Module 2-6

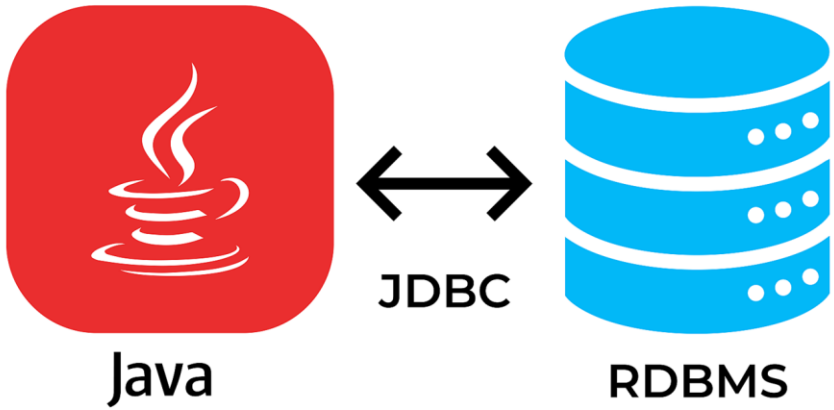
JDBC and DAO Pattern

Objectives

- JDBC
- Spring JDBC
- Making database connections
- Executing SQL statements
- Parameterized Queries
- DAO pattern

JDBC

Java Database Connectivity



JDBC Introduction

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

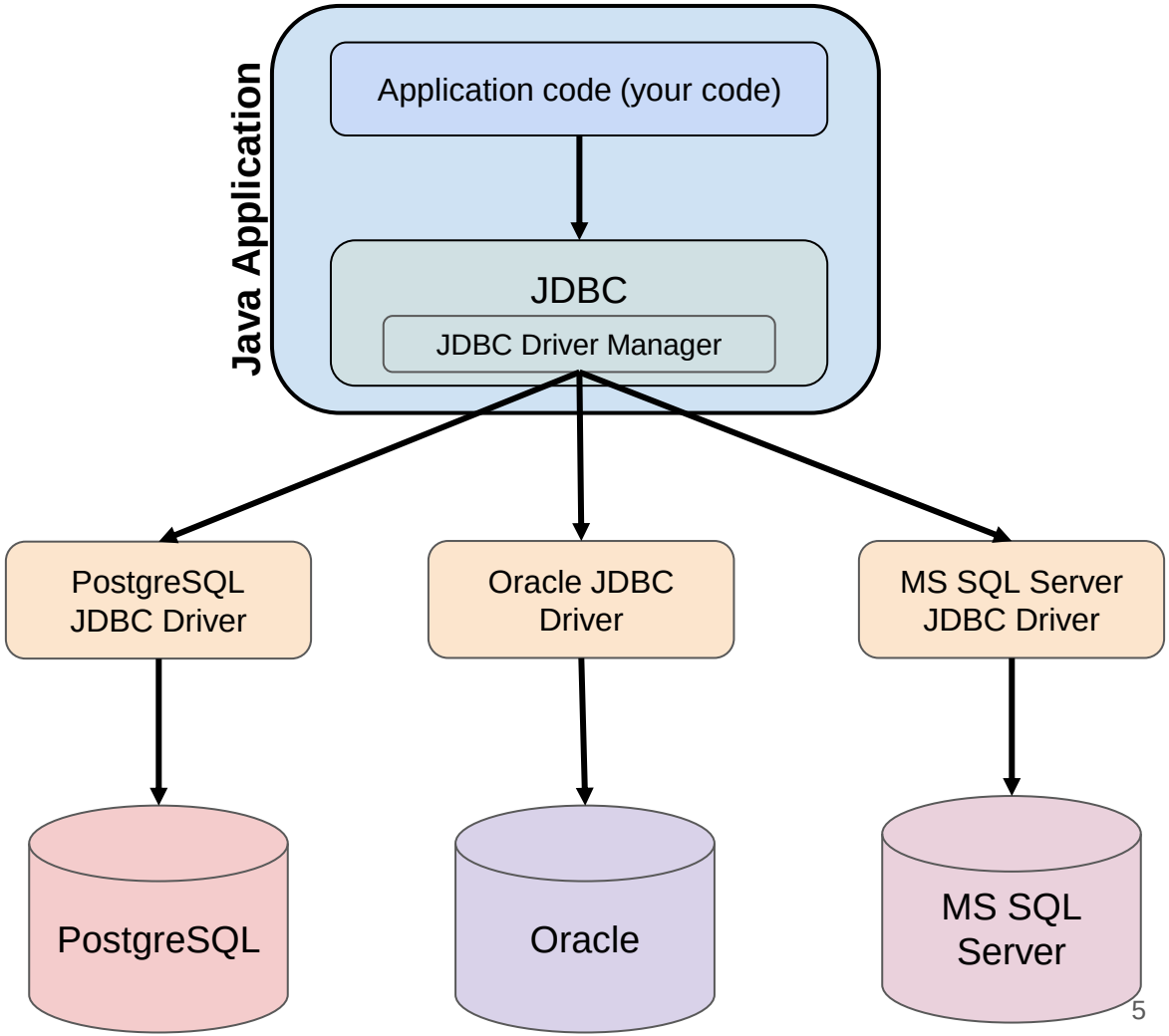
- Our main task in this lecture is to understand the collaborator classes and methods that will be needed to talk to a Postgresql database.

