TEKsystems | PER SCHOLAS

A Partnership for Progress

Applying DAO and Object-Oriented Programming to Java Database Connectivity

Data Access Object Pattern and JDBC Application

Learning Objectives:

In this presentation, you will explore the Data Access Object (DAO) pattern, and apply the Object-Oriented Programming (OOP) concept and DAO pattern in JDBC application.

By the end of this lesson, learners will be able to:

- Describe the Design pattern and DAO Design pattern.
- Demonstrate the DAO pattern.
- Demonstrate how to apply the OOP concept and DAO pattern in JDBC application.

Agenda/Topics

- Overview of Design Pattern.
- Usage of Design Pattern.
- Introduction of Data Access Object (DAO).
- Components in Data Access Object (DAO) Pattern.
- Applying OOP and DAO to JDBC application

Overview of Design Pattern

- Authors, ***Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides*** published a book titled *Design Patterns - Elements of Reusable Object-Oriented Software (1994),* which initiated the concept of **Design Patterns in Software development**. The authors are collectively known as Gang of Four (GOF), and according to them, design patterns are primarily based on the following principles of object oriented design (OOD):
  - Program to an interface, not an implementation.
  - Favor object composition over inheritance.
- The **software/application design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations.
- Design patterns are best practices that programmers can employ to overcome common design difficulties, and are solutions to general problems that software developers faced during software and application development. These solutions were obtained by trial and error by numerous software developers over a substantial period of time.

Usage of Design Pattern

Design Patterns have two main uses in software development:

1. **Common platform for developers.**

Design patterns provide standard terminology, and are specific to a particular scenario. For example, a single design pattern signifies use of a single object so that all developers familiar with a single design pattern will make use of a single object, and can verify that the program is following a single pattern.

2. **Best Practices for developers.**

Design patterns have evolved over a long period of time, and they provide best solutions to certain problems faced during software development. Learning these patterns helps inexperienced developers learn software design in a fast and easy way.

Data Access Object Patterns

Introduction–

Data Access Object (DAO) is a pattern that provides an abstract interface to some type of database or other persistence mechanism.

- DAO Design Pattern is used to separate the data persistence logic into a separate layer. This way, the service remains completely in the dark about how the low-level operations access the database. This is known as the principle of *Separation of Logic*.
- DAO is used to separate ***low-level data accessing API (like JDBC)*** or ***operations from high-level business services*** or the idea is to abstract or hide the database logic from the business layer.

Click here for more information about DAO.

Data Access Object Patterns (continued)

