

Lektion 8

Datateknik GR(B), Java III, 7,5 högskolepoäng

Syfte:	Att känna till vad JDBC är och kunna använda dess API för att koppla upp sig mot en databas och ställa frågor mot den. Frågor för att både uppdatera databasen och hämta data från den.
Att läsa:	Kursboken, finns inget kapitel om JDBC The Java™ Tutorial, JDBC(TM) Database Access http://docs.oracle.com/javase/tutorial/jdbc/



Java III



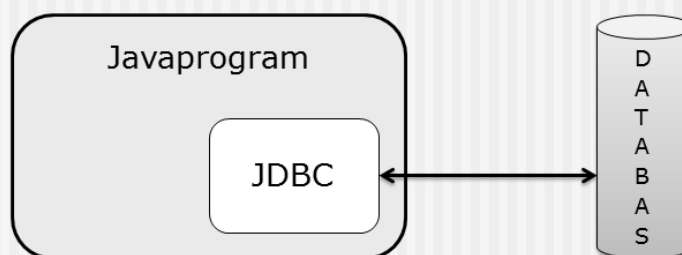
Lektion 8 - JDBC

Lektionen förutsätter att du redan har grundläggande kunskaper om databaser

Vad är JDBC?



- API för att hantera databaser på ett enhetligt sätt i Java
- Program blir (i stort sett) helt oberoende av vilken databas som används





Vad är JDBC?

- JDBC standardiserar hur vi
 - Ansluter till databasen
 - Ställer frågor till databasen (SQL)
 - Hanterar resultatet från en fråga
 - Uppdaterar databasen (SQL)
- Kan komma åt alla typer av data i tabellform
- Används främst för data som lagras i en relationsdatabas



Vad är JDBC?

- JDBC är ett namn och inte en förkortning av något
- Sägs ändå ofta stå för:
Java Database Connectivity
- Består av två paket som finns tillgängliga i JDK:

```
java.sql
```

```
javax.sql
```



Vad är JDBC?

- JDBC består av följande huvuddelar
 - JDBC Drivers
 - Connections
 - Statements
 - ResultSets
- Grundläggande användningsområden
 - Ställa frågor mot databasen
 - Uppdatera databasen
 - Utföra transaktioner

JDBC Drivers - En JDBC-drivrutin är en samling av klasser i Java som gör att du kan ansluta till en viss databas. De flesta databastillverkare tillhandahåller egna JDBC-drivrutiner för sina databasprodukter. Till exempel har både MySQL och PostgreSQL sina egna JDBC-drivrutiner. En JDBC-drivrutin implementerar en hel del av de gränssnitt (interface) som finns i API. När du sen i din kod använder en viss JDBC-drivrutin, är det faktiskt bara dessa gränssnitt som används. Den konkreta JDBC-drivrutinen som används är dold bakom gränssnittet. Således kan du plugga in en ny JDBC-drivrutin utan din kod märker det. Som vi skrev tidigare blir program tack vare detta i stort sett helt oberoende av vilken databas (drivrutin) som används. Dock kan de förekomma vissa skillnader i vilka funktioner de olika drivrutinerna stöder.

Connections - När en JDBC-drivrutin har lästs in och initierats, måste vi upprätta en anslutning till databasen. Detta görs genom att skapa ett Connection-objekt med hjälp av JDBC API:t och den laddade drivrutinen. All kommunikation med databasen sker sedan via denna Connection. En applikation kan ha mer än en Connection öppen till en (eller flera olika) databaser i taget.

Statements - Ett Statement är vad vi använder för att köra frågor och uppdateringar mot databasen. Det finns några olika typer av Statement som vi kan använda oss av. Varje Statement motsvarar en enda fråga eller uppdatering.

ResultSet - När vi ställer en fråga om mot databasen kommer vi att som svar få ett ResultSet. Detta ResultSet innehåller all data som frågan resulterade i och via detta ResultSet kan vi stega igenom och utvinna all data.

Dessa används sen för följande grundläggande användningsområden.

Ställa frågor mot databasen - Ett av de vanligaste, om inte det vanligaste, användningsområdet är att läsa data från en databas. Läsning av data från en databas kallas att ställa frågor mot databasen. En fråga kan t.ex. vara: Lista alla anställda som heter Andersson i efternamn.

Uppdatera databasen - En annat mycket vanligt användningsområde är att uppdatera databasen. Uppdatering av databasen innebär att skriva data till databasen. Oftast innebär det att nya data läggs till i databasen, men det kan även vara att uppdatera befintlig data och att skapa de tabeller som ska lagra data.

Utföra transaktioner - En transaktion grupper flera uppdateringar och eventuellt förfrågningar till en enda händelse. Antingen genomförs alla åtgärder eller så genomförs ingen av dem.

