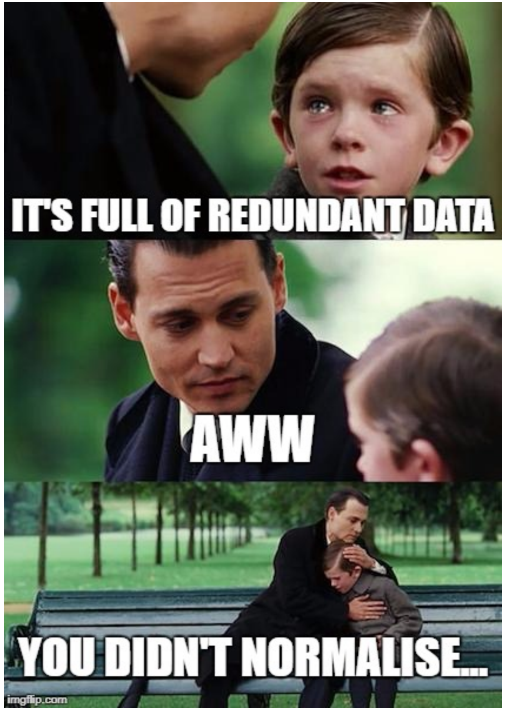

IT'S FULL OF REDUNDANT DATA
AWW
YOU DIDN'T NORMALISE...
imgflip.com

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

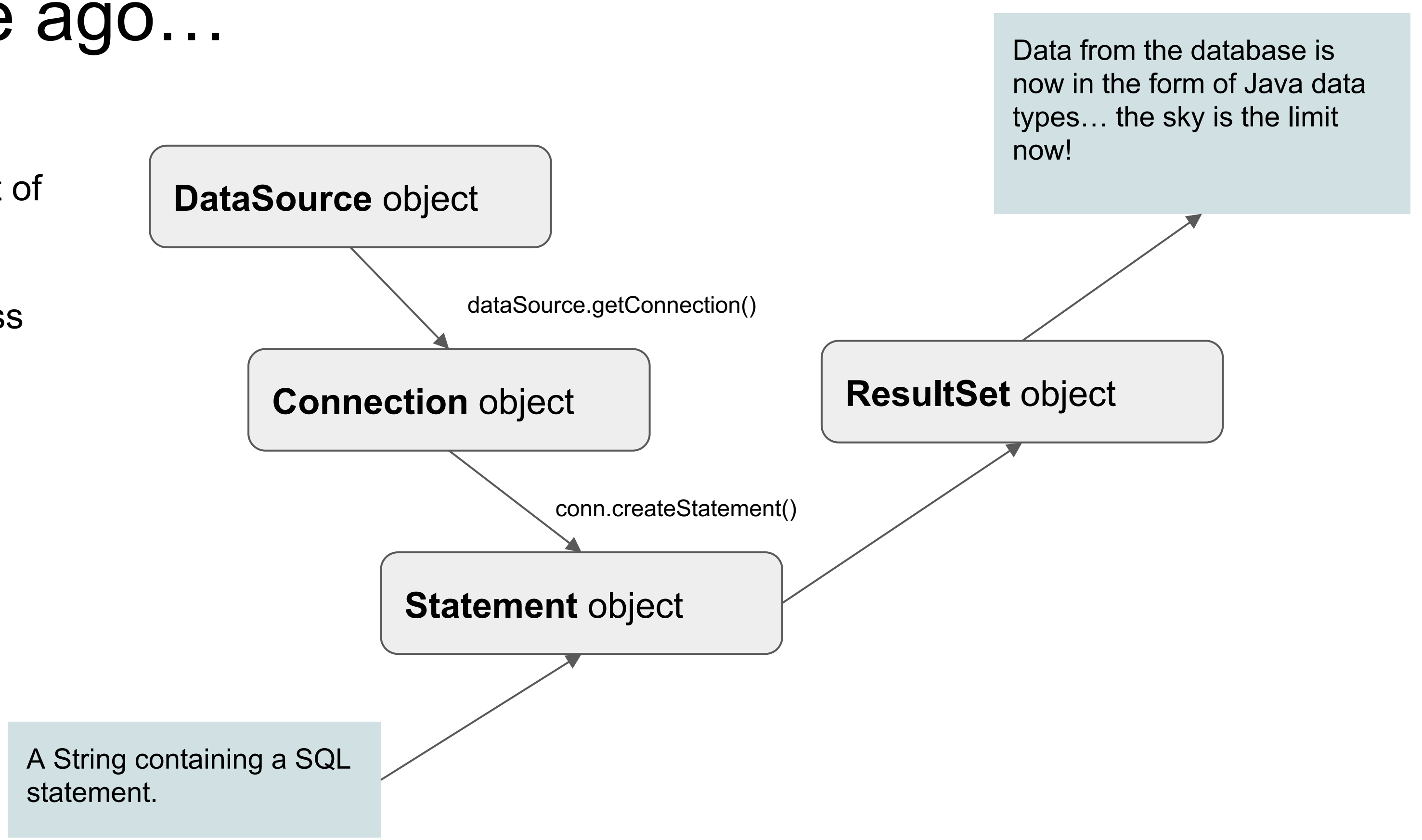
- The DataSource class is responsible for creating a connection to a database.
- There are 4 methods we will be concerned with:
    - **.setURL(<<String with URL>>)**: Sets the network location of the database, it could be a localhost connection to a database on your own workstation.
    - **.setUsername**(**<<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");
```

A long time ago…

dataSource is an object of class DataSource.

conn is an object of class Connection.



DataSource object

dataSource.getConnection()

Connection object

conn.createStatement()

Statement object

A String containing a SQL statement.

ResultSet object

