## Welcome

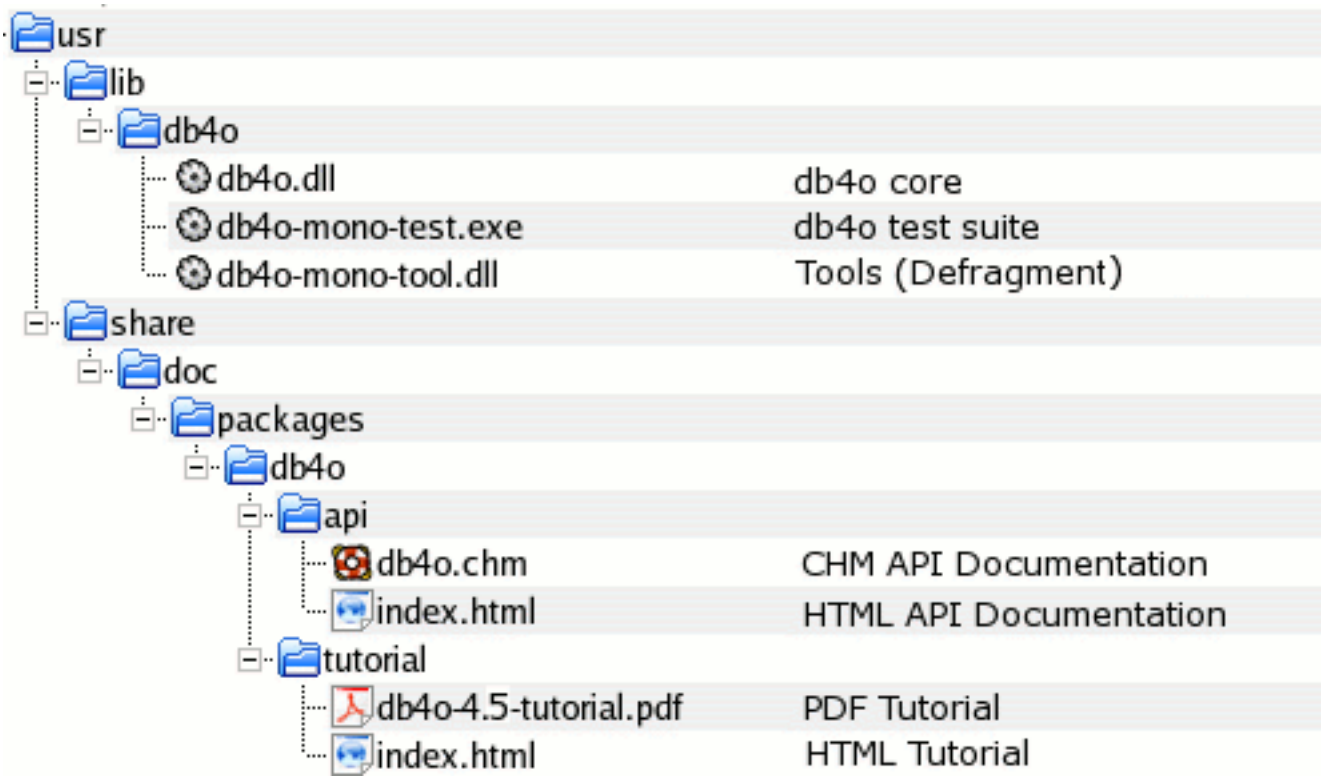
**db4o is the native Java, .NET and Mono open source object database.**

This documentation and tutorial is intended to get you started with db4o and to be a reliable companion while you develop with db4o. Before you start, please make sure that you have downloaded the latest db4o distribution from the db4objects website.

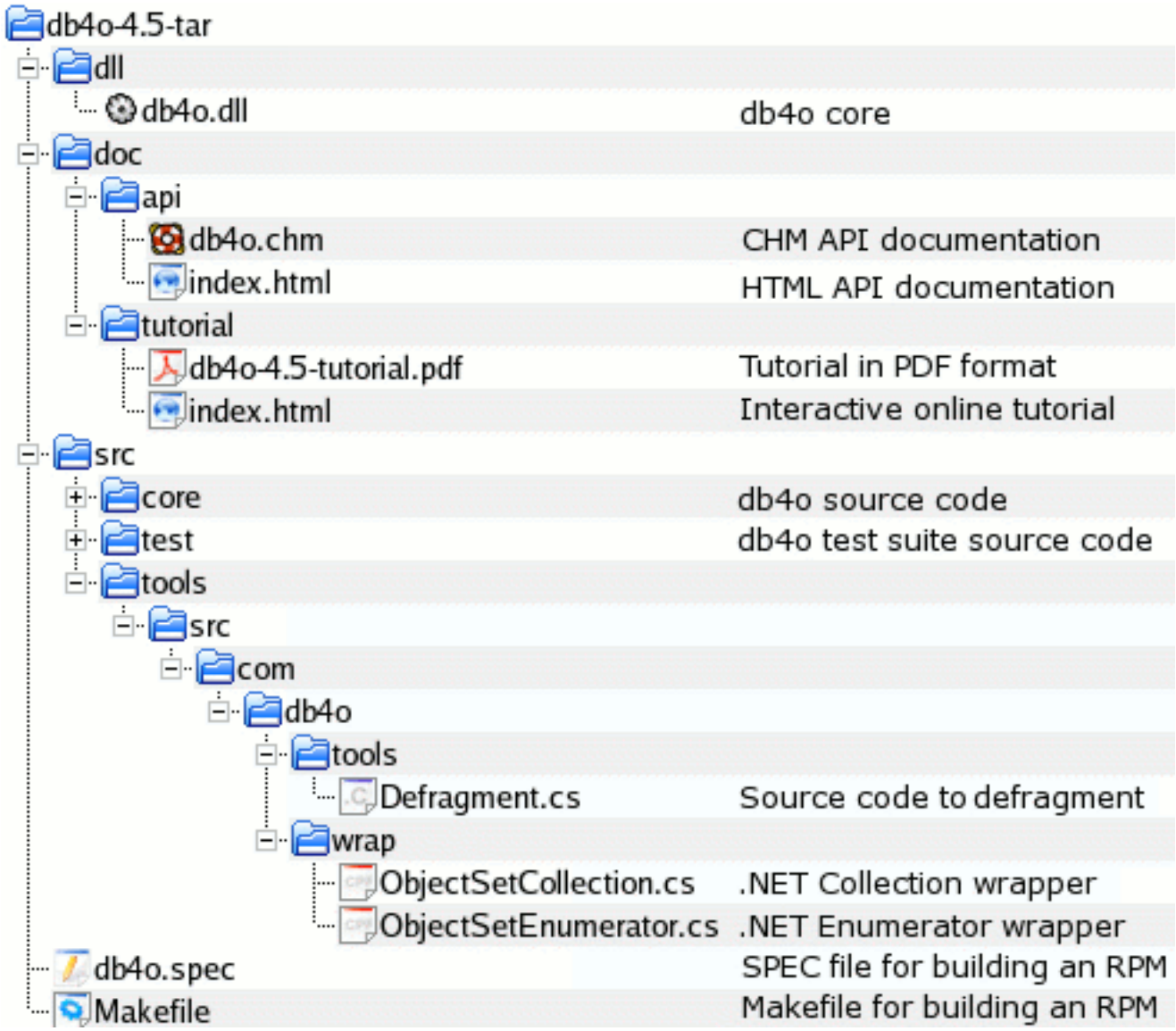
You are invited to join the db4o community in the public db4o newsgroup at news://news.db4odev.com/db4o.users and to ask for help at any time. A searchable archive of previous postings is available here . You may also find the db4o knowledgebase helpful for keyword searches.