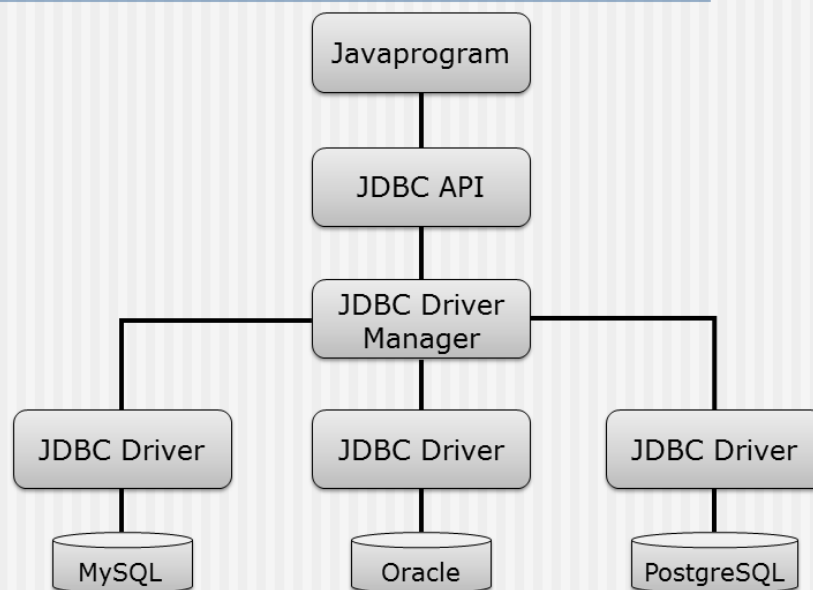


JDBC-version

- JDK 8 använder version 4.2 av JDBC
- Kompatibel med tidigare versioner
- Består av två delar:
 - JDBC core API (java.sql): är ofta fullt tillräckligt för flesta databaslösningar
 - JDBC Optional Package API (javax.sql): för större och mer avancerade lösningar, ofta implementerade i J2EE
- Dokumenteras i JSR221



JDBC arkitekturen

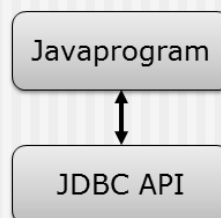




JDBC arkitekturen

■ Javaprogram

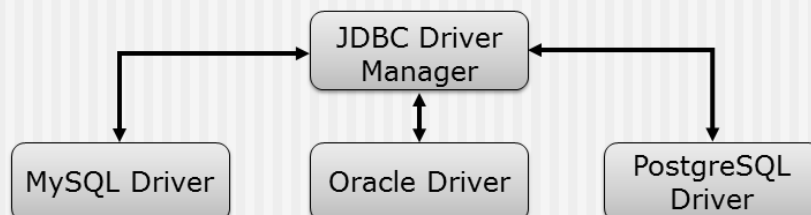
- Skapade av oss som utvecklare
- Alla anrop till databasen sker via ett antal gränssnitt i JDBC API:t
- Oavsett underliggande databas används samma anrop



JDBC arkitekturen

■ JDBC DriverManager

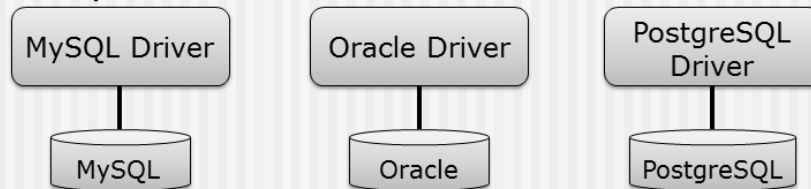
- Fungerar som länk mellan programmen och drivrutinen
- Laddar och väljer ut lämpliga drivrutiner
- Kan kommunicera med vilken drivrutin som helst som följer JDBC Driver API:t





JDBC arkitekturen

- JDBC drivrutinen
 - Länk mellan programmen och underliggande databas
 - Tillhandahållen av databastillverkaren eller annan tredje part
 - Konverterar JDBC-kod till databas-specifika databaskommandon





JDBC drivrutin

- Talas om fyra olika typer av drivrutin
 - Typ 1: JDBC-ODBC bridge driver
 - Typ 2: Java + Native code driver
 - Typ 3: All Java + Middleware translation driver
 - Typ 4: All Java driver
- Idag är typ 4 vanligast



En typ 1 JDBC-drivrutin består av en Java-del som översätter anrop till JDBC-gränssnitten till anrop i ODBC. En ODBC brygga anropar sedan ODBC-drivrutinen för den angivna databasen. Typ 1 är (var) främst avsedda att användas i början när Java var nytt, då det inte fanns några drivrutiner av typ 4.

En typ 2 JDBC-drivrutin är som en typ 1, förutom att ODBC-delen ersätts med en kod skriven för den plattform programmet körs i. Den plattformaberoende koden är dessutom skriven för att passa mot en viss databas.

En typ 3 JDBC-drivrutin är en drivrutin skriven helt i Java. Den skickar anrop från gränssnitten i JDBC till en mellanliggande server. Denna server ansluter sedan till databasen på uppdrag av JDBC-drivrutinen.

