EyeDB Getting Started

Version 2.8.0

January 2006

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

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

Contents

Getting Started

We will introduce EyeDB by going through some simple operations such as creating a database, defining an ODL schema, creating and updating objects, querying objects with OQL, adding indexes and constraints, and then writing simple C++ and Java client programs.

We assume that EyeDB has been correctly installed on your computer. Refer to the installation guide for installation information.

1 Starting the server

In the following sections, we assume that you are running all the EyeDB tools under the same Unix user as the one used when installing EyeDB , in order not to be forced to create a new EyeDB user and give this new user the necessary authorizations to create a database. In case this assumption is not valid, please refer to the administration guide for further information about creating a user and assigning a user database creation permission.

For any EyeDB operation, a server must run on your computer. To check if a server is running, use the following command:

```
% eyedbctl status
```

If a server is running, this command will print a message like:

```
Starting EyeDB Server
 Version      V2.7.5
 Compiled     Jan 28 2006 04:06:17
 Architecture linux-x86-64
 Program Pid  9159
```

If no server is running, it will print an error message such as:

```
No EyeDB Server is running on localhost:6240
```

To start a server, just do this:

```
% eyedbctl start
```

Then, you may try again `eyedbctl status`.

If you get any trouble at this step, refer to the installation and to the administration manuals.

2 Creating a database

The next step is to create a database to perform our tests.

Before creating a database, you can check that you are authorized to perform this operation, using the `eyedbuserlist` command, as in:

```
% eyedbuserlist
name      : "francois" [strict unix user]
sysaccess : SUPERUSER_SYSACCESS_MODE
```

If you are running the `eyedbuserlist` command under the same Unix user as the one used when installing EyeDB , the command output will be a message like the one above, showing that you have `superuser` priviledge and are thus allowed to create a database.

Creating a database is performed using the `eyedbcreate` tool, as in:

```
% eyedbdbcreate foo
```

where `foo` is the name of the database.

Similarly, deleting a database is performed using the `eyedbdelete` tool, as in:

```
% eyedbdbdelete foo
```

where `foo` is the name of the database.

3 Defining a simple schema with ODL

Now that a database has been created, we are going to populate it with objects.

The first step is to define the database schema.

A standard example in databases is the well known `Person` class (or table in relational system) which contains a few attributes such as a firstname, a lastname, an age, an address, a spouse and a set of children.

We will show the inheritance feature through the simple class `Employee` which inherits from the `Person` class and will contains a simple attribute: salary.

Here is simple ODL schema for the classes `Address`, `Person` and `Employee`:

```
//
// person.odl
//

class Address {
  int num;
  string street;
  string town;
  string country;
};

class Person {
  string firstname;
  string lastname;
  int age;
  Address addr;
  Person * spouse inverse Person::spouse;
  set<Person *> children;
};

class Employee extends Person {
  long salary;
};
```

A few comments about this schema:

- the `Address` class contains four attributes, one integer and three strings.
  - **integer**: there are three types of ODL integers:
    1. 16-bits integer, named `int16` or `short`
    2. 32-bits integer, named `int32` or `int`
    3. 64-bits integer, named `int64` or `long`

    so the `num` attribute is a 32-bits integer.
  - **string**: an ODL string is under the form: `string` or `string<N>`. The first form means that the string is not bounded, the second form means that the string contains at most `N` characters.
- the `Person` class contains six attributes: two strings, one 32-bits integer, one `Person` object and one set of `Person` objects.

  - the third attribute `addr` is of `Address` type and is a litteral because there is no `*` before the attribute name. A litteral is an object without identifier: the `addr` attribute is tied to a `Person` instance, it has no proper existence.
  - the `spouse` attribute is an object, not a litteral, because it is preceded by a `*`. An object has an identifier and has its proper existence. The `*` means a reference or pointer to an object. The directive after the attribute name `inverse Person::spouse` is a relationship directive.
  - the `children` attribute is a collection set of `Person` objects.

- the `Employee` contains seven attributes: the six `Person` attributes because `Employee` inherits from `Person` and 64-bits integer attribute: `salary`.

To add the previous schema in the `foo` database, you need to use the `eyedbodl` tool as follows:

```
% eyedbodl -d foo -u person.odl
Updating 'person' schema in database foo...
Adding class Address
Adding class Person
Adding class Employee

Done
```

Note that you must pass the following command line options to the `eyedbodl` command: `-d foo` to specify to which database you are applying the schema and `-u` to update the database schema.