JDBC
Java Database Connectivity

JDBC (Java Database Connectivity) is a library that allows Java applications to access SQL databases in a generic way.

The vendor of the database provides a Driver that implements an interface that JDBC is expecting and knows how to communicate with the specific database.

Java Applications use the JDBC library and JDBC calls the vendor supplied driver to access the specified database.



Database Connections in Java

To create a connection in a Java Application a DataSource is needed.

The DataSource must be given a connection string to tell it what database to connect to and where it is located.

Connection String

- Similar to a web site address (URL).
- Includes details on how to connect to the database, including the **vendor driver to use**, **where the database is located**, and the **name of the database**.

```
jdbc:postgresql://localhost:5432/MovieDB
```

DataSource

An object that represents a database.

Allows a Java application to connect to the database and use SQL commands.

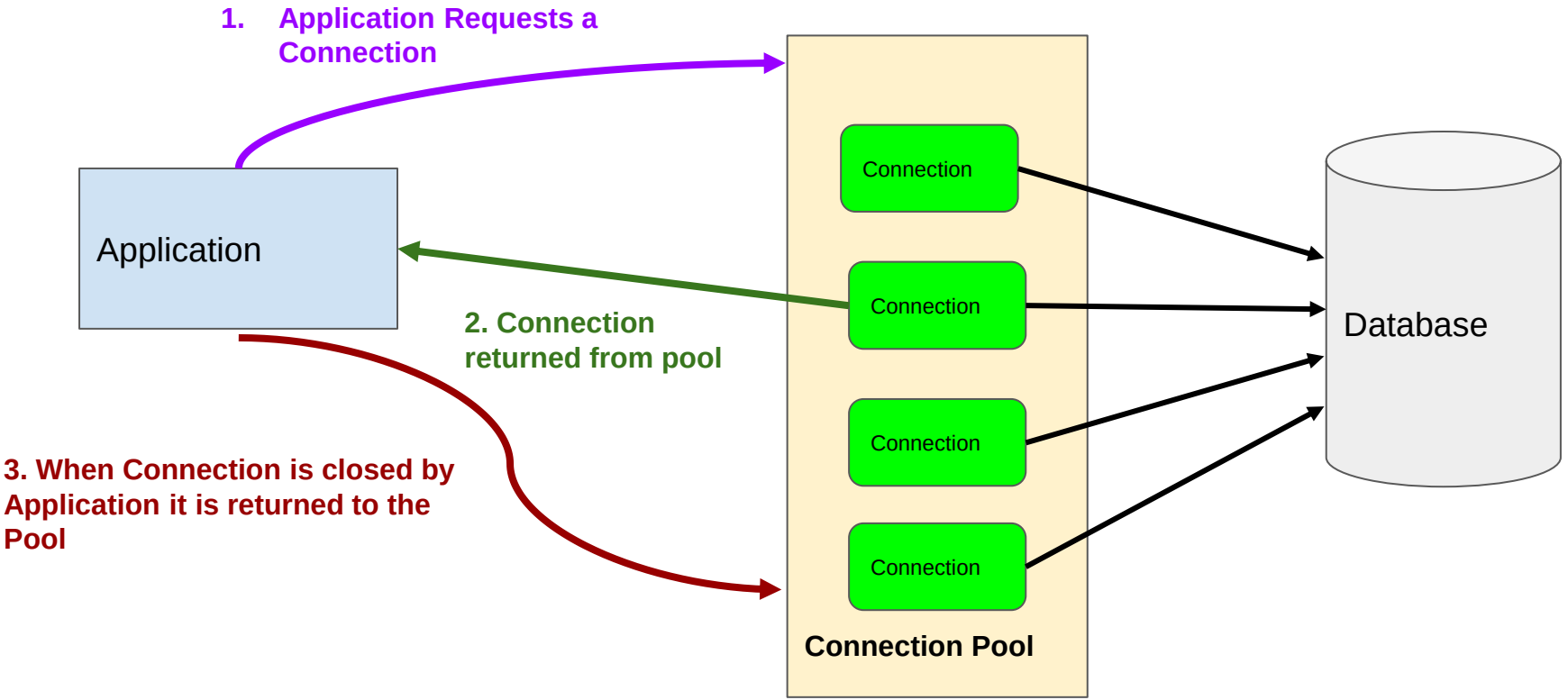
```
BasicDataSource movieDBDataSource = new BasicDataSource();
dvdstoreDataSource.setUrl("jdbc:postgresql://localhost:5432/movieDB");
dvdstoreDataSource.setUsername("postgres");
dvdstoreDataSource.setPassword("postgres1");
```