En typ 4 JDBC-drivrutin är också en drivrutin skriven helt i Java. Denna typ av drivrutin ansluter själv direkt till databasen. Drivrutinen är specifik för varje databas. Idag är de flesta JDBC-drivrutiner av typen 4.



Ladda drivrutin

- Drivrutinerna (jar-filen) måste finnas i classpath
- I JDBC version 4+ laddas drivrutinen automatiskt
- I tidigare versioner måste vi ladda via kod

```
Class.forName("namn.på.drivrutinen");
```

- Går att göra även i version 4, men onödigt!



Lista laddade drivrutiner

■ Exempel:

```
import java.util.*;
import java.sql.*;

public class ListDriversLoaded {
    public static void main(String[] args) {
        // Hämta alla laddade drivrutiner
        Enumeration<Driver> drivers =
            DriverManager.getDrivers();

        // Loopa alla drivrutiner och skriv ut dem på skärmen
        while (drivers.hasMoreElements()) {
            Driver driver = drivers.nextElement();
            System.out.println(driver.getClass().getName());
        }
    }
}
```

I alla program som använder JDBC måste vi importera paketet `java.sql`. För att få en lista över laddade JDBC-drivrutiner anropar vi den statiska metoden `getDrivers` i klassen `DriverManager`. Vi loopar sedan igenom denna lista och skriver ut drivrutinens klassnamn på skärmen.

Se exemplet **ListDriversLoaded.java**.



Lista laddade drivrutiner

■ Körning:

```
java ListDriversLoaded
```

■ Utskrift:

```
sun.jdbc.odbc.JdbcOdbcDriver
```

■ Körning:

```
java -cp .;drivers\postgresql-9.3-1102.jdbc41.jar  
ListDriversLoaded
```

■ Utskrift:

```
sun.jdbc.odbc.JdbcOdbcDriver  
org.postgresql.Driver
```



Skapa en anslutning

- JDBC version 4 erbjuder två sätt att ansluta sig till en databas
 - Via DriverManager
 - Via DataSource
- DataSource är att föredra
- Vi använder dock DriverManager
 - För det är lättare att använda
 - Vi behöver inte den extra funktionalitet som DataSource erbjuder



Skapa en anslutning

```
String url = "jdbc:mysql:/test"; // databasspecifik url
String username = "namnPåAnvändaren";
String password = "användarensLösenord";
Connection connection =
    DriverManager.getConnection(url, username, password);
```

- URL är en sträng som drivrutinen behöver för att ansluta till databasen
 - Anger var databasen finns
 - Namnet på databasen som ska anslutas
 - (samt olika konfigurationsegenskaper)
- Exakt syntax tillhandahålls av varje databasleverantör



Skapa en anslutning

■ Exempel:

```
String url =
    "jdbc:postgresql://localhost:5432/dt066g";
String username = "dt066g";
String password = "RaUc7PDrW";

Connection connection =
    DriverManager.getConnection(url, username,
    password);

// inträffade inget SQLException är vi anslutna
// och vi kan nu ställa frågor m.m. mot databasen

// stäng alltid anslutningen när vi är klar
connection.close();
```

Se exemplet **ConnectionTest.java**.



URL till Miuns server

- Mittuniversitetet har en PostgreSQL-server som kursen utnyttjar
- Använd följande URL

Drivrutin av PostgreSQL	Adress till servern	Lyssnar- port	Databasen vi ska ansluta till	Anslut med SSL aktiverat
<code>jdbc:postgresql://webblabb.miun.se:5432/dt066g?ssl=true&sslfactory=org.postgresql.ssl.NonValidatingFactory</code>				

För att slippa installera ett certifikat på datorn anger vi att validering av certifikat inte ska göras



Tabellen person

■ Skapa

```
CREATE TABLE person(  
  id serial NOT NULL,  
  firstname text NOT NULL,  
  lastname text NOT NULL,  
  phone text,  
  email text,  
  CONSTRAINT person_pk  
    PRIMARY KEY (id)  
);
```

■ Lägga in data

```
INSERT INTO person (firstname,  
  lastname, phone, email)  
VALUES("Bertil", "Andersson",  
  "555-1234", "bean@test.se");  
...  
INSERT INTO person (firstname,  
  lastname, phone, email)  
VALUES("Sara", "Larsson",  
  null, "sara@larsson.net");
```

id	firstname	lastname	phone	email
1	Bertil	Andersson	555-1234	bean@test.se
2	Sofia	Nilsson	063-56744	nilsson.s@mail.se
3	Kalle	Karlsson	555-5544	
4	Anna	Andersson	555-1234	anna@andersson.com
5	Lars	Svensson		
6	Sara	Larsson		sara@larsson.net

Tabellen person kommer att användas som exempel i resten av lektionen. Här ges exempel på hur tabellen skapats med SQL och hur data har lagts in i tabellen. Ansluter du till Mittuniversitetets databasserver så finns denna tabell redan skapad. Om du däremot har installerat en egen databas lokalt på datorn måste du skapa denna tabell själv. Enklast gör du det genom att titta på exemplet **PersonDBTable.java**. Ändra url, användare och lösenord så att uppgifterna stämmer med din installation.



