Java Database Connectivity (1)

- what is a better alternative to using SQL for making complex stats queries?
- in the **real world** you would rarely access a database in its own right
  - instead it is used ***within*** a programming language as part of a program
- Different lanaguegs have different APIs for different databases
  - **Java however has the JDBC for nearly all of them**

JDBC

- Its library is in java.sql and javax.sql packages
- looks similar to Oracle SQL (Oracle used to own Java)
- supports *prepared statements*

Simplye JDBC program to create a table:

```
import java.sql.*;

try (final Connection conn = DriverManager.getConnection("jdbc:sqlite:database.db")) {
    conn.createStatement()
            .executeQuery("CREATE TABLE users(username TEXT PRIMARY KEY, password TEXT)");
} catch (final SQLException err) {
    System.out.println(err);
}
```

- if we try and say `import java.sql.*;` - basically import everything

The `final Connection conn` line is basically aksing how you want to connect to the database

`.getConnection("jdbc:NAMEOFENGINE:database.db"`

- replace nameofengine with the specific one e.g. SQLite

OUTPUT:

```
java.sql.SQLException: No suitable driver found for jdbc:sqlite:database.db
```

> And when you've found a suitable driver and added it to your CLASSPATH...

```
import java.sql.*;

try (final Connection conn = DriverManager.getConnection("jdbc:sqlite:database.db")) {
    conn.createStatement()
        .executeQuery("CREATE TABLE users(username TEXT PRIMARY KEY, password TEXT)");
} catch (final SQLException err) {
    System.out.println(err);
}

org.sqlite.SQLiteException: [SQLITE_ERROR] SQL error or missing database (table users already exists)
```

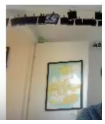
- this new exception is saying the users table already exists
- but we have successfully connected to the database

Adding Users to the database:

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

final var users = new HashMap<String, String>();
users.put("Joseph", "password");
users.put("Matt", "password1");
users.put("Partha", "12345");

try (final Connection conn = DriverManager.getConnection("jdbc:sqlite:database.db")) {
    conn.createStatement().executeUpdate("DELETE FROM users");
    final var statement = conn.prepareStatement("INSERT INTO users VALUES(?, ?)");
    for (final var user : users.keySet()) {
        statement.setString(1, user);
        statement.setString(2, users.get(user));
        statement.executeUpdate();
    }
} catch (final SQLException err) {
    System.out.println(err);
}
```



And list them back out...

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

System.out.println("|User | Password");
try (final Connection conn = DriverManager.getConnection("jdbc:sqlite:database.db")) {
    final var results = conn.createStatement()
        .executeQuery("SELECT * FROM users");
    while (results.next())
        System.out.println("| "+results.getString(1)
                           +" | "+results.getString(2));
} catch (final SQLException err) {
    System.out.println(err);
}
```

User	Password
Matt	password1
Joseph	password
Partha	12345

- remmeber youd have to stick the imports at the top and all the code inside a class
    - changed for the purposes of the presentation doesnt follow this

Prepared Statements

- when adding all the users Jo used a **PreparedStatement**

```
final var statement = conn.prepareStatement("INSERT INTO users VALUES(?, ?)");
for (final var user : users.keySet()) {
    statement.setString(1, user);
    statement.setString(2, users.get(user));
    statement.executeUpdate();
}
```

Isnt there an easier alternative?

- concatenate values by using '+' and get java to joint he strings for you?

```
for (final var user : users.keySet())
    conn.createStatement()
        .executeUpdate("INSERT INTO users "+"VALUES ('"+user+"', '"+users.get(user)+"')");
```

NO this isnt easier

- poses **security risks...**

This leads to a horrible vulnerability:

SQL Injection Attack

The use of a prepared statement basically says: when you say set this thing to be a string, it does all the implicit conversion of this stuff into a string

The basic idea behind an SQL injection attack, including those executed through string concatenation in Java or any other programming language, involves injecting malicious SQL code into a database query. The goal is often to manipulate the database into performing unauthorized actions, such as extracting passwords, other sensitive data, or altering database information.

