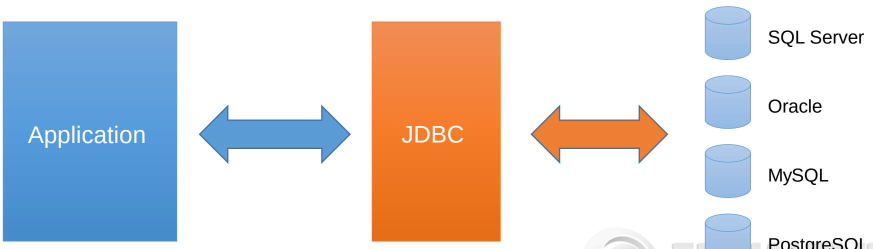
JDBC

Java Database Connectivity (JDBC)

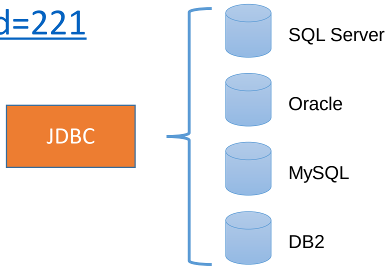
JDBC is how we connect to databases in Java

JDBC provides a standard **API** for connecting to many different types of databases.



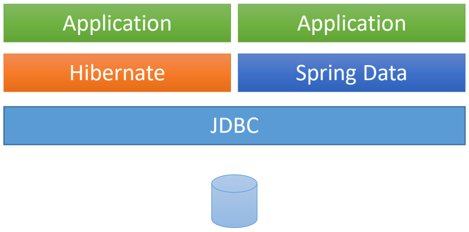
Why JDBC?

- Standard Interface
- Can write portable code against many RDBMSs
- Lower learning curve/more consistency across Java projects.
- JDBC v4 specified in JSR 221 https://jcp.org/en/jsr/detail?id=221

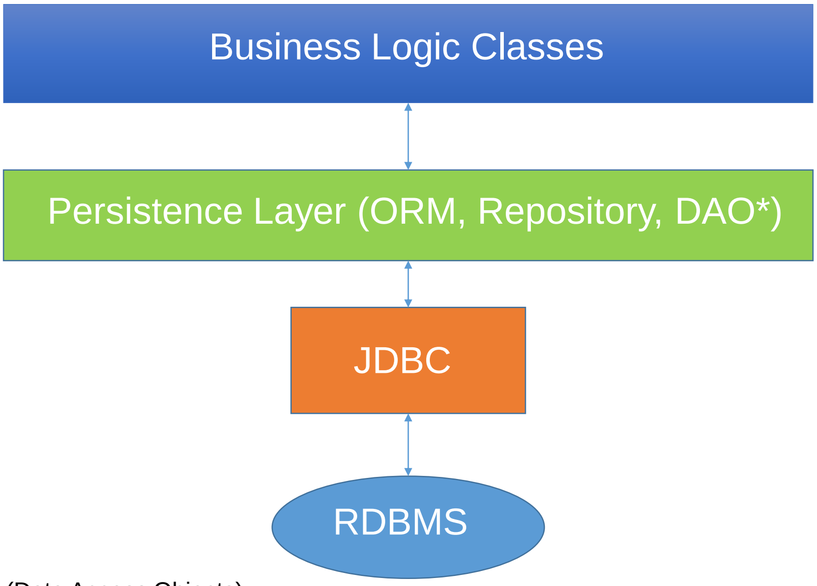


JDBC and Multiple DB Support

- You can write SQL queries that run on all supported DB's
- JDBC provides a common set of data types & interfaces.
- Your application doesn't have to know the db targeted
- Other DB-independent libraries (like Hibernate, Spring Data) can run on top of JDBC



The Big Picture


Business Logic Classes

Persistence Layer (ORM, Repository, DAO*)

JDBC

RDBMS

*DAO (Data Access Objects)

Connecting to the Database

Connecting Logic

The basic code to connect to a database looks like this:

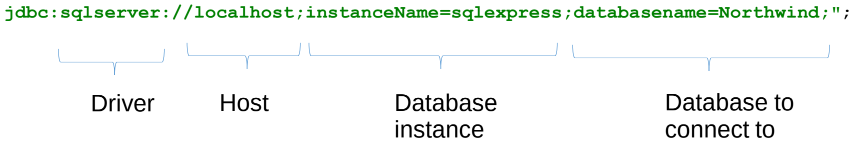
```
String connstr = "jdbc:sqlserver://localhost;instanceName=sqlexpress;databasename=Northwind;
                  user=sqluser;password=password";

Connection dbconn = null;
try {
    dbconn = DriverManager.getConnection(connstr);

    //Write your DB logic here

} catch (SQLException e) {
    e.printStackTrace();
}
finally {
    if(dbconn!=null)
        dbconn.close();
}
```

The connection string

Make sure we always close the connection

The Connection String

The Connection String tells us how and where to connect



```
jdbc:sqlserver://localhost;instanceName=sqlexpress;databasename=Northwind;";
```

Driver

Host

Database instance

Database to connect to

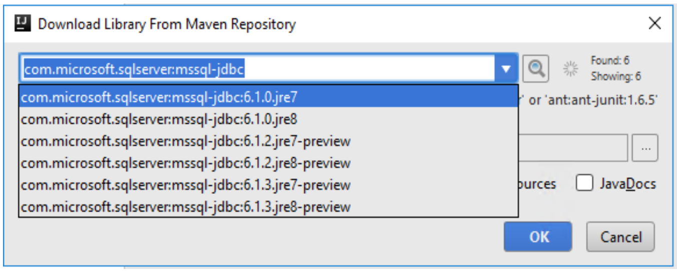
- Connection strings are URIs
- Start with jdbc:[driver]://
- Drivers have specific options
- https://www.connectionstrings.com/

Driver Registration

To use a **driver**, we first have to load it

The easiest is to import it using **Maven**:

```
com.microsoft.sqlserver:mssql-jdbc
```



Download Library From Maven Repository

com.microsoft.sqlserver:mssql-jdbc

com.microsoft.sqlserver:mssql-jdbc:6.1.0.jre7
com.microsoft.sqlserver:mssql-jdbc:6.1.0.jre8
com.microsoft.sqlserver:mssql-jdbc:6.1.2.jre7-preview
com.microsoft.sqlserver:mssql-jdbc:6.1.2.jre8-preview
com.microsoft.sqlserver:mssql-jdbc:6.1.3.jre7-preview
com.microsoft.sqlserver:mssql-jdbc:6.1.3.jre8-preview

Connection Exceptions

Most JDBC commands, if unsuccessful, throw a checked **SQLException**

You **must** handle these!

```
try {
    dbconn = DriverManager.getConnection(constr);
    //do some stuff
    dbconn.close();
} catch (SQLException e) {
    e.printStackTrace();
    throw new RuntimeException(e.getMessage());
};
```

Try with Resources

Fortunately, JDBC also works with Java’s new “try with resources” paradigm:

```
try (dbconn = DriverManager.getConnection(connstr)){
    // do some stuff here with the db
    // Connection is automatically closed during
    // garbage collection at end of try block.
} catch (SQLException e) {
    e.printStackTrace();
    throw new RuntimeException(e.getMessage());
};
```

https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

Exercise

Let's do exercise 1-4

Executing Utility Statements

Understanding Query Types

When we run a query against a database, we are usually interest in one of two things:

1. We are interested in the results of the query, or
2. We are interested in the query performing a task

Handling the results will be discussed in the next section, but for now let’s just talk about doing things.

executeUpdate

The `executeUpdate(String)` method returns an int.

Use it when you do not want the returned results.

This particularly applies to utility statements (think of “updating” the database structure).

Also used for many insert/update/delete statements.

```
try (Statement sth = dbconn.createStatement()){
    sth.executeUpdate(“CREATE TABLE test (id int)");
} catch (SQLException e) {
    e.printStackTrace();
    throw new RuntimeException(e.getMessage());

}
```

Querying and Manipulating Data

PreparedStatement and Parameterization

So we want to look up a beer by name

```
String stmt = "SELECT * FROM beers where name = " + userInput;
try {
    Statement sth = dbconn.createStatement();
    ResultSet results = sth.executeQuery(stmt);
    // process results
} catch (SQLException e) {
    e.printStackTrace();
    throw new RuntimeException(e.getMessage());
}
```

This seems to work, but why is it a bad idea?
What could a user submit that would cause a problem?