The db4o Mono distribution is available from the db4o download center in 3 versions: *db4o-4.5-mono.tar.gz* (binary and source tarball), *db4o-4.5-mono.noarch.rpm* (binary RPRM), *db4o-4.5-mono.src.rpm* (source RPM). The noarch RPM can be built from the source RPM. After installing/unzipping you will find the following directory structure on your machine:

**db4o-4.5-mono.noarch.rpm**



db4o-4.5-mono.tar.gz




This tutorial comes in multiple versions. Make sure that you use the right one for the right purpose.

/usr/share/doc/packages/db4o/tutorial/index.html

The tutorial in HTML format. The Java and .NET db4o distributions also provide the HTML documentation with live execution capabilities.

/usr/share/doc/packages/db4o/tutorial/db4o-4.x-tutorial.pdf

The PDF version of the tutorial allows best fulltext search capabilities.

Java, .NET and Mono

db4o is available for Java, for .NET and for Mono. This tutorial was written for Mono. The structure of

the other distributions may be considerably different, so please use the tutorial for the version that you plan to experiment with first.

# 1. First Glance

Before diving straight into the first source code samples let's get you familiar with some basics.

## 1.1. The db4o engine...

The db4o object database engine consists of one single DLL. This is all that you need to program against. The version supplied with the distribution can be found in /usr/lib/db4o/.☐

## 1.2. Installation

To use db4o in a development project, you only need to add one of the above db4o.dll files to your project references.

Here is how to do this if you are using MonoDevelop:
- Right-click on "References" in the Solution Tab
- choose "Edit References"
- select the ".NET Assembly" tab and then "Browse"
- select /usr/lib/db4o/db4o.dll
- click "Open"
- click "OK"

## 1.3. Object Manager

### 1.3.1. Installation

The db4o Object Manager, a GUI tool to browse and query database files, is packaged separately for each supported platform. Choose from the following links in the db4o Download Center for your platform:

- db4o ObjectManager for Windows IKVM (Java VM included)
- db4o ObjectManager for Windows no Java VM
- db4o ObjectManager for Linux

Once you have downloaded the appropriate Object Manager build, create a folder called Object Manager in an appropriate location and unpack the downloaded zip file there.

### 1.3.2. Running

#### 1.3.2.1. Windows IKVM

Object Manager for Windows IKVM includes the open-source IKVM Java virtual machine in the download.  Simply double-click the objectmanager.bat file to start Object Manager.

#### 1.3.2.2. Windows no Java VM

This build assumes that your computer already has a Sun Java Virtual Machine version 1.3 or later installed and that your operating system path already lists the directory containing your java.exe file. If this is true, you can simply double-click the objectmanager.bat file to start Object Manager. Otherwise, you will need to edit objectmanager.bat and specify the full path and file name for your java.exe file on the first line.

#### 1.3.2.3. Linux

This build assumes that your computer already has a Sun Java Virtual Machine version 1.3 or later installed and that your PATH variable already lists the directory containing the java binary.  If this is not the case, you will
need to edit the objectmanager.sh file and specify the full path and file name of the Java binary on the "export VMEXE" line".  Since the zip archive does not preserve the executable permissions for objectmanager.sh, you will need
to `chmod +x objectmanager.sh`.  Once this is complete, running objectmanager.sh will start Object Manager.

### 1.4. API

The API documentation for db4o is supplied as a set of HTML pages in /usr/share/doc/packages/db4o/api/index.html. While you read through this tutorial, it may be helpful to look into the API documentation occasionaly. For the start, the namespaces com.db4o and com.db4o.query are all that you need to worry about.

Let's take a first brief look at one of the most important interfaces:

```
com.db4o.ObjectContainer
```

This will be your view of a db4o database:
- An ObjectContainer can either be a database in single-user mode or a client to a db4o server.
- Every ObjectContainer owns one transaction. All work is transactional. When you open an ObjectContainer, you are in a transaction, when you commit() or rollback(), the next transaction is started immediately.
- Every ObjectContainer maintains it's own references to stored and instantiated objects. In doing so, it manages object identities.

In case you wonder why you only see very few methods in an ObjectContainer, here is why: The db4o interface is supplied in two steps in two namespaces, com.db4o and com.db4o.ext for the following reasons:
- It's easier to get started, because the important methods are emphasized.
- It will be easier for other products to copy the basic db4o interface.
- We hint how a very-light-version of db4o should look like.

Every com.db4o.ObjectContainer object also always is a com.db4o.ext.ExtObjectContainer. You can cast to ExtObjectContainer or you can call the #ext() method if you want to use advanced features.

## 2. First Steps

Let us get started as simple as possible. We are going to learn how to store, retrieve, update and delete instances of a single class that only contains primitive and String members. In our example this will be a Formula One (F1) pilot whose attributes are his name and the F1 points he has already gained this season.

First we create a native class such as:

```
namespace com.db4o.f1.chapter1
{
    public class Pilot
    {
        string _name;
        int _points;

        public Pilot(string name, int points)
        {
            _name = name;
            _points = points;
        }

        public string Name
        {
            get
            {
                return _name;
            }
        }

        public int Points
        {
            get
            {
                return _points;
            }
        }
```

```
        public void AddPoints(int points)
        {
            _points += points;
        }

        override public string ToString()
        {
            return _name + "/" + _points;
        }
    }
}
```

Note that this class does not contain any db4o related code.

### 2.1. Storing objects

To access a db4o database file or create a new one, call Db4o.openFile(), providing the path to your file as the parameter, to obtain an ObjectContainer instance. ObjectContainer will be your primary interface to db4o. Closing the container will release all resources associated with it.

```
[accessDb4o]

ObjectContainer db=Db4o.openFile(Util.YapFileName);
    try
    {
        // do something with db4o
    }
    finally
    {
        db.close();
    }
```

For the following examples we will assume that our environment takes care of opening and closing the ObjectContainer automagically.

