

**What is the most used
language in programming?**

Profanity

Module 2-6

JDBC and DAO Pattern

Objectives

- Making Connections
- Executing SQL statements
- Parameterized Queries
- DAO pattern

JDBC Basics

JDBC Introduction

JDBC (Java Database Connectivity) is an API that is part of standard Java, made available to facilitate connections to a database.

- Our main task in this lecture is to understand the collaborator classes and methods that will be needed to talk to a Postgresql database.

The DataSource Class

- The DataSource class is responsible for creating a connection to a database.
- There are 4 methods we will be concerned with:
 - **.setURL(<<String with URL>>)**: Sets the network location of the database, it could be a localhost connection to a database on your own workstation.
 - **.setUsername(<<Username String>>)**: Sets the username for the database.
 - **.setPassword(<<Password String>>)**: Sets the password for the database.
 - **.getConnection()**: returns a connection object that will be used for running queries.
- Here is an example of a DataSource class being initialized and some of the above methods invoked:

```
BasicDataSource dataSource = new BasicDataSource();  
dataSource.setUrl("jdbc:postgresql://localhost:5432/dvdstore");  
dataSource.setUsername("postgres");  
dataSource.setPassword("postgres1");
```

The DataSource Class

There is a more elegant way to manage the credentials and connection strings. This will be a topic for when we discuss “Dependency Injection” later in Module 2.

The Connection Class

- The Connection class creates a session for any database transactions.
 - As a sidenote, typically connections are “pooled,” meaning they are stored within the JVM’s memory and reused during a given session. This is meant to reduce the amount of heavy lifting needed to establish a database connection.
- The connection object can be instantiated by simply having a DataSource object “pass the ball” through the aforementioned getConnection() method, like so:

```
Connection conn = dataSource.getConnection();
```


The Statement Class

- The Statement object is responsible for the execution of the actual SQL command.
- The object can be instantiated by having the connection “pass the ball” with the `createStatement` method.

```
Statement stmt = conn.createStatement();
```

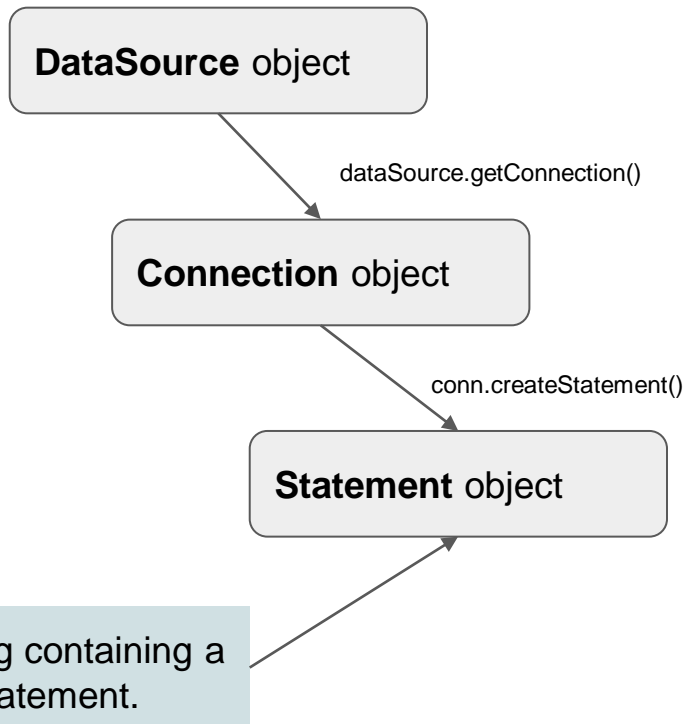
- The **.executeQuery**(<<String containing SQL>>) method is used to run the SQL command.

```
String aSQLStatement = "SELECT name from country";  
stmt.executeQuery(aSQLStatement);
```

The story so far...

dataSource is an object of class DataSource.

conn is an object of class Connection.



But how do we get the results back and do something meaningful with Java?

