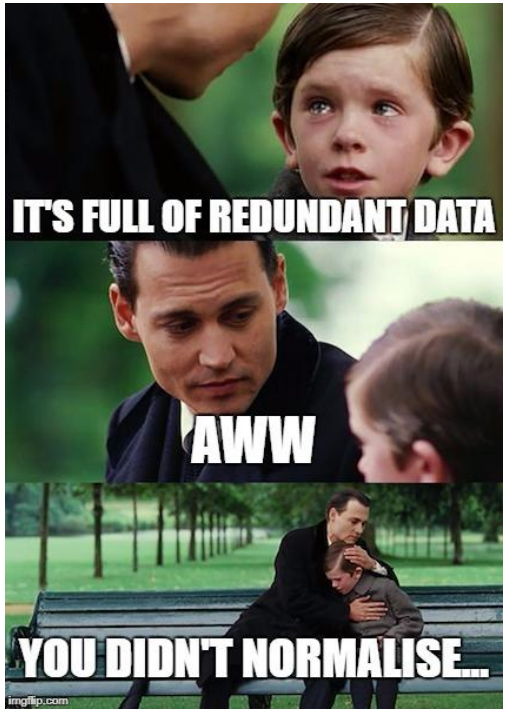

IT'S FULL OF REDUNDANT DATA
AWW
YOU DIDN'T NORMALISE...
imgflip.com

Database Connectivity - DAO

Module 2: 07

Today’s Objectives

1. DAO (Data Access Object) Pattern
2. JDBC
3. SpringJDBC
    a. Making Connections
        i. Connection Pools
    b. Executing SQL Statements that return results
        i. Parameterized Queries
        ii. Getting Results
    c. Executing SQL Statements that return no results
    d. Executing SQL Statements that return a single result

DAO Pattern

Data Access Object (DAO) is a structural design pattern that abstracts and encapsulates the details of persistent storage (like a database) to isolate it from our application code.

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

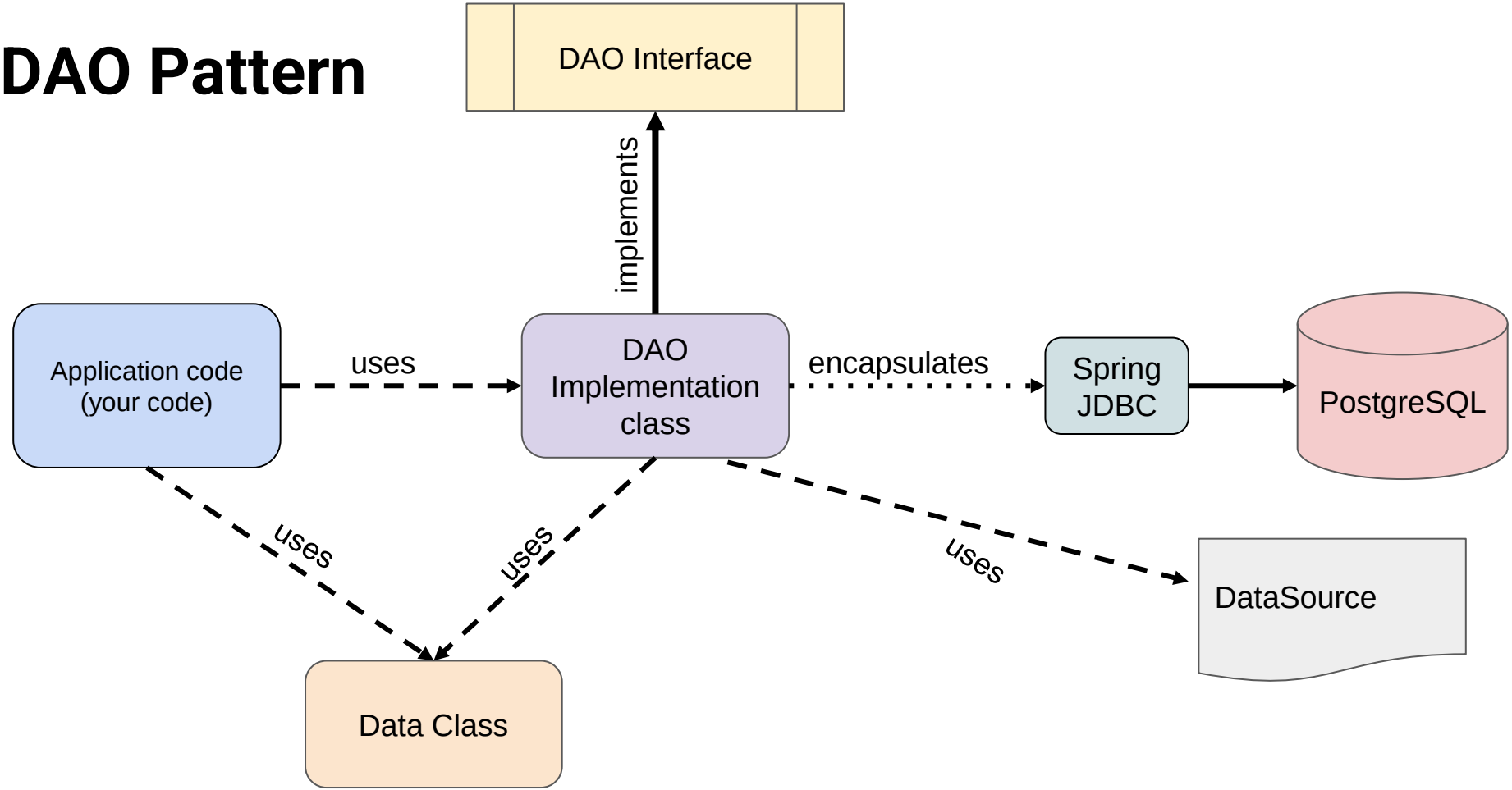
The DAO Pattern breaks the responsibility of data access into 3 parts.