To store an object, we simply call set() on our database, passing the object as a parameter.

```
[storeFirstPilot]

Pilot pilot1 = new Pilot("Michael Schumacher", 100);
    db.set(pilot1);
    Console.WriteLine("Stored " + pilot1);
```

**OUTPUT:**

```
Stored Michael Schumacher/100
```

We'll need a second pilot, too.

```
[storeSecondPilot]

Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
    db.set(pilot2);
    Console.WriteLine("Stored " + pilot2);
```

**OUTPUT:**

```
Stored Rubens Barrichello/99
```

### 2.2. Retrieving objects

To query the database for our pilot, we shall use *Query by Example* (QBE) for now. This means we will create a prototypical object for db4o to use as an example. db4o will retrieve all objects of the given type that contain the same (non-default) field values as the candidate. The result will be handed as an ObjectSet instance. We will use a convenience method 'listResult', inherited by all our main example classes, to display a result's content and reset it for further use:

```
public static void listResult(ObjectSet result)
    {
        Console.WriteLine(result.size());
        while (result.hasNext())
        {
            Console.WriteLine(result.next());
        }
    }
```

To retrieve all pilots from our database, we provide an 'empty' prototype:

```
[retrieveAllPilots]

Pilot proto = new Pilot(null, 0);
    ObjectSet result = db.get(proto);
    listResult(result);
```

```
OUTPUT:
2
Michael Schumacher/100
Rubens Barrichello/99
```

Note that our results are not constrained to have 0 points, as 0 is the default value for int fields.

To query for a pilot by name:

```
[retrievePilotByName]

Pilot proto = new Pilot("Michael Schumacher", 0);
    ObjectSet result = db.get(proto);
    listResult(result);
```

```
OUTPUT:
1
Michael Schumacher/100
```

Let's retrieve a pilot by exact points:

```
[retrievePilotByExactPoints]

Pilot proto = new Pilot(null, 100);
    ObjectSet result = db.get(proto);
    listResult(result);
```

```
OUTPUT:
1
Michael Schumacher/100
```

Of course there's much more to db4o queries. We'll come to that in a moment.

### 2.3. Updating objects

To update an object already stored in db4o, just call set() again after modifying it.

```
[updatePilot]

ObjectSet result = db.get(new Pilot("Michael Schumacher", 0));
    Pilot found = (Pilot)result.next();
    found.AddPoints(11);
    db.set(found);
    Console.WriteLine("Added 11 points for " + found);
    retrieveAllPilots(db);
```

```
OUTPUT:
Added 11 points for Michael Schumacher/111
2
Michael Schumacher/111
Rubens Barrichello/99
```

Note that it is necessary that db4o already 'knows' this pilot, else it will store it as a new object. 'Knowing' an object basically means having it set or retrieved during the current db4o session. We'll explain this later in more detail.

To make sure you've updated the pilot, please return to any of the retrieval examples above and run them again.

### 2.4. Deleting objects

Objects are removed from the database using the delete() method.

```
[deleteFirstPilotByName]

ObjectSet result = db.get(new Pilot("Michael Schumacher", 0));
    Pilot found = (Pilot)result.next();
    db.delete(found);
    Console.WriteLine("Deleted " + found);
    retrieveAllPilots(db);
```

```
OUTPUT:
Deleted Michael Schumacher/111
1
Rubens Barrichello/99
```

Let's delete the other one, too.

```
[deleteSecondPilotByName]

ObjectSet result = db.get(new Pilot("Rubens Barrichello", 0));
    Pilot found = (Pilot)result.next();
    db.delete(found);
    Console.WriteLine("Deleted " + found);
    retrieveAllPilots(db);
```

**OUTPUT:**

```
Deleted Rubens Barrichello/99
0
```

Please check the deletion with the retrieval examples above.

Again, the object to be deleted has to be known to db4o. It is not sufficient to provide a prototype object with the same field values.

### 2.5. Conclusion

That was easy, wasn't it? We have stored, retrieved, updated and deleted objects with a few lines of code. But what about complex queries? Let's have a look at the restrictions of QBE and alternative approaches in the next chapter .

### 2.6. Full source

```
namespace com.db4o.f1.chapter1
{
    using System;
    using System.IO;
    using com.db4o;
    using com.db4o.f1;

    public class FirstStepsExample : Util
    {
        public static void Main(string[] args)
```

```
    {
        File.Delete(Util.YapFileName);
        accessDb4o();
        File.Delete(Util.YapFileName);
        ObjectContainer db = Db4o.openFile(Util.YapFileName);
        try
        {
            storeFirstPilot(db);
            storeSecondPilot(db);
            retrieveAllPilots(db);
            retrievePilotByName(db);
            retrievePilotByExactPoints(db);
            updatePilot(db);
            deleteFirstPilotByName(db);
            deleteSecondPilotByName(db);
        }
        finally
        {
            db.close();
        }
    }

    public static void accessDb4o()
    {
     ObjectContainer db=Db4o.openFile(Util.YapFileName);
     try
     {
         // do something with db4o
     }
     finally
     {
         db.close();
     }
}

    public static void storeFirstPilot(ObjectContainer db)
    {
        Pilot pilot1 = new Pilot("Michael Schumacher", 100);
        db.set(pilot1);
        Console.WriteLine("Stored " + pilot1);
    }
```

```
        public static void storeSecondPilot(ObjectContainer db)
        {
            Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
            db.set(pilot2);
            Console.WriteLine("Stored " + pilot2);
        }

        public static void retrieveAllPilots(ObjectContainer db)
        {
            Pilot proto = new Pilot(null, 0);
            ObjectSet result = db.get(proto);
            listResult(result);
        }

        public static void retrievePilotByName(ObjectContainer db)
        {
            Pilot proto = new Pilot("Michael Schumacher", 0);
            ObjectSet result = db.get(proto);
            listResult(result);
        }

        public static void retrievePilotByExactPoints(ObjectContainer
db)
        {
            Pilot proto = new Pilot(null, 100);
            ObjectSet result = db.get(proto);
            listResult(result);
        }

        public static void updatePilot(ObjectContainer db)
        {
            ObjectSet result = db.get(new Pilot("Michael Schumacher",
0));
            Pilot found = (Pilot)result.next();
            found.AddPoints(11);
            db.set(found);
            Console.WriteLine("Added 11 points for " + found);
            retrieveAllPilots(db);
        }
```

```
        public static void deleteFirstPilotByName(ObjectContainer db)
        {
            ObjectSet result = db.get(new Pilot("Michael Schumacher",
0));
            Pilot found = (Pilot)result.next();
            db.delete(found);
            Console.WriteLine("Deleted " + found);
            retrieveAllPilots(db);
        }

        public static void deleteSecondPilotByName(ObjectContainer
db)
        {
            ObjectSet result = db.get(new Pilot("Rubens Barrichello",
0));
            Pilot found = (Pilot)result.next();
            db.delete(found);
            Console.WriteLine("Deleted " + found);
            retrieveAllPilots(db);
        }
    }
}
```

## 3. Query API

We have already seen how to retrieve objects from db4o via QBE. While this approach is easy and intuitive, there are situations where it is not sufficient.

