

EYEDB Java Binding

Version 2.8.0

January 2006

Copyright © 2001-2006 SYSRA

Published by SYSRA
30, avenue Général Leclerc
91330 Yerres - France

home page: <http://www.eyedb.org>

Contents

1	Getting Started	5
1.1	Programming Steps	5
1.2	The ODL Schema	6
1.3	Compiling the ODL Schema	6
1.4	The Java produced	7
1.5	Compiling the Java stubs	8
1.6	Writing a Client	8
1.7	Compiling the client	10
1.8	Updating the schema	10
1.9	Running the client	10
1.10	An advanced example	11

The Java Binding

The Java binding maps the EYEDB object model into Java by introducing a generic API and a tool to generate a specific Java API from a given schema, built upon the generic API.

The generic Java API is made up of about one hundred and fifty classes such as some abstract classes as the `object` and `class` classes and some more concrete classes such as the `database` and `image` classes.

Each type in the EYEDB object model is implemented as a Java class within the Java API: there is a one for one mapping between the object model and the Java API.

This mapping follows a very simple naming schema: each Java class mapped from a type has the name of this type prefixed by `org.eyedb..`

For instance, the `object` type in the EYEDB object model is mapped on the `org.eyedb.Object` Java class, while the `agregat` is mapped to the `org.eyedb.Agregat` Java class.

The use of the Java language for an EYEDB binding has been motivated by several reasons:

- Java is architecture independent.
- Java is indeed valuable for distributed network environment.
- Java has a very rich builtin library.
- Java is secure.
- Java is easier to program than C++.

The Java binding is very close from the C++ binding: the class interfaces are identical, the functionalities are the same; only the language is slightly different.

1 Getting Started

We will introduce the Java binding through a simple example of defining an ODL schema, generating the java stubs and then writing a client program. We will explain only what is necessary to understand the example to avoid begin swamped by unnecessary details.

The example that we will develop in this section is the same that has been developped in the C++ binding chapter, a couple of classes `Person` and `Car`.

1.1 Programming Steps

The following programming steps are typically required to write a Java client program dealing with the given schema:

1. Define the schema, using the EYEDB Object Definition Language.
2. Generate the Java stubs, using the `eyedbodl` tool.
3. Write a client program using the generated Java package.

We will illustrate these steps in the remainder of this chapter.

1.2 The ODL Schema

The ODL schema for our application can be described as follows:

```
/* person.odl */

class Person {
    string name;
    int age;
    Person *spouse inverse Person::spouse;
    Person *father;
    set<Person *> children inverse Person::father;
    set<Car *> cars inverse Car::owner;

    index on name;
    index on age;
    constraint<nonnull> on name;
    constraint<unique> on name;
};

class Car {
    string brand;
    int num;
    Person *owner inverse Person::cars;

    index on brand;
    index on num;
};
```

This schema describes two classes **Person** and **Car**, one 1 to 1 relationship (attribute **spouse** in class **Person**) and two 1 to many relationships (from **Person::father** to **Person::children** and from **Car::owner** to **Person::cars**).

1.3 Compiling the ODL Schema

The ODL schema must be compiled, both to check the schema and to bind it into some Java code so that it can be used in a client program.

The **person.odl** file can be compiled using the following command:

```
eyedbodl --gencode=JAVA --package=person person.odl
```

or :

```
eyedbodl --gencode=JAVA --package=person -d person
```

The **eyedbodl** tool contains a lot of command line options to control the generated code.

There are two mandatory options:

```
odlfile|-|-d dbname|--database=dbname : Input ODL file (or - for standard input) or the database name
--package=package                      : Package name
```