To verify that the update has correctly worked, you can generate the ODL schema from the database, as follows:

```
% eyedbodl -d foo --gencode=ODL

//
// EyeDB Version 2.7.5 Copyright (c) 1995-2005 SYSRA
//
// UNTITLED Schema
//
// Automatically Generated by eyedbodl at Sat Jan 28 04:21:32 2006
//

#if defined(EYEDBNUMVERSION) && EYEDBNUMVERSION != 207005
#error "This file is being compiled with a version of eyedb different from that used to create it (2.7.5)"
#endif

class Address (implementation <hash, hints = "key_count = 2048;">) {
        attribute int32 num;
        attribute string street;
        attribute string town;
        attribute string country;
};

class Person (implementation <hash, hints = "key_count = 2048;">) {
        attribute string firstname;
        attribute string lastname;
        attribute int32 age;
        attribute Address addr;
        relationship Person* spouse inverse Person::spouse;
        attribute set<Person*> children;
};

class Employee (implementation <hash, hints = "key_count = 2048;">) extends Person {
        attribute int64 salary;
};
```

Note that the exact output may differ a bit from what is displayed above, depending on the EyeDB version.

By default, `eyedbodl` generates the ODL on the standard output. You see here that the displayed ODL is very similar to the original ODL except that the keywords `attribute` and `relationship` have been added before each attribute declaration. The `relationship` keyword means that the attribute has an `inverse` directive.

Note that these two keywords are optional: it is why we have not use them in our example.

Another way to check that the schema has been created within the database, is to use the `eyedboql` tool, as follows:

```
% eyedboql -d foo -c "select schema" --print
= bag(2546.2.120579:oid, 2553.2.112046:oid, 2568.2.515951:oid)
struct Address {2546.2.120579:oid} : struct : agregat : instance : object {
        attribute int32 num;
        attribute string street;
        attribute string town;
        attribute string country;
};
struct Person {2553.2.112046:oid} : struct : agregat : instance : object {
        attribute string firstname;
        attribute string lastname;
        attribute int32 age;
        attribute Address addr;
        relationship Person* spouse inverse Person::spouse;
        attribute set<Person*> children;
};
struct Employee {2568.2.515951:oid} : Person : struct : agregat : instance : object {
        attribute string Person::firstname;
        attribute string Person::lastname;
        attribute int32 Person::age;
        attribute Address Person::addr;
        relationship Person* Person::spouse inverse Person::spouse;
        attribute set<Person*> Person::children;
        attribute int64 salary;
};
```

Again, note that the exact output may differ a bit from what is displayed above, depending on the EyeDB version.

Note that the object identifiers (`oid`) of the classes are displayed.

4 Creating and updating objects with the OQL interpreter

Once a schema has been created in the database, we can create and update `Person` and `Employee` instances.

Using the `eyedboql` monitor, we are going to perform the following operations:

1. create a person named “john wayne”
2. create a person named “mary poppins”
3. mary them
4. create 3 “john wayne” children named “baby1”, “baby2” and “baby3”

Here is the way to perform the first three step:

```
% eyedboql -d foo -w
Welcome to eyedboql.
  Type ‘\help’ to display the command list.
  Type ‘\copyright’ to display the copyright.
? john := Person(firstname : "john", lastname : "wayne", age : 72);
= 2585.2.196439:oid
? mary := Person(firstname : "mary", lastname : "poppins", age : 68);
= 2587.2.702511:oid
? john.spouse := mary;
= 2587.2.702511:oid
```

Note the `-w` option on the `eyedboql` command line, specifying that you open the `foo` database in write mode.

A few comments: - `?` is the `eyedboql` prompt: of course, do not type this string! - `:=` is the affectation operator. - each time you create an object, its identifier (`oid`) is displayed on your terminal. - because of the relationship

integrity constraint on the `spouse` attribute, the operation `john.spouse := mary` is equivalent to `mary.spouse := john`.

