Object Oriented Databases

A guide for implementing your own Object Oriented Database with the use of Java Data Objects

ASHLEY NELSON
# Table of Contents

PREREQUISITES ......................................................................................................................... 3

SETUP .......................................................................................................................................... 3

INTRODUCTION ............................................................................................................................. 3

OBJECT ORIENTED DATABASES .................................................................................................. 3

- Benefits of Object Oriented Databases ..................................................................................... 4
  - No Disassembly or Reassembly ................................................................................................. 4
  - Support for Object Relationships ........................................................................................... 5

DRAWBACKS OF OBJECT ORIENTED DATABASES ......................................................................... 6

JAVA DATA OBJECTS (JDO) .......................................................................................................... 6

- Setting up PersistenceManagerFactory ..................................................................................... 7
- Creating our Message Object to be Stored ................................................................................. 7
- Making a Java Class Persistent ................................................................................................... 8
- Storing a Message ....................................................................................................................... 9
- JDOQL ......................................................................................................................................... 10
- Inheritance with JDO .................................................................................................................. 11

FURTHER YOUR LEARNING .......................................................................................................... 12

WORKS CITED ............................................................................................................................... 14
Prerequisites

In order to best understand the information presented in this tutorial, it is recommended that you have experience with the Java programming language and a basic understanding of relational databases. No prior experience with object oriented databases is needed.

Setup

You will need to install an object oriented database that supports the Java Data Objects API in order to complete the following tutorial. There are plenty of free options, including:

- ObjectDB
- JDOInstruments
- Google App Engine
- db4o

Follow the JDO setup instructions given in the documentation for the database of your choice. This will typically involve copying JDO JAR files to your project and setting up an XML configuration file.

Introduction

Welcome to this tutorial on Object Oriented Databases and Java Data Objects! This guide aims to provide you with an understanding of what object oriented databases are, along with when and how to use them. It will also introduce Java Data Objects, a popular interface used to make Java objects compatible with object oriented databases. You will use these technologies to create a project that simulates a message board.

Object Oriented Databases

Simply put, an object oriented database, or OODBMS (Object Oriented Database Management System), is a database that can store objects. When queried, these databases return the objects in their entirety, which means a returned object’s attributes and methods are as usable as they were before the object was ever stored in the database. There are a variety of OODMBS platforms available to the
public, as well as a great number of APIs and query languages to use with them. This is because there is no prominent standard for object oriented databases, unlike the widely-accepted SQL standard for relational databases.

**Benefits of Object Oriented Databases**

**No Disassembly or Reassembly**

Currently, the most popular database in use is a relational database. Relational databases store their data in tables, with each row in a table corresponding to a record and each column representing the record’s properties. Their main distinction is that they can only hold primitive data types, such as an integer or text. They are not capable of storing more complex data types unless the data is serialized or broken up into primitive components.

To illustrate this idea, let’s say that we have an object of type Cat that we want to store in a database. If we happened to have an object oriented database at our disposal, we would simply store the Cat object as is, and the data inside Cat would be kept in its original form, as seen in *Figure 1* below.

![Figure 1 - Cat in Object Oriented Database](Source: Hand and Chandler)

On the other hand, if we were to store this Cat object into a relational database, we would have to do a little bit of work to make it compatible. Seeing as “Cat” is not a primitive data type, we would either have to serialize the entire object and store it all in one field of type “text”, or we would have to break it down into its properties and store it piece by piece, as shown in *Figure 2*. 
Serializing the entire object is not practical, as that would make it impossible to query the database and find any object’s record based on its properties. Therefore, the best method would be to break it up into primitive properties. In Figure 2, you can see that the Cat object was broken up into body part properties, with one body part in each field. The developer using this database had to take extra time to go through each attribute, determine the primitive data type that best suits it, and store it as such. As objects increase in complexity, this process turn out to be much more tedious, and that is when object oriented databases become necessary.

Support for Object Relationships

Another important and beneficial feature of object oriented databases is their support for storing relationships between objects. If we again refer to the Cat example, we can see in Figure 2 that the cat is made up of many parts, such as two eyes, two ears, and a tail. Though it would have been inconvenient for the developer to disassemble and store the Cat object in a relational database, it was certainly not impossible; we assume in that example that the cat’s properties are all primitive and can be stored together in one table.

Consider, however, the possibility that each of the Cat object’s properties contains properties of their own. For example, rather than a tail merely being stored as “tail”, what if it were an object of type Tail that contained properties such as length, thickness, and color? In the case of a relational database, this would result in the creation of a new table to hold Tail and all of its properties, with the “tail” column in the Cat table holding some sort of value to link it to the correct tail record in the Tail table. Therefore, any query regarding the Tail properties of a Cat would require that the two tables be joined, which can get costly as more tables are involved.
Object oriented databases, on the other hand, make object relationships such as Cat and Tail very simple to navigate. This is because objects can be stored as properties of other objects through the use of pointers. Rather than joining tables or performing multiple queries to retrieve information about a cat’s tail, an object database would need to perform just one query for the Cat object whose Tail we want to know more about. Once this Cat object is returned, we would simply gain access to the Tail object through the pointer stored in the Cat object.

