Vad är JDBC? Varför är JDBC bra?

- Ett API skrivet i Java för att kunna komma åt data från tabeller
- **JDBC** står numera för "**Java DataBase Connectivity**" (i dagligt tal)
  Från början var det bara ett varumärke
- Gör det enkelt att skicka SQL-frågor till relationsdatabaser
- Har i stort sätt stöd för alla dialekter av SQL
- Kan mera än SQL - kan användas till olika datakällor som t.ex. en textfil som innehåller tabelldata

Vad gör JDBC API'et?

- Enkelt uttryckt gör det möjligt med tre saker:

  1. Upprätta en koppling till en datakälla

  2. Skicka frågor eller uppdaterings -"statements" till datakällan

  3. Behandla resultatet

Hur kan JDBC fungera mot olika databaser?

- Olika databaser (MySQL, Oracle, Jersey) tillhandahåller olika **Drivers** som implementerar **JDBC-intefacen** på olika sätt.

- Användaren arbetar mot interfacen och är därmed avskärmad från implementationsdetaljerna.

Vad måste jag ha för att använda JDBC?

- Från och med J2SDK 1.4 är **JDBC 3.0** inkluderat i API'et
- Någon form av datakälla - vanligtvis en databas
- En JDBC-driver som implementerar de olika interfacen som JDBC-api'et definierar och som arbetar mot den datakälla du använder

Äntligen! Self Crash Course

- Skapa en Java-klass som kopplar upp sig mot din databas och hämtar information från den

- Använd internet för att hitta information om hur du gör

- Inget krav på att du skall lyckas

- Du har 2 timmar på dig

Några av JDBC's huvudbeståndsdelar

- **`DriverManager`**
- **`Driver`**
- **`Connection`**
- **`Statement`**
- **`Resultset`**

DriverManager

- Är en konkret klass till skillnad från många av de andra delarna i JDBC
- **Huvuduppgift:**

  Hålla reda på de olika Driver implementeringarna som är registrerade

  Tillhandahålla en passande Driver som matchar den URL som klienten frågar efter

  Har metoder för att registrera och avregistrera en Driver

  `getDrivers()` - ger en enumeration över de Driver's som är registrerade

Driver

- Ett interface som implementerande klasser använder för att skapa en koppling till en databas
- Kravet är att en klass som implementerar detta interface skall registrera sig själv hos `DriverManager` när den skapas
- Utvecklare som inte skapar egna drivers behöver i stort sätt aldrig använda detta interface direkt
- **För den som vill veta:**
  Implementerande klasser har ofta ett static-initieringsblock som anropas när klassen laddas i ClassLoadern. I det blocket sker registreringen hos DriverManager

Connection

- Ett interface som representerar en koppling mot en databas
- Används för att skicka SQL-satser till databasen
- En applikation kan använda sig av mera än en `Connection` till en eller flera olika databaser
- Följande skapar en `Connection` till en datakälla:

```
String url = "jdbc:mysql://localhost/userdatabase";
Connection con = DriverManager.getConnection(url, "anca01", "password");
```

localhost: host där datakällan finns
userdatabase: namnet på datakällan

Statement

- Är ett interface som representerar ett SQL-statement t.ex. "SELECT * FROM users”
- Definierar tre viktiga metoder:

executeQuery():
Tar en SQL fråga i from av en String som argument och returnerar svaret på frågan som ett ResultSet. Används där det förväntas ett svar från databasen

[con är en öppen Connection]

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT username FROM user");
```

executeUpdate():
Används för att exekvera update-statements (ex. INSERT, UPDATE, DELETE). Tar också en String som argument samt returnerar antalet rader som berördes av kommandot

```
Statement stmt = con.createStatement();
String sql = "UPDATE user SET username='brutus' WHERE username='anca01'”;
int rows = stmt.executeUpdate(sql);
```

execute():
Används då man inte vet om det är en fråga eller en update som kommer att behöva exekvers. Returnerar **true** om det var en fråga som skapade ett ResultSet (dvs. ett resultat) annars **false**.

```
Statement stmt = con.createStatement();
boolean result = stmt.execute("SELECT username FROM user");
```

ResultSet

- Kapslar in resultatet av ett `Statement` (en SQL-fråga) m.a.o. de rader från en tabell som uppfyller SQL-frågan
- Tillhandahåller många metoder för att komma åt kolumner i detta svar
- Hanterar svar bestående av mera än en rad men dock bara en rad åt gången
- Har en get-metod för att komma åt innehållet i en viss kolumn

  get-metoden är överlagrad och har följande signatur: **return-type get[Type](int | String)**

  `getString(1)` returnerar String-värdet i kolumn 1 i ett ResultSet *(notera att första kolumnen är 1 och inte 0)*

  `getString(”username")` returnerar String-värdet i kolumnen ”username" i ett `ResultSet` (svarar mot en kolumn i tabellen)

  Metoden next():
  ResultSet har en *cursor* (pekare) som initialt pekar på positionen innan första raden.
  `next()` gör att cursor stegar fram till nästa rad. Returnerar **`true`** om det gick att stega fram en rad annars **`false`**.

  Följande skapar ett `ResultSet`:

```
[con är en öppen Connection]
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT email FROM user);
```

Att tänka på - SQL NULL != Java NULL

- Om en fråga resulterar i att ett **SQL NULL**-värde returneras är detta inte alltid desamma som Java NULL

- Märks speciellt när en kolumn som lagrar SQL INTEGER tillåter SQL NULL. Det går då inte att mappa från SQL NULL till Java NULL eftersom Java **`int`** inte kan representeras som NULL

- För att få reda på om det är ett SQL NULL-värde som returnerats när du anropat t.ex. **`getInt("SSNO")`** på ett ResultSet anropar du **`wasNull()`** efter det att du anropat get-metoden

  Därför: Gör alltid en koll innan du använder ett värde från en sådan kolumn

```
int result = rs.getInt("size");
if(!rs.wasNull()){
...
}
```

  Om **`rs.getInt("size")`** returnerar **SQL NULL** kommer **result** att innehålla **0** och inte **java null**

Att tänka på - Stäng resurser när du är klar

- Ha alltid ett **finally-block** som stänger de resurser du använt
- Stängningsordningen borde vara `ResultSet`, `Statement`, `Connection`
- Om du stänger en `Connection` stänger den `Statement` automatiskt
- Om du stänger ett `Statement` stänger den `ResultSet` automatiskt
- En del drivers gör dock inte detta därför skall DU alltid göra det

```
finally {
 if (rs != null) {
 rs.close();
}
 if (stmt != null) {
 stmt.close();
}
if (con != null) {
 con.close();
 }
}
```

CoreDev
SYSTEM DEVELOPMENT AND SUPPORT

Övning

- Om du inte redan gjort det laddar du hem **MySQL** och **Sequel Pro**
- Installera dessa och skapa en ny **databas** och en ny **användare** som har alla rättigheter till den
- Databasen skall innehålla tabeller för att lagra **användare** och deras **adressuppgifter**
- Skapa ett sql-script som skapar dessa tabeller och lägger in data i dessa (ungefär 5 användare)
- Skapa klasser i Java som motsvarar en **användare** och **adressuppgifter** (en användare har ett adressuppgiftsobjekt i sig)
- Skapa en klass som kopplar upp sig mot din databas och exponerar metoder för att lagra och hämta användare i databasen

Du kommer att arbeta databasen som du skapar lokalt på den dator du sitter vid. Detta för att du skall ha total kontroll över databasen.