To create the three "john wayne" children:

```
? add Person(firstname : "baby1", age : 2) to john->children;
= 2589.2.36448:oid
? add Person(firstname : "baby2", age : 3) to john->children;
= 2595.2.683802:oid
? add Person(firstname : "baby3", age : 4) to john->children;
= 2597.2.134950:oid
```

At this stage, it is interesting to perform the following operation: in another terminal, launch another `eyedboql` command on the same database `foo` and query all persons, as follows:

```
% eyedboql -d foo -w -c "select Person;"
= bag()
```

It may seem surprising that no person instance is returned, but in fact it is not: each interaction with the database occurs within a *transaction*, and as long as this transaction has not been *committed*, the database is not modified by the operations that have been done since the beginning of the transaction. To perform effectively these operations, you must *commit* the transaction, by typing in the first `eyedboql` session:

```
? \commit
```

If you now query the person instances in your second `eyedboql` session, the five person instances will be returned:

```
eyedboql -d foo -w -c "select Person;"
= bag(2597.2.134950:oid, 2595.2.683802:oid, 2589.2.36448:oid, 2587.2.702511:oid, 2585.2.196439:oid)
```

You can now quit the first `eyedboql` session with the following command:

```
? \quit
```

5 Querying objects using the OQL interpreter

To query all persons in the database, launch an `eyedboql` session as in:

```
% eyedboql -d foo
Welcome to eyedboql.
  Type '\help' to display the command list.
  Type '\copyright' to display the copyright.
? select Person;
= bag(2597.2.134950:oid, 2595.2.683802:oid, 2589.2.36448:oid, 2587.2.702511:oid, 2585.2.196439:oid)
```

To query all persons whose firstname is "john":

```
? select Person.firstname = "john";
= bag(2585.2.196439:oid)
? \print
Person {2585.2.196439:oid} = {
        firstname = "john";
        lastname = "wayne";
        age = 72;
        addr Address = {
                num = NULL;
                street = NULL;
                town = NULL;
                country = NULL;
        };
        *spouse = {2587.2.702511:oid};
        children set<Person*> = set {
                name = "";
                count = 3;
        };
};
```

Note that the
`print` command allows to display the contains of the last objects returned on your terminal.

To query all persons whose firstname contains a `y`:

```
? select Person.firstname ~ "y";
= bag(2597.2.134950:oid, 2595.2.683802:oid, 2589.2.36448:oid, 2587.2.702511:oid)
```

6 Manipulating objects using OQL

The OQL interpreter can be used to manipulate object, for instance updating the attributes of objects returned by a query.

First, launch an `eyedboql` session as in:

```
% eyedboql -d foo -w
Welcome to eyedboql.
  Type '\help' to display the command list.
  Type '\copyright' to display the copyright.
?
```

The database must be opened in write mode, because we are going to modify the objects stored in the database.

To change the `lastname` attribute of the person whose firstname is `mary`:

```
? (select Person.firstname = "mary").lastname := "stuart";
= bag("stuart")
```

To increment the `age` attribute of all persons, we use a `for` loop to iterate on the result of a query:

```
? select Person.age;
= bag(4, 3, 2, 68, 72)
? for (p in (select Person)) { p.age += 1 ; };
? select Person.age;
= bag(5, 4, 3, 69, 73)
```

7 Updating the database schema

Once created, a database schema can be updated, to add or remove attributes, add or remove classes or schema, add indexes or contraints.

7.1 Adding indexes

To introduce the necessity of indexes, we propose to perform the following operations:

```
? for (x in 1 <= 50000) new Person(firstname : "xx" + string(x));
? select Person.firstname = "xx20";
= bag(23336.2.420154:oid)
? select Person.firstname = "xx10";
= bag(23316.2.824639:oid)
```

The first operation creates 50000 person instances: as you can notice, this operation takes a few seconds. The two last operations query person instance according to their firstname attribute. These operations also take a few seconds to perform and take a significant amount of CPU.

A good idea is to affect an index on the attributes - for instance `firstname`, `lastname` and `age` - for which one wants to perform efficient query.

This is very simple:

- add index specification to the class `Person` in the `person.odl` file as follows:

  ```
  class Person {
    string firstname;
    char lastname;
    int age;
  ```

```
  Address addr;
  ...
  set<Person *> children;

  index on firstname;
  index on lastname;
  index on age;
};
```

- then, use the `eyedbodl` tool to update the database schema:

```
% eyedbodl -d foo -u person.odl
Updating 'person' schema in database foo...
Creating [NULL] hashindex 'index<type = hash, propagate = on> on Person.firstname' on class 'Person'...
Creating [NULL] hashindex 'index<type = hash, propagate = on> on Person.lastname' on class 'Person'...
Creating [NULL] btreeindex 'index<type = btree, propagate = on> on Person.age' on class 'Person'...

Done
```

Now, you can try again to query Person instances according to its firstname, lastname or age:

```
% eyedboql -d foo -w
? select Person.firstname = "xx20";
= bag(23336.2.420154:oid)
? select Person.firstname = "xx10";
= bag(23316.2.824639:oid)
```

and you will notice that these operations are immediate.