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.

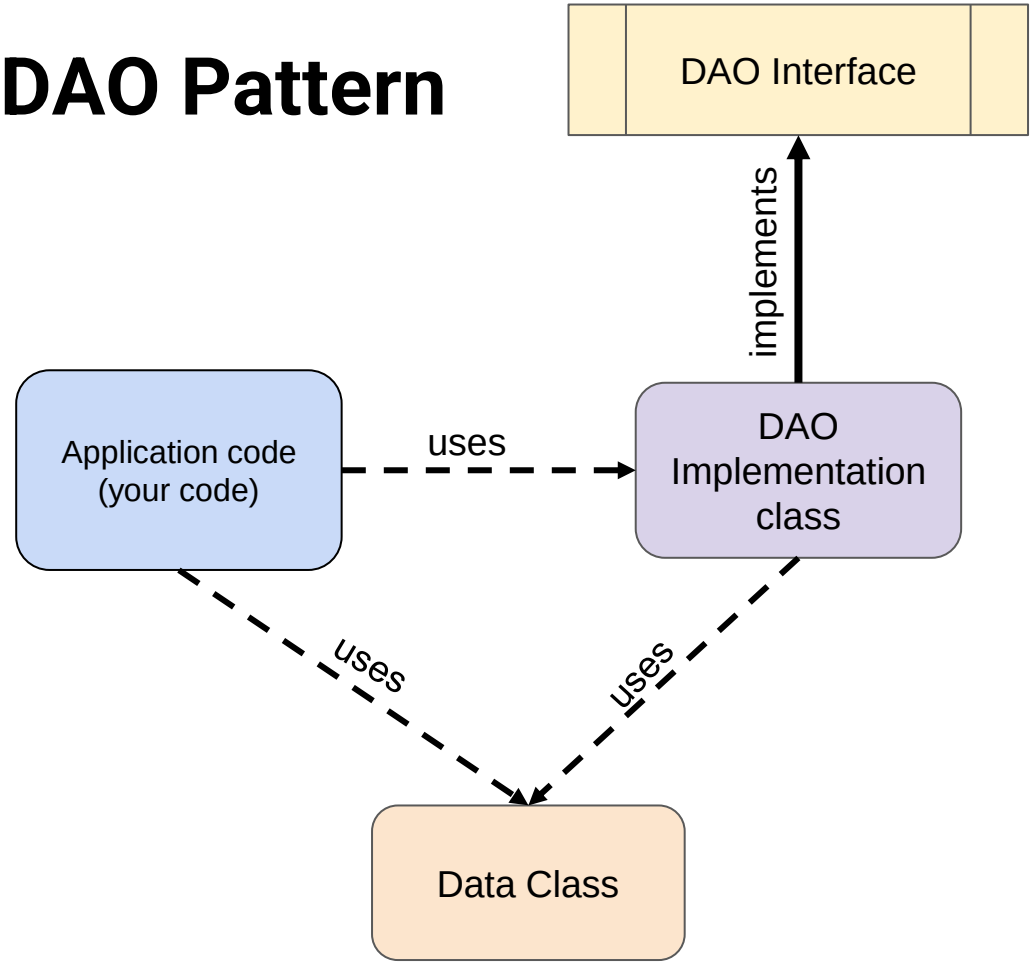
2. **DAO Interface** - An interface that defines the methods for the CRUD operations that will be available. These methods will generally use either the DAO Domain Object or simple types (String, long, int, etc) as return types and arguments.