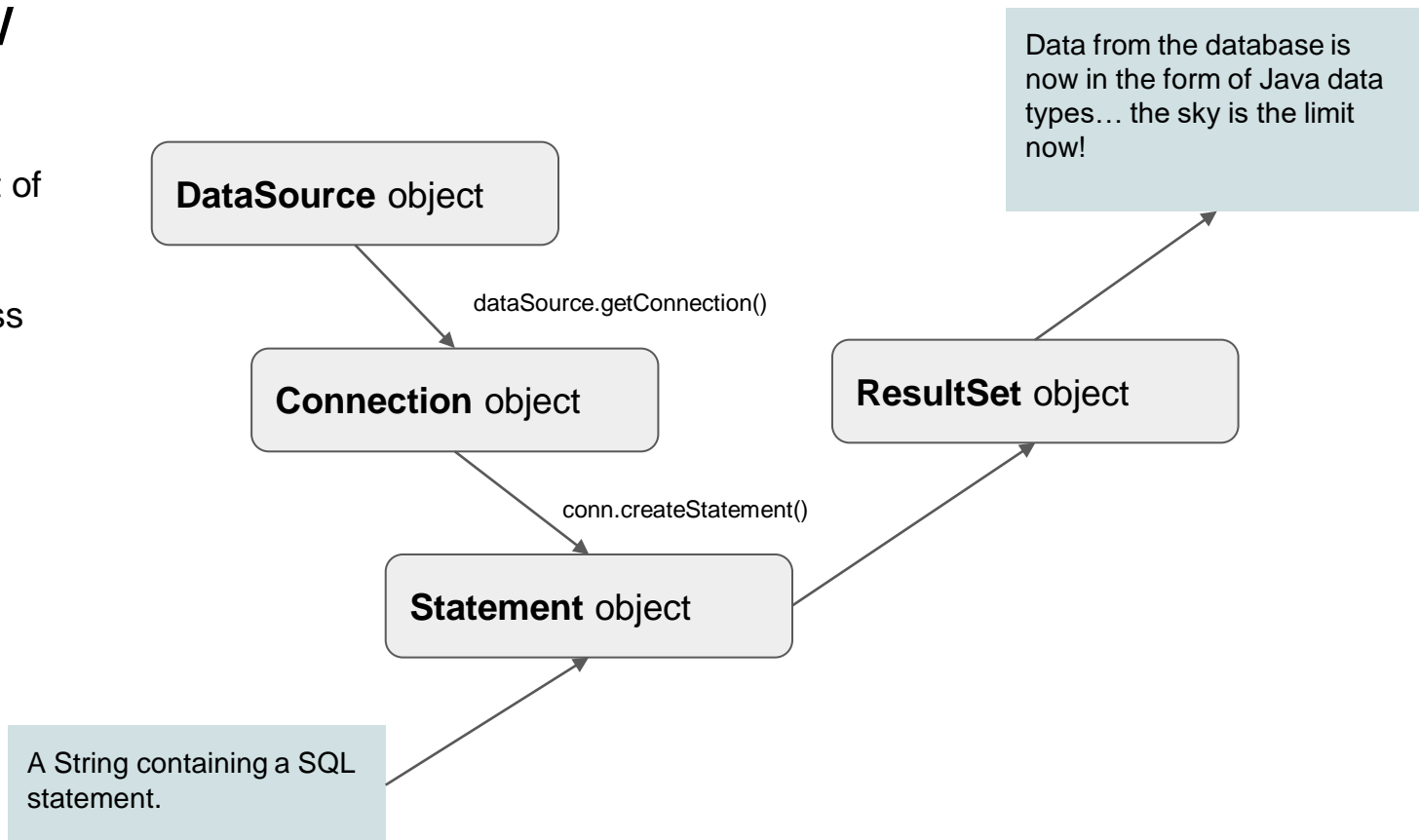
The ResultSet Class

- The ResultSet class is the collaborator in charge of storing the results from the query.
- It contains several meaningful methods:
 - **.next()**: This method allows for iteration if the SQL operation returns multiple rows. Using next is very similar to the way we dealt with file processing.
 - **.getString(<<name of column in SQL result>>)** , **getInt(<<name of column in SQL result>>)**, **getBoolean(<<name of column in SQL result>>)** ,etc. : These get the values for a given column, for a given row.

JDBC Flow

dataSource is an object of class DataSource.

conn is an object of class Connection.



Example

Spring JDBC

JDBC Introduction

You might have noticed that the end to end process previously described involved multiple steps and collaborators, a process that is repetitive and could be error prone.

- Spring is a popular Java framework that abstracts various operations (i.e. querying a database) to a higher level such that it's easier for developers to work with.
- Spring provides a **JdbcTemplate** class that accomplishes the previous operation in less lines of code.