`BasicDataSource` - The `org.apache.commons.dbcp2.BasicDataSource` provides the ability to make a database connection and creates a *Connection Pool*.

The **`setUrl`** takes a connection string as an argument

`setUsername()` and **`setPassword()`** methods set the username and password to use when connecting to the database

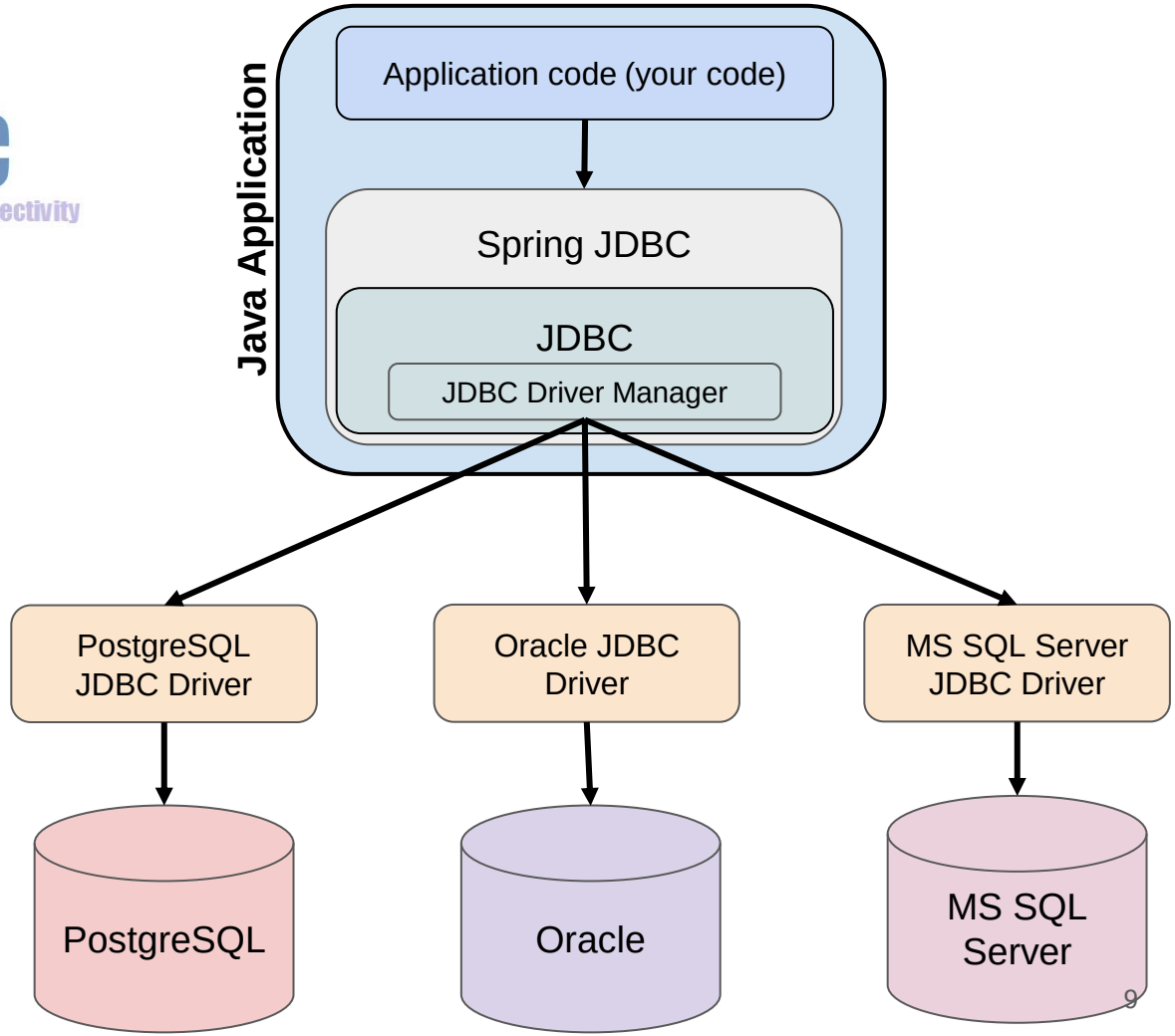
A **Connection Pool** works like a library of connections. The Connection Pool makes multiple connections to the database and keeps them open. When an application needs a connection it *"checks one out"* from the pool. When it is finished with it it *returns it* to be reused. This reduces the overhead of a new connection each database access is needed. Multiple Applications can use the same Connection Pool, allowing for fewer overall connections.