Data from the database is now in the form of Java data types… the sky is the limit now!

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 operations 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 and SqlRowSet

- The .queryForRowSet(<<String containing SQL>>)method will execute the SQL query.
  - Extra parameter constructor are available as well, allowing for any prepared statement placeholders.

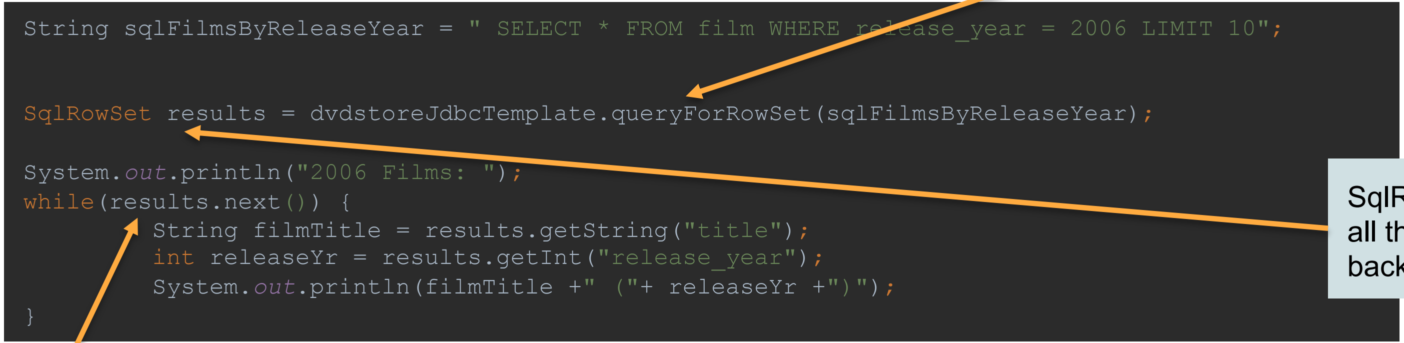
```
String sqlString = “SELECT name from country”;
SqlRowSet results = jdbcTemplate.queryForRowSet(sqlString);
```

- 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

QueryForRowSet – performs query to the database



```
String sqlFilmsByReleaseYear = " SELECT * FROM film WHERE release_year = 2006 LIMIT 10";


SqlRowSet results = dvdstoreJdbcTemplate.queryForRowSet(sqlFilmsByReleaseYear);