- There are queries that simply cannot be expressed with QBE: Retrieve all pilots with more than 100 points, for example.
- Creating a prototype object may have unwanted side effects.
- Default values (e.g. null) may not be accepted by the domain class constructor.
- We may want to query for field default values.

db4o provides a dedicated query API that can be used in those cases.

We need some pilots in our database again to explore it.

```
[storeFirstPilot]

Pilot pilot1 = new Pilot("Michael Schumacher", 100);
    db.set(pilot1);
    Console.WriteLine("Stored " + pilot1);
```

**OUTPUT:**

```
Stored Michael Schumacher/100
```

```
[storeSecondPilot]

Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
    db.set(pilot2);
    Console.WriteLine("Stored " + pilot2);
```

**OUTPUT:**

```
Stored Rubens Barrichello/99
```

### 3.1. Simple queries

First, let's see how our familiar QBE queries are expressed within the query API. This is done by retrieving a 'fresh' Query object from the ObjectContainer and adding Constraint instances to it. To find all Pilot instances, we constrain the query with the Pilot class object.

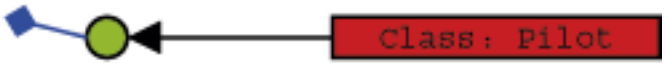
```
[retrieveAllPilots]

Query query = db.query();
    query.constrain(typeof(Pilot));
    ObjectSet result = query.execute();
    listResult(result);
```

**OUTPUT:**

```
2
Michael Schumacher/100
Rubens Barrichello/99
```

Basically, we're exchanging our 'real' prototype for a meta description of the objects we'd like to hunt down: a **query graph** made up of query nodes and constraints. A query node is a placeholder for a candidate object, a constraint decides whether to add or exclude candidates from the result.

Our first simple graph looks like this.




We're just asking any candidate object (here: any object in the database) to be of type Pilot to aggregate our result.

To retrieve a pilot by name, we have to further constrain the candidate pilots by descending to their name field and constraining this with the respective candidate String.

```
[retrievePilotByName]

Query query = db.query();
    query.constrain(typeof(Pilot));
    query.descend("_name").constrain("Michael Schumacher");
    ObjectSet result = query.execute();
    listResult(result);
```

**OUTPUT:**

```
1
Michael Schumacher/100
```

What does 'descend' mean here? Well, just as we did in our 'real' prototypes, we can attach constraints to child members of our candidates.


So a candidate needs to be of type Pilot and have a member named 'name' that is equal to the given String to be accepted for the result.

Note that the class constraint is not required: If we left it out, we would query for all objects that contain a 'name' member with the given value. In most cases this will not be the desired behavior, though.

Finding a pilot by exact points is analogous, we just have to cross the Java primitive/object divide.

```
[retrievePilotByExactPoints]

Query query = db.query();
```

```
    query.constrain(typeof(Pilot));
    query.descend("_points").constrain(100);
    ObjectSet result = query.execute();
    listResult(result);
```

```
OUTPUT:
1
Michael Schumacher/100
```

### 3.2. Advanced queries

Now there are occasions when we don't want to query for exact field values, but rather for value ranges, objects not containing given member values, etc. This functionality is provided by the Constraint API.

First, let's negate a query to find all pilots who are not Michael Schumacher:

```
[retrieveByNegation]

Query query = db.query();
    query.constrain(typeof(Pilot));
    query.descend("_name").constrain("Michael Schumacher").not();
    ObjectSet result = query.execute();
    listResult(result);
```

```
OUTPUT:
1
Rubens Barrichello/99
```

Where there is negation, the other boolean operators can't be too far.

```
[retrieveByConjunction]

Query query = db.query();
    query.constrain(typeof(Pilot));
    Constraint constr = query.descend("_name")
            .constrain("Michael Schumacher");
    query.descend("_points")
            .constrain(99).and(constr);
    ObjectSet result = query.execute();
    listResult(result);
```

**OUTPUT:**

```
0
```

```
[retrieveByDisjunction]

Query query = db.query();
    query.constrain(typeof(Pilot));
    Constraint constr = query.descend("_name")
            .constrain("Michael Schumacher");
    query.descend("_points")
            .constrain(99).or(constr);
    ObjectSet result = query.execute();
    listResult(result);
```

**OUTPUT:**

```
2
Michael Schumacher/100
Rubens Barrichello/99
```

We can also constrain to a comparison with a given value.

```
[retrieveByComparison]

Query query = db.query();
    query.constrain(typeof(Pilot));
    query.descend("_points")
            .constrain(99).greater();
    ObjectSet result = query.execute();
    listResult(result);
```

```
OUTPUT:
1
Michael Schumacher/100
```

The query API also allows to query for field default values.

```
[retrieveByDefaultFieldValue]

Pilot somebody = new Pilot("Somebody else", 0);
    db.set(somebody);
    Query query = db.query();
    query.constrain(typeof(Pilot));
    query.descend("_points").constrain(0);
    ObjectSet result = query.execute();
    listResult(result);
    db.delete(somebody);
```

```
OUTPUT:
1
Somebody else/0
```

It is also possible to have db4o sort the results.

```
[retrieveSorted]

Query query = db.query();
    query.constrain(typeof(Pilot));
    query.descend("_name").orderAscending();
    ObjectSet result = query.execute();
    listResult(result);
    query.descend("_name").orderDescending();
    result = query.execute();
    listResult(result);
```

```
OUTPUT:
2
Michael Schumacher/100
Rubens Barrichello/99
2
Rubens Barrichello/99
Michael Schumacher/100
```

All these techniques can be combined arbitrarily, of course. Please try it out.

To prepare for the next chapter, let's clear the database.

```
[clearDatabase]

ObjectSet result = db.get(new Pilot(null, 0));
    while (result.hasNext())
    {
        db.delete(result.next());
    }
```

```
OUTPUT:
```

### 3.3. Conclusion

Now we know how to build arbitrarily complex queries. But our domain model is not complex at all, consisting of one class only. Let's have a look at the way db4o handles object associations in the next chapter .