**Drawbacks of Object Oriented Databases**

In spite of the conveniences described above, there are still some disadvantages that come with using object oriented databases, as well as scenarios in which a relational database would be more suitable.

First of all, there is no mistaking the fact that relational databases are much simpler than object oriented databases; they only hold primitives and their data is stored very uniformly. Therefore, when dealing with very simplistic data that can be contained in just one or two tables, it is more efficient to use a relational database.

Another disadvantage to object oriented databases is the fact that they have no set standards. This means that there are fewer user tools for object databases, and less support in general for developers. The multitude of language-specific APIs also makes it more difficult to port an object database to an application that is written in another programming language, because it would most likely involve the use of a different API.

**Java Data Objects (JDO)**

Now that you have a basic understanding of object oriented databases, it is time to put one into practice! We will be using the Java Data Objects (JDO) API to store Java objects into an object database and explore some of its object oriented capabilities in our creation of a simple message board.

Before we continue, make sure you have an object oriented database installed on your computer with an empty (except for configuration files) Java project that is configured for Java Data Objects. See the *Setup* section if you have not yet done this.
Setting up PersistenceManagerFactory

Every implementation of JDO must have a PersistenceManagerFactory (PMF) that provides a PersistenceManager (PM) whenever a database operation is to be done. Once a PMF has been created, the PM instances that it provides are the only way that a developer programmatically interacts with the database. They allow "transactions," or atomic operations, to be carried out on the database, and must be closed immediately after the transaction has occurred.

There should be a separate PMF for each database being used in an application. In the case of our application, we will be using just one database, so we will therefore need just one instance of a PMF.

1. Create a singleton class called **PMF** that maintains a single instance of PersistenceManagerFactory and returns that instance when its method `get()` is called. Use the code below to create this instance of PersistenceManagerFactory, and be sure to replace the generic property values with your own.

```java
Properties properties = new Properties();
properties.setProperty("javax.jdo.PersistenceManagerFactoryClass", "(my_implementation_pmf_class)");
properties.setProperty("javax.jdo.option.ConnectionDriverName", "com.mysql.jdbc.Driver");
properties.setProperty("javax.jdo.option.ConnectionURL", "jdbc:mysql://localhost/myDB");
properties.setProperty("javax.jdo.option.ConnectionUserName", "login");
properties.setProperty("javax.jdo.option.ConnectionPassword", "password");
PersistenceManagerFactory pmf = JDOHelper.getPersistenceManagerFactory(properties);
```

Creating our Message Object to be Stored

Before we move any further with Java Data Objects, we should create the class for the object that we plan on storing in our database. Since we will be creating a message board, it makes sense that the object being stored should be a message.

2. Create a class titled **Message**, and include the following instance variables along with getter methods for each:
   - `java.util.Date date`
   - `String author`
• String message

You should also include a constructor that takes in those three variables as arguments.

**Making a Java Class Persistent**

JDO provides a convenient way to prepare Java classes for persistent storage without changing much of their code. This is done using annotations, provided in the `javax.jdo.annotations` package.

3. In order to tell our application that we plan to store our Message object in its database, annotate it as such:

```java
import javax.jdo.annotations.PersistenceCapable;

@PersistenceCapable
public class Message {
    // ...
}
```

However, declaring the class as persistence-capable is not enough. JDO also requires that you specify the persistence for each instance variable within the class that you intend on storing. This is done using the `@Persistent` annotation. If an instance variable is not marked as persistent, then its data will not be transferred to the database with the rest of the object.

4. Add this annotation above every instance variable declaration in your Message class, similar to what is shown below:

```java
import javax.jdo.annotations.Persistent;

// ...

@Persistent
private Date date;
```
Now that you have declared the Message class as persistent, along with all of its instance variables, there is one last thing you must do – create an instance variable to hold the object’s key. This is because the database needs a unique property that it can use to identify objects, just as a relational database does.

5. Create the key instance variable as shown below:

```java
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.PrimaryKey;

// ...
@PrimaryKey
@Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
private Long key;
```

The above code uses the annotation @PrimaryKey to point out which instance variable is responsible for holding the key of the object. The @Persistent annotation is also included with an argument specifying which mechanism should be used to generate the key. The current mechanism will increment the key by 1 for each new object added. Because this key is automatically generated by the database, you should not add this instance variable to the arguments within your constructor.

Storing a Message

Your Message class should now be perfectly persistent and therefore ready to store in the database. As mentioned earlier, the PersistenceManager (PM) is responsible for all interactions with the database, which means you need to get an instance of PM in order to store your Message object. This can be done using our PMF class.