Ställa en fråga

- Först måste vi ha en Connection
- Skapa sen ett Statement-objekt

```
Statement statement = connection.createStatement();
```

- Därefter kan vi exekvera SQL-frågor

```
String sql = "SELECT * FROM person";  
ResultSet result = statement.executeQuery(sql);
```

- Vi får ett ResultSet som innehåller resultatet av frågan
- Stäng både ResultSet och Statement



ResultSet

- Innehåller en "tabell" med rader och kolumner av data
- Vi stegar igenom resultatet så här

```
while(result.next()) {  
    int id = result.getInt("id");  
    String firstName = result.getString("firstname");  
    String lastName = result.getString("lastname");  
    String phone = result.getString("phone");  
    String email = result.getString("email");  
  
    Person p = new Person(id, firstName, lastName,  
        phone, email);  
    System.out.println(p.toString());  
}
```

Se exemplet **ListAllPersons.java**.



ResultSet

- All dataaccess sker via ResultSet
- Har metoder för att
 - Navigera (framåt/bakåt) i resultatet
 - Komma åt data
 - Stänga resultatet när vi är klar
- Har en "markör" (cursor) som pekar på aktuell rad i resultatet
- Initialt pekar markören före första raden



ResultSet

- Ett ResultSet kan skapas med tre olika egenskaper
 1. Type
 2. Concurrency
 3. Cursor holdability
- Anges redan när du skapar ditt Statement-objekt

```
createStatement(int resultSetType,  
                int resultSetConcurrency,  
                int resultSetHoldability)
```




ResultSet - Type

- **TYPE_FORWARD_ONLY**
 - Markören kan enbart flyttas framåt
 - Data i resultatet ändras inte utan ser ut så som databasen såg ut när frågan ställdes
 - Är standardvärdet
- **TYPE_SCROLL_INSENSITIVE**
 - Markören kan flyttas både framåt och bakåt
 - Data i resultatet ändras inte om underliggande data ändras i databasen
- **TYPE_SCROLL_SENSITIVE**
 - Markören kan flyttas både framåt och bakåt
 - Data i resultatet ändras om underliggande data ändras i databasen



ResultSet - Concurrency

- Vissa databaser och JDBC drivrutiner stödjer att databasen uppdateras via ett ResultSet
- Concurrency avgör den uppdateringsfunktionalitet som stöds
- Finns två värden
 - **CONCUR_READ_ONLY**
 - Ett ResultSet kan enbart läsas
 - **CONCUR_UPDATABLE**
 - Ett ResultSet kan både läsas och uppdateras



ResultSet - Holdability

- Avgör om ett ResultSet ska stängas (close) när commit anropas på underliggande Connection
- Inte alla databaser och JDBC drivrutiner stödjer alla värden
 - CLOSE_CURSORS_OVER_COMMIT
 - ResultSet stängs om connection.commit() anropas på den Connection som skapade ResultSet
 - HOLD_CURSORS_OVER_COMMIT
 - Stänger inte ResultSet om commit anropas



Navigera i ett ResultSet

- Metoden next():
 - Flyttar fram markören en rad
 - Returnerar true så länge markören kommer befinna sig på en rad
 - False när det inte finns några fler rader

Initial position av markören →

	ResultSet		
	Bertil	Andersson	555-1234
markör →	Sofia	Nilsson	063-56744
Position efter next() →	Kalle	Karlsson	555-5544
next() returnerar false här →	Anna	Andersson	555-1234



Navigera i ett ResultSet

- Vid TYPE_SCROLL_INSENSITIVE eller TYPE_SCROLL_SENSITIVE kan även följande metoder användas:
 - first() – Flyttar markören till första raden
 - beforeFirst() – Flyttar markören innan första raden
 - last() – Flyttar markören till sista raden
 - afterLast() – Flyttar markören efter sista raden



Navigera i ett ResultSet

- Vid TYPE_SCROLL_INSENSITIVE eller TYPE_SCROLL_SENSITIVE kan även följande metoder användas:
 - previous() – Flyttar markören ett steg bakåt från nuvarande rad
 - absolute(int) – Flyttar markören till radnumret som ges som argument
 - relative(int) – Flyttar markören ett relativt antal steg från nuvarande position



Hämta data från ResultSet

- Det finns ett stort antal get-metoder i ResultSet för att returnera data
 - getInt(...) – returnerar data som en int
 - getString(...) returnerar som en sträng
 - etc
- Vilken kolumn som ska hämtas från anges som namn eller index
- Index börjar på 1



Hämta data från ResultSet

ResultSet

namn →	firstname	lastname	phone
	Bertil	Andersson	555-1234
markör →	Sofia	Nilsson	063-56744
	Kalle	Karlsson	555-5544
	Anna	Andersson	555-1234
index →	1	2	3

```
String firstName = result.getString("firstname"); // Sofia
String lastName = result.getString(2);           // Nilsson

int phoneIndex = result.findColumn("phone");      // 3
String phone = result.getString(phoneIndex);      // 063-56744
```