SpringJDBC is a framework that provides an *abstraction for JDBC* making it easier to use.

Provides a consistent way to create queries, handle results, deal with exception, and automatically provides transactions.

Java Applications use the **SpringJDBC** Framework, which will utilize *JDBC* to call the *Vendor Supplied JDBC Driver* to access the *Database*.



JDBCTemplate

org.springframework.jdbc.core.JdbcTemplate

- The central class of SpringJDBC
- Simplifies using JDBC and helps to avoid common errors.
- Allows execution of SQL Queries
- Provides a uniform way to retrieve results.

JDBCTemplate requires a **DataSource** as a constructor argument when instantiated.

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(datasource);
```

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/MovieDB");
dataSource.setUsername("postgres");
dataSource.setPassword("postgres1");

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
```

Let’s look at the code

JDBCTemplate Class and SqlRowSet

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

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

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

JDBCTemplate Class

QueryForRowSet – performs query to the database

```
String sqlMoviesByReleaseYear = "SELECT * FROM movie WHERE release_date >= '01/01/2006' LIMIT 10";


SqlRowSet results = movieDBJdbcTemplate.queryForRowSet(sqlMoviesByReleaseYear);

System.out.println("Movies since 2006: ");
while(results.next()) {
        String movieTitle = results.getString("title");
        int releaseYr = results.getInt("release_year");
        System.out.println(movieTitle +" ("+ 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 of 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.

Let’s look at the code

SQL Parameters

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

```
String sqlMovieByReleaseYear = " SELECT * FROM film WHERE release_year >= " +
     movieReleaseYear + " LIMIT 10";
```

```
String sqlMovieByReleaseYear = " SELECT * FROM film WHERE release_year >= ? LIMIT 10";

int movieReleaseYear = 2006;
SqlRowSet results = movieDBJdbcTemplate.queryForRowSet(sqlMoviesByReleaseYear, movieReleaseYear);