### 3.4. Full source

```
namespace com.db4o.f1.chapter1
{
    using System;
    using com.db4o;
    using com.db4o.query;
    using com.db4o.f1;

    public class QueryExample : Util
    {
        public static void Main(string[] args)
        {
            ObjectContainer db = Db4o.openFile(Util.YapFileName);
            try
            {
                storeFirstPilot(db);
                storeSecondPilot(db);
                retrieveAllPilots(db);
                retrievePilotByName(db);
                retrievePilotByExactPoints(db);
                retrieveByNegation(db);
                retrieveByConjunction(db);
                retrieveByDisjunction(db);
                retrieveByComparison(db);
                retrieveByDefaultFieldValue(db);
                retrieveSorted(db);
                clearDatabase(db);
            }
            finally
            {
                db.close();
```

```
            }
        }

        public static void storeFirstPilot(ObjectContainer db)
        {
            Pilot pilot1 = new Pilot("Michael Schumacher", 100);
            db.set(pilot1);
            Console.WriteLine("Stored " + pilot1);
        }

        public static void storeSecondPilot(ObjectContainer db)
        {
            Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
            db.set(pilot2);
            Console.WriteLine("Stored " + pilot2);
        }

        public static void retrieveAllPilots(ObjectContainer db)
        {
            Query query = db.query();
            query.constrain(typeof(Pilot));
            ObjectSet result = query.execute();
            listResult(result);
        }

        public static void retrievePilotByName(ObjectContainer db)
        {
            Query query = db.query();
            query.constrain(typeof(Pilot));
            query.descend("_name").constrain("Michael Schumacher");
            ObjectSet result = query.execute();
            listResult(result);
        }

        public static void retrievePilotByExactPoints(ObjectContainer
db)
        {
            Query query = db.query();
            query.constrain(typeof(Pilot));
            query.descend("_points").constrain(100);
            ObjectSet result = query.execute();
```

```
            listResult(result);
        }

        public static void retrieveByNegation(ObjectContainer db)
        {
            Query query = db.query();
            query.constrain(typeof(Pilot));
            query.descend("_name").constrain("Michael
Schumacher").not();
            ObjectSet result = query.execute();
            listResult(result);
        }

        public static void retrieveByConjunction(ObjectContainer db)
        {
            Query query = db.query();
            query.constrain(typeof(Pilot));
            Constraint constr = query.descend("_name")
                    .constrain("Michael Schumacher");
            query.descend("_points")
                    .constrain(99).and(constr);
            ObjectSet result = query.execute();
            listResult(result);
        }

        public static void retrieveByDisjunction(ObjectContainer db)
        {
            Query query = db.query();
            query.constrain(typeof(Pilot));
            Constraint constr = query.descend("_name")
                    .constrain("Michael Schumacher");
            query.descend("_points")
                    .constrain(99).or(constr);
            ObjectSet result = query.execute();
            listResult(result);
        }

        public static void retrieveByComparison(ObjectContainer db)
        {
            Query query = db.query();
            query.constrain(typeof(Pilot));
```

```
            query.descend("_points")
                    .constrain(99).greater();
            ObjectSet result = query.execute();
            listResult(result);
        }

        public static void
retrieveByDefaultFieldValue(ObjectContainer db)
        {
            Pilot somebody = new Pilot("Somebody else", 0);
            db.set(somebody);
            Query query = db.query();
            query.constrain(typeof(Pilot));
            query.descend("_points").constrain(0);
            ObjectSet result = query.execute();
            listResult(result);
            db.delete(somebody);
        }

        public static void retrieveSorted(ObjectContainer db)
        {
            Query query = db.query();
            query.constrain(typeof(Pilot));
            query.descend("_name").orderAscending();
            ObjectSet result = query.execute();
            listResult(result);
            query.descend("_name").orderDescending();
            result = query.execute();
            listResult(result);
        }

        public static void clearDatabase(ObjectContainer db)
        {
            ObjectSet result = db.get(new Pilot(null, 0));
            while (result.hasNext())
            {
                db.delete(result.next());
            }
        }
    }
}
```

## 4. Structured objects

It's time to extend our business domain with another class and see how db4o handles object interrelations. Let's give our pilot a vehicle.

```
namespace com.db4o.f1.chapter2
{
    public class Car
    {
        string _model;
        Pilot _pilot;

        public Car(string model)
        {
            _model = model;
            _pilot = null;
        }

        public Pilot Pilot
        {
            get
            {
                return _pilot;
            }

            set
            {
                _pilot = value;
            }
        }

        public string Model
        {
            get
            {
                return _model;
            }
        }
```

```
        override public string ToString()
        {
            return _model + "[" + _pilot + "]";
        }
    }
}
```

### 4.1. Storing structured objects

To store a car with its pilot, we just call set() on our top level object, the car. The pilot will be stored implicitly.

```
[storeFirstCar]

Car car1 = new Car("Ferrari");
    Pilot pilot1 = new Pilot("Michael Schumacher", 100);
    car1.Pilot = pilot1;
    db.set(car1);
```

Of course, we need some competition here. This time we explicitly store the pilot before entering the car - this makes no difference.

```
[storeSecondCar]

Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
    db.set(pilot2);
    Car car2 = new Car("BMW");
    car2.Pilot = pilot2;
    db.set(car2);
```

### 4.2. Retrieving structured objects

### 4.2.1. QBE

To retrieve all cars, we simply provide a 'blank' prototype.

```
[retrieveAllCarsQBE]

Car proto = new Car(null);
    ObjectSet result = db.get(proto);
    listResult(result);
```

```
OUTPUT:
2
BMW[Rubens Barrichello/99]
Ferrari[Michael Schumacher/100]
```

We can also query for all pilots, of course.

```
[retrieveAllPilotsQBE]

Pilot proto = new Pilot(null, 0);
    ObjectSet result = db.get(proto);
    listResult(result);
```

```
OUTPUT:
2
Rubens Barrichello/99
Michael Schumacher/100
```

Now let's initialize our prototype to specify all cars driven by Rubens Barrichello.

```
[retrieveCarByPilotQBE]

Pilot pilotproto = new Pilot("Rubens Barrichello",0);
    Car carproto = new Car(null);
    carproto.Pilot = pilotproto;
    ObjectSet result = db.get(carproto);
    listResult(result);
```

```
OUTPUT:
1
BMW[Rubens Barrichello/99]
```

What about retrieving a pilot by car? We simply don't need that - if we already know the car, we can simply ask it for its pilot directly.

### 4.2.2. Query API

To query for a car given its pilot's name we have to descend one level deeper in our query.

```
[retrieveCarByPilotNameQuery]

Query query = db.query();
    query.constrain(typeof(Car));
    query.descend("_pilot").descend("_name")
        .constrain("Rubens Barrichello");
    ObjectSet result = query.execute();
    listResult(result);
```

```
OUTPUT:
1
BMW[Rubens Barrichello/99]
```

We can also constrain the pilot field with a prototype to achieve the same result.