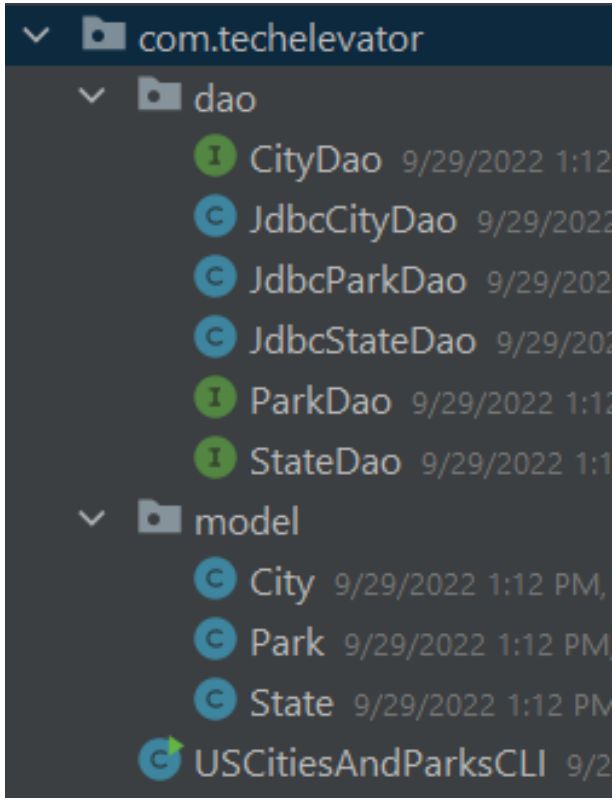
3. **Implementation Class (JDBCDao)** will implement the DAO Interface and provide functionality for the persistence mechanism, like a database.

DAO Pattern


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

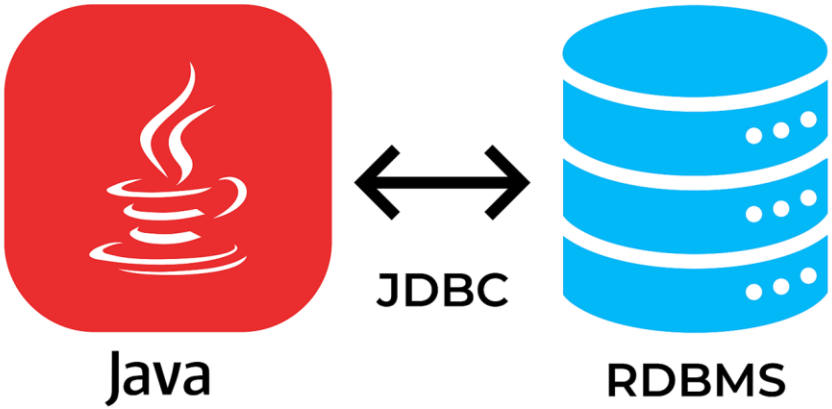
DAO Pattern


DAO Interface
implements
Application code
(your code)
uses
DAO
Implementation
class
uses
uses
Data Class