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

}
```

DAO Pattern

- A database table can sometimes map fully or partially to an existing class in Java. This is known as **Object-Relational Mapping (ORM)**.
- We implement the Object Relation Mapping with a design pattern called DAO, which is short for **Data Access Object**. This design pattern abstracts and encapsulates the details of persistent storage (like a database) to isolate it from our application code.
- 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.
- The purpose is to hide the complexities of the CRUD (Create, Read, Update, Delete) of the storage mechanism from the rest of the code. This allows both to evolve without knowing anything about the other.

Parts of the DAO Pattern

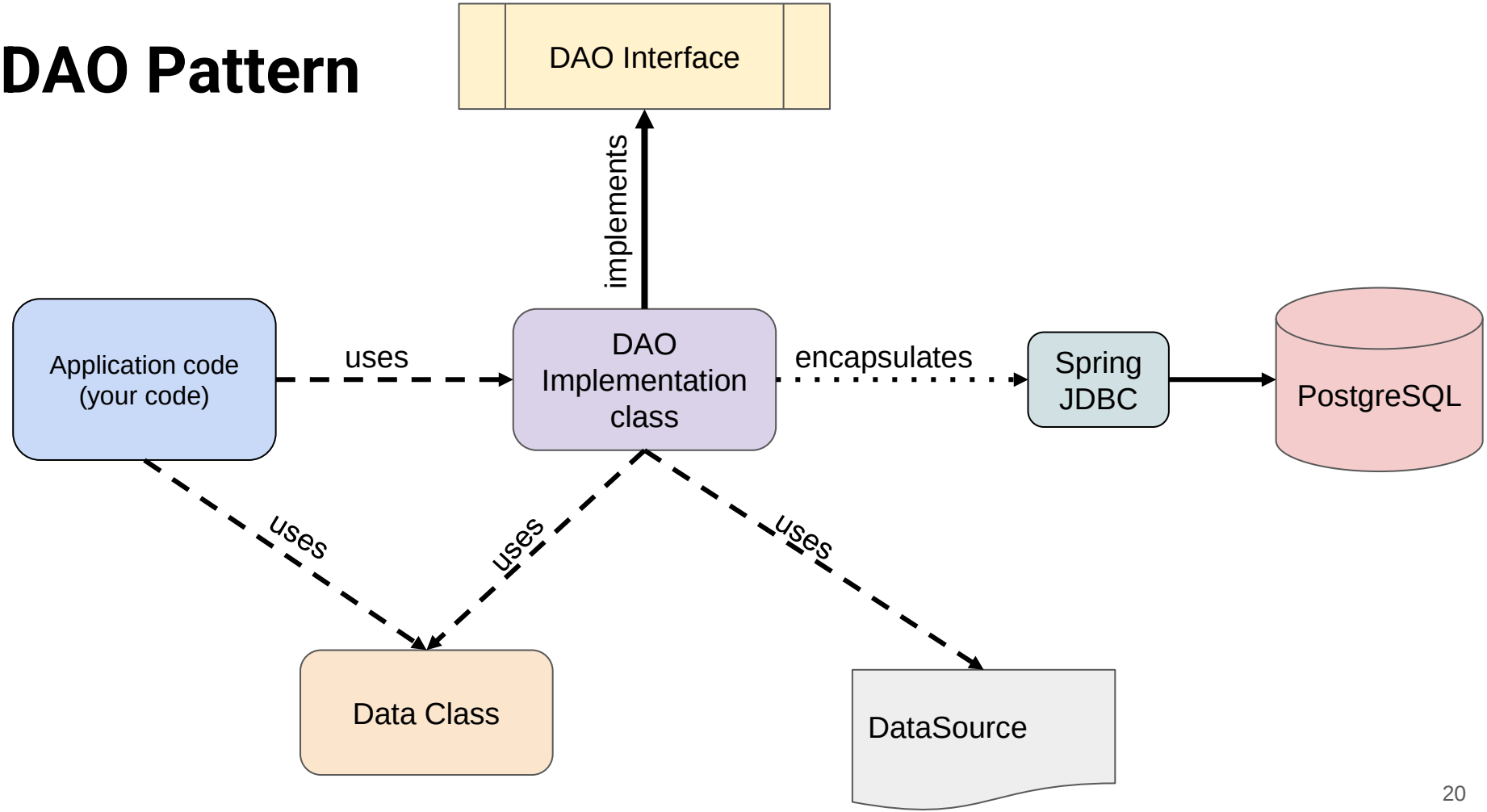
The DAO Pattern breaks the responsibility of data access into 3 parts.

1. **Data Object** - a simple data object (*POJO*) often called a *domain or business object*. This data object will act as a data type that represents a single entity of data.

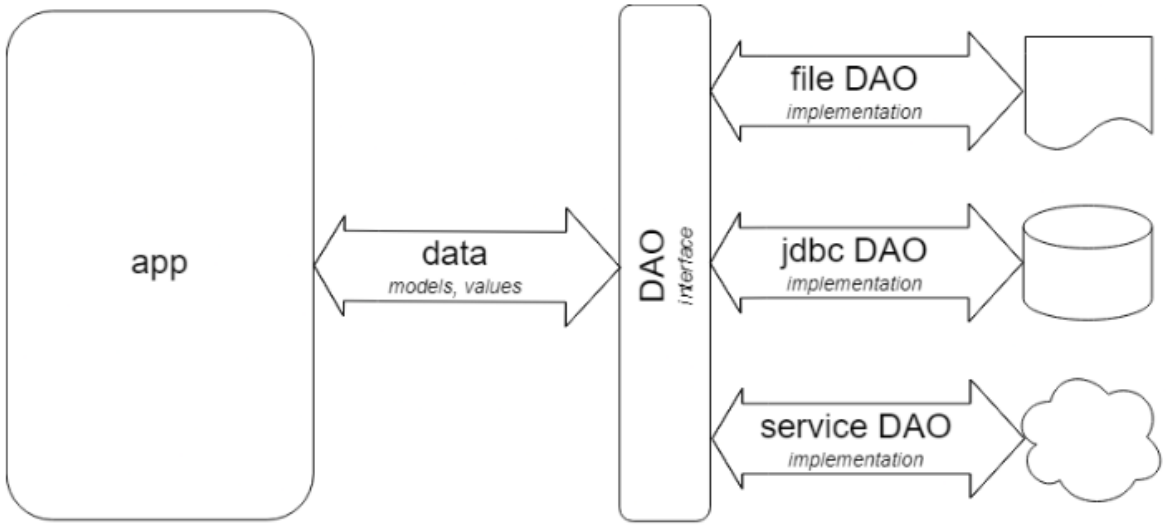
2. **DAO Interface** - An interface that defines the methods for the CRUD operations that will be available. These methods will generally use either the DAO Domain Object or simple types (String, long, int, etc) as return types and arguments.

3. **Implementation Class (JDBCDao)** will implement the DAO Interface and provide functionality for the persistence mechanism, like a database.

DAO Pattern


DAO Interface
implements
Application code
(your code)
uses
DAO
Implementation
class
encapsulates
Spring
JDBC
PostgreSQL
uses
uses
uses
Data Class
DataSource

DAO Pattern


app
data
models, values
DAO
interface
file DAO
implementation
jdbc DAO
implementation
service DAO
implementation

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 createCity(City city);  // c - create
     public City getCity(long cityId);  // 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 createCity(City city) {