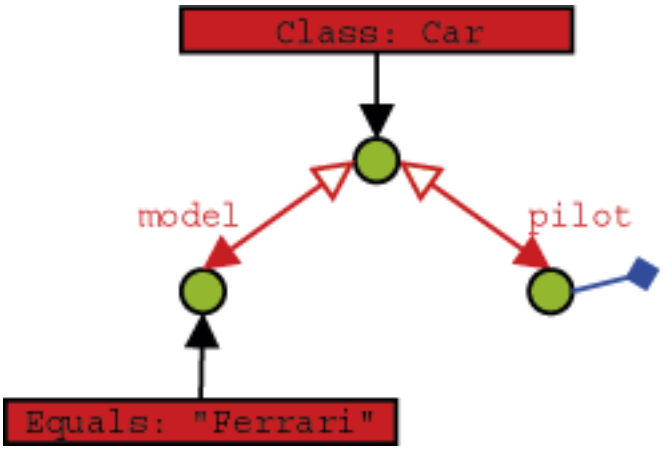
```
[retrieveCarByPilotProtoQuery]

Query query = db.query();
    query.constrain(typeof(Car));
    Pilot proto = new Pilot("Rubens Barrichello", 0);
    query.descend("_pilot").constrain(proto);
    ObjectSet result = query.execute();
    listResult(result);
```

```
OUTPUT:
1
BMW[Rubens Barrichello/99]
```

We have seen that descending into a query provides us with another query. You may also have noticed that the associations between query nodes look somehow bidirectional in our diagrams. Let's move upstream.

```
[retrievePilotByCarModelQuery]

Query carquery=db.query();
    carquery.constrain(typeof(Car));
    carquery.descend("_model").constrain("Ferrari");
    Query pilotquery=carquery.descend("_pilot");
    ObjectSet result=pilotquery.execute();
    listResult(result);
```

```
OUTPUT:
1
Michael Schumacher/100
```




### 4.3. Updating structured objects

To update structured objects in db4o, we simply call set() on them again.

```
[updateCar]

ObjectSet result = db.get(new Car("Ferrari"));
    Car found = (Car)result.next();
    found.Pilot = new Pilot("Somebody else", 0);
    db.set(found);
    result = db.get(new Car("Ferrari"));
    listResult(result);
```

**OUTPUT:**

```
1
Ferrari[Somebody else/0]
```

Let's modify the pilot, too.

```
[updatePilotSingleSession]

ObjectSet result = db.get(new Car("Ferrari"));
```

```
    Car found = (Car)result.next();
    found.Pilot.AddPoints(1);
    db.set(found);
    result = db.get(new Car("Ferrari"));
    listResult(result);
```

```
OUTPUT:
1
Ferrari[Somebody else/1]
```

Nice and easy, isn't it? But wait, there's something evil lurking right behind the corner. Let's see what happens if we split this task in two separate db4o sessions: In the first we modify our pilot and update his car, in the second we query for the car again.

```
[updatePilotSeparateSessionsPart1]

ObjectSet result = db.get(new Car("Ferrari"));
    Car found = (Car)result.next();
    found.Pilot.AddPoints(1);
    db.set(found);
```

```
[updatePilotSeparateSessionsPart2]

ObjectSet result = db.get(new Car("Ferrari"));
    listResult(result);
```

```
OUTPUT:
1
Ferrari[Somebody else/2]
```

Looks like we're in trouble. What's happening here and what can we do to fix it?

### 4.3.1. Update depth

Imagine a complex object with many members that have many members themselves. When updating this object, db4o would have to update all its children, grandchildren, etc. This poses a severe performance penalty and will not be necessary in most cases - sometimes, however, it will.

To be able to handle this dilemma as flexible as possible, db4o introduces the concept of update depth to control how deep an object's member tree will be traversed on update. The default update depth for all objects is 1, meaning that only primitive and String members will be updated, but changes in object members will not be reflected.

db4o provides means to control update depth with very fine granularity. For our current problem we'll advise db4o to update the full graph for Car objects by setting cascadeOnUpdate() for this class accordingly.

```
[updatePilotSeparateSessionsImprovedPart1]

Db4o.configure().objectClass(typeof(Car))
        .cascadeOnUpdate(true);
```

```
[updatePilotSeparateSessionsImprovedPart2]

ObjectSet result = db.get(new Car("Ferrari"));
    Car found = (Car)result.next();
    found.Pilot.AddPoints(1);
    db.set(found);
```

```
[updatePilotSeparateSessionsImprovedPart3]

ObjectSet result = db.get(new Car("Ferrari"));
```

```
    listResult(result);
```

```
OUTPUT:
1
Ferrari[Somebody else/3]
```

This looks much better.

Note that container configuration must be set before the container is opened.

We'll cover update depth as well as other issues with complex object graphs and the respective db4o configuration options in more detail in a later chapter.

### 4.4. Deleting structured objects

As we have already seen, we call delete() on objects to get rid of them.

```
[deleteFlat]

ObjectSet result = db.get(new Car("Ferrari"));
    Car found = (Car)result.next();
    db.delete(found);
    result = db.get(new Car(null));
    listResult(result);
```

```
OUTPUT:
1
BMW[Rubens Barrichello/99]
```

Fine, the car is gone. What about the pilots?

```
[retrieveAllPilotsQBE]

Pilot proto = new Pilot(null, 0);
    ObjectSet result = db.get(proto);
    listResult(result);
```

```
OUTPUT:
1
Rubens Barrichello/99
```

Ok, this is no real surprise - we don't expect a pilot to vanish when his car is disposed of in real life, too. But what if we want an object's children to be thrown away on deletion, too?

### 4.4.1. Recursive deletion

You may already suspect that the problem of recursive deletion (and perhaps its solution, too) is quite similar to our little update problem, and you're right. Let's configure db4o to delete a car's pilot, too, when the car is deleted.

```
[deleteDeepPart1]

Db4o.configure().objectClass(typeof(Car))
        .cascadeOnDelete(true);
```

```
[deleteDeepPart2]

ObjectSet result = db.get(new Car("BMW"));
    Car found = (Car)result.next();
    db.delete(found);
    result = db.get(new Car(null));
    listResult(result);
```

```
OUTPUT:
0
```

Again: Note that all configuration must take place before the ObjectContainer is opened.

Let's have a look at our pilots again.

```
[retrieveAllPilotsQBE]

Pilot proto = new Pilot(null, 0);
    ObjectSet result = db.get(proto);
    listResult(result);
```

```
OUTPUT:
0
```

### 4.4.2. Recursive deletion revisited

But wait - what happens if the children of a removed object are still referenced by other objects?

```
[deleteDeepRevisited]

ObjectSet result = db.get(new Pilot("Michael Schumacher", 0));
    Pilot pilot = (Pilot)result.next();
    Car car1 = new Car("Ferrari");
    Car car2 = new Car("BMW");
    car1.Pilot = pilot;
    car2.Pilot = pilot;
    db.set(car1);
    db.set(car2);
    db.delete(car2);
```

```
    result = db.get(new Car(null));
    listResult(result);
```

**OUTPUT:**

```
1
Ferrari[null]
```

```
[retrieveAllPilotsQBE]

Pilot proto = new Pilot(null, 0);
    ObjectSet result = db.get(proto);
    listResult(result);
```

**OUTPUT:**

```
0
```

Houston, we have a problem - and there's no simple solution at hand. Currently db4o does **not** check whether objects to be deleted are referenced anywhere else, so please be very careful when activating this feature.