com.techelevator
dao
CityDao 9/29/2022 1:12
JdbcCityDao 9/29/2022
JdbcParkDao 9/29/202
JdbcStateDao 9/29/20
ParkDao 9/29/2022 1:1
StateDao 9/29/2022 1:1
model
City 9/29/2022 1:12 PM,
Park 9/29/2022 1:12 PM
State 9/29/2022 1:12 PM
USCitiesAndParksCLI 9/2

JDBC

Java Database Connectivity



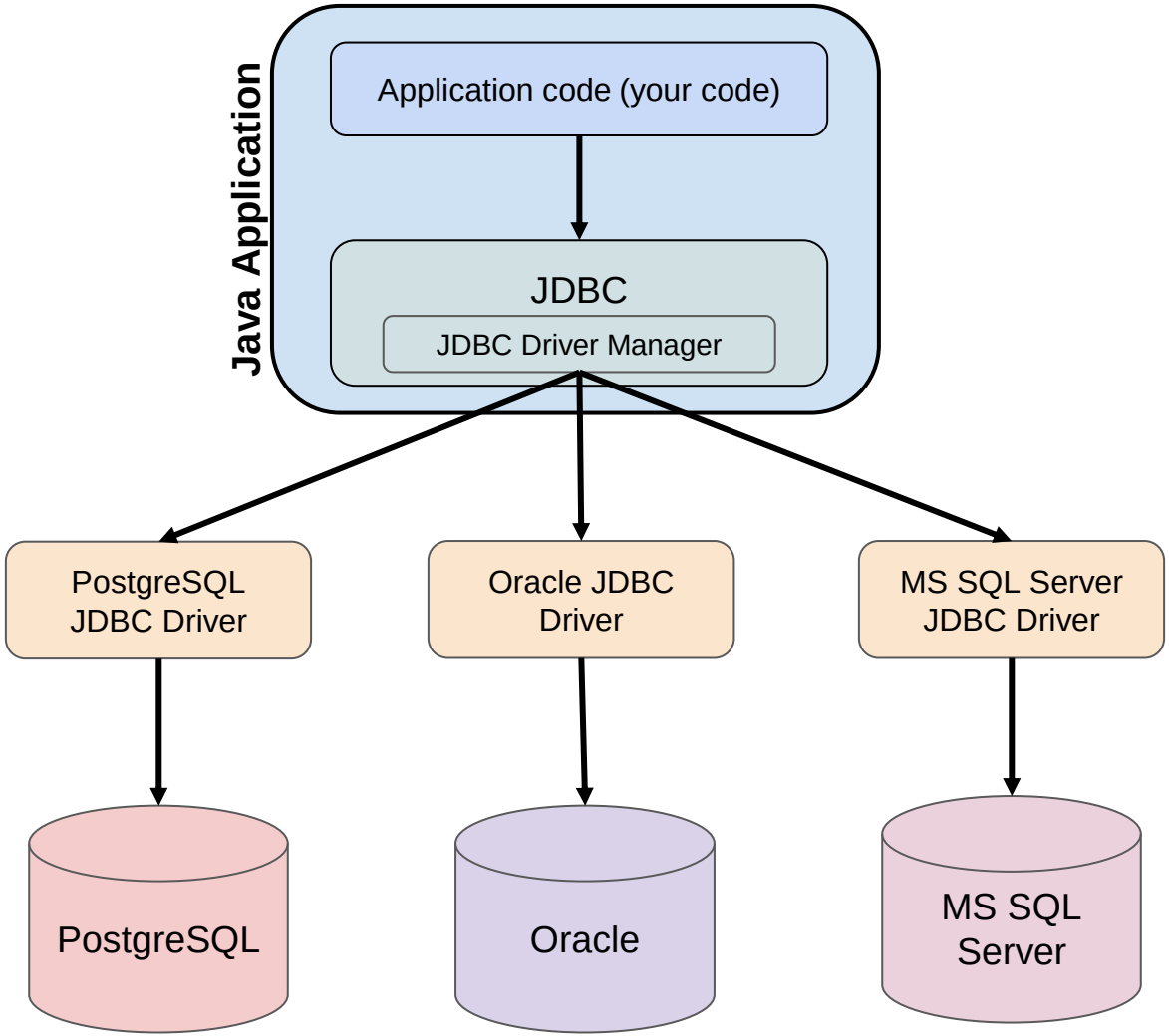
JDBC

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.

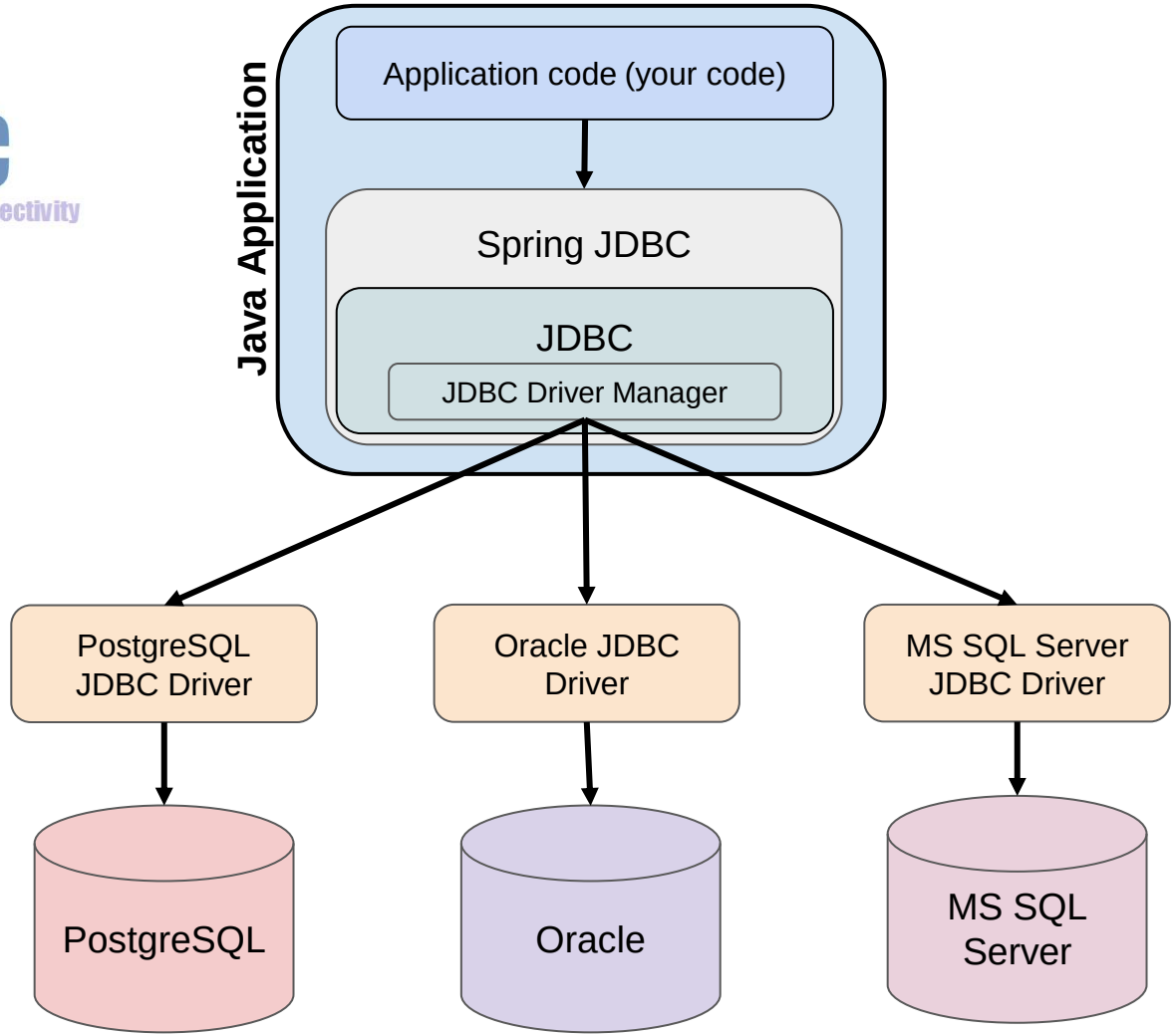
Java Applications use the JDBC library and JDBC calls the vendor supplied driver to access the specified database.