- it ensures that the things you add are what you say they are
- suppose we have some login code:

```
SELECT username FROM users
WHERE username = "Joseph"
AND password = "password";
```

- this returns the user that is alloweed to login with that password and username combo

Suppose the username and password are taken from a website login form...

- What happens if I try and login with a password of:

```
" OR 1 OR password = "heheh
```

With a prepared statement:

```
SELECT username FROM users
WHERE username = "Joseph"
AND password = """ OR 1 OR password = ""heheh";
```

Without a prepared statement:

```
SELECT username FROM users
WHERE username = "Joseph"
AND password = ""OR 1 OR password = "heheh";
```

username
Matt
Joseph
Partha

- the OR statement without prepared statement will effectively always be true
- we will find all the user names letting you log in as everyone
- the compiler will ususally warn you if you dont use prepared statements
    - SQL injection is however still a problem

Using prepared statements ensures that user input is treated as data and not executable code

- and ensures only expected data types are processed

Transactions

- JDBC makes transactions easy

what are transactions?

Say you were making lots of insertaions and deletions/updates to a database, and then halfway through your computer crashes, or something else goes wrong?

- you could roll back all the new data and changes you made
  - this is tedious
  - automate it!

So this is what transactions are

instead of making a load of single little updates, lets batch together a whole bunch of them and then commit the work once all of them have finished

- if error occurs halfway through wipe all of them at once and start over from scratch

Transaction Workflow

1. start new transaction
2. do your work
3. commit to it when done
4. rollback if errors occur

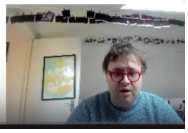
Transactions in JAVA

- `Connection conn` = connection to a database

And in Java please?

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

try (final Connection conn = DriverManager.getConnection("jdbc:sqlite:database.db")) {
    conn.setAutoCommit(false);
    final var save = conn.setSavepoint();
    try {
        conn.createStatement() .executeQuery("INSERT INTO users VALUES ('Alice', 'pa55w0rd')");
        conn.createStatement() .executeQuery("INSERT INTO users VALUES ('Bob', 'Pa55w0Rd7')");
        if (true) throw new Exception("Whoops!");
        conn.createStatement() .executeQuery("INSERT INTO users VALUES ('Eve', 'backd00r')");
        conn.commit();
    } catch (final Exception err) {
        conn.rollback(save);
    } finally {
        conn.setAutoCommit(true);
    }
} catch (final SQLException err) {
    System.out.println(err);
}
```



- look at befroe the catch statement, conn.commit();
  - i.e. commit to changes, if someoen goes wrong catch it then: conn.rollback(save);
    - this take syou back to the save point at the beginning of the code: final var save = conn.setSavePoint();