Let's clear our database for the next chapter.

```
[deleteAll]

ObjectSet cars=db.get(new Car(null));
    while(cars.hasNext()) {
        db.delete(cars.next());
    }
    ObjectSet pilots=db.get(new Pilot(null,0));
    while(pilots.hasNext()) {
        db.delete(pilots.next());
```

```
    }
```

### 4.5. Conclusion

So much for object associations: We can hook into a root object and climb down its reference graph to specify queries. But what about multi-valued objects like arrays and collections? We will cover this in the next chapter .

### 4.6. Full source

```
namespace com.db4o.f1.chapter2
{
    using System;
    using System.IO;
    using com.db4o;
    using com.db4o.f1;
    using com.db4o.query;

    public class StructuredExample : Util
    {
        public static void Main(String[] args)
        {
            File.Delete(Util.YapFileName);

            ObjectContainer db = Db4o.openFile(Util.YapFileName);
            try
            {
                storeFirstCar(db);
                storeSecondCar(db);
                retrieveAllCarsQBE(db);
                retrieveAllPilotsQBE(db);
                retrieveCarByPilotQBE(db);
                retrieveCarByPilotNameQuery(db);
                retrieveCarByPilotProtoQuery(db);
             retrievePilotByCarModelQuery(db);
                updateCar(db);
                updatePilotSingleSession(db);
```

```
            updatePilotSeparateSessionsPart1(db);
            db.close();
            db=Db4o.openFile(Util.YapFileName);
            updatePilotSeparateSessionsPart2(db);
            db.close();
            updatePilotSeparateSessionsImprovedPart1(db);
            db=Db4o.openFile(Util.YapFileName);
            updatePilotSeparateSessionsImprovedPart2(db);
            db.close();
            db=Db4o.openFile(Util.YapFileName);
            updatePilotSeparateSessionsImprovedPart3(db);
            deleteFlat(db);
            db.close();
            deleteDeepPart1(db);
            db=Db4o.openFile(Util.YapFileName);
            deleteDeepPart2(db);
            deleteDeepRevisited(db);
        }
        finally
        {
            db.close();
        }
    }

    public static void storeFirstCar(ObjectContainer db)
    {
        Car car1 = new Car("Ferrari");
        Pilot pilot1 = new Pilot("Michael Schumacher", 100);
        car1.Pilot = pilot1;
        db.set(car1);
    }

    public static void storeSecondCar(ObjectContainer db)
    {
        Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
        db.set(pilot2);
        Car car2 = new Car("BMW");
        car2.Pilot = pilot2;
        db.set(car2);
    }
```

```
        public static void retrieveAllCarsQBE(ObjectContainer db)
        {
            Car proto = new Car(null);
            ObjectSet result = db.get(proto);
            listResult(result);
        }

        public static void retrieveAllPilotsQBE(ObjectContainer db)
        {
            Pilot proto = new Pilot(null, 0);
            ObjectSet result = db.get(proto);
            listResult(result);
        }

        public static void retrieveCarByPilotQBE(ObjectContainer db)
        {
            Pilot pilotproto = new Pilot("Rubens Barrichello",0);
            Car carproto = new Car(null);
            carproto.Pilot = pilotproto;
            ObjectSet result = db.get(carproto);
            listResult(result);
        }

        public static void
retrieveCarByPilotNameQuery(ObjectContainer db)
        {
            Query query = db.query();
            query.constrain(typeof(Car));
            query.descend("_pilot").descend("_name")
                .constrain("Rubens Barrichello");
            ObjectSet result = query.execute();
            listResult(result);
        }

        public static void
retrieveCarByPilotProtoQuery(ObjectContainer db)
        {
            Query query = db.query();
            query.constrain(typeof(Car));
            Pilot proto = new Pilot("Rubens Barrichello", 0);
            query.descend("_pilot").constrain(proto);
```

```
            ObjectSet result = query.execute();
            listResult(result);
        }

        public static void
retrievePilotByCarModelQuery(ObjectContainer db)
        {
         Query carquery=db.query();
         carquery.constrain(typeof(Car));
         carquery.descend("_model").constrain("Ferrari");
         Query pilotquery=carquery.descend("_pilot");
         ObjectSet result=pilotquery.execute();
         listResult(result);
        }

        public static void updateCar(ObjectContainer db)
        {
            ObjectSet result = db.get(new Car("Ferrari"));
            Car found = (Car)result.next();
            found.Pilot = new Pilot("Somebody else", 0);
            db.set(found);
            result = db.get(new Car("Ferrari"));
            listResult(result);
        }

        public static void updatePilotSingleSession(ObjectContainer
db)
        {
            ObjectSet result = db.get(new Car("Ferrari"));
            Car found = (Car)result.next();
            found.Pilot.AddPoints(1);
            db.set(found);
            result = db.get(new Car("Ferrari"));
            listResult(result);
        }

        public static void
updatePilotSeparateSessionsPart1(ObjectContainer db)
        {
            ObjectSet result = db.get(new Car("Ferrari"));
            Car found = (Car)result.next();
```

```
            found.Pilot.AddPoints(1);
            db.set(found);
        }

        public static void
updatePilotSeparateSessionsPart2(ObjectContainer db)
        {
            ObjectSet result = db.get(new Car("Ferrari"));
            listResult(result);
        }

        public static void
updatePilotSeparateSessionsImprovedPart1(ObjectContainer db)
        {
            Db4o.configure().objectClass(typeof(Car))
                .cascadeOnUpdate(true);
        }

        public static void
updatePilotSeparateSessionsImprovedPart2(ObjectContainer db)
        {
            ObjectSet result = db.get(new Car("Ferrari"));
            Car found = (Car)result.next();
            found.Pilot.AddPoints(1);
            db.set(found);
        }

        public static void
updatePilotSeparateSessionsImprovedPart3(ObjectContainer db)
        {
            ObjectSet result = db.get(new Car("Ferrari"));
            listResult(result);
        }

        public static void deleteFlat(ObjectContainer db)
        {
            ObjectSet result = db.get(new Car("Ferrari"));
            Car found = (Car)result.next();
            db.delete(found);
            result = db.get(new Car(null));
            listResult(result);
```

```
        }

        public static void deleteDeepPart1(ObjectContainer db)
        {
            Db4o.configure().objectClass(typeof(Car))
                .cascadeOnDelete(true);
        }

        public static void deleteDeepPart2(ObjectContainer db)
        {
            ObjectSet result = db.get(new Car("BMW"));
            Car found = (Car)result.next();
            db.delete(found);
            result = db.get(new Car(null));
            listResult(result);
        }

        public static void deleteDeepRevisited(ObjectContainer db)
        {
            ObjectSet result = db.get(new Pilot("Michael Schumacher",
0));
            Pilot pilot = (Pilot)result.next();
            Car car1 = new Car("Ferrari");
            Car car2 = new Car("BMW");
            car1.Pilot = pilot;
            car2.Pilot = pilot;
            db.set(car1);
            db.set(car2);
            db.delete(car2);
            result = db.get(new Car(null));
            listResult(result);
        }

        public static void deleteAll(ObjectContainer db) {
         ObjectSet cars=db.get(new Car(null));
         while(cars.hasNext()) {
             db.delete(cars.next());
         }
         ObjectSet pilots=db.get(new Pilot(null,0));
         while(pilots.hasNext()) {
             db.delete(pilots.next());
```

```
            }
        }
    }
}
```

