Föreläsning 4

JavaBeans

Taglib

JDBC

JavaBeans

- Stora scriptlets på html-sidorna har några nackdelar:
  - det blir snabbt oöverskådligt och svårt att felsöka
  - webbdesignern måste kunna Java
- Man önskar separera presentation (html) från logik(t e x jsp) vilket kan åstadkommas med s k javaBeans.
- En JavaBean är identiskt med en vanlig javaklass men följer JavaBean designmönster:
  - public konstruktor utan argument
  - publika set och get metoder för att sätta/läsa egenskaper
  - implements serializable

Fördefinierade JavaBean taggar

- jsp:useBean tag används för att referera till en javaBean på en JSP-sida, syntax:
- *<jsp:useBean id = "valbart namn att referera till bönan" scope = "page|request|session|application" class = "namnet på .class-filen" />*
- scope (räckvidd):
  - page/request: bönan är sidglobal
  - session: Om bönan är skapad återanvänds den, om inte skapas den och sparas i HTTP-sessionsobjektet
  - application: webbserverns livscykel
- För att sätta en egenskap används setProperty
  - *<jsp:setProperty name="tidigare valt namn att referera till bönan" property="instansvariabel|*">*
- För att läsa en egenskap används getpropery
  - *<jsp:getProperty name="tidigare valt namn att referera till bönan" property="instansvariabel(oftast)">*

Exempel 4:1


```
package bean;

public class DiceBean{
   private int utfall;

   public DiceBean(){
   }

   public void kasta(){
     utfall = 1 + (int)(Math.random()*6);
   }

   public int getUtfall(){
     return utfall;
   }
}
```

Exempel 4:2

```
<html>
<head><title>Exempel 4</title></head>
<body>
<center>

<jsp:useBean class="bean.DiceBean" id="db" scope="session"/>
<% db.kasta(); %>
<h2>Tärningskast</h2>
Tärningens utfall:
<p><img src="<jsp:getProperty name="db" property="utfall">.gif">
<form><input type=submit value="Kasta"></form>
</center>
</body>
</html>
```

Exempel 5:2

```
package bean;

public class SavingsBean{
   private double deposit;
   private double time;
   private double interest = 3;

   public void setDeposit(double deposit){
            this.deposit = deposit;
   }

   public void setTime(double time){
            this.time = time;
   }

   public int getWithdrawal(){
            return (int)Math.round(deposit*Math.pow((100+interest)/100,time));
   }

   public double getInterest(){
            return interest;
   }
}
```

Exempel 5:3

```
<html>
<head><title>Exempel 5</title></head>
<body>
<center>

<h2>Sparandeberäkning</h2>

<form action="exempel5.jsp">
<p>Insättning första året: <input type=text name="deposit">
<p>Antal år att förräntas: <input type=text name="time">
<p><input type=submit value="Beräkna">
</form>

</center>
</body>
</html>
```

Exempel 5:4

```
<html>
<head><title>Exempel 5</title></head>
<body>
<center>

<jsp:useBean class="bean.SavingsBean" id="sb" scope="session"/>
<jsp:setProperty name="sb" property="*">
<h2>Resultat</h2>
<p>Du får ut <jsp:getProperty name="sb" property="withdrawal"> kr.
<p>Beräkningen är baserad på <jsp:getProperty name="sb"
    property="interest">% ränta.
</center>
</body>
</html>
```

JSP-taggar

- I JSP finns möjligheten att skapa egna taggar, med dessa taggar kan man inkapsla komplext beteende i enkla anrop, och p s s ytterligare separera presentation från logik
- Istället för att varje jsp-sida skall börja med all html och logik för att genererar menyn så skulle man vilja skriva
  - *<util:savings deposit="1000" time="10"/>*
- Tre filer för jsp-tag:
  - En “Tag Handler”-klass (.class)
  - En “Tag Library Descriptor”-fil (.tld)
  - Slutligen JSP-sidan som anropar taggen
- Då taggen ärver av TagSupport har man en instans av *PageContext* som heter *pageContext* genom vilken man kan komma åt viktiga objekt inuti den anropande jsp-sidan:
  - pageContext.getOut();
  - pageContext.getRequest();
  - pageContext.getResponse();