          String sqlInsertCity = "INSERT INTO city(id, name, countrycode, district, population) " +
                                 "VALUES(?, ?, ?, ?, ?)";
          newCity.setId(getNextCityId());
          jdbcTemplate.update(sqlInsertCity, city.getCityName(), city.getStateAbbreviation(),
                                 city.getPopulation(), city.getArea());
     }

     @Override
     public City  getCity(long id) {
         City theCity = null;
         String sqlFindCityById = "SELECT city_id, city_name, state_abbreviation, population, area "+
                                 "FROM city "+
                                 "WHERE city_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(dataSource);
```

DAO Pattern Step 3

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

```
City theCity = dao.getCity(id);
```

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

Let’s code!

Objectives

- Making Connections

```
BasicDataSource dataSource = new BasicDataSource();
dataSource.setUrl("jdbc:postgresql://localhost:5432/MovieDB");
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

```
String sqlMovieByReleaseYear = " SELECT * FROM film WHERE release_year >= ? LIMIT 10";

int movieReleaseYear = 2006;
SqlRowSet results = movieDBJdbcTemplate.queryForRowSet(sqlMoviesByReleaseYear,
movieReleaseYear);

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

}
```

Objectives

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

```
public interface CityDAO {  // CRUD – create, read, update, delete
     public void createCity(City city);  // c - create
     public City getCity(long cityId);  // r - read
}
```

```
public class City {
   private long cityId;
   private String cityName;
   private long population;
   private double area;

…
}
```

```
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.createCity(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 createCity(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;
}
```