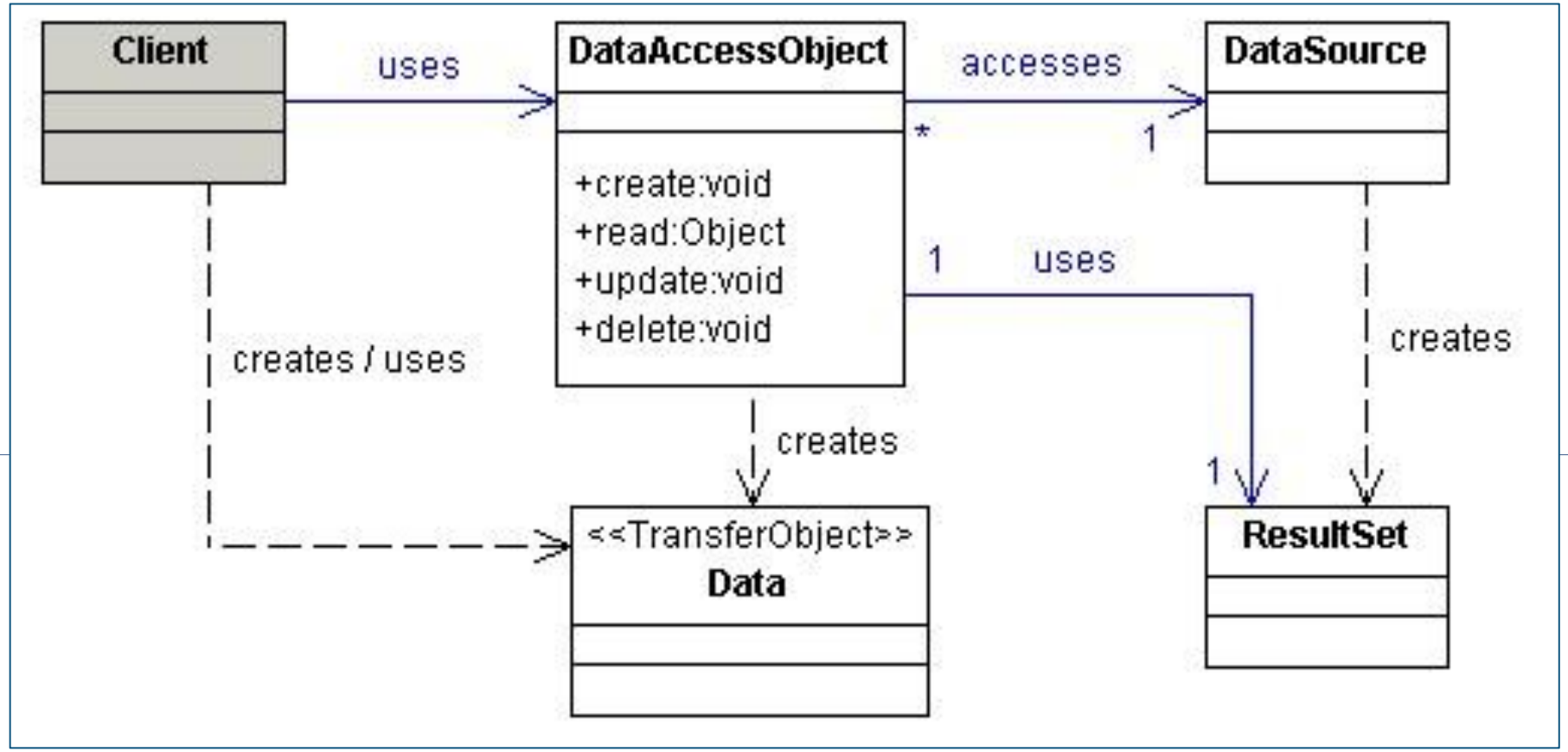
- The advantage of using data access objects is the relatively simple and rigorous separation between two important parts of an application that do, but should not know of each other, and which can be expected to evolve frequently and independently.
- If we need to change the underlying persistence mechanism, we only have to change the DAO layer, and not all the places in the domain logic where the DAO layer is used.

Data Access Object Pattern Components

- **DAO - Data Access Object Interfaces** - This interface defines the standard operations to be performed on model object(s). The interfaces provide a flexible design.
- **DAO - Data Access Object classes(Business/Controller layer)** - This class implements above the interface. This class is responsible for getting data from a data source, which can be database / xml or any other storage mechanism. The interface implementation is a concrete implementation of the persistence logic.
- **DAL - Data-Access Layer** - This is a layer in an application that provides easy and simplified access to data stored in persistent storage, such as an entity-relational database or any database. This layer exists between the Business/Controller Logic Layer (BLL) and the storage layer.
- **Models Class / DTO** - This object is simple POJO class containing getter and setter methods to store data retrieved using DAO class.
- **DataSource** - A data source could be a database such as an RDBMS, File,, XML repository, flat file system, or any other data source. A data source can also be another system service or some kind of repository.

Data Access Object Pattern Components (continued)

High-level class diagram representing the relationships for the DAO Pattern.