PreparedStatement and Parameterization

Using the **prepareStatement** interface, we can parameterize statements.

This prevents SQL injection attacks.

```
String stmt = "SELECT * FROM beers where id = ?";
try {
    PreparedStatement sth = dbconn.prepareStatement(stmt);
    sth.setInt(1, beerId);    ← Setting the parameter noted by ?

    ResultSet results = sth.executeQuery();
    // TODO, see next slide
} catch (SQLException e) {
    e.printStackTrace();
    throw new RuntimeException(e.getMessage());
}
```

Always parameterize statements when possible

Cursors and ResultSets

When we want results back, we use **executeQuery**()
and process a **ResultSet**.

```
String stmt = "SELECT * FROM beers where id = ?";
try {
    PreparedStatement sth = dbconn.prepareStatement(stmt);
    sth.setInt(1, beerId);

    ResultSet results = sth.executeQuery();
    // TODO

} catch (SQLException e) {
    e.printStackTrace();
    throw new RuntimeException(e.getMessage());
}
```

Cursors and ResultSets

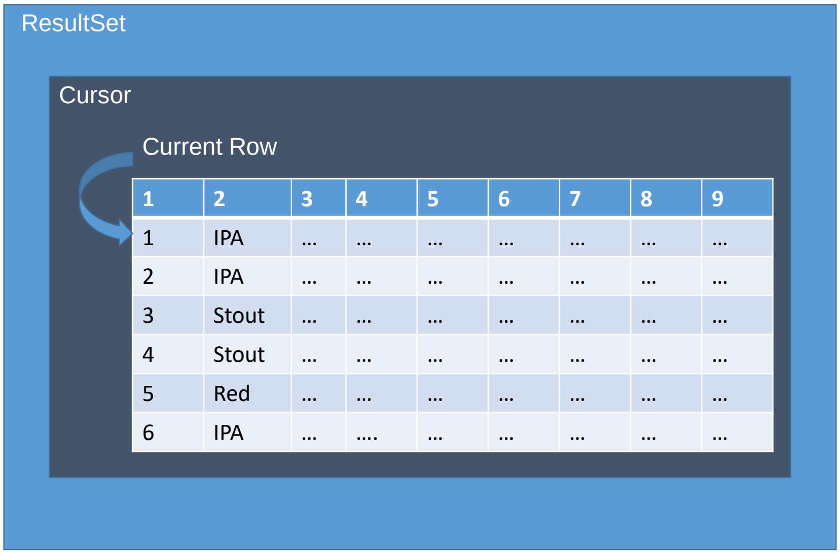
To access the **ResultSet** we start by calling **next()**

Then we can access elements by their index.

```
String stmt = "SELECT * FROM beers where id = ?";
try {
    PreparedStatement sth = dbconn.prepareStatement(stmt);
    sth.setInt(1, beerId);

    ResultSet results = sth.executeQuery();

    results.next(); // advances to first row

    beer = new Beer(res.getInt(1), res.getString(2))

} catch (SQLException e) {
    e.printStackTrace();
    throw new RuntimeException(e.getMessage());
}
```

Cursors and ResultSets

The **ResultSet** wraps a cursor



1	2	3	4	5	6	7	8	9
1	IPA	…	…	…	…	…	…	…
2	IPA	…	…	…	…	…	…	…
3	Stout	…	…	…	…	…	…	…
4	Stout	…	…	…	…	…	…	…
5	Red	…	…	…	…	…	…	…
6	IPA	…	….	…	…	…	…	…

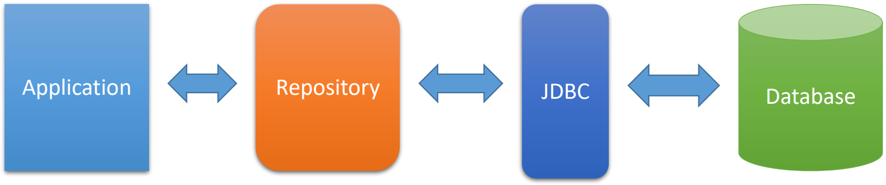
Repository Pattern

The Repository Explained

Some programs benefit from an abstraction layer around object persistence.

One such approach is the **repository** pattern.

- The repository is responsible for all data operations
- The repository has no knowledge of object internals
- Objects do not care about their own persistence



Exercise

Let's do exercise 5-9