<http://www.jpgtutorials.com/data-access-object-design-pattern-in-j2ee>

DAO design pattern works with Data transfer object also known as value object. DTO is a java class with properties, getter and setter methods. Let us examine a practical scenario.

You are developing a simple Employee management application. Your data is residing in RDBMS, say MySQL. There is one table called **user** which stores employee information.

user  
——-  
userid                     int not null primary key column  
name                      varchar (100) not null  
designation         varchar (100) not null  
age                          int not null

The schema for user table is given above. Now let us create a DTO class.

|  |
| --- |
| **import** java.io.Serializable;    **public** **class** User **implements** Serializable {    **private** **static** **final** **long** serialVersionUID = 1L;    **private** **int** userId;  **private** String name;  **private** String designation;  **private** **int** age;    **public** **int** getUserId() {  **return** userId;  }  **public** **void** setUserId(**int** userId) {  **this**.userId = userId;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** String getDesignation() {  **return** designation;  }  **public** **void** setDesignation(String designation) {  **this**.designation = designation;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }    } |

If you look at the class definition you could be able to see that User class has some properties declared as private and corresponding getter/setter methods. In normal situation you may think of creating DTO as mapping to tables if your persistence storage is RDBMS. Create properties which map to corresponding columns in table. This is not always the case but in our example we would like to follow the same. The class is serializable because if we want to pass DTO between different JVM’s using RMI or any other protocol, it should implement Serializable interface.

Well, our DTO is ready. Now let us get into DAO. For implementing DOA, we have to implement the following steps.

* An interface which defines methods for various operations related to DTO.
* Concreate classes which implement DAO interface
* Factory/Abstract Factory class to get a reference to DAO object.

Let us first define interface. You can think of various operations related to user object  as follows,

* Get a list of all users
* Get a particular user information when we supply unique id.
* Add/Update/Delete user.

The DAO interface definition is given below.

|  |
| --- |
| **public** **interface** UserDAO {    **public** **void** insert(User user);  **public** **void** update(User user);  **public** **void** delete(**int** userId);  **public** User[] findAll();  **public** User findByKey(**int** userId);  } |

Well, now let us define a derived class. For accessing data, we may be using JDBC, object relational mapping tools like Hibernate etc. For each implementation we have to create separate class. The skeleton for one such implementation is given below.

|  |
| --- |
| **public** **class** UserDAOImpl **implements** UserDAO {    @Override  **public** **void** delete(**int** userId) {  *// delete user from user table*    }    @Override  **public** User[] findAll() {  *// get a list of all users from user table*  **return** **null**;  }    @Override  **public** User findByKey(**int** userId) {  *// get a user information if we supply unique userid*  **return** **null**;  }    @Override  **public** **void** insert(User user) {  *// insert user into user table*    }    @Override  **public** **void** update(User user) {  *// update user information in user table*    }    } |

Well, now let us get into factory class. If you don’t know anything about factory method design pattern follow the tutorial,

[Factory Method Design Pattern in Java](http://www.jpgtutorials.com/factory-method-design-pattern-in-java)  
The sample factory class is given below.

|  |
| --- |
| **public** **class** UserDAOFactory {    **public** **static** UserDAO getUserDAO(String type) {    **if** (type.equalsIgnoreCase("jdbc")) {  **return** **new** UserDAOImpl();  } **else** {  **return** **new** UserDAOImpl();  }  }    } |

In the above factory class we define a static method to get a reference to DAO derived class based on parameter. It returns appropriate class based on condition.

Well our DAO implementation is ready. Now let us examine the client code to insert user information into user table.

|  |
| --- |
| *//instantiate user object*  User user=**new** User();  user.setUserId(1);  user.setName("Jinoy P George");  user.setDesignation("Programmer");  user.setAge(35);  *//get a reference to UserDAO object*  UserDAO userDAO=UserDAOFactory.getUserDAO("jdbc");  *//call insert method by passing user object*  userDAO.insert(user); |

In the above code snippet, we create user object and assign properties. Then we call static method of factory class to get a reference to UserDAO object. The factory class returns appropriate derived class, in our case UserDAOImpl. It then calls the insert() method DAO object. Here DAO provides a clean abstraction for data access from architecture point of view.

Well, I think the tutorial is lengthy though I wanted to write a short one. Sorry for wasting your time.

cheers,