

databeans

fully object oriented persistence for java

version 2.0

1. [Introduction](#)
 1. [Different kinds of persistence](#)
2. [Persistence for java](#)
3. [databeans](#)
4. [The databeans](#)
 1. [What is a databean ?](#)
5. [Coding a databean](#)
6. [Using a databean](#)
7. [Data structures](#)
8. [Notifications](#)
9. [The backing store \(heapspace\)](#)
 1. [Memory allocation](#)
10. [Garbage collection](#)
11. [Connections and transactions](#)
12. [Administration tasks](#)
 1. [Importing and exporting the backing store](#)
13. [Other administration tasks](#)
14. [Advanced topics](#)
 1. [databeans constructors with more than zero args](#)
15. [Note on overriding the init method](#)
16. [Persistent classes and the way they handle java classes definition changes](#)
17. [References](#)
 1. [Persistence](#)
18. [JavaBeans](#)
19. [RMI](#)
20. [Security](#)
21. [Transactions](#)
22. [Memory allocation & garbage collection](#)
23. [Sources of sample data](#)

Introduction

This is the user manual for databeans, which is a new, fully object oriented persistence framework for java. First we will try to situate databeans amongst the various existing persistence schemes. These can be arranged according to the following criteria:

- orthogonal/API-based

- object-relational/full object (depending on the nature of the backing store)
- coarse grained (all at once) to fine grained (incremental)

Different kinds of persistence

Orthogonal means that there is no difference between not persistent and persistent. All objects are made persistent. In other words, a persistent object doesn't have to implement some specific class (there is no data access object). The advantage is that the developer has no side effects when using persistence [1], but it entails either an extension of the virtual machine, or some persistence support from the underlying OS [2].

Object-relational versus full object means that so-called object-relational mappings are to be defined, with the accompanying complexity of the underlying databases (which on the other hand are robust).

Incremental versus all-at-once means that not all objects have to be written to/read from the backing store at the same time. Ofcourse the more fine-grained is the better, for obvious performance reasons.

Persistence for java

Regarding java, there is one persistence mechanism that was standardized, it is called serialization. It is an orthogonal, full object, all-at-once scheme. All the other java orthogonal attempts (that I know of) have stopped around year 2000, be them all-at-once or, most interestingly, incremental. The best effort in this latter area was the PJama project [3]. This or similar projects stalled apparently because of the difficulty to make them survive the JVM updates [2].

So now all the java persistence products that exist in the mainstream (JDO, Hibernate, JPA) are non orthogonal, API-based ones. It should be noted however that there is a move back to orthogonality with the Plain Old Java Object (POJO) concept. It mainly consists in annotating objects to draw the persistence related statements out of the code, and seem promising.

The aforementioned mainstream products are all object-relational ones. There is one API-based, full object product : ozone-db. But it is getting a bit old (it exists since 1999) and, on the status page of the latest (experimental) version, one can read that "enhanced collections api with support for lazy loading" is a work in progress. One can infer from that that the mechanism is not as fine-grained as one may wish.

databeans

Hence there is room for yet another project in the not orthogonal, not relational category, and that is the purpose of the databeans project. It had to provide both a backing store and data access objects. The former is called "heapspace" (in reference to databases' tablespaces) and the latter are the databeans themselves. Both are described in detail in the sequel.

The databeans

What is a databean ?

A databean is a Data Access Object (DAO). Every persistent object must extend a class that is provided in the API, namely the `PersistentObject` class.

In addition, a databean implements the Bean pattern : data access is performed through the use of the javabeans accessor methods to read/write data on disk in place of in class instance fields. This is the main concept of the databeans framework and it is rather powerful, as you will see for yourself.

Coding a databean

Beside extending `PersistentObject`, a databean has the following characteristics:

- it has no constructor other than the implicit no-arg constructor, which is left unimplemented.
- other constructors are replaced by `init` methods (see the advanced topics regarding the few implications)
- it has no instance fields, which are replaced (not supplemented as in the regular Bean pattern) by accessor methods, namely a pair of getter-setter methods for each property
- the accessor methods delegate their work to build-in `get` and `set` methods that take the property name as argument and perform actual data storage access.