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

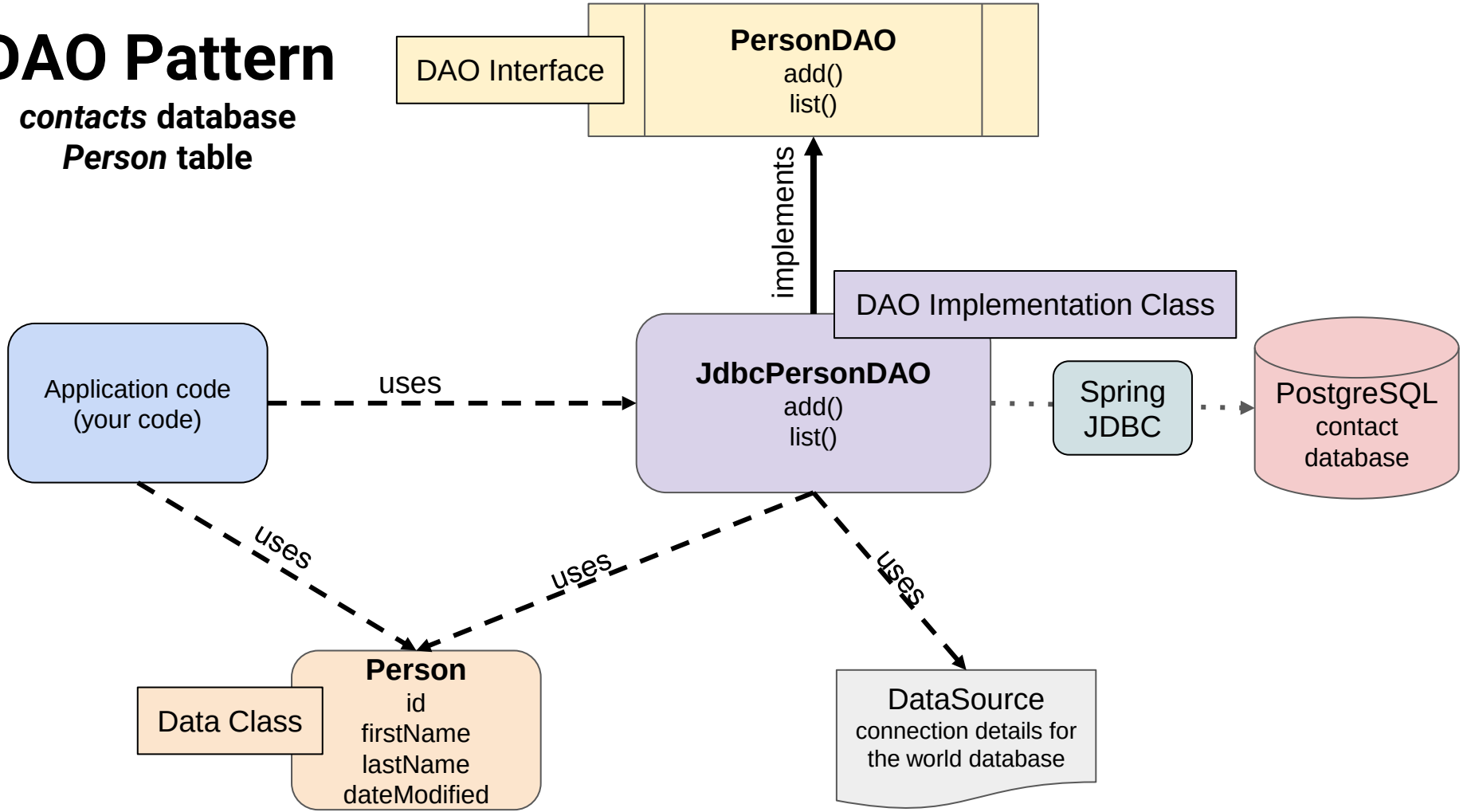
Provides a consistent way to create queries, handle results, deal with exception, and automatically provides transactions.