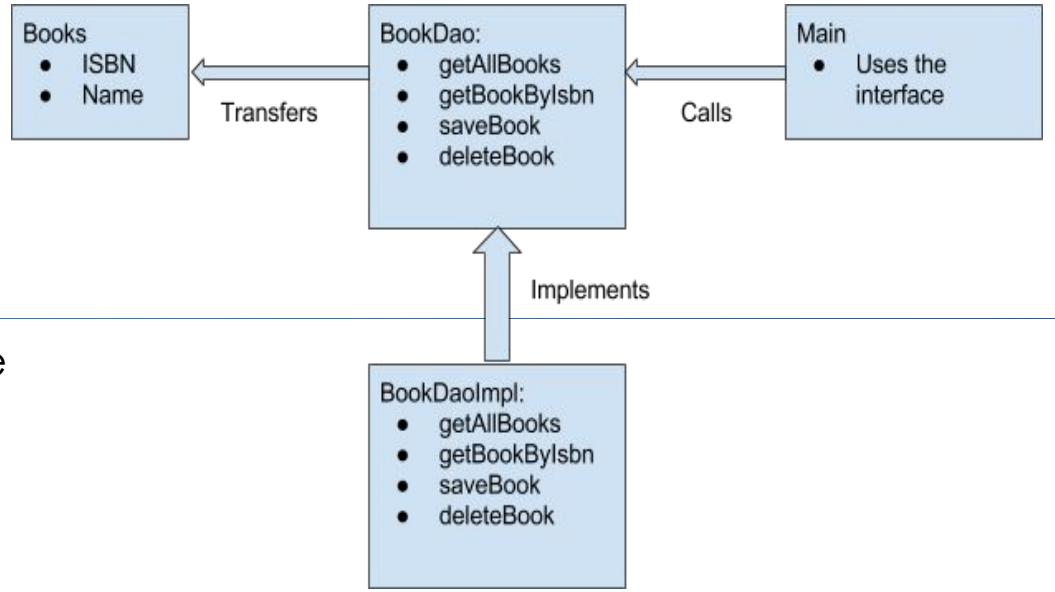
Example 1: Implementing the DAO Pattern

Let's use ***Books*** as an example. With DAO design pattern, we have the following components on which our design depends:

- The **Model** class or **the transfer object,** which is transferred from one layer to the other.
- The **interfaces,** which provides a flexible design.
- The **interface** implementation, which is a concrete implementation of the persistence logic.

We will use four components here:

1. The Book model class, which is transferred from one layer to the other.
2. The BookDao interface that provides a flexible design and API to implement.
3. The BookDaoImpl class, which is an implementation of the BookDao interface.
4. The **main()** method, which is the entrypoint of the application.



Example 1: Implementing the DAO Pattern (continued)

Step 1: Create a new project and add a JDBC driver in the project. Then create a package named **model** under ***src*** folder.

Step 2: Under **model** package, create a class named **Books**. Write the code on the right in class.

This will be our model class. It has a simple object with just two properties to keep things simple.

Books.java class

```
package model;
public class Books {
   private int isbn;
   private String bookName;
   public Books() {   }
   public Books(int isbn, String bookName) {
       this.isbn = isbn;
       this.bookName = bookName;
   }
   // getter setter methods
   public int getIsbn() {
       return isbn;
   }

   public void setIsbn(int isbn) {
       this.isbn = isbn;
   }

   public String getBookName() {
       return bookName;
   }

   public void setBookName(String bookName) {
       this.bookName = bookName;
   }
}
```

continue…

Example 1: Implementing the DAO Pattern (continued)

Step 3: Create a package named **DAOinterface** under ***src*** folder.

Step 4: Under **DAOinterface** package, create an ***interface*** named **BookDao**. Write the code on the right in the class.

```
import java.util.List;
import model.Books;
public interface BookDao {
	List<Books> getAllBooks();
	Books getBookByIsbn(int isbn);
	void saveBook(Books book);
	void deleteBook(Books book);
}
```

BookDao.java interface

Example 1: Implementing the DAO Pattern (continued)

Step 5: Create a package named **Controller** under the ***src*** folder.

Step 6: Under **Controller** package, create a class named **`BookDaoImpl`**. Add write the code on the right in the class.

Finally here is how we use our DAO to manage Books.

`BookDaoImpl`class supports getallbooks, save(insert), update, and delete operations. We can add more operations in future if needed. The business layer remains unaware of the actual persistence logic.

In this class, **just for the sake of simplicity, we have used an *ArrayList* as a data store instead of interacting with some DB.** In a real-world situation, we will interact with some database.

`BookDaoImpl.java class`

```
import java.util.ArrayList;
import java.util.List;
import DAOinterface.BookDao;
import model.Books;
public class BookDaoImpl implements BookDao  {
   //list is working as a database
   private ArrayList<Books> booksObj;
   public BookDaoImpl() {
      booksObj = new ArrayList<Books>();
      booksObj.add(new Books(1, "Java Book"));
      booksObj.add(new Books(2, "Python Book"));
      booksObj.add(new Books(3, "Android Book"));
   }
   @Override
   public List<Books> getAllBooks() {
       return this.booksObj;
   }
   @Override
   public Books getBookByIsbn(int isbn) {
       return this.booksObj.get(isbn);
   }
   @Override
   public void saveBook(Books book) {
      this.booksObj.add(book);
   }
   @Override
   public void deleteBook(Books book) {
      this.booksObj.remove(book);
   }
}
```

Example 1: Implementing the DAO Pattern (continued)

Step 7: Create a package named **Runner** under the ***src*** folder.