7.2 Adding constraints

In the same way, you can add a `notnull` and an `unique` constraint on the `lastname` attribute within the class `Person`:

- add the constraint specification to the class `Person` within the `person.odl` file as follows:

```
class Person {
  string firstname;
  string lastname;
  int age;
  Address addr;
  ...

  index on firstname;
  index on lastname;
  index on age;
  constraint<notnull> on lastname;
  constraint<unique> on lastname;
};
```

- then, use the `eyedbodl` tool to update the database schema:

```
% eyedbodl -d foo -u person.odl
Updating 'person' schema in database foo...
Creating [NULL] notnull_constraint 'constraint<notnull, propagate = on> on Person.lastname' on class 'Person'...
Creating [NULL] unique_constraint 'constraint<unique, propagate = on> on Person.lastname' on class 'Person'...

Done
```

Now try to create two person instances with the same `lastname` attribute:

```
% eyedboql -d foo -w
? new Person(lastname : "curtis");
= 79902.2.884935:oid
? new Person(lastname : "curtis");
near line 2: 'new Person(lastname : "curtis")' => oql error: new operator 'new<oql$db> Person(lastname:"curtis"); ' : u
```

or with no `lastname` attribute:

```
? new Person();
near line 3: 'new Person()' => oql error: new operator 'new<oql$db> Person(); ' : notnull[] constraint error: attribute
```

7.3 Removing classes and schema

It is possible to remove a class in a schema using `eyedbodl`. For instance, to remove the class `Employee` in the already introduced schema:

```
% eyedbodl -d foo -u --rmcls=Employee
Updating 'UNTITLED' schema in database foo...
Removing class Employee

Done
```

You can then check the class removal by:

```
% eyedbodl -d foo --gencode=ODL

//
// EyeDB Version 2.7.5 Copyright (c) 1995-2005 SYSRA
//
// UNTITLED Schema
//
// Automatically Generated by eyedbodl at Fri Jan 27 22:51:26 2006
//

#if defined(EYEDBNUMVERSION) && EYEDBNUMVERSION != 207005
#error "This file is being compiled with a version of eyedb different from that used to create it (2.7.5)"
#endif

class Address (implementation <hash, hints = "key_count = 2048;">) {
        attribute int32 num;
        attribute string street;
        attribute string town;
        attribute string country;
};

class Person (implementation <hash, hints = "key_count = 2048;">) {
        attribute string firstname;
        attribute string lastname;
        attribute int32 age;
        attribute Address addr;
        relationship Person* spouse inverse Person::spouse;
        attribute set<Person*> children;

        index<type = hash, hints = "key_count = 4096; initial_size = 4096; extend_coef = 1; size_max = 4096;", propagat
        index<type = hash, hints = "key_count = 4096; initial_size = 4096; extend_coef = 1; size_max = 4096;", propagat
        constraint<unique, propagate = on> on Person.lastname;
        constraint<notnull, propagate = on> on Person.lastname;
        index<type = btree, hints = "degree = 128;", propagate = on> on Person.age;
};
```

It is as well possible to remove entirely the database schema:

```
% eyedbodl -d foo -u --rmsch
Updating 'UNTITLED' schema in database foo...
Removing [2570.2.500986:oid] hashindex 'index<type = hash, hints = "key_count = 4096; initial_size = 4096; extend_coef
Removing [2585.2.286352:oid] hashindex 'index<type = hash, hints = "key_count = 4096; initial_size = 4096; extend_coef
Removing [2599.2.7912:oid] btreeindex 'index<type = btree, hints = "degree = 128;", propagate = on> on Person.age' from
Removing [2625.2.396262:oid] unique_constraint 'constraint<unique, propagate = on> on Person.lastname' from class 'Pers
Removing [2620.2.240536:oid] notnull_constraint 'constraint<notnull, propagate = on> on Person.lastname' from class 'Pe
Removing class Address
Removing class Person
Removing class set<Person*>

Done
```

The result can be checked with:

```
% eyedbodl -d foo --gencode=ODL

//
// EyeDB Version 2.7.5 Copyright (c) 1995-2005 SYSRA
//
// UNTITLED Schema
//
// Automatically Generated by eyedbodl at Fri Jan 27 22:52:07 2006
//

#if defined(EYEDBNUMVERSION) && EYEDBNUMVERSION != 207005
#error "This file is being compiled with a version of eyedb different from that used to create it (2.7.5)"
#endif
```

8 Using the C++ Binding

We are going to introduce now the C++ binding through the same schema and examples as previously.