PreparedStatement

- Ärver från klassen Statement
- Kan användas när vi vill:
 - Exekvera samma SQL-fråga flera gånger (blir mer effektivt med ett PreparedStatement)
 - Kunna ange parametrar i SQL-frågan
- Skapas med en SQL-fråga direkt

```
String sql = "SELECT * FROM person";  
  
PreparedStatement preparedStatement =  
    connection.prepareStatement(sql);
```



PreparedStatement

- När SQL-frågor exekveras på databasservern sker det i fyra steg
 - Analys av SQL-frågan
 - Kompilering av SQL-frågan
 - Planering hur frågan ska exekveras
 - Exekvera SQL-frågan
- Om samma fråga upprepas måste alla fyra steg göras om
- Inte alltid så effektivt
- För PreparedStatement sker analys, kompilering och planering redan vid skapandet av objektet
- Endast exekvering behöver sen utföras oavsett hur många gånger samma SQL-fråga ska köras



PreparedStatement

- Där du behöver ange en parameter i SQL-frågan skriver du ett ?

```
String sql = "SELECT * FROM person WHERE id=?";  
  
PreparedStatement preparedStatement =  
    connection.prepareStatement(sql);
```

- Efter att PreparedStatement är skapat kan vi lägga till parametrar på frågetecknets plats



PreparedStatement

- Görs genom att använda en av många set-metoder:
 - `setInt(index, value)`
 - `setString(index, value)`
 - etc
- Ange index på frågetecknet och vilket värde som ska sättas in

```
preparedStatement.setInt(1, 2);  
ResultSet result = preparedStatement.executeQuery();
```

→ **ResultSet**

id	firstname	lastname	phone	email
2	Sofia	Nilsson	063-56744	nilsson.s@mail.se

Se exemplet **ListPersonWithId.java**.



PreparedStatement

- Kan ange fler än en parameter
- Använd bara fler ?

```
String sql = "SELECT * FROM person WHERE firstname=  
'?' AND lastname='?'";
```

```
PreparedStatement preparedStatement =  
    connection.prepareStatement(sql);
```

```
preparedStatement.setString(1, "Bertil");  
preparedStatement.setString(2, "Andersson");  
ResultSet result = preparedStatement.executeQuery();
```

→ ResultSet

id	firstname	lastname	phone	email
1	Bertil	Andersson	555-1234	bean@test.se



Uppdatera databasen

- Uppdateringar kan bland annat vara
 - Skapa/ta bort databaser
 - Skapa/ta bort tabeller
 - Lägga till/ta bort rad i en tabell
 - Uppdatera en eller flera rader i tabellen
- Använd ett Statement

```
int result = statement.executeUpdate(sql);
```

- Returnerar antingen
 - antal rader som påverkades eller
 - 0 för SQL som inte returnerar något



Uppdatera databasen

■ Skapa tabell

```
Connection connection =  
DriverManager.getConnection(URL, USERNAME, PASSWORD);  
  
Statement statement = connection.createStatement();  
  
String sql = "CREATE TABLE bil ( " +  
    "regnr CHAR(6) NOT NULL, " +  
    "märke VARCHAR(20) NOT NULL, " +  
    "år CHAR(4), " +  
    "CONSTRAINT bil_pk PRIMARY KEY (regnr));";  
  
int result = statement.executeUpdate(sql);  
  
// result blir 0 om allt gick bra
```

Se exempel UpdateDBCreateTable.java.



Uppdatera databasen

- Lägga till data (Statement)
- Om endast ett fåtal rader

```
String sql = "INSERT INTO bil(regnr, märke, år) " +  
    "VALUES(?, ?, ?);";  
  
Statement statement = connection.createStatement();  
  
int result = statement.executeUpdate(sql);  
  
// result blir 1 eftersom en rad påverkades  
// (en ny rad lades till)
```




Uppdatera databasen

- Lägga till data (PreparedStatement)
- Om många rader ska läggas till

```
String sql = "INSERT INTO bil(regnr, märke, år) " +  
            "VALUES(?, ?, ?);";  
  
PreparedStatement preparedStatement =  
    connection.prepareStatement(sql);  
  
preparedStatement.setString(1, "AAA111");  
preparedStatement.setString(2, "Volvo");  
preparedStatement.setString(3, "2013");  
  
int result = preparedStatement.executeUpdate();  
// result blir 1 eftersom en rad påverkades  
// (en ny rad lades till)
```

Se exempel **UpdateDBInsertData.java**.



Uppdatera databasen

- Uppdatera data

```
String sql = "UPDATE bil SET märke = 'Opel' " +  
            "WHERE märke = 'Volvo';";  
  
Statement statement = connection.createStatement();  
  
int result = statement.executeUpdate(sql);  
  
// result blir 2 eftersom det finns två volvobilar  
// som ändras till Opel
```

Se exempel **UpdateDBUpdateData.java**.