and some optionnal options:

```
--output-dir=dirname          : Output directory for generated files
--output-file-prefix=prefix   : Ouput file prefix (default is the package name)
--class-prefix=prefix         : Prefix to be put at the beginning of each runtime class
--db-class-prefix=prefix      : Prefix to be put at the beginning of each database class
--attr-style=implicit         : Attribute methods have the attribute name
--attr-style=explicit         : Attribute methods have the attribute name prefixed by get/set (default)
--schema-name=schname        : Schema name (default is package)
--export                      : Export class instances in the .h file
--dynamic-attr                : Uses a dynamic fetch for attributes in the get and set methods
--down-casting=yes            : Generates the down casting methods (the default)
```

```

--down-casting=no           : Does not generate the down casting methods
--attr-cache=yes           : Use a second level cache for attribute value
--attr-cache=no           : Does not use a second level cache for attribute value (the default)
--namespace=namespace     : Define classes with the namespace namespace
--c-suffix=suffix         : Use suffix as the C file suffix
--h-suffix=suffix         : Use suffix as the H file suffix
--gen-class-stubs         : Generates a file class_stubs.h for each class
--class-enums=yes         : Generates enums within a class
--class-enums=no          : Do not generate enums within a class (default)
--gencode-error-policy=status : Status oriented error policy (the default)
--gencode-error-policy=exception : Exception oriented error policy
--rootclass=rootclass     : Use rootclass name for the root class instead of the package name
--no-rootclass            : Does not use any root class

```

1.4 The Java produced

The ODL compiler will then produce the following Java files in the **person** directory:

- Person.java
- Car.java
- Database.java
- set_class_Car_ref.java
- set_class_Person_ref.java

The Person.java file produced by the ODL compiler contains the following (only the method interfaces are shown, not the body):

```

//
// class Person
//
// package person
//
// Automatically Generated by eyedbodl at ...
//

package person;

public class Person extends org.eyedb.Struct {

    public Person(org.eyedb.Database db);
    public Person(org.eyedb.Struct x, boolean share);
    public Person(Person x, boolean share);

    public void setName(String _name);
    public void setName(int a0, char _name) throws org.eyedb.Exception;
    public String getName();

    public void setAge(int _age) throws org.eyedb.Exception;
    public int getAge() throws org.eyedb.Exception;

    public void setSpouse(Person _spouse) throws org.eyedb.Exception;
    public Person getSpouse() throws org.eyedb.Exception;
    public void setSpouse_oid(org.eyedb.Oid _oid) throws org.eyedb.Exception;
    public org.eyedb.Oid getSpouse_oid() throws org.eyedb.Exception;

    public org.eyedb.CollSet getChildrenColl() throws org.eyedb.Exception;
    public int getChildrenCount() throws org.eyedb.Exception;
    public Person getChildrenAt(int ind) throws org.eyedb.Exception;
    public void setChildrenColl(org.eyedb.CollSet _children) throws org.eyedb.Exception;
    public void addToChildrenColl(Person _children) throws org.eyedb.Exception;
    public void addToChildrenColl(org.eyedb.Oid _oid) throws org.eyedb.Exception;

```

```

public void rmvFromChildrenColl(Person _children) throws org.eyedb.Exception;nn
public void rmvFromChildrenColl(org.eyedb.Oid _oid) throws org.eyedb.Exception;nn

// and so on. father and cars set and get methods.
// ...

// protected and private parts
// ...
};

```

eyedbodl has generated three constructors (including two copy constructors), and get and set methods for each attributes.

1.5 Compiling the Java stubs

The generated code must be compiled using the `javac` compiler.

The Java CLASSPATH must include both the EYEDB Java class path and the current directory. The EYEDB Java class is generally installed at `libdir/eyedb/java/eyedb.jar` where `libdir` is the object code library directory, usually `prefix/lib`.

```

export CLASSPATH=<libdir>/eyedb/java/eyedb.jar
javac -depend -d. person/Database.java

```

1.6 Writing a Client

Here is a simple example of a Java client which opens a database, creates 2 person instances and mary them:

```

//
// class TestP.java
//

import person.*;

class TestP {
    public static void main(String args[]) {

        // Initialize the eyedb package and parse the default eyedb options
        // on the command line
        String[] outargs = org.eyedb.Root.init("TestP", args);

        // Check that a database name is given on the command line
        int argc = outargs.length;
        if (argc != 1) {
            System.err.println("usage: java TestP dbname");
            System.exit(1);
        }

        try {
            // Initialize the person package
            person.Database.init();

            // Open the connection with the backend
            org.eyedb.Connection conn = new org.eyedb.Connection();

            // Open the database named outargs[0]
            person.Database db = new person.Database(outargs[0]);
            db.open(conn, org.eyedb.Database.DBRW);

            db.transactionBegin();
            // Create two persons john and mary
            Person john = new Person(db);
            john.setName("john");
            john.setAge(32);

            Person mary = new Person(db);
            mary.setName("mary");

```



```

    mary.setAge(30);

    // Mary them :-)
    john.setSpouse(mary);

    // Store john and mary in the database
    john.store(org.eyedb.RecMode.FullRecurs);

    db.transactionCommit();
}
catch(org.eyedb.Exception e) { // Catch any eyedb exception
    e.print();
    System.exit(1);
}
}
}

```

The client contains the followin line at the beginning of all its modules:

```
import person.*;
```

This line means that you are imported the generated **person** package.