Exempel 6:1

```
<html>
<head><title>Exempel 6</title></head>
<body>
<center>

<%@ taglib uri="/WEB-INF/taglib.tld" prefix="util"%>
<p><util:savings deposit="1000" time="10"/>

</center>
</body>
</html>
```

```
<taglib>
        <short-name>util</short-name>
        <tag>
        <name>savings</name>
        <tagclass>tag.SavingsTag</tagclass>
        <attribute>
                <name>deposit</name>
                <required>true</required>
                <rtexprvalue>false</rtexprvalue>
        </attribute>
        <attribute>
                <name>time</name>
                <required>true</required>
                <rtexprvalue>false</rtexprvalue>
        </attribute>
        </tag>

</taglib>
```

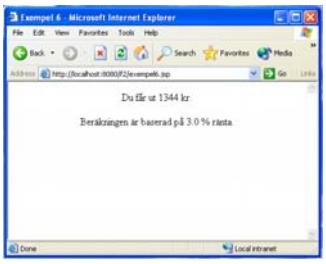
Exempel 6:2

```
package tag;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class SavingsTag extends TagSupport{
    private double deposit;
    private double time;
    private static double interest = 3;

    public SavingsTag(){
    }

    public void setDeposit(double deposit){
            this.deposit = deposit;
    }

    public void setTime(double time){
            this.time = time;
    }
```

Exempel 6:3

```
    public int doStartTag()throws JspTagException{
        try{
            JspWriter out = pageContext.getOut();
                    out.write("<p>Du får ut " + getWithdrawal() + " kr.");
                    out.write("<p>Beräkningen är baserad på " + getInterest() + " % ränta.");
        }
        catch (java.io.IOException e) {
            System.out.println(e.getMessage());
            throw new JspTagException(e.getMessage());
        }
        return EVAL_PAGE;
    }

    public int getWithdrawal(){
            return (int)Math.round(deposit*Math.pow((100+interest)/100,time));
    }

    public double getInterest(){
            return interest;
    }
}
```

Exempel 6:4



JDBC - Innehåll

- JNDI – Java Naming and Directory Interface
  - En slags DNS för Objekt
- JDBC Grunderna
  - Uppkoppling
  - Exekvera SQL-satser
  - Ta hand om resulterande ResultSets
- Finesser
  - PreparedStatement
  - Commit & Rollback

Java Naming and Directory Interface(JNDI)

- För att distribuerade applikationers komponenter skall kunna hitta varandra behövs någon tjänst som hjälper till med detta, en s k namngivningstjänst (Naming Service)
- JNDI mappar namn mot objekt (jämför med DNS).
- JNDI är ett interface som är beroende av en underliggande implementation t.ex. LDAP för att fungera
- JNDI använder ett fåtal objekt, främst Context & InitialContext
- Ett Context objekt har metoder för att binda namn till objekt, lista existerande namn, ta bort och döpa om

javax.naming.Context

- Viktigaste metoder:
  - `void bind(String stringName, Object object)`
  - `Object lookup(String stringName)`

javax.sql.Datasource

- Datasource är ett Interface och implementeras av Driverförsäljaren
- Man får normalt tag på ett Datasource objekt genom en JNDI-uppslagning.
- Är en abstraktion av drivrutinen som ligger under orion/lib och konfigureras under orion/config/data-sources.xml

Vad är JDBC?

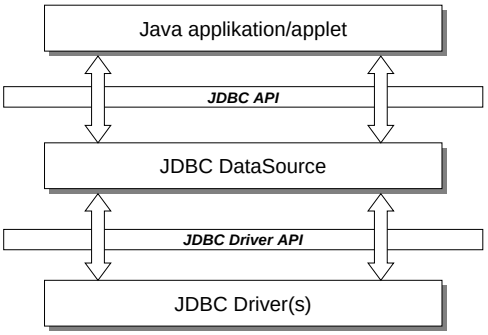
- Ett API för att kommunicera med databaser från javaapplikationer
- Relevanta paket:
  - java.sql
  - javax.sql