Step 8: Under **Runner** package, create a class named **AccessBook**. This class contains **main()** method.

Write a code in class, available on the right side. Run this class.

Output

```
############## show list of all books ##########
Book ISBN : 1 and Book Name : Java Book
Book ISBN : 2 and Book Name : Python Book
Book ISBN : 3 and Book Name : Android Book
------After added/ inserted new book ----------
Book ISBN : 1 and Book Name : Java Book
Book ISBN : 2 and Book Name : Python Book
Book ISBN : 3 and Book Name : Android Book
Book ISBN : 4 and Book Name : SQL Book
--------- After update book name ----------
Book ISBN : 1 and Book Name : Java Book
Book ISBN : 2 and Book Name : Algorithms Book
Book ISBN : 3 and Book Name : Android Book
Book ISBN : 4 and Book Name : SQL Book
```

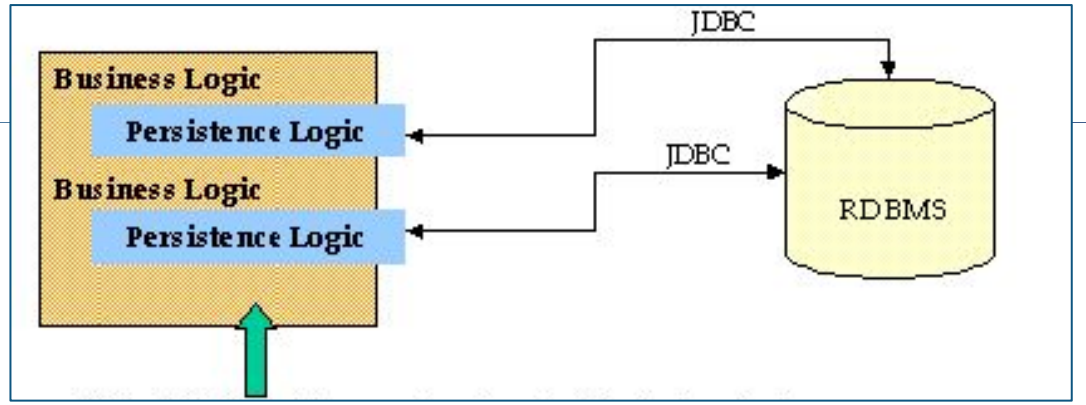
```
import Controller.BookDaoImpl;
import DAOinterface.BookDao;
import model.Books;
import javaa.awt.print.Book;
import java.util.ArrayList;

public class AccessBook {
  public static void main(String[] args) {
        System.out.println( "############## show list of all books ########## ");
        BookDao bookDao = new BookDaoImpl();
        for (Books b : bookDao.getAllBooks()) {
            System.out.println("Book ISBN : " + b.getIsbn() +" and Book Name : " +
b.getBookName() );
        }
        System.out.println("--------- After added/ inserted new book ----------");
        Books b1 = new Books();
        b1.setIsbn(4);
        b1.setBookName("SQL Book");
        bookDao.saveBook(b1);

        for (Books b : bookDao.getAllBooks()) {
System.out.println("Book ISBN : " + b.getIsbn() +" and Book Name : " + b.getBookName() );
        }
        System.out.println("--------- After update book name ----------");
        Books bookupdate = bookDao.getAllBooks().get(1);
        bookupdate.setBookName("Algorithms Book");

        for (Books b : bookDao.getAllBooks()) {
            System.out.println("Book ISBN : " + b.getIsbn() +" and Book Name : " +
b.getBookName() );
        }
    }
}
```

Applying OOP and DAO to JDBC Application

Now we will explore how to use DAO. Let's use the DAO pattern in a simple JDBC CRUD application.