NOTE : those package importations are not essential, as you can refer to the generated classes using the **person.** prefix; and that you can refer to the standard EYEDB package using the **org.eyedb.** prefix.

Before any EYEDB method call, you need to initialize the EYEDB package by calling the **org.eyedb.Root.init** method, as follows:

```
String[] outargs = org.eyedb.Root.init("TestP", args);
```

This method will take out from **args** all the EYEDB options such as **--host=<host>**, **--port=<port>** (refer to the environment chapter).

The returned array **outargs** will contain the command line arguments except those that have been recognized as EYEDB options.

If you do not call the **org.eyedb.Root.init** method, the EYEDB java binding will not work.

Then, you need to initialiaze the person generated package, as follows:

```
person.Database.init()
```

Once again, if you do not call this method, the EYEDB java binding will not work properly.

Then, you need to open a connection with the EYEDB backend

```
org.eyedb.Connection conn = new org.eyedb.Connection();
```

The constructor **org.eyedb.Connection()** will try to connect to the backend using the host and port given on the command line.

In case of failure, an **org.eyedb.Exception** is thrown.

To open the database whose name is **outargs[0]**, you must first create a **person.Database** object, then call the **open** method on this object as follows:

```
person.Database db = new person.Database(outargs[0]);
db.open(conn, org.eyedb.Database.DBRW);
```

The **org.eyedb.Database.DBRW** flag indicates that we wants to open the database in the read/write mode.

Once that the database has been opened, we begin a transaction as follows:

```
db.transactionBegin();
```

This method call is necessary for any database access in read or write mode.

To create two persons whose name are **john** and **mary** and to mary them, you need to use the generated constructors and methods as follows:

```

Person john = new Person(db);
john.setName("john");
john.setAge(32);

Person mary = new Person(db);
mary.setName("mary");
mary.setAge(30);

john.setSpouse(mary);

```

Note that at this step, the persons `john` and `mary` have **not** been stored in the database. Those person references are runtime references, not persistent references.

To store them permanently in the database:

```
john.store(org.eyedb.RecMode.FullRecurs);
```

The `org.eyedb.RecMode.FullRecurs` argument means that all the object references included in the `john` object will be stored too ; so the `mary` reference which is the `john` spouse, will be stored too.

At this step, if you exit the program, the current transaction will be automatically aborted (rolled back). So the `john` and `mary` persons will not be effectively stored in the database.

To commit the transaction, you must do the following:

```
db.transactionCommit();
```

Note that all the constructors and methods previously called, may throw an `org.eyedb.Exception` in case of failure. The following code catches the exceptions, print them and exit the program:

```

catch(org.eyedb.Exception e) {
    e.print();
    System.exit(1);
}

```

1.7 Compiling the client

The client must be compiled as follows:

```
javac -d . TestP.java
```

1.8 Updating the schema

Before running the client for the first time, we need to create a test database and to update its schema according to the ODL description.

Note that to create a database, the EYEDB user under which you are working, must have the `dbcreate` system mode (refer to the administration section).

To create a database `foo`:

```

eyedbdbcreate foo
user authentication : <user name>
password authentication : <user passwd>

```

To update the schema `person.odl` in the database `foo`:

```

eyedbodl --u -d foo person.odl
Updating <unnamed> Schema in database foo...
Done

```

1.9 Running the client

To run properly, an EYEDB program needs some environment information such as the EYEDB host and port, the EYEDB user and password.

When we run a C++ program, the EYEDB runtime takes this environment from the UNIX environment variables.

There is not such mechanisms for a Java program: the `getenv()` function does not exist in Java.