System.out.println("2006 Films: ");
while(results.next()) {
         String filmTitle = results.getString("title");
         int releaseYr = results.getInt("release_year");
         System.out.println(filmTitle +" ("+ releaseYr +")");
}
```

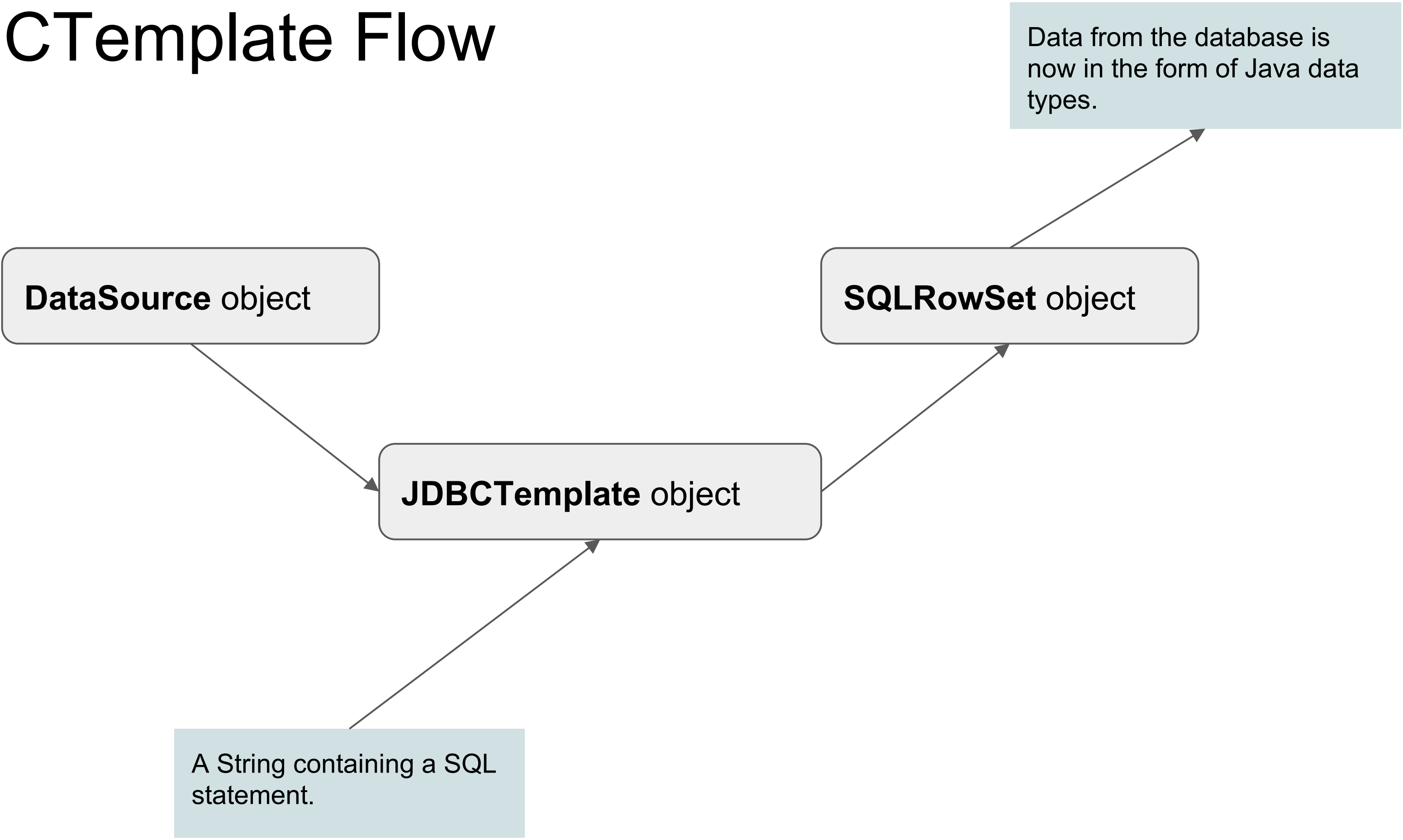
SqlRowSet is a set containing all the data (rows) coming back from database

While loop loops through the results and turns the data being returned into Java data types to be displayed

JDBCTemplate Class