There are two ways to use the C++ binding:

1. using the generic C++ binding
2. using both the generic C++ binding and the specific `Person` C++ code generated from the ODL schema

We will explain here only the second way, as it is far more simple and pratical than the first one. For more information on the generic C++ binding, please refer to the C++ binding manual.

Writing a C++ program that can create, retrieve, modify and delete person instances that are stored in an EyeDB database involves the following steps:

1. generates the specific `Person` binding using the `eyedbodl` tool
2. write the C++ client program
3. compile the generated binding and the client program

This example is located in the `examples/GettingStarted` subdirectory.

8.1 Generating the specific C++ binding

To generate the specific C++ binding, run the `eyedbodl` tool as follow:

```
% eyedbodl --gencode=C++ --package=person schema.odl
```

The `--package` option is mandatory: you may give any name you want, this name will be used as the prefix for generated files names. Without the `--package` option, the prefix used will be the name of the ODL file without its extension.

`eyedbodl` generates a few files, all prefixed by `person`, the most important being `person.h` and `person.cc`.

If you have a look to the file `person.h`, you will notice that the following classes have been generated:

1. the class `person`
2. the class `personDatabase`
3. the class `Root`
4. the class `Address`
5. the class `Person`
6. the class `Employee`

The first class, `person`, is the package class:

```
class person {
 public:
  static void init();
  static void release();
  static eyedb::Status updateSchema(eyedb::Database *db);
  static eyedb::Status updateSchema(eyedb::Schema *m);
};
```

it is used to perform package initialization and schema update. Before any use of the `person` package, you need to call `person::init`.

The second class, `personDatabase` is used to open, close and manipulate objects within a database containing the `person` schema.

The `open` method has two purposes: the first one is to open the database, as the standard `eyedb::Database` will do; the second one is to check that the database schema is consistant with the generated runtime schema. Although it is possible to use the standard `Database` class to open a database containing the `person` schema, it is strongly recommended to use the `personDatabase` class.

The third class, `Root`, is the root class for all the generated classes. This class is useful to perform safe down-casting during object loading.

The three last classes, `Address`, `Person` and `Employee` are generated from the `person.odl` class specifications: for each attribute in the `person.odl`, a set of get and set methods is generated.

For instance, for the `firstname` attribute, the following methods are generated:

```
  eyedb::Status setFirstname(const std::string &);
  std::string getFirstname(eyedb::Bool *isnull = 0, eyedb::Status * = 0) const;
  eyedb::Status setFirstname(unsigned int a0, char);
  char getFirstname(unsigned int a0, eyedb::Bool *isnull = 0, eyedb::Status * = 0)  const;
```

The two first methods manipulate the `firstname` attribute as a string while the two last ones manipulate each character within this string.

There are two `set` methods and two `get` methods.

8.2 A minimal client program

We are now going to write a minimal client program which will perform the following operations:

1. initialize the EyeDB package and the `person` package
2. open a connection with the EyeDB server
3. open a database
4. perform error management
5. release the EyeDB package and the `person` package

Here is the code for this minimal client:

```
#include "person.h"

int
main(int argc, char *argv[])
{
  eyedb::init(argc, argv);      // initializes EyeDB package
  person::init();               // initializes person package

  eyedb::Exception::setMode(eyedb::Exception::ExceptionMode); // use exception mode

  try {
    eyedb::Connection conn;

    conn.open();                // opens the connection
```

```
    personDatabase db(argv[1]); // creates a database handle
    db.open(&conn, eyedb::Database::DBRW); // opens the database in read/write mode
  }

  catch(Exception &e) {      // catch any exception and print it
    e.print();
  }

  person::release();            // releases person package
  eyedb::release();             // releases EyeDB package

  return 0;
}
```

Note that statement `Exception::setMode(...)` is mandatory if you want to use the exception error policy.