JDBC Arkitekturen

Tre huvudkomponenter i JDBC program

- **Java applikation**
  - Skapad av Java utvecklaren
  - Alla anrop till databasen enligt JDBC API:t
- **JDBC DataSource**
  - Slår upp med JNDI
  - Fungerar som länk mellan applikationen och drivrutinen
- **JDBC drivrutinen**
  - Tillhandahållen av DB företaget eller tredje part
  - Konverterar JDBC kod till DB specifika databaskommandon

JDBC Arkitekturen



JDBC - klasserna

- Dessa tre klasser är viktigast att känna till i JDBC:
  - `Connection`
  - `Statement`
  - `ResultSet`

DatabasURL:er

- **URL:en bidrar med nödvändig information**
  - Signalerar att det är en DB-URL
  - Identifierar databasens sub-protokoll
  - Lokaliserar databasen
- **Generell syntax**
  - **`jdbc : <driverns sub-protokoll> : <db sökvägen>`**

Exempel på DatabasURL:er

- ODBC Datakälla
  jdbc:odbc:test
- mySQL datakälla
  jdbc:mysql://localhost:3306/test
- Informix Databas
  jdbc:informix-sqli://dupond.nada.kth.se:1557/stene:informixserver=course_2000
- Kolla i orion/config/data-sources.xml

JDBC: Steg för steg

- Steg 1: Gör en JNDI-uppslagning av drivrutinen
- Steg 2: Skapa ett Connection-objekt
- Steg 3: Skapa ett Statement-objekt och utför en förfrågan
- Steg 4: Ta hand om ett resulterande ResultSet

Steg 1: JNDI-uppslagning

```
<%@ page import="javax.naming.*,java.sql.*,javax.sql.*" %>

<%
    Context env = null;
    DataSource source = null;
    try{
       env = new InitialContext();
       source = (DataSource)env.lookup("jdbc/InformixCoreDS");
    }
    catch(NamingException e){
       out.println(e.getMessage());
    }
    try{
       Connection con = source.getConnection();
       Statement s = con.createStatement();
       String sql = "select * from studenter order by enamn";
       ResultSet rs = s.executeQuery(sql);
        while(rs.next()){
                out.println(rs.getString("enamn"));
        }
       rs.close();
       s.close();
       con.close();
    }
     catch(SQLException e) {
       out.println("SQLException: "+e.getMessage());
     }
%>
```

JDBC: Steg för steg

- Steg 1: Ladda databasdrivern
- Steg 2: Skapa ett Connection-objekt
- Steg 3: Skapa ett Statement och utför en förfrågan
- Steg 4: Ta hand om ett resulterande ResultSet

Steg 2: Skapa ett Connection objekt

```
<%@ page import="javax.naming.*,java.sql.*,javax.sql.*" %>

<%
     Context env = null;
    DataSource source = null;
    try{
       env = new InitialContext();
       source = (DataSource)env.lookup("jdbc/InformixCoreDS");
    }
    catch(NamingException e){
       out.println(e.getMessage());
    }
    try{
       Connection con = source.getConnection();
       Statement s = con.createStatement();
       String sql = "select * from studenter order by enamn";
       ResultSet rs = s.executeQuery(sql);
        while(rs.next()){
                out.println(rs.getString("enamn"));
        }
       rs.close();
       s.close();
       con.close();
    }
     catch(SQLException e) {
       out.println("SQLException: "+e.getMessage());
     }
%>
```

JDBC: Steg för steg

- Steg 1: Ladda databasdrivern
- Steg 2: Skapa ett Connection-objekt
- Steg 3: Skapa ett Statement-objekt och utför en förfrågan
- Steg 4: Ta hand om det resulterande ResultSet

java.sql.Statement objektet

- Ett Statement objekt
  - Används för att skicka förfrågningar till databasen
  - Skapas via metoden *`createStatement()`*
    - `Statement stmt = con.createStatement()`
- Via statement sköts exekveringen av SQL

Steg 3: statement-objekt / sql-fråga

```
<%@ page import="javax.naming.*,java.sql.*,javax.sql.*" %>

<%
     Context env = null;
    DataSource source = null;
    try{
       env = new InitialContext();
       source = (DataSource)env.lookup("jdbc/InformixCoreDS");
    }
    catch(NamingException e){
       out.println(e.getMessage());
    }
    try{
       Connection con = source.getConnection();
       Statement s = con.createStatement();
       String sql = "select * from studenter order by enamn";
       ResultSet rs = s.executeQuery(sql);
        while(rs.next()){
                out.println(rs.getString("enamn"));
        }
       rs.close();
       s.close();
       con.close();
    }
     catch(SQLException e) {
       out.println("SQLException: "+e.getMessage());
     }
%>
```