Here is a sample databean:

```
public class Employee extends PersistentObject {
    public String getName() { return (String)get("name"); }
    public void setName(String s) { set("name",s); }

    public Department getDepartment() { return
(Department)get("department"); }
    public void setDepartment(Department d) { set("department",d); }

    public String getLocation() { return (String)get("location"); }
    public void setLocation(String s) { set("location",s); }

    public double getSalary() { return
((Double)get("salary")).doubleValue(); }
    public void setSalary(double d) { set("salary",new Double(d)); }

    public Employee getManager() { return (Employee)get("manager"); }
    public void setManager(Employee e) { set("manager",e); }

    public String getJob() { return (String)get("job"); }
    public void setJob(String s) { set("job",s); }
}
```

This sample has the three possible kinds of property types : primitive (salary), reference-not persistent (name, location, job) and reference-persistent (department, manager). (Indeed Department is another persistent type of our example.)

The properties with reference-non persistent type are simply serialized into the backing store. The choice of using a persistent versus a non-persistent type depends upon whether it is mutable or not. If it is mutable, it is worth coding a dedicated databean. If it is immutable, one can just use the suitable regular `Serializable` java class.

Using a databean

The bottom line is that, the creation excepted, a databean works just like any other javabean.

A databean is not created by a new call, but instead by calling one of the create methods of the `Connection` object (obtaining a connection and using its transaction facilities is covered in the suitable chapter):

```
PersistentObject create(String name);
PersistentObject create(Class clazz);
PersistentObject create(Class clazz, Class types[], Object args[]);
PersistentArray create(Class componentType, int length);
PersistentArray create(Object component[]);
```

(As you can see there is a persistent array type. The functioning is roughly the same as that of other persistent classes, except that its accessor methods take an index argument in place of a property name.)

The factory methods above can take as arguments either:

- the fully qualified name of the remote object's class
- the remote object's class
- the remote object's class plus the types and arguments of a `init` method, if present

For persistent arrays, the arguments are either:

- the component type (primitive or reference) and the length
- a java array, the contents of which the new array will be initialized with

Data structures

The databeans API provides persistent implementations of the `Collection` classes. These classes work exactly as their transient counterpart. Persistent data can be processed transparently just like in-memory data.

Notifications

One can have persistent objects notified of property changes to another persistent object, by that

object extending `PersistentObject`'s subclass `NotifiedObject`. Client objects register themselves as `PropertyChangeListeners` or `VetoableChangeListeners`, and receive `PropertyChangeEvent`s accordingly. This mechanism emulates the trigger concept of relational databases.

See the important Note on overriding the `init` method in the advanced topics, as `NotifiedObject` is concerned.

The backing store (heap space)

Memory allocation

The backing store is basically an on-disk heap. Space for persistent object's images is allocated using a `alloc()` call and freed using `free()`. There are two memory models available: 32 and 64 bits. The default is 64 bits, with a total addressed space of 2^{64} bytes. A single object image is limited to 2^{32} bytes in both memory models.

The allocation algorithm is of *best-first with coalescing* type [13] with the improvement that it uses a red-black tree [14] to hold the free chunks instead of a hashtable.

Garbage collection

The garbage collector is a reference counting one, backed by a mark-and-sweep GC for remaining self-referencing structures.

The garbage collection chronology is as follows:

1. distributed garbage collection
 1. a client stub is garbage collected (locally)
2. the server object's lease is not renewed after the configured time [7]
3. the server object (which is specific to this particular client connection) is unexported, unless there remain other stubs to it
4. garbage collection local to the server
 1. the server object is garbage collected
5. the "accessor" of the object, which is shared amongst all client connections, is in turn garbage collected, unless it is used by other clients
6. the on-disk image of the object is marked as unused
7. this object's image is freed from the on-disk heap, unless it is still referenced by other objects' images
8. on disk garbage collection
 1. the images this object's image was referring to have their reference count decremented
9. if it's null and they are not used by any client, they are in turn freed from the heap. If they are part of one or more self-referencing structure, they will have to wait for the mark-and-sweep garbage collection to take place, which is either when the heap space gets exhausted, or when the administrator issues a `gc()` call (or also when the database server is restarted)

Connections and transactions

Connections are obtained as follows:

```
Connections.getConnection("//[host]/store");
```

The features of the returned Connection object are very similar to those of a `java.sql.Connection`:

```
int getTransactionIsolation();
void setTransactionIsolation(int level);
boolean isReadOnly();
void setReadOnly(boolean readOnly);
boolean isAutoCommit();
void setAutoCommit(boolean autoCommit);

void commit();
void rollback();
void close();
boolean isClosed();

XAResource getXAResource();
```

In addition the following methods are available:

```
Object root();
void setRoot(Object obj);
```

, which are used to retrieve/set the object that one chooses to be the root of the persistent store. This can be an object as simple as a new `Date()` for instance or anything from it to domain models of arbitrary complexity.