JdbcTemplate Class

- The JDBC template's constructor requires a data source. You can pass it the same data source object described in the regular JDBC workflow:

```
BasicDataSource dataSource = new BasicDataSource();  
dataSource.setUrl("jdbc:postgresql://localhost:5432/dvdstore");  
dataSource.setUsername("postgres");  
dataSource.setPassword("postgres1");  
  
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource)
```


JdbcTemplate Class

- Instead of having to create a separate connection and a separate Connection and Statement object, the JdbcTemplate essentially does it all.
- The .queryForRowSet(<<String containing SQL>>)method will execute the SQL query.
 - Extra parameter constructor are available as well, allowing for any prepared statement placeholders.

```
String sqlString = "SELECT name from country";  
SqlRowSet results = jdbcTemplate.queryForRowSet(sqlString);
```

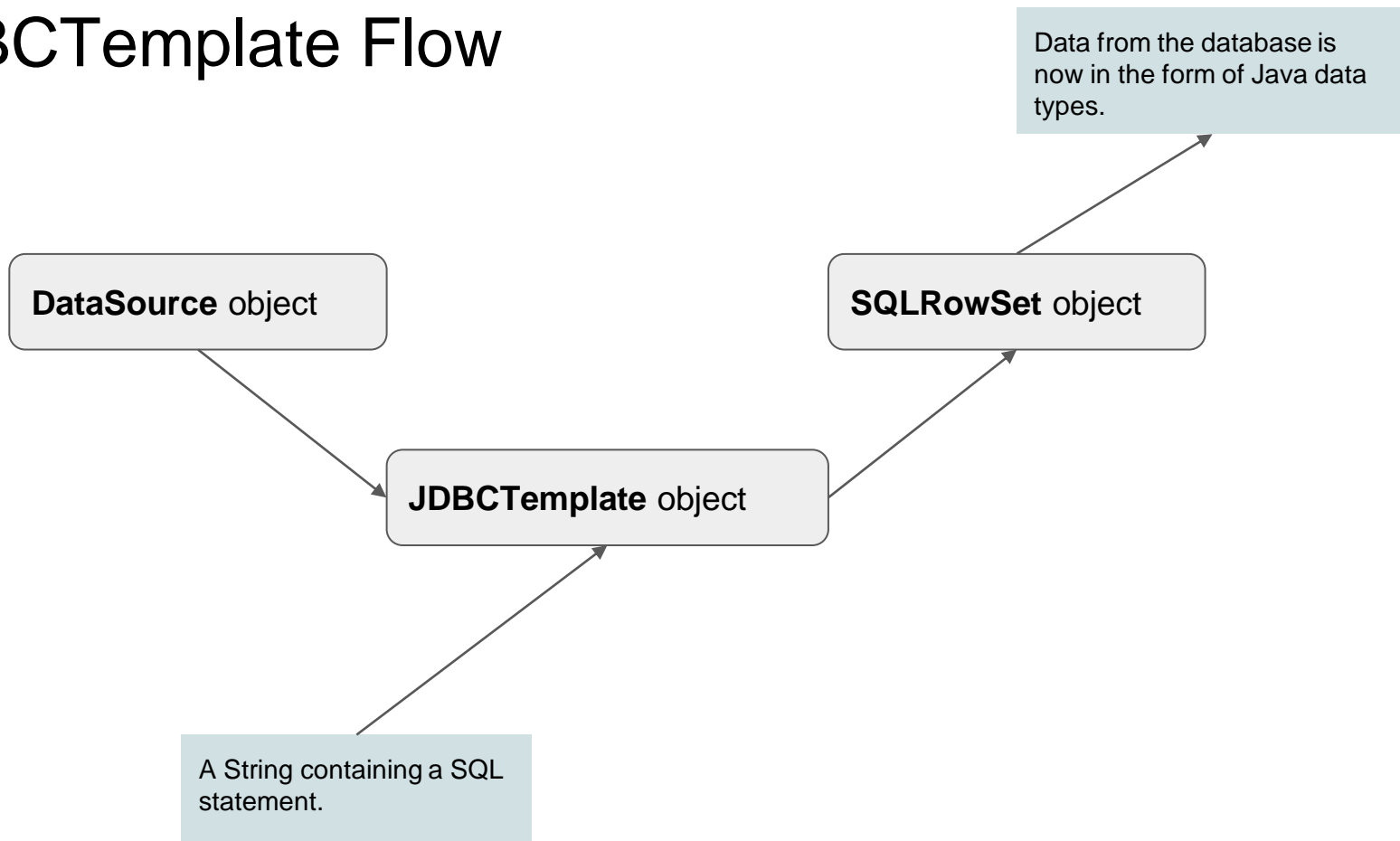
- For UPDATE, INSERT, and DELETE statements we will use the **.update** method instead of the .queryForRowSet method.

```
SqlRowSet results = jdbcTemplate.update(sqlString);  
// Where sqlString contains an UPDATE, INSERT, or DELETE.
```