6. Create a new class called MessageBoardRunner and give it a main() method. Place the below code in this method to obtain a PM instance and store your message.

```java
PersistenceManager pm = PMF.get().getPersistenceManager();
Message message = new Message(new Date(), ”Your Name”, ”Hello!”);
try {
    pm.makePersistent(message);
```
As you can see, the PM merely calls its `makePersistent()` method with the object to store as its argument, and the object is stored in the database. It is also important to note that the PM was closed immediately after storage of the object. This must be done to complete the operation.

If your database comes with a user interface for viewing its contents, you should be able to open it and see the Message object that you just stored. You should also see a generated key of “1”.

**JDOQL**

JDO also comes with its own query language, called Java Data Objects Query Language (JDOQL). This language has all of the capabilities that SQL has, and its query structure is almost exactly the same. One of the only differences is that the `FROM` clause now refers to an object class, rather than a table. Since we have begun storing messages in our database, querying them is the next logical step, for it allows us to retrieve the objects we have stored and display them.

This tutorial will have you directly print out the contents that you query in the database to the console, but feel free to create your own user interface that acts as a working message board instead.

7. Below the code that you just added in your `main()` method, place the following query. Don’t forget to close the PM when you are done:

```java
import javax.jdo.Query;
// ...
PersistenceManager pm = PMF.get().getPersistenceManager();
Query q = pm.newQuery(Message.class);
q.setOrdering("date desc");
List<Message> messages = (List<Message>) q.execute();
```

Again, since we are interacting with the database, we needed to get a new instance of a PM. We then used the declarative API provided through the Query class to formulate our query, rather than writing the entire query string ourselves. We told the query to look for objects of class Message and place them
in order of descending date. We then executed this query, which returned a List of Message objects that matched our query. In this case, it should return every Message object stored in the database. In the future, however, it would make sense to impose a limit on the number of results returned as more and more messages are added. This can be done using the `setRange(...)` method of Query, which takes in a start and end index and returns the results that fall within that range.

8. Iterate through the list and print out the details of each Message found

You are encouraged to check out the [JDOQL Documentation](#) for a more complete understanding of its capabilities.

### Inheritance with JDO

One of the coolest things about object oriented databases is that they support the entire structure of object oriented languages, including features such as inheritance and polymorphism. In this section, we are going to see exactly how our object oriented database handles a subclass of Message being stored through the PM.

9. Create a class that extends your Message class, and add a new instance variable of type String called “email”. Be sure to add the same persistence annotations as you did in the [Making a Java Class Persistent](#) section. However, you do not need to create another key variable because it is already included in the class you extended.

By default, the subclass that we create will be stored alongside its superclass’ objects in the database, and will be returned whenever its superclass is queried. There are a number of options besides this, such as separating the classes entirely, but we will stick with the default. To accommodate this behavior, we need to add an annotation to the original Message class telling the database to add a “discriminator” based on class name. This means that if you were to look at the contents of your database, you would now see an extra column called “DISCRIMINATOR” that contains the actual class name of the object, which helps us distinguish between our original Message object and its subclass.

10. Add the following discriminator annotation to your Message class, as shown:
import javax.jdo.annotations.Discriminator;
import javax.jdo.annotations.DiscriminatorStrategy;

@PersistenceCapable
@Discriminator(strategy=DiscriminatorStrategy.CLASS_NAME)
public class Message {

11. Now add code in your main() method that will store an object of the
subclass that you just created, right after the code that adds your original
Message object.

12. Run your project so you have another object of type Message stored in
your database, as well as an object of your Message subclass.

If you use the user interface that your database provides, you should now see
two extra columns in your Message structure – one being the Discriminator field
and the other being the Email field. All of the new properties in the Message
subclass have now become a part of the Message table structure.

Further your Learning

Though this tutorial covered a number of topics regarding object oriented
databases and the Java Data Objects API, there is still so much to learn. Here is a
list of sources that are recommended for furthering your understanding of some
of the topics covered above.

- Introduction to Object-Oriented Databases – This is a relatively old PDF
about object oriented databases, but it has a lot of information that
remains true even a decade later. It was helpful in laying out the benefits
and disadvantages of object oriented databases, and contained nice visuals
to demonstrate these ideas.

- Java Data Object Documentation – This tutorial was only able to briefly
touch on different features of the Java Data Objects API, but there are so
many more features that were not brought up. If you are curious about
what else JDO can do, please look into its documentation for the full
library.

- JDO with Google App Engine – Although Java Data Objects can be used
with a variety of platforms, I found the App Engine documentation and
tutorial on using JDO to be a bit more helpful than the rest. It provides an overview on getting JDO set up that is not specific to App Engine itself, with the exception of the configuration files.
Works Cited

<http://odbms.org/download/005.01%20Chandler%20Introduction%20to%20Object-Oriented%20Databases%20September%201998.pdf>.