JDBC: Steg för steg

- Steg 1: Ladda databasdrivern
- Steg 2: Skapa ett Connection objekt
- Steg 3: Skapa ett Statement och utför en "query"
- Steg 4: Ta hand om det resulterande ResultSet

Datamanipulation i JDBC

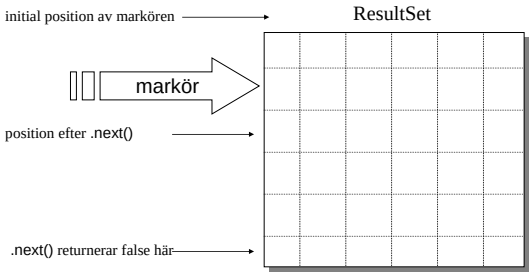
- Vid UPDATE, INSERT, DELETE används
- `executeUpdate(String sql)`
- som returnerar antal berörda tupler
- Vid SELECT används istället
- *ResultSet executeQuery(String sql)*
- som returnerar ett Resultset att navigera i

Det finns också en execute(String)-metod som returnerar en boolean om det finns ett ResultSet att hämta.

ResultSet

- All dataaccess sker genom en instans av ResultSet
- `ResultSet` har medoder för
  - navigering i ResultSet
  - åtkomst av datat

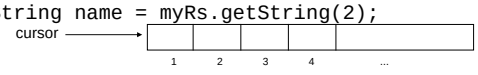
Navigation genom ResultSet



Läsning av data-innehåll

- Data avläses via ett antal typbestämda `getXXX` metoder
  - `getInt(...)` returnerar datan som en int
  - `getString(...)` returnerar data som en sträng
  - ...
- Vilken kolumn man ska läsa från identifieras av *namn* eller *index*
  - index number startar på 1

```
String name = myRs.getString(2);
```



Steg 4: Ta hand om resultset

```
<%@ page import="javax.naming.*,java.sql.*,javax.sql.*" %>

<%
      Context env = null;
     DataSource source = null;
     try{
        env = new InitialContext();
        source = (DataSource)env.lookup("jdbc/InformixCoreDS");
     }
     catch(NamingException e){
        out.println(e.getMessage());
     }
     try{
        Connection con = source.getConnection();
        Statement s = con.createStatement();
        String sql = "select * from studenter order by enamn";
        ResultSet rs = s.executeQuery(sql);
        while(rs.next()){
              out.println(rs.getString("enamn"));
        }
        rs.close();
        s.close();
        con.close();
     }
      catch(SQLException e) {
        out.println("SQLException: "+e.getMessage());
     }
%>
```

PreparedStatement

- I en s k förberedd SQL-sats läggs selektionsvärden in som vaiabler, p s s kan samma kod återanvändas vid ett senare tillfälle.
- Det finns en annan god anledning till att använda PreparedStatement (PS), nämligen säkerhet. Om man t e x har ett formulär där man ska välja att kunna radera en rad i en databas är det inte så bra om användaren skriver ”in (select * from tabellnamn)”
- PS ser automatiskt till att konvertera special tecken så som ’ så att det inte tolkas att databasen.
- PreparedStatement ps;
- ps = con.prepareStatement(
- “SELECT * FROM studenter WHERE enamn = ?”);
- ...
- ps.setString(1, “Bengtsson”);
- rs = ps.executeQuery();

Transaktions stöd

- När uppdatering i en tabell är beroenda av uppdateringen i en annan tabell
- Exempel:
  - En student ska registreras i ett kursregister och på minst en kurs. Två tabeller finns (studenter, kurser) och båda insert-satserna måste fungera.