JdbcTemplate Class

- The results are stored in an object of class `SqlRowSet`, this behaves the exact same way as a `ResultSet` object:
- The same methods available to `ResultSet` are also available here:
 - **.next()**: This method allows for iteration if the SQL operation returns multiple rows. Using `next` is very similar to the way we dealt with file processing.
 - **.getString(<<name of column in SQL result>>)** , **getInt(<<name of column in SQL result>>)**, **getBoolean(<<name of column in SQL result>>)** ,etc. : These get the values for a given column, for a given row.

JdbcTemplate Flow



Example

DAO Pattern

DAO Pattern

- A database table can sometimes map fully or partially to an existing class in Java. This is known as **Object-Relational Mapping**.
- We implement the Object Relation Mapping with a design pattern called DAO, which is short for **Data Access Object**.
- We do this in a very specific way using Interfaces so that future changes to our data infrastructure (i.e. migrating from 1 database platform to another) have minimal changes on the our business logic.

DAO Pattern Step 1

- We start off with a Interface specifying that a class that chooses to implement the interface must implement methods to communicate with a database (i.e. search, update, delete). Consider the following example:

```
public interface CityDAO {  
    public void save(City newCity);  
    public City findCityById(long id);  
}
```

DAO Pattern Step 2

- Next, we want to go ahead and create a concrete class that implements the interface:

DAO Pattern Step 2

```
public class JDBCCityDAO implements CityDAO {  
  
    private JdbcTemplate jdbcTemplate;  
  
    public JDBCCityDAO(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
  
    @Override  
    public void save(City newCity) {  
        String sqlInsertCity = "INSERT INTO city(id, name, countrycode, district, population) " +  
                                "VALUES(?, ?, ?, ?, ?)";  
        newCity.setId(getNextCityId());  
        jdbcTemplate.update(sqlInsertCity, newCity.getId(), newCity.getName(), newCity.getCountryCode(),  
                            newCity.getDistrict(), newCity.getPopulation());  
    }  
  
    @Override  
    public City findCityById(long id) {  
        City theCity = null;  
        String sqlFindCityById = "SELECT id, name, countrycode, district, population " +  
                                  "FROM city " +  
                                  "WHERE id = ?";  
        SqlRowSet results = jdbcTemplate.queryForRowSet(sqlFindCityById, id);  
        if(results.next()) {  
            theCity = mapRowToCity(results);  
        }  
        return theCity;  
    }  
}
```

The contractual obligations of the interface are met.

DAO Pattern Step 3

- In our orchestrator class, we will be using a polymorphism pattern to declare our DAO objects:

```
CityDAO dao = new JDBCCityDAO(worldDataSource);
```

The Interface Reference


The Concrete Class Constructor

DAO Pattern Step 3

- In our orchestrator class, we will be using a polymorphism pattern to declare our DAO objects:

```
City smallville = new City();  
smallville.setCountryCode("USA");  
smallville.setDistrict("KS");  
smallville.setName("Smallville");  
smallville.setPopulation(42080);  
  
dao.save(smallville);  
  
City theCity = dao.findCityById(smallville.getId());
```

We can now call the methods that are defined in concrete class and required by the interface.



Example

DAO Pattern – different way of returning object

```
public class JDBCCityDAO implements CityDAO {

    private JdbcTemplate jdbcTemplate;

    public JDBCCityDAO(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    @Override
    public void save(City newCity) {
        String sqlInsertCity = "INSERT INTO city(name, countrycode, district, population) " +
            "VALUES(?, ?, ?, ?) RETURNING id";

        Long id = jdbcTemplate.queryForObject(sqlInsertCity, new Object[] { newCity.getName(),
            newCity.getCountryCode(), newCity.getDistrict(), newCity.getPopulation() }, Long.class );
        // you can either return the city id or you can update the object (newCity) and return the new object
    }

    @Override
    public City findCityById(long id) {
        City theCity = null;
        String sqlFindCityById = "SELECT id, name, countrycode, district, population " +
            "FROM city " +
            "WHERE id = ?";

        SqlRowSet results = jdbcTemplate.queryForRowSet(sqlFindCityById, id);
        if(results.next()) {
            theCity = mapRowToCity(results);
        }
        return theCity;
    }
}
```

Create an object from the values of the City that match the column names in the INSERT statement.