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



DAO Pattern

***contacts* database**
***Person* table**



Making a Connection

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/dvdstore
```

DataSource

An object that represents a database.

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

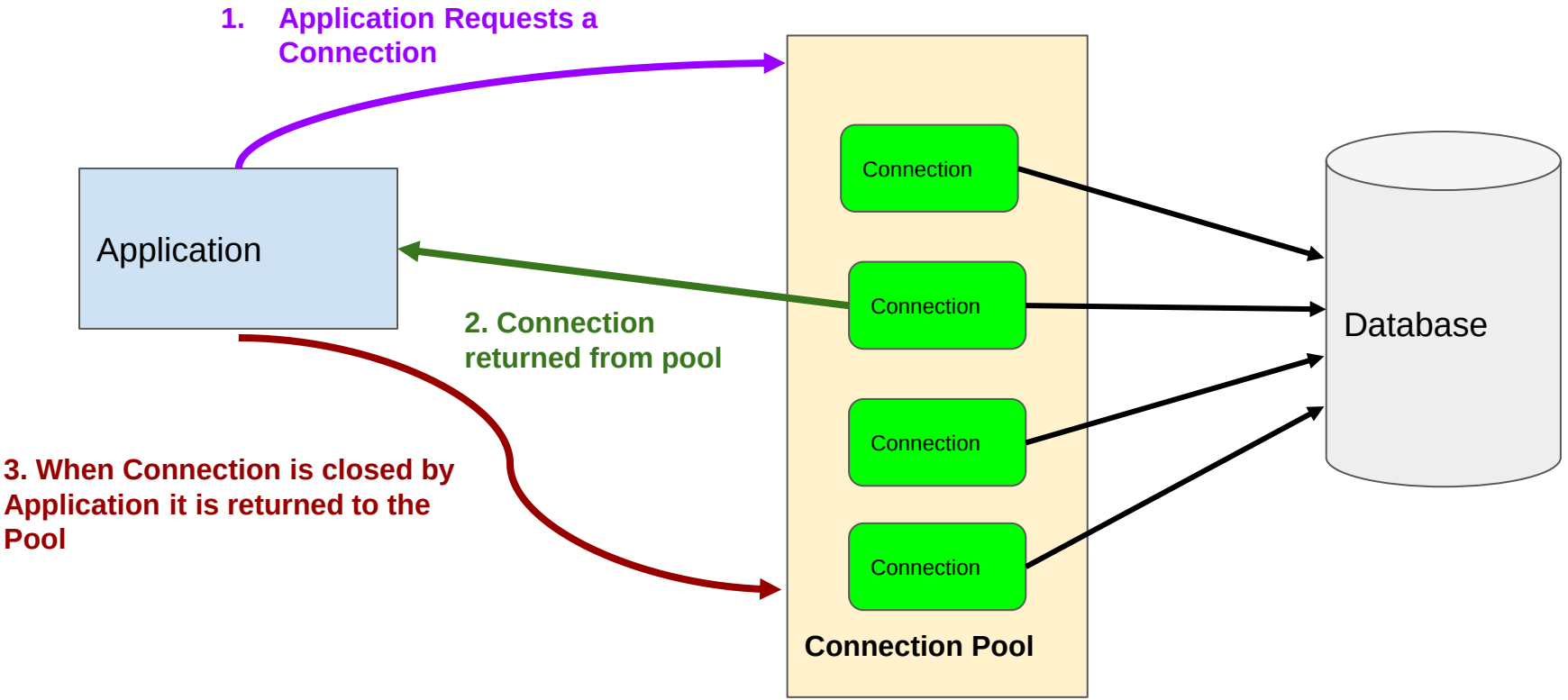
```
BasicDataSource dvdstoreDataSource = new BasicDataSource();
dvdstoreDataSource.setUrl("jdbc:postgresql://localhost:5432/dvdstore");
dvdstoreDataSource.setUsername("postgres");
dvdstoreDataSource.setPassword("postgres1");
```

`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.



Executing Queries

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);
```