So, we need to give all the environment using the command line arguments as follows:

```
eyedbjrun foo
```

After the program has run, let's verify that the objects have really been created in the `foo` database, using the `eyedboql` tool:

```
eyedboql
Welcome to eyedboql. Type '!help' to display the command list.
? \open foo
? select Person;
47886.20.803967:oid, 47887.20.2361599:oid
? \print full
47886.20.803967:oid Person = {
    name = "john";
    age = 32;
    *spouse 47887.20.2361599:oid Person = {
        name = "mary";
        age = 30;
        *spouse 47886.20.803967:oid Person = {
            <trace cycle>
        };
        *father NULL;
        *children NULL;
        *cars NULL;
    };
    *father NULL;
    *children NULL;
    *cars NULL;
};
47887.20.2361599:oid Person = {
    name = "mary";
    age = 30;
    *spouse 47886.20.803967:oid Person = {
        name = "john";
        age = 32;
        *spouse 47887.20.2361599:oid Person = {
            <trace cycle>
        };
        *father NULL;
        *children NULL;
        *cars NULL;
    };
    *father NULL;
    *children NULL;
    *cars NULL;
};
? \quit
```

As there is a unique constraint on the `Person` `name` attribute, if you run the program again, you will catch the following exception:

```
TestP.run foo
org.eyedb.StoreException: unique[] constraint error : attribute
'name' in the agregat class 'Person'
```

1.10 An advanced example

You can use EYEDB OQL withing the Java Binding, using the `org.eyedb.OQL` class.

For instance, the following program looks for the `Person` instance whose name is given on the command line. Then, it looks for the `Person` instances whose age is less than 3 years old, adds them to the children collection of the previous `Person` instance, and increments their age.

```
//
// class TestPC.java
//

import person.*;

class TestPC {
    public static void main(String args[]) {

        // Initialize the eyedb package and parse the default eyedb options
        // on the command line
        String[] outargs = org.eyedb.Root.init("TestPC", args);

        // Check that a database name is given on the command line
        int argc = outargs.length;
        if (argc != 2) {
            System.err.println("usage: java TestPC dbname person-name");
            System.exit(1);
        }

        try {
            // Initialize the person package
            person.Database.init();

            // Open the connection with the backend
            org.eyedb.Connection conn = new org.eyedb.Connection();

            // Open the database named outargs[0]
            person.Database db = new person.Database(outargs[0]);
            db.open(conn, org.eyedb.Database.DBRW);

            db.transactionBegin();
            // Looks for the Person john

            String pname = outargs[1];
            org.eyedb.OQL q = new org.eyedb.OQL(db, "select Person.name = \"" + pname + "\"");

            org.eyedb.ObjectArray obj_array = new org.eyedb.ObjectArray();

            q.execute(obj_array);

            if (obj_array.getCount() == 0) {
                System.err.println("TestPC: cannot find person '" + pname + "'");
                System.exit(1);
            }

            Person john = (Person)obj_array.getObjects()[0];

            // Looks for Person whose age is less than 3
            // and add them to the john children collection

            q = new org.eyedb.OQL(db, "select Person.age < 3");

            obj_array = new org.eyedb.ObjectArray();

            q.execute(obj_array);

            for (int i = 0; i < obj_array.getCount(); i++) {
                Person child = (Person)obj_array.getObjects()[i];
                child.setAge(child.getAge() + 1);
                john.addToChildrenColl(child);
            }
        }
    }
}
```

```

        // Update john and its children in the database
        john.store(org.eyedb.RecMode.FullRecurs);

        db.transactionCommit();
    }
    catch(org.eyedb.Exception e) { // Catch any eyedb exception
        e.print();
        System.exit(1);
    }
}
}

```

The following steps (and code) are identical to the previous example:

- importing package `person`
- initializing EYEDB package and `person` packages
- opening the connection
- opening the database
- beginning a transaction

The first difference comes using the `org.eyedb.OQL` constructor call:

```

String pname = outargs[1];
org.eyedb.OQL q = new org.eyedb.OQL(db, "select Person.name = \"" + pname + "\"");

```

This constructor makes a EYEDB OQL query in the opened database.

Nothing is returned from this query, but an exception is thrown if the query is invalid.

To get back the result of this query (i.e. the Person whose `name` is `pname`), one needs to scan the query.