Before DAO - Process of Java JDBC API

- Register the Driver - *Same every time / Repeat.*
- Connect to the Database - *Same every time / Repeat.*
- Prepare the Statement.
- Execute the Statement.
- Obtain data from the Database.
- Close the connection - *Same every time / Repeat.*

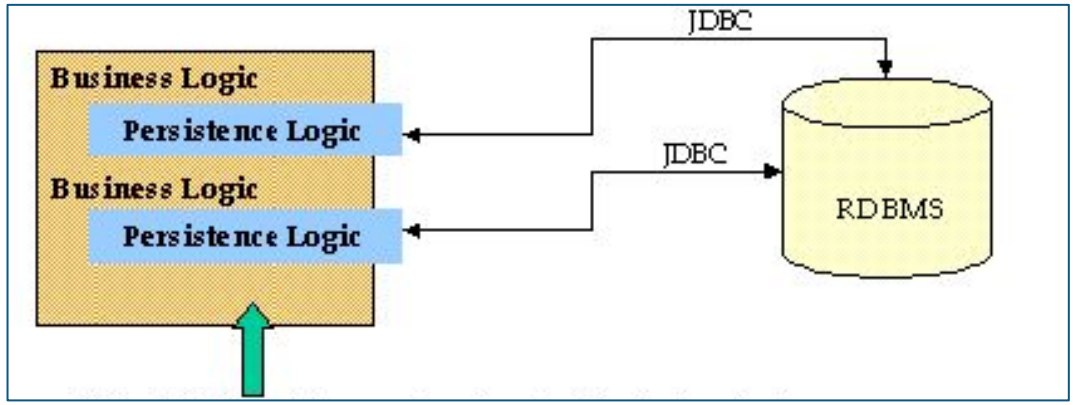


Before DAO, persistence code scattered within business logic.

After DAO - Process of Java JDBC API

The New Process for JDBC Application with DAO

- **Start with AbstractDAO**
  - Registers Driver.
  - Connects to Database.
  - Safely closes connection.
- **Create Interface for each DAO**
  - Contains queries.
  - Structure for methods(CRUD).
- **Implement DAO**
  - Write logic for interface methods.
  - Start with register/connect method.
  - End with close method.
- **Use business logic anywhere in the application.**



Before DAO, persistence code scattered within business logic.

After DAO, new layer to encapsulate interaction with persistent.

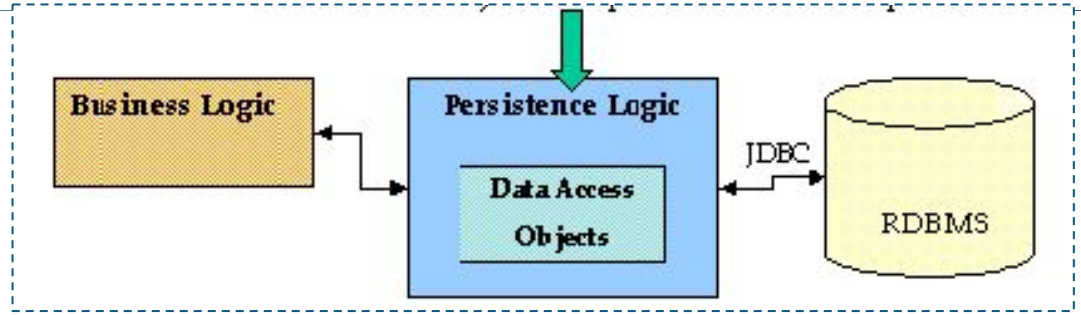
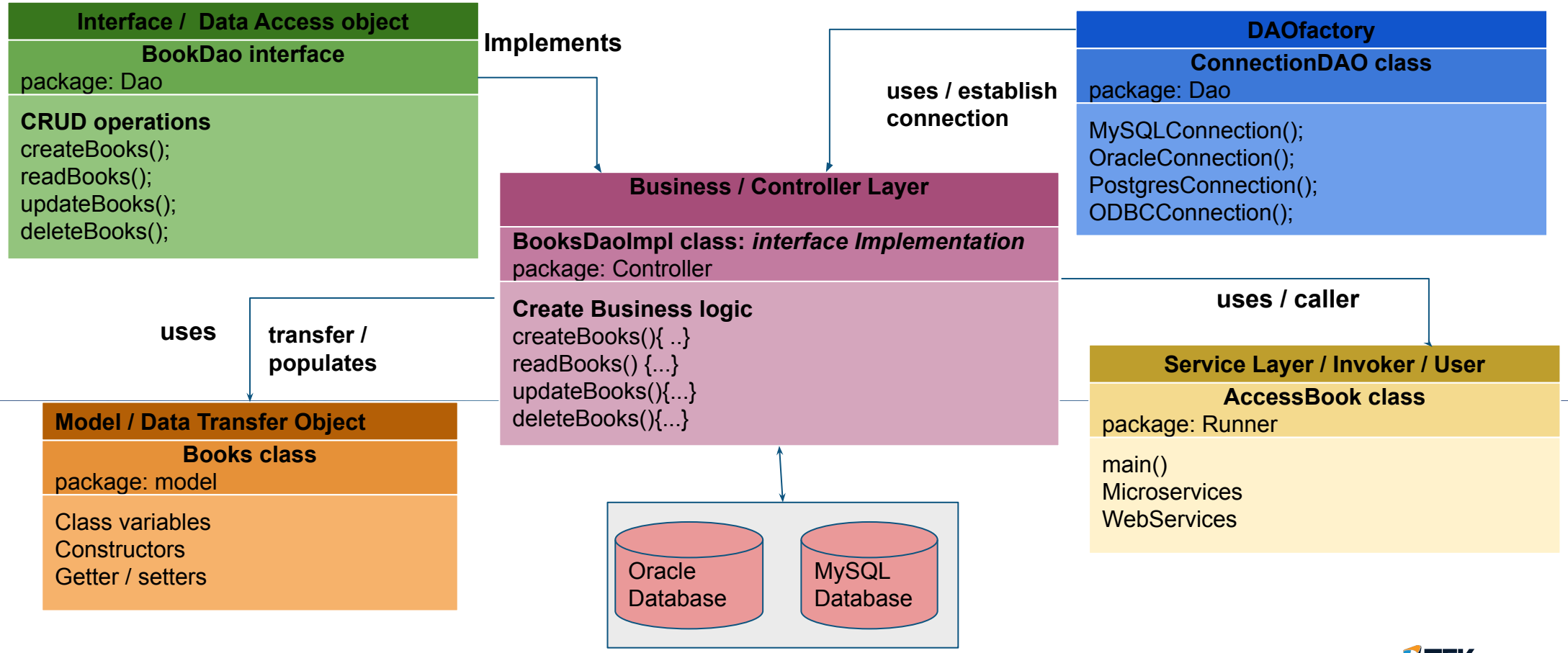