To use this client, you must first compile it: `eyedbodl` has generated a makefile called Makefile.*package* which can be used as is or can help you to design your own makefile.

A template C++ file (`template_`*package*`.cc`) has also been generated, closed to the previous minimal client program, which can be compiled with the generated makefile.

Here is the generated `Makefile.person` ((`<<datadir>>` is the data directory, usually /usr/share):

```
#
# Makefile.person
#
# person package
#
# Example of template Makefile that can help you to compile
# the generated C++ file and the template program
# Generated by eyedbodl at Sat Jan 28 17:53:48 2006
#

include <<datadir>>/eyedb/Makefile.eyedb

CXXFLAGS += $(EYEDB_CXXFLAGS) $(EYEDB_CPPFLAGS)
LDFLAGS  += ${EYEDB_LDFLAGS}
LDLIBS   += ${EYEDB_LDLIBS}

# if you use gcc
GCC_FLAGS = -Wl,-R$(EYEDB_LIBDIR)

# Example for compiling a client program:

client_program = template_person

$(client_program): person.o $(client_program).o
        $(CXX) $(LDFLAGS) $(GCC_FLAGS) -o $@ $^ $(LDLIBS)
```

Important note: you need a recent version of GNU make to use this makefile. This makefile does not work with the standard SUN make.

Once compiled, you can execute the program as follows:

```
% ./persontest foo
```

We are going now to add a function to manipulate `Person` instances:

1. create a person named “john wayne”
2. create a person named “mary poppins”
3. mary them
4. create 3 “john wayne” children named “baby1”, “baby2” and “baby3”

These operations are performed in the following function:

```
static void
create(eyedb::Database *db)
{
  db->transactionBegin(); // starts a new transaction

  Person *john = new Person(db);
  john->setFirstname("john");
  john->setLastname("wayne");
  john->setAge(32);
  john->getAddr()->setStreet("courcelles");
  john->getAddr()->setTown("Paris");

  Person *mary = new Person(db);
  mary->setFirstname("mary");
  mary->setLastname("poppins");
  mary->setAge(30);
  mary->getAddr()->setStreet("courcelles");
  mary->getAddr()->setTown("Paris");

  // mary them
  john->setSpouse(mary);

  // creates children
  for (int i = 0; i < 5; i++) {
    std::string name = std::string("baby") + str_convert(i+1);
    Person *child = new Person(db);
    child->setFirstname(name.c_str());
    child->setLastname(name.c_str());
    child->setAge(1+i);
    john->addToChildrenColl(child);
    child->release(); // release the allocated pointer
  }

  // store john and all its related instances within the database
  john->store(eyedb::FullRecurs);

  // release the allocated pointers
  mary->release();
  john->release();

  db->transactionCommit(); // commits the current transaction
}
```

A few remarks about this code:

- all operations - setting, getting attributes, storing, querying instances in a database - must be performed within a transaction. A transaction is initiated using the `Database::transactionBegin` method and is committed (resp. aborted) using the `Database::transactionCommit` (resp. `Database::transactionAbort`) method.
- to store any instance in the database, you need to call the emphstore (or `realize`) method on this instance. In our case, we use the argument `FullRecurs` indicating that we want all related instances (through relationship or indirect attribute) to be stored in the database.
- all runtime pointers allocated with the `new` operator must be deleted using the `release` method. The `delete` operator is forbidden: if you try to use it, an exception will be thrown at runtime.

We are now going to query and display all the person instances.

Here is the corresponding code:

```
static void
read(eyedb::Database *db, const char *s)
{
  db->transactionBegin();
```

```
  eyedb::OQL q(db, "select Person.lastname ~ \"%s\"", s);

  eyedb::ObjectArray obj_arr;
  q.execute(obj_arr);

  for (int i = 0; i < obj_arr.getCount(); i++) {
    Person *p = Person_c(obj_arr[i]);
    if (p)
      printf("person = %s %s, age = %d\n", p->getFirstname(),
             p->getLastname(), p->getAge());
  }

  db->transactionCommit();
}
```

An OQL construct can be used within the C++ code using the `OQL(Database *, const char *fmt, ...)` constructor. For instance, in the above example, assuming `s` is equal to `baby`, the code:

```
  eyedb::OQL q(db, "select Person.lastname ~ \"%s\"", s);
```

will send the query `select Person.lastname ~ "baby"` to the OQL interpreter.