- The results are stored in an object of class SqlRowSet which give us method to let us read the results from the set of data:
    - **.next()**: This methods 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



DataSource object

JDBCTemplate object

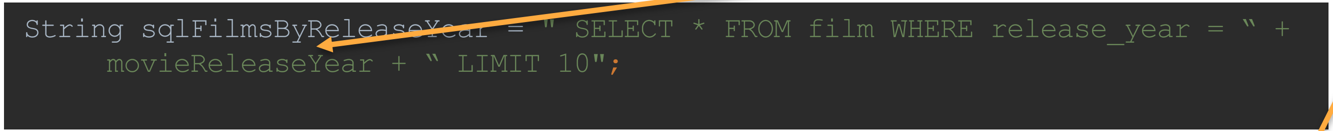
A String containing a SQL statement.

SQLRowSet object

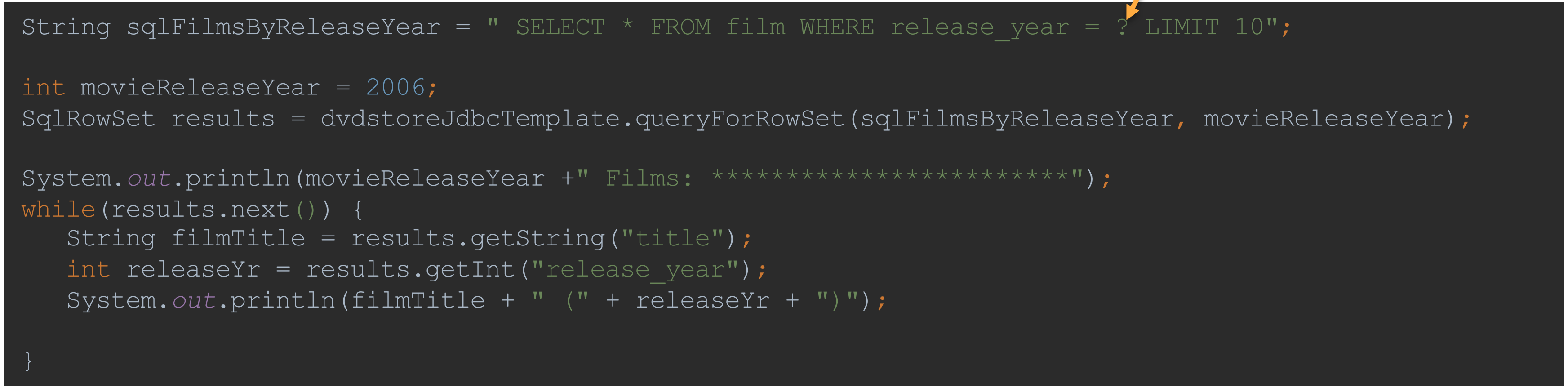
Data from the database is now in the form of Java data types.

Parameterized Queries

It is not a good idea to use the concatenation - better to use parameters