Executing SQL Statements that return results

The JdbcTemplate queryForRowSet() methods executes a SQL query and returns the results as a SqlRowSet

```
SqlRowSet results = jdbcTemplate.queryForRowSet(sql, params1...paramsN)
```

The Danger of Concatenation

Insecure code!!

```
String sql = "SELECT * FROM actor WHERE first_name = " + firstName;
```

Even though SQL Statements are created as Strings in Java, ***values used in a query should not be concatenated***, unless the source of the value is a known as safe. A literal value is safe, but arguments passed to a method are always unsafe! Unsafe concatenation can open an application to **SQL Injection**, which is a critical security vulnerability that has been responsible for some of the largest data breaches in history.

Secure Code with Parameters

```
String sql = "SELECT * FROM actor WHERE first_name = ? AND last_name = ?";


jdbcTemplate.queryForRowSet (sql, firstName, LastName );
```

To allow for variable data, the methods that jdbcTemplate provides allow parameters.

Parameters are represented in the SQL String with a ?, and passed as a list to the jdbcTemplate methods in the same order as the ? appear in the query. Passing values as parameters secures against Sql Injection.

Getting Results with SqlRowSet

- Each row is returned as an entry in a Set, which can be looped through using next().
- Each time next() is called it moves to the next row in the set. next() returns true/false if more data exists. **next() must always be called once to access the first row**.
- Data can be accessed for a row by using the provided getters with column name returned in the query.

```
SqlRowSet results = jdbcTemplate.queryForRowSet(sql);

while (results.next()) {

                String filmTitle = results.getString("title");

                int releaseYear = results.getInt("release_year");

        }
```

Detailed Description of using Next() in Cheatsheets

Executing SQL Statements that return no results

Queries that do not return results, like UPDATE, DELETE, and INSERT, *when a sequence is not being used to generate a primary key*, use the update() method.

update() does not return a SqlRowSet since there are no results.

```
jdbcTemplate.update(sql, params1...paramsN)
```

Executing SQL Statements that return a single result

Some Queries return a single piece of data.

A common usage is using RETURNING with an insert statement to have the generated id (sequence) returned. queryForObject() can be used to retrieve a single value.

```
String sql = "INSERT INTO city(id, name, countrycode, district,
population) VALUES(DEFAULT, ?, ?, ?, ?) RETURNING id"

Integer cityId = jdbcTemplate.queryForObject(sql, Integer.class,
                                "Smallville", "USA", "Kansas", 45001);
```

Note: DEFAULT is a command that tells SQL to use the default value. We could also do this without including the ID in the query.

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 (ORM)**.
- 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 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

- Example, we use a polymorphism pattern to declare our DAO objects:

```
CityDao dao = new JdbcCityDao(dataSource);
```

The Interface Reference

The Concrete Class Constructor

DAO Pattern Step 3

- Example, we use a polymorphism pattern to declare our DAO objects:

```
City smallville = new City();
smallville.setCityName("Smallville");
smallvill.e.setStateAbbreviation(“KS”);
smallville.setPopulation(42080);
smallville.setArea(4.5);

Long id = dao.createCity(smallville);

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

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


Finally, we found the DAO ...