```
SELECT * FROM users;
```

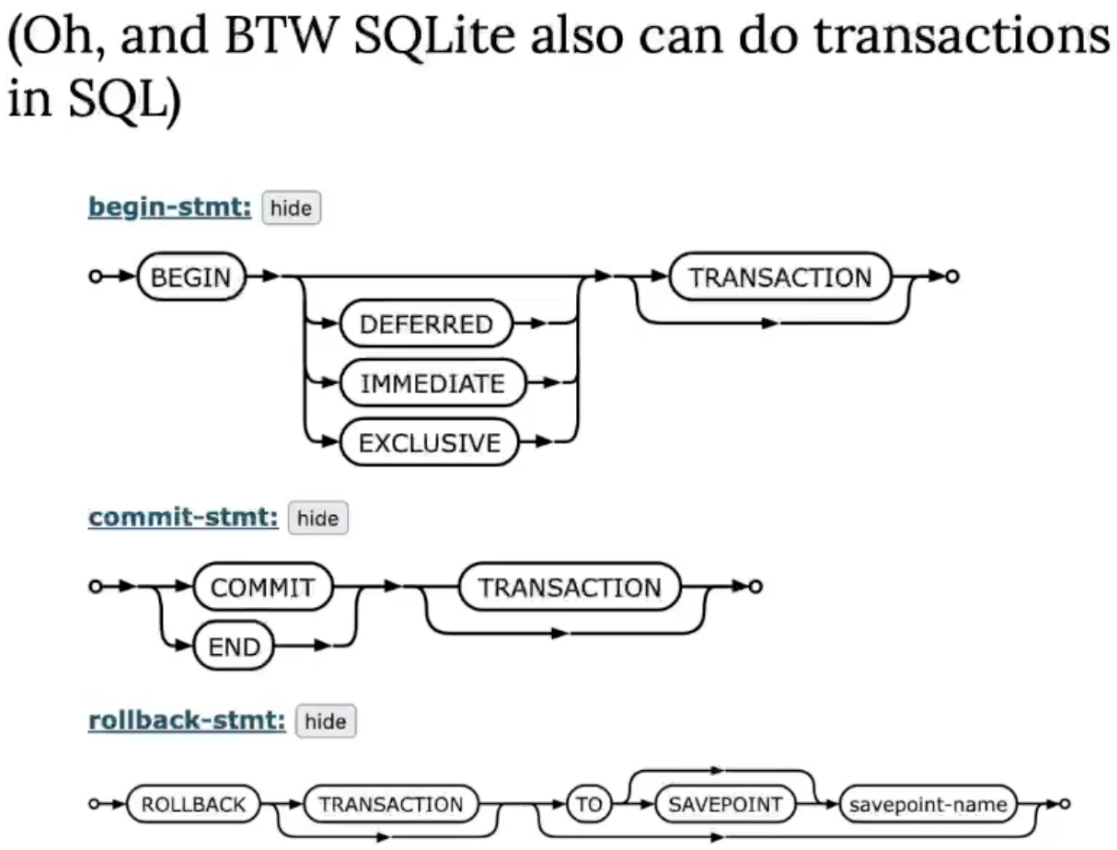
username	password
Matt	password1
Joseph	password
Partha	12345

Our table *remains* unaltered... the whole transaction was *rolled back*.

- becasue the above Java code was designed to rollback, the new users and their passwords (see code snippet) were never added, hence the output in the

image directly above

Transactions in SQLite:

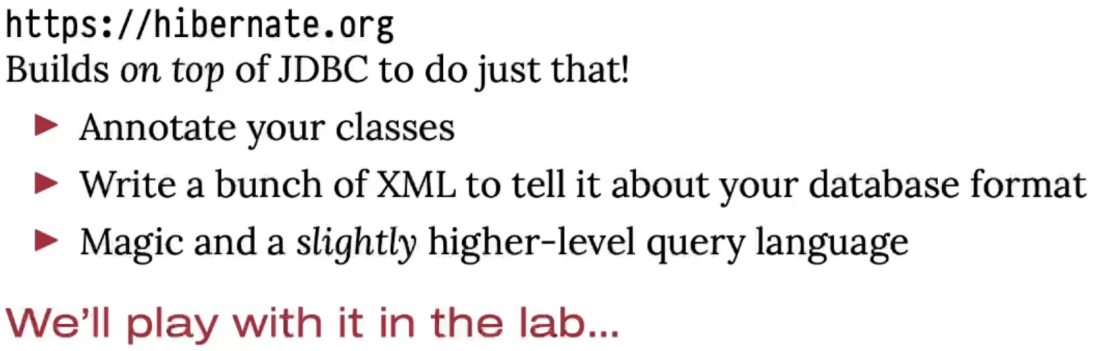


Java Classes as Entity Relationship Diagrams

- it would be nice if we could take a Java class and get the database importing and saving all handled for us
    - i.e. here is a database, map it to these series of classes
- we can...

Hibernate

- this is a library for doing just this - creating entity relationship diagrams (they are classes)



Conclusion

- JDBC lets you access SQL from Java
  - make sure youre in the right driver
  - catch SQL exceptions
  - use prepared statements and transactions to prevent errors or security risks
  - can use an Object Relation Manager (ORM) such as Hibernate to help you if you want