Image Source: oracle.com

Example: After DAO - Process of Java JDBC API

Illustration of the complete project

Let's take the same example of books. There are several ways to implement DAO and OOPs in our JDBC application. We are trying to illustrate it in a simple way. Below is a class level diagram for how to implement the DAO, OOP concept and JDBC in application.



Example: After DAO - Process of Java JDBC API

Click here for LAB - JDBC - 2: Building CRUD Application using DAO pattern, OOP concept, and JDBC.

Practice Assignment Activity

- Create Table in Database:
  - **Customer table**:
    - id(int), email(varchar), fname(varchar), lname(varchar)
  - **Item table:**
    - id(int), name(varchar), price(decimal(4,2)).
- Create models for each table.
- Create DAOs with the following methods:
  - **CustomerDAO**
    - getCustomerById(int id) - Returns the customer object for the given id.
    - addCustomer(Customer c) - Adds a customer with given information.
    - removeCustomerById(int id) - Removes a customer with the given id.
  - **ItemDAO**
    - getAllItems() - Returns a list of all item objects in the database.
    - addItem(Item i) - Adds an item with given information.
    - removeItemById(int id) - Removes an item with the given id.
- Create additional classes and attributes.

Summary

The Data Access Object (DAO) pattern is now a widely accepted mechanism to abstract away the details of persistence in an application. The idea is that instead of having the domain logic, communicate directly with the database, file system, web service, or whatever persistence mechanism your application uses. The domain logic speaks to a DAO layer instead.

- Every module (or group of routes example: /auth/<signup, login,etc.>) should have its own DAO layer.
- Every module (or group of routes example: /auth/<signup, login,etc.>) should have its own set of Models.
- DAO layers should use database queries as much as possible for various data manipulations.
- Models should have their own getters and setters functions, which should be used by the Controller / DAO for data updation and retrieval.

Glossary/Terminology

DAO: Data Access Object

DTO: Data Transfer Object

DAL: Data: Access Layer

BLL: Business/controller Logic Layer

POJO: Plain Old Java object

CRUD: Create, Read/Retrieve, Update and Delete

GOF: Gang of Four

References

- http://www.oracle.com/techn
- https://en.wikipedia.org/wiki/Data_access_objectetwork/java/dataaccessobject-138824.html
- https://en.wikipedia.org/wiki/Data_access_object
- https://www.linkedin.com/pulse/data-access-object-dao-design-pattern-java-mohammed/

Opening Questions ?