Transaktions stöd (forts)

- Connection har en metod setAutoCommit(boolean autoCommit)

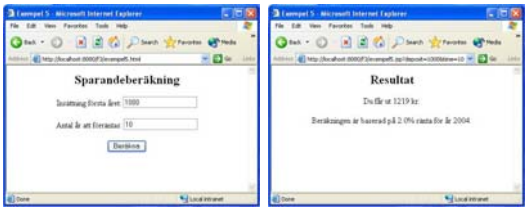
```
Statement s = con.createStatement();
con.setAutoCommit(false);

String sql = "INSERT INTO studenter VALUES(3,'Goran','Henriksson',3)";
System.out.println(sql);
s.executeUpdate(sql);
int transaktion1 = s.getUpdateCount();

sql = "INSERT INTO kurser VALUES(3,'2D1311','Programmeringsteknik med PBL',4)";
System.out.println(sql);
s.executeUpdate(sql);
int transaktion2 = s.getUpdateCount();

if((transaktion1 == 1) && (transaktion2 == 1)) {
      con.commit();
      System.out.println("Båda tabellerna uppdaterade!");
}
else{
      con.rollback();
      System.out.println("Ingen tabell uppdaterad då ett fel har uppstått!");
}
```

Glöm inte att sätta tillbaka autoCommit!

SQL syntax (fort...)

- *UPDATE studenter SET fnamn = ’Adam’ WHERE fnamn=’Ada’*
- *UPDATE kurser SET credits = 4 WHERE credits < 4*
- *DELETE from studenter WHERE enamn like ’%sson’*
- *INSERT INTO studenter VALUES (4,’Ada’,’Bengtsson’)*
- *SELECT * from studenter where fnamn = ’Ada’*
- *CREATE TABLE users (username varchar(50), password varchar(50))*

Exempel 1:5



Exempel 2:5

```
<html>
<head><title>Exempel 5</title></head>
<body>
<center>

<h2>Sparandeberäkning</h2>

<form action="exempel5.jsp">
<p>Insättning första året: <input type=text name="deposit">
<p>Antal år att förräntas: <input type=text name="time">
<p><input type=submit value="Beräkna">
</form>

</center>
</body>
</html>
```

```
<html>
<head><title>Exempel 5</title></head>
<body>
<center>

<jsp:useBean class="bean.SavingsBean" id="sb"
scope="session"/>
<jsp:setProperty name="sb" property="*">
<% sb.compute(); %>
<h2>Resultat</h2>
<p>Du får ut <jsp:getProperty name="sb"
property="withdrawal"> kr.
<p>Beräkningen är baserad på <jsp:getProperty
name="sb" property="interest">% ränta för år
<jsp:getProperty name="sb" property="year">.
</center>
</body>
</html>
```

Exempel 3:5

```
package bean;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

import java.text.*;

public class SavingsBean{
    private double deposit;
    private double time;
    private int withdrawal;
    private double interest;
    private int year;

    public SavingsBean(){
    }

    public void setDeposit(double deposit){
        this.deposit = deposit;
    }

    public void setTime(double time){
        this.time = time;
    }
```

Exempel 4:5

```
public void compute(){
      SimpleDateFormat formatter = new SimpleDateFormat("yyyy");
       java.util.Date rightNow = new java.util.Date();
       String dateString = formatter.format(rightNow);
       Context env = null;
      DataSource source = null;
      try{
         env = new InitialContext();
         source = (DataSource)env.lookup("jdbc/InformixCoreDS");
          //source = (DataSource)env.lookup("jdbc/MySqlCoreDS");
      }
      catch(NamingException e){
         System.out.println("<p>"+e.getMessage());
      }
      try{
         Connection con = source.getConnection();
         Statement s = con.createStatement();
         String sql = "select * from interest where year = " + dateString;
         ResultSet rs = s.executeQuery(sql);
          while(rs.next()){
                  year = Integer.parseInt(rs.getString("year"));
                  interest = Double.parseDouble(rs.getString("rate"));
          }
          rs.close();s.close();con.close();
      }
       catch(SQLException e) {
         System.out.println("SQLException: "+e.getMessage());
      }
       withdrawal = (int)Math.round(deposit*Math.pow((100+interest)/100,time));
   }
```

Exempel 5:5

```
 public int getWithdrawal(){
     return withdrawal;
    }

    public double getInterest(){
     return interest;
    }

    public int getYear(){
     return year;
    }
}
```