Administration tasks

Importing and exporting the backing store

Two beanshell scripts are provided to export/import the backing store to/from a XML file:

```
java -classpath databeans.jar bsh.Interpreter bsh/export.bsh [file.xml]
java -classpath databeans.jar bsh.Interpreter bsh/import.bsh [file.xml]
```

The used underlying mechanism is derived from the `java.beans` long term persistence scheme introduced in java 1.4 [4].

Other administration tasks

All administration tasks are available through an `AdminConnection` object, obtained as follows:

```
Connections.getAdminConnection("//[host]/store");
```

, with the methods:

```
PersistentSystem system();
void changePassword(String username, String password);
void changePassword(String username, String oldPassword, String
newPassword);
void addUser(String username, String password);
void deleteUser(String username);
void inport(String name);
void export(String name);
void shutdown();
void gc();
long allocatedSpace();
long maxSpace();
```

, which are self-explanatory. Note that the system itself is a persistent object, and can be browsed and modified at will, which is powerful but can be dangerous.

Advanced topics

databeans constructors with more than zero args

The java class of a persistent object is special in that it has to provide support not only for the creation, as for regular java objects, but also for the "re-creation" of the persistent object, when it is retrieved from the backing store. Both warrant an object instantiation, but they have to be handled in different ways. The creation will perform possible initializations (for instance to assign default values to some properties), whereas the re-creation will have to do nothing except instantiate the in-memory java object. The creation is achieved by `init` methods. The re-creation is achieved by the implicit no-arg constructor. The following rules must be obeyed:

- the re-creation constructor is left unimplemented, and hence has no arguments and is `public`, as per java rules ; it does nothing
- a creation `init` method can be omitted, in which case one with no arguments will be inherited from `PersistentObject`
- the creation `init` methods can take any number of arguments of any types.

Note on overriding the `init` method

Care must be taken, when overriding an `init` method, to call the superclass' no-arg `init` if no other is specified, and in first place, to emulate the implicit call to `super()` that takes place in regular constructors. See for instance the case of `NotifiedObject`, whose `init` has

initializations to perform (namely create a `PropertyChangeSupport` and a `VetoableChangeSupport`).

Persistent classes and the way they handle java classes definition changes

At the same time that an object is being persisted, its definition is recorded into the backing store as a "persistent class" object, which will then be used to map the property names to their actual location in the disk object images. Should the definition of the persistent object's java class change, there are three cases:

1. a property is added : a `PersistentException` will be thrown when the access attempt is made (either for reading or writing)
2. a property is removed : the property will remain in the object's image, it will simply become unavailable to the users
3. a property type is changed. There are two subcases:
 1. the new type is assignable from the old type : the change will be transparent
4. the new type is not assignable from the old type : a `ClassCastException` will be thrown when a read access attempt is made (the write access will still work, except for primitive types)

The persistent objects that are created after the definition change will have the same persistent class as the persistent object created before the definition change (in fact a single instance of the persistent class is shared by all persistent objects of one type), unless all instances of the type are erased from the store and the databeans server is restarted (the unused persistent classes are then discarded) before the new creations.

References

Persistence

- [1] [Wikipedia article on orthogonality](#)
- [2] [Straightforward Java Persistence Through Checkpointing](#)
- [3] [Pjama](#) an experimental prototype that implements *Orthogonal Persistence for the Java platform* (OPJ)

JavaBeans

- [4] [JavaBeans Documentation](#)

RMI

- [5] [Getting Started Using RMI](#)
- [6] [RMI Specification](#)

[7] [RMI FAQ](#)

Security

[8] [JAAS Tutorials](#)

[9] [JAAS Login Configuration File](#)

[10] [Default Policy Implementation and Policy File Syntax](#)

Transactions

[11] [The SQL92 standard](#) specifically : chapter 4.28 SQL-transactions

[12] [Java Transaction API \(JTA\)](#)

Memory allocation & garbage collection

[13] [A Memory Allocator](#)

[14] [Wikipedia article on Red-black tree](#)

[15] [GC FAQ](#) -- draft

[16] [No Silver Bullet](#) - Garbage Collection for Java in Embedded Systems

Sources of sample data

[17] [Oracle8i SQL Reference - SQL Statements: SELECT and subquery - Examples](#)

modified on Sat Mar 1 2008