## 5. Collections and Arrays

We will slowly move towards real-time data processing now by installing sensors to our car and collecting their output.

```
namespace com.db4o.f1.chapter3
{
    using System;
    using System.Text;

    public class SensorReadout
    {
        double[] _values;
        DateTime _time;
        Car _car;

        public SensorReadout(double[] values, DateTime time, Car car)
        {
            _values = values;
            _time = time;
            _car = car;
        }

        public Car Car
        {
            get
            {
                return _car;
            }
        }

        public DateTime Time
        {
            get
            {
                return _time;
            }
        }
```

```
        public int NumValues
        {
            get
            {
                return _values.Length;
            }
        }

        public double GetValue(int idx)
        {
            return _values[idx];
        }

        override public string ToString()
        {
            StringBuilder builder = new StringBuilder();
            builder.Append(_car);
            builder.Append(" : ");
            builder.Append(_time.TimeOfDay);
            builder.Append(" : ");
            for (int i=0; i<_values.Length; ++i)
            {
                if (i > 0)
                {
                    builder.Append(", ");
                }
                builder.Append(_values[i]);
            }
            return builder.ToString();
        }
    }
}
```

A car may produce its current sensor readout when requested and keep a list of readouts collected during a race.

```
namespace com.db4o.f1.chapter3
{
    using System;
    using System.Collections;

    public class Car
    {
        string _model;
        Pilot _pilot;
        IList _history;

        public Car(string model) : this(model, new ArrayList())
        {
        }

        public Car(string model, IList history)
        {
            _model = model;
            _pilot = null;
            _history = history;
        }

        public Pilot Pilot
        {
            get
            {
                return _pilot;
            }

            set
            {
                _pilot = value;
            }
        }

        public string Model
        {
            get
            {
                return _model;
```

```
            }
        }

        public SensorReadout[] GetHistory()
        {
            SensorReadout[] history = new
SensorReadout[_history.Count];
            _history.CopyTo(history, 0);
            return history;
        }

        public void Snapshot()
        {
            _history.Add(new SensorReadout(Poll(), DateTime.Now,
this));
        }

        protected double[]  Poll()
        {
            int factor = _history.Count + 1;
            return new double[] { 0.1d*factor, 0.2d*factor,
0.3d*factor };
        }

        override public string ToString()
        {
            return _model + "[" + _pilot + "]/" + _history.Count;
        }
    }
}
```

We will constrain ourselves to rather static data at the moment and add flexibility during the next chapters.

### 5.1. Storing

This should be familiar by now.

```
[storeFirstCar]

Car car1 = new Car("Ferrari");
    Pilot pilot1 = new Pilot("Michael Schumacher", 100);
    car1.Pilot = pilot1;
    db.set(car1);
```

The second car will take two snapshots immediately at startup.

```
[storeSecondCar]

Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
    Car car2 = new Car("BMW");
    car2.Pilot = pilot2;
    car2.Snapshot();
    car2.Snapshot();
    db.set(car2);
```

### 5.2. Retrieving

#### 5.2.1. QBE

First let us verify that we indeed have taken snapshots.

```
[retrieveAllSensorReadouts]

SensorReadout proto = new SensorReadout(null, DateTime.MinValue,
null);
    ObjectSet result = db.get(proto);
    listResult(result);
```

```
OUTPUT:
2
BMW[Rubens Barrichello/99]/2 : 1115341840857 : 0.1,0.2,0.3
BMW[Rubens Barrichello/99]/2 : 1115341840857 : 0.2,0.4,0.6
```

As a prototype for an array, we provide an array of the same type, containing only the values we expect the result to contain.

```
[retrieveSensorReadoutQBE]

SensorReadout proto = new SensorReadout(new double[] { 0.3, 0.1 },
DateTime.MinValue, null);
    ObjectSet result = db.get(proto);
    listResult(result);
```

```
OUTPUT:
1
BMW[Rubens Barrichello/99]/2 : 1115341840857 : 0.1,0.2,0.3
```

Note that the actual position of the given elements in the prototype array is irrelevant.

To retrieve a car by its stored sensor readouts, we install a history containing the sought-after values.

```
[retrieveCarQBE]

SensorReadout protoreadout = new SensorReadout(new double[] { 0.6,
0.2 }, DateTime.MinValue, null);
    IList protohistory = new ArrayList();
    protohistory.Add(protoreadout);
    Car protocar = new Car(null, protohistory);
    ObjectSet result = db.get(protocar);
    listResult(result);
```

```
OUTPUT:
1
BMW[Rubens Barrichello/99]/2
```

We can also query for the collections themselves, since they are first class objects.

```
[retrieveCollections]

ObjectSet result = db.get(new ArrayList());
    listResult(result);
```

```
OUTPUT:
2
[]
[BMW[Rubens Barrichello/99]/2 : 1115341840857 : 0.1,0.2,0.3,
BMW[Rubens Barrichello/99]/2 : 1115341840857 : 0.2,0.4,0.6]
```

This doesn't work with arrays, though.

```
[retrieveArrays]

ObjectSet result = db.get(new double[] { 0.6, 0.4 });
    listResult(result);
```

```
OUTPUT:
0
```

### 5.2.2. Query API

Handling of arrays and collections is analogous to the previous example.

```
[retrieveSensorReadoutQuery]

Query query = db.query();
    query.constrain(typeof(SensorReadout));
    Query valuequery = query.descend("_values");
    valuequery.constrain(0.3);
    valuequery.constrain(0.1);
    ObjectSet result = query.execute();
    listResult(result);
```

```
OUTPUT:
1
BMW[Rubens Barrichello/99]/2 : 1115341840857 : 0.1,0.2,0.3
```

```
[retrieveCarQuery]

Query query = db.query();
    query.constrain(typeof(Car));
    Query historyquery = query.descend("_history");
    historyquery.constrain(typeof(SensorReadout));
    Query valuequery = historyquery.descend("_values");
    valuequery.constrain(0.3);
    valuequery.constrain(0.1);
    ObjectSet result = query.execute();
    listResult(result);
```

```
OUTPUT:
1
BMW[Rubens Barrichello/99]/2
```

### 5.3. Updating and deleting

This should be familiar, we just have to remember to take care of the update depth.

```
[updateCarPart1]

Db4o.configure().objectClass(typeof(Car)).cascadeOnUpdate(true);
```

```
[updateCarPart2]

ObjectSet result = db.get(new Car("BMW", null));
    Car car = (Car)result.next();
    car.