This interpreter will perform the query and returned all the found objects. The returned objects can be found using the `OQL::execute` method as follows:

```
  eyedb::ObjectArray obj_arr;
  q.execute(obj_arr);
```

The returned objects are of type `eyedb::Object`, so you cannot use the `Person` methods such as `getFirstname()`, `getAge()`... To use them, you need to perform a down-cast using the `Person_c` static function as follows:

```
  for (int i = 0; i < obj_arr.getCount(); i++) {
    Person *p = Person_c(obj_arr[i]);
    if (p) ...
```

If the object `obj_arr[i]` is not of type `Person`, the returned pointer will be null. It is why we make a test on the value of `p`. If `p` is not null, we can use all the `Person` methods as follows:

```
    printf("person = %s %s, age = %d\n", p->getFirstname(),
           p->getLastname(), p->getAge());
```

To have more information about the C++ binding, please refer to the EyeDB C++ binding manual.

9 Using the Java Binding

Although the C++ binding is more complete than the Java binding - essentially according to the administrative operations - the Java bindings allow to manipulate data without limitations.

Using the Java binding is very similar to the C++ binding. Writing a Java program that can create, retrieve, modify and delete person instances that are stored in an EyeDB database involves the following steps:

1. generates the specific `Person` binding using the `eyedbodl` tool
2. write the Java client program
3. compile the generated binding and the client program

This example is located in the `examples/GettingStarted` subdirectory.

9.1 Generating the Java code

The Java code is generated from the ODL schema definition using the following command:

```
% eyedbodl --gencode=Java --package=person person.odl
```

The `--package` option is mandatory: this name will be used as the name of the Java package to which all generated Java classes will belong.

This command will generate a number of Java file in subdirectory `person/`, each generated file containing a Java class having the same name.

If you have a look to the files in sub-directory `person`, you will notice that the following classes have been generated:

1. the class `Address`
2. the class `Database`
3. the class `Employee`
4. the class `Person`
5. the class `set_class_Person_ref`

9.2 A minimal client program

We are now going to write a minimal client program which will perform the following operations:

1. initialize the EyeDB and `person` packages
2. connect to the EyeDB server
3. open a database
4. creates two person instances and mary them

Here is the code for this minimal client:

```
//
// Persontest.java
//

import person.*;

class PersonTest {
  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("PersonTest", args);

    // Check that a database name is given on the command line
    int argc = outargs.length;
    if (argc != 1) {
        System.err.println("usage: java PersonTest 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.setFirstname("john");
      john.setLastname("travolta");
```

```
      john.setAge(26);

      Person mary = new Person(db);
      mary.setFirstname("mary");
      mary.setLastname("stuart");
      mary.setAge(22);

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

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

      john.trace();

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

To use this client, you must first compile it using a standard Makefile, as follows (replace `<<datadir>>` with the data directory, usually /usr/share):

```
include <<datadir>>/eyedb/Makefile.eyedb

all: PersonTest.class

person/Database.java: schema.odl
        $(EYEDB_ODL) --gencode=Java --package=person --output-dir=person $<

PersonTest.class: PersonTest.java person/Database.java
        CLASSPATH=$(EYEDB_CLASSPATH):. javac *.java person/*.java
```

Once compiled, you can execute the program as follows:

```
% CLASSPATH=. eyedbjrun PersonTest person_g
```

The `eyedbjrun` script is a helper script that wraps the call to the Java virtual machine with an appropriate CLASSPATH environment variable containing the path to `eyedb.jar` and passes the necessary options to the `PersonTest` class.

A few remarks about the Java code:

- all operations - setting, getting attributes, storing, querying instances in a database - must be performed within a transaction. A transaction is initiated using the `Database::transactionBegin` method and is committed (resp. aborted) using the `Database::transactionCommit` (resp. `Database::transactionAbort`) method.
- to store any instance in the database, you need to call the emphstore (or `realize`) method on this instance. In our case, we use the argument `FullRecurs` indicating that we want all related instances (through relationship or indirect attribute) to be stored in the database.

The Java binding support both the standalone applications and the applets.

To have more information about the Java binding, please refer to the EyeDB Java binding manual.

10 Learning more about EyeDB

We have briefly introduce in this manual some of the main features of EyeDB.

More detailled information can be found in the other parts of the EyeDB manual:

- Object Definition Language (ODL) manual

- Object Query Language (OQL) manual
- C++ Binding manual
- Java Binding manual