```java
String sqlFilmsByReleaseYear = " SELECT * FROM film WHERE release_year = “ +
     movieReleaseYear + “ LIMIT 10";
```



```java
String sqlFilmsByReleaseYear = " SELECT * FROM film WHERE release_year = ? LIMIT 10";

int movieReleaseYear = 2006;
SqlRowSet results = dvdstoreJdbcTemplate.queryForRowSet(sqlFilmsByReleaseYear, movieReleaseYear);

System.out.println(movieReleaseYear +" Films: ************************");
while(results.next()) {
   String filmTitle = results.getString("title");
   int releaseYr = results.getInt("release_year");
   System.out.println(filmTitle + " (" + releaseYr + ")");

}
```

Let’s Code!

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 {  // CRUD - create, read, update, delete
     public void save(City newCity);  // c - create
     public City findCityById(long id);  // r - read
}
```

DAO Pattern Step 2

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

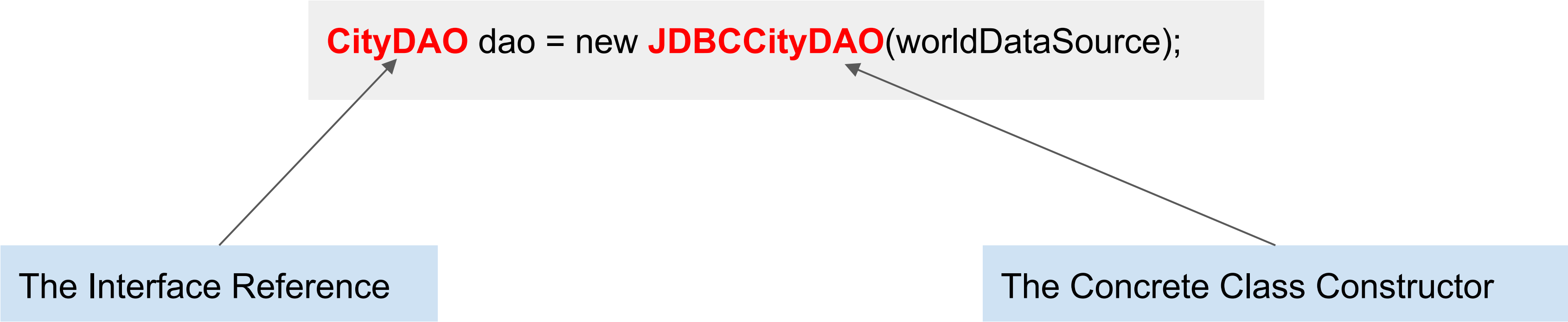
DAO Pattern Step 2

```
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.

What is the most used language in programming?

Profanity

Objectives

- Making Connections

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

Objectives

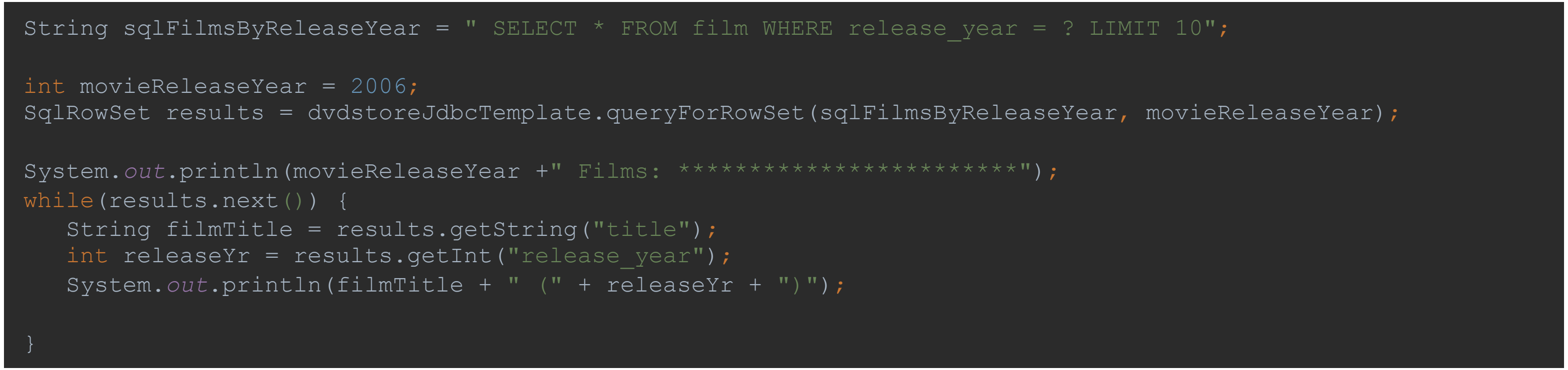
- Making Connections
- Executing SQL statements

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

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

Objectives

- Making Connections
- Executing SQL statements
- Parameterized Queries



```java
String sqlFilmsByReleaseYear = " SELECT * FROM film WHERE release_year = ? LIMIT 10";

int movieReleaseYear = 2006;
SqlRowSet results = dvdstoreJdbcTemplate.queryForRowSet(sqlFilmsByReleaseYear, movieReleaseYear);

System.out.println(movieReleaseYear +" Films: ************************");
while(results.next()) {
   String filmTitle = results.getString("title");
   int releaseYr = results.getInt("release_year");
   System.out.println(filmTitle + " (" + releaseYr + ")");

}
```

Objectives

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

```
public interface CityDAO {  // CRUD - create, read, update, delete
     public void save(City newCity);  // c - create
     public City findCityById(long id);  // r - read
}
```

```
public class City {
   private Long id;
   private String name;
   private String countryCode;
   private String district;
   private int population;

   public City() {
   }
…
}
```

```
public class DAOExample {

   public static void main(String[] args) {

     BasicDataSource worldDataSource = new BasicDataSource();
     worldDataSource.setUrl("jdbc:postgresql://localhost:5432/world");
     worldDataSource.setUsername("postgres");
     worldDataSource.setPassword("postgres1");

     CityDAO dao = new JDBCCityDAO(worldDataSource);

     City smallville = new City();
     smallville.setCountryCode("USA");
     smallville.setDistrict("Kansas");
     smallville.setName("Smallville");
     smallville.setPopulation(42080);

     dao.save(smallville);

     City theCity = dao.findCityById(smallville.getId());

   }
}
```

```
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;
}
```