Several `org.eyedb.OQL` methods allows us to scan the query:

```

public void scan(org.eyedb.ValueArray value_array)
    throws org.eyedb.TransactionException;

public void scan(org.eyedb.OidArray oid_array)
    throws org.eyedb.TransactionException;

public void scan(org.eyedb.ObjectArray obj_array)
    throws org.eyedb.TransactionException, org.eyedb.LoadObjectException;

public void scan(org.eyedb.ObjectArray obj_array, org.eyedb.RecMode rcm)
    throws org.eyedb.TransactionException, org.eyedb.LoadObjectException;

public void scan(org.eyedb.ValueArray value_array, int count, int start)
    throws org.eyedb.TransactionException;

public void scan(org.eyedb.OidArray oid_array, int count, int start)
    throws org.eyedb.TransactionException;

public void scan(org.eyedb.ObjectArray obj_array, int count, int start)
    throws org.eyedb.TransactionException, org.eyedb.LoadObjectException;

public void scan(org.eyedb.ObjectArray obj_array, int count, int start,
    org.eyedb.RecMode rcm)
    throws org.eyedb.TransactionException, org.eyedb.LoadObjectException;

```

The `scan` methods which deals with a `org.eyedb.ValueArray` are the most general.

These methods allows us to get back anything: integer, float, string, char, oid.
For instance, if you make a query such as:

```
org.eyedb.OQL q = new org.eyedb.OQL(db, \"list(1, 2, 3, \"hello world\",
3.45, 'c', (select Person.age < 3));
```

the returned values will be:

- 3 integers
- 1 string
- 1 float
- 1 character
- some Person oids

In this case, it seems *raisonnable* to use a `scan` method which deals with a `org.eyedb.ValueArray`, as follows:

```
org.eyedb.ValueArray valarr = new org.eyedb.ValueArray();
q.execute(valarr);

for (int i = 0; i < valarr.getCount(); i++) {
    org.eyedb.Value value = valarr.getValues()[i];
    if (value.type == org.eyedb.Value.INT)
        // ...
    else if (value.type == org.eyedb.Value.FLOAT)
        // ...
    else if (value.type == org.eyedb.Value.OID)
        // ...
    else if (value.type == org.eyedb.Value.CHAR)
        // ...
}
```

In the `TestPC.java` example, as we know that only objects could be returned, a more simple `scan` method may be used, as follows:

```
org.eyedb.OQL q = new org.eyedb.OQL(db, "select Person.name = \"\" + pname + \"\"");

org.eyedb.ObjectArray obj_array = new org.eyedb.ObjectArray();

q.execute(obj_array);

if (obj_array.getCount() == 0) {
    System.err.println("TestPC: cannot find person \" + pname + "\"");
    System.exit(1);
}
```

In the case of no objects has been found, we display an error message and then leaves the program.

In the case of, at least, one object has been found, we cast the `org.eyedb.Object` instance in a `Person` instance, as follows:

```
Person john = (Person)obj_array.getObjects()[0];
```

This cast is a *raisonnable* cast for several reasons:

1. As the query is `"Person.name = \"\" + pname + \"\""`, the expected returned values are `Person` instances.
2. As the database has been opened using an instance of the `person.Database` class, a `Person` instance has been actually constructed using the generated stubs.
If we had used an instance of the generic `org.eyedb.Database` class, this cast would have been illegal, as the `Person` class is unknown from the generic EYEDB package!
3. Contrary to C++, Java performs secure dynamic casting, so if the returned object is not a `Person` instance, an invalid cast exception will be thrown.

Then:

- we look for the `Person` instances whose `age` is less than 3:

```
q = new org.eyedb.QL(db, "select Person.age < 3");  
  
obj_array = new org.eyedb.ObjectArray();  
  
q.execute(obj_array);
```

- one increments their `age`.

```
for (int i = 0; i < obj_array.getCount(); i++) {  
    Person child = (Person)obj_array.getObjects()[i];  
    child.setAge(child.getAge() + 1);  
}
```

- one adds them to the `children` collection of the previous `Person` instance found.

```
    john.addToChildrenColl(child);  
}
```

- one stores all in the database.

```
john.store(org.eyedb.RecMode.FullRekurs);
```

- one commits the transaction.

```
db.transactionCommit();
```