Uppdatera databasen

■ Ta bort data

```
String sql = "UPDATE bil SET märke = 'Opel' " +  
            "WHERE märke = 'Volvo';";  
  
Statement statement = connection.createStatement();  
  
int result = statement.executeUpdate(sql);  
  
// result blir 2 eftersom det finns två volvobilar  
// som ändras till Opel
```

Se exempel **UpdateDBDeleteData.java**.



Uppdatera databasen

■ Ta bort tabell

```
String sql = "DROP TABLE bil;";  
  
Statement statement = connection.createStatement();  
  
int result = statement.executeUpdate(sql);  
  
// result blir 0 om allt är ok
```

Se exempel **UpdateDBDeleteTable.java**.



Metadata

- Är data om data
- Dvs information om den data som är lagrad i databasen
- Kan vara:
 - Namn på alla tabeller
 - Namn på kolumnerna i en tabell
 - Vilken datatyp en kolumn har
 - Om databasen/drivrutinen stöder en viss funktion (t.ex. navigera bakåt)
 - etc



Läsa metadata

- Gränssnittet DatabaseMetaData lagrar metadata
 - Fås på följande sätt:
- ```
DatabaseMetaData metaData = connection.getMetaData();
```
- Data fås via ett antal get-metoder
  - Namn och version på databasprodukt

```
String productName =
 metaData.getDatabaseProductName();

String productVersion =
 metaData.getDatabaseProductVersion();
```



## Läsa metadata

- Namn och version på drivrutin

```
String driverName = metaData.getDriverName();
String driverVersion = metaData.getDriverVersion();
```

- Stöds en viss typ av ResultSet?

```
boolean scrollable =
metaData.supportsResultSetType(TYPE_SCROLL_SENSITIVE);
```

- Max antal Connection

```
int maxConnections = metaData.getMaxConnections();
```

- Med flera (titta i API)

Se exemplet **MetaData.java**.



## Läsa metadata

- Vilka tabeller som finns i databasen

```
ResultSet result = metaData.getTables(null, null,
null, null);

while(result.next()) {
 String tableName = result.getString(3);
}
```

- Kolumner som finns i tabellen person

```
ResultSet result = metaData.getColumns(null, null,
"person", null);

while(result.next()) {
 String columnName = result.getString(4);
 int columnDatatype = result.getInt(5);
}
```

Se exemplen **MetaDataListTables.java** och **MetaDataListColumns.java**.



## JTable

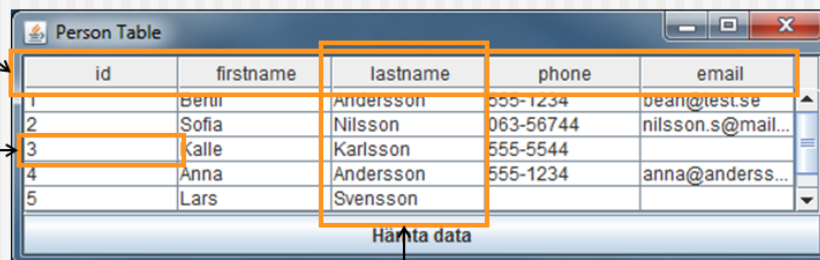
- Är en grafisk komponent
- Kan visa data i tabellform
- Möjlighet att redigera data
- Möjlighet att sortera data
- Möjlighet att filtrera data
- Placeras vanligtvis i en JScrollPane
- Justerar som standard bredden på kolumnerna så den passar 100%



## JTable

Sidhuvud med  
namn på  
kolumnerna

Varje cell visar  
ett datavärde



| id | firstname | lastname  | phone     | email             |
|----|-----------|-----------|-----------|-------------------|
| 1  | Berit     | Andersson | 555-1234  | bean@test.se      |
| 2  | Sofia     | Nilsson   | 063-56744 | nilsson.s@mail... |
| 3  | Kalle     | Karlsson  | 555-5544  |                   |
| 4  | Anna      | Andersson | 555-1234  | anna@anderss...   |
| 5  | Lars      | Svensson  |           |                   |

Hämta data

Varje kolumn  
visar en typ  
av data

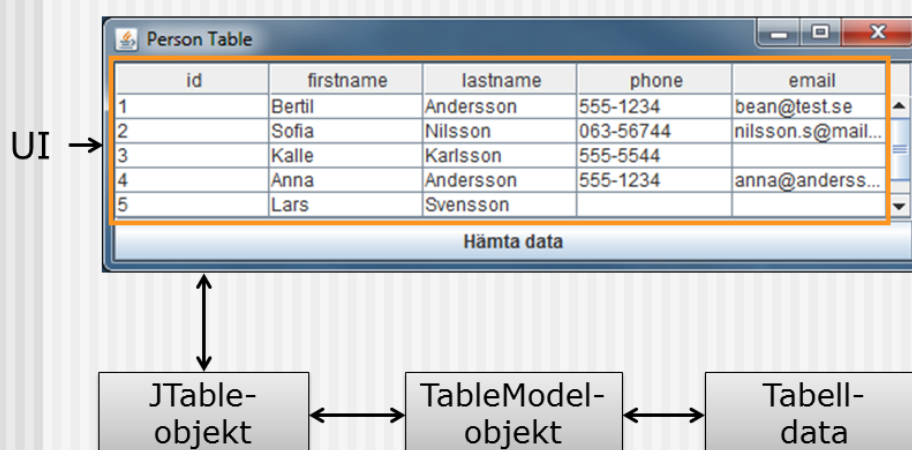


## TableModel

- JTable har många egenskaper som gör det möjligt att anpassa rendering (av UI) och redigering
- Bestäms av gränssnittet TableModel
- Varje tabell har en tabellmodell som hanterar den underliggande data
- DefaultTableModel används om användaren inte själv anger någon



## TableModel





## DefaultTableModel

- Har två konstruktörer som direkt sätter data som ska visas

```
DefaultTableModel(Object[][] data, Object[] columnNames)
```

```
DefaultTableModel(Vector data, Vector columnNames)
```

- Argumentet data är tabellens data
- I det senare fallet är det en: vektor och vektorer av Object-objekt

```
Vector<Vector<Object>> data = new Vector<Vector<Object>>();
```

- Argumentet columnNames är namn på alla kolumner



## Skapa DefaultTableModel

- Skapa först vektor/array med namn

```
Vector<String> n = new Vector<String>();
v.add("id"); v.add("firstname"); ...; v.add("email");
```

- Skapa sen vektor/array med data

```
Vector<Vector<Object>> d = new Vector<Vector<Object>>();
Vector<Object> v = new Vector<Object>();
v.add(1); v.add("Bertil"); ...; v.add("bean@test.se");
d.add(v);
v = new Vector<Object>();
v.add(2); v.add("Sofia"); ...; v.add("nilsson.s@mail.se");
d.add(v);
...
```



## Skapa DefaultTableModel

### ■ Skapa table model

```
DefaultTableModel tableModel =
 new DefaultTableModel(d, n);
```

### ■ Skapa JTable och sätt table model

```
JTable table = new JTable();
table.setModel(tableModel);
container.add(new JScrollPane(table));
```



| id | firstname | lastname  | phone     | email              |
|----|-----------|-----------|-----------|--------------------|
| 1  | Bertil    | Andersson | 555-1234  | bean@test.se       |
| 2  | Sofia     | Nilsson   | 063-56744 | nilsson.s@mail.se  |
| 3  | Kalle     | Karlsson  | 555-5544  |                    |
| 4  | Anna      | Andersson | 555-1234  | anna@andersson.com |
| 5  | Lars      | Svensson  |           |                    |
| 6  | Sara      | Larsson   |           | sara@larsson.net   |

Se exemplet **JTableExample.java**.



## Visa databastabell i JTable

- En SwingWorker som skapar en DefaultTableModel genom att skapa
  1. En Connection till databasen
  2. Ett Statment för att ställa fråga
  3. Ett ResultSet som innehåller alla rader i tabellen person
  4. En vektor med alla kolumnnamn från ResultSet
  5. En vektor av vektorer med all data
- Använd denna i en JTable





# Visa databastabell i JTable

## ■ SwingWorker som en inre klass

```
public class ListAllPersonsGUI extends {
 private JTable personsTable;
 private JButton loadButton;

 class LoadDataWorker extends SwingWorker<DefaultTableModel, Void> {
 @Override
 public DefaultTableModel doInBackground() {
 DefaultTableModel tableModel = createTableModel();
 return tableModel;
 }

 @Override
 public void done() {
 loadButton.setEnabled(true);
 DefaultTableModel tableModel = get();
 personsTable.setModel(tableModel);
 }
 }
}
```

Eftersom databasaccess kan vara relativt tidskrävande måste detta ske i en bakgrundstråd för att inte blockera händelsetråden (event dispatch thread). Eftersom vi skapar ett grafiskt användargränssnitt i Swing är det enklast att implementera bakgrundstråden som en `SwingWorker`. Denna skriver vi som en inre klass vilket ger den direkt åtkomst till medlemmar i den yttre klassen.

I metoden `doInBackground` kopplar vi upp oss mot databasen och skapar en `DefaultTableModel` genom att anropa metoden (som vi skriver själva) `createTableModel`. I denna metod kopplar vi upp oss mot databasen och hämtar alla rader från tabellen person. Från data som hämtas skapar vi vektorer för kolumnnamn och själva data som sen används för att till sist skapa det `DefaultTableModel`-objekt som metoden returnerar.

I metoden `done` hämtar vi denna table model genom ett anrop till metoden `get` (kom ihåg att den returnerar samma värde som `doInBackground` returnerar). Vi använder till sist denna table model i vår `JTable personsTable`.



# Visa databastabell i JTable

## ■ Metoden createTableModel

```
private DefaultTableModel createTableModel() throws SQLException {
 // Försök ansluta mot databasen
 Connection connection = DriverManager.getConnection(
 URL, USERNAME, PASSWORD);

 // STEG 1: Exekvera SQL-fråga för att lista alla personer
 // Skapa ett Statement som vi kan använda för SQL-frågan
 Statement statement = connection.createStatement();

 // SQL-fråga för att lista alla personer
 String sql = "SELECT * FROM person;";

 // Skicka frågan till databasen
 ResultSet result = statement.executeQuery(sql);
 ...
}
```

Metoden `createTableModel` låter vi kasta vidare eventuella `SQLException` som kan uppstå i metoden. I själva metoden börjar vi med att ansluta till databasen genom att skapa ett `Connection`-objekt. Därefter följer steg 1 vilket involverar att skapa ett `Statement`-objekt för att exekvera en fråga som listar alla personer i tabellen `person`. Resultatet av frågan lagras i ett `ResultSet` precis som vanligt.



# Visa databastabell i JTable

## ■ Metoden createTableModel

```
private DefaultTableModel createTableModel() throws SQLException {
 ...
 // STEG 2: Ta reda på kolumnnamn i ResultSet
 // Hämta metadata om ResultSet
 ResultSetMetaData metaData = result.getMetaData();

 // En vektor med namn på kolumnerna
 Vector<String> columnNames = new Vector<String>();

 // Antal kolumner i resultatet
 int columnCount = metaData.getColumnCount();

 // Loopa igenom alla kolumner
 for (int column = 0; column < columnCount; column++) {
 // Hämta namnet på kolumnen på och lägg in det i vektorn
 columnNames.add(metaData.getColumnName(column + 1));
 }
 ...
}
```

I steg 2 skapar vi den vektor med namn på alla kolumner som ska visas i vår JTable. Detta görs genom att vi skapar ett ResultSetMetaData-objekt genom att anropa metoden `getMetaData` på resultatet av SQL-frågan. Ett ResultSetMetaData-objekt påminner till stor del om `DatabasMetaData` vi tittat på tidigare. Vi tar reda på hur många kolumner resultatet innehåller genom att anropa `getColumnCount`. I en loop loopar vi sen lika många gånger som det fanns kolumner och i varje varv lägger vi till kolumnens namn i en vektor.



# Visa databastabell i JTable

## ■ Metoden createTableModel

```
private DefaultTableModel createTableModel() throws SQLException {
 ...
 // STEG 3: Skapa en vektor med resultatet
 Vector<Vector<Object>> data = new Vector<Vector<Object>>();

 while (result.next()) {
 // En vektor som motsvarar en rad i tabellen
 Vector<Object> vector = new Vector<Object>();

 // Loopa igenom alla kolumner
 for (int i = 0; i < columnCount; i++) {
 // Lägg till data i vektorn
 vector.add(result.getObject(i + 1));
 }

 // Lägg till "raden" i data-vektorn
 data.add(vector);
 }
 ...
}
```

Steg 3 handlar om att skapa en vektor (vektor av vektorer av `Object`-objekt) för att lagra tabellens data i. I en `while`-loop loopar vi så länge som det finns mer rader i resultatet. För varje nytt varv i `while`-loopen börjar vi med att skapa en vektor av `Object` som kommer att innehålla varje rads data. I en `for`-loop loopar vi igenom alla kolumner i resultatet och lägger in data i vektorn. I slutet av `while`-loopen lägger vi sen in vektorn som innehåller rad-data i vektorn för resultatet.



# Visa databastabell i JTable

## ■ Metoden createTableModel

```
private DefaultTableModel createTableModel() throws SQLException {
 ...
 // stäng allt
 result.close();
 statement.close();
 connection.close();

 // Skapa en DefaultTableModel av data och kolumnnamnen
 DefaultTableModel tableModel = new DefaultTableModel(data, columnNames);

 return tableModel;
}

@Override
public void done() {
 loadButton.setEnabled(true);
 DefaultTableModel tableModel = get();
 personsTable.setModel(tableModel);
}
```

Sist i metoden stänger vi alla öppna resurser (ResultSet, Statement och Connection). Vi skapar en DefaultTableModel av våra vektorer och returnerar denna. När metoden doInBackground är klar kommer metoden done att automatiskt anropas. I denna hämtar vi den table model som doInBackground returnerar och använder den som table model i vår JTable.

Det som återstår är nu att skapa själva applikationsfönstret (JFrame). I användargränssnittet har vi en knapp (JButton) loadButton som när användaren trycker på den skapar ett objekt av vår SwingWorker och anropas dess execute-metoden. Innan bakgrundstråden skapas sätter vi att knappen inte ska vara klickabar för att förhindra att flera samtidiga SwingWorker exekveras. I metoden done sätter vi knappen att vara klickabar igen.

Se exemplet **PersonsGUI.java**.