NamedParameterJdbcTemplate(dataSource);
}
//using Map<String,?>
public int nrEleviClasa(String clasa) {
    String sql = "select count(*) from elevi where clasa = :class";
    Map<String, Object> param = new HashMap<String,Object>();
     param.put("class", clasa);
    return namedParameterJdbcTemplate.queryForInt(sql, param);
}
//using SqlParameterSource
public int nrEleviClasa(String clasa) {
    String sql = "select count(*) from elevi where clasa = :class";
    SqlParameterSource param = new MapSqlParameterSource("class",
    clasa);
    return namedParameterJdbcTemplate.queryForInt(sql, param);
}
```

NamedParameterJdbcTemplate -Examples

```
//using  SqlParameterSource, with multiple parameters
public int nrEleviClasa(String clasa, String prenume) {
    String sql = "select count(*) from elevi where clasa = :class
   and prenume= :prenume";
    SqlParameterSource param = new MapSqlParameterSource("class",
   clasa);
    param.addValue("prenume", prenume)
    return namedParameterJdbcTemplate.queryForInt(sql, param);
}
```

Remark:

When using `SqlParameterSource` (and `MapSqlParameterSource`) you can also specify the SQL type of the paramater.

DaoSupport

- For each of an application's JDBC-based DAO classes, we need to add a `JdbcTemplate`/`NamedParameterJdbcTemplate` property and setter method (field and corresponding method).
- If there are multiple DAOs, a lot of code will be duplicated.
- To avoid code duplication, Spring framework contains a few classes: `JdbcDaoSupport` and `NamedParameterJdbcDaoSupport` that already define and implement setting the property.
- The DAO classes from the application can extend the corresponding DaoSupport, implement the interface and define the operations .
- In the configuration file the code for setting the property will still repeat (duplicate).

DaoSupport

- JdbcDaoSupport : getJdbcTemplate()
- NamedParameterJdbcDaoSupport: getNamedParameterJdbcTemplate()

```
public class JdbcEleviDao extends JdbcDaoSupport implements
   EleviDao{

  //you do not need to declare JdbcTemplate property anymore

  public int nrElevi(String clasa){
     return getJdbcTemplate().queryForInt("select count(*) from
   elevi");
//...
  }
}
```

CLASSPATH

- In order to work with Spring and Spring JDBC you have to add the following jar files to your CLASSPATH:
- Spring
  - `spring.jdbc-3.2.2.RELEASE.jar`
  - `spring.tx-3.2.2.RELEASE.jar`
  - `spring.context.support-3.2.2.RELEASE.jar`
  - `spring.beans-3.2.2.RELEASE.jar`
  - `spring.context-3.2.2.RELEASE.jar`
  - `spring.core-3.2.2.RELEASE.jar`
  - `spring.expression-3.2.2.RELEASE.jar`
- Apache-Commons:
  - `commons-dbcp.jar`
  - `commons-logging.jar`
  - `commons-pool.jar`

Spring References

- Java Spring Framework documentation and download

  http://www.springsource.org

  or

  http://spring.io

- Craig Walls, *Spring in Action*, Third Edition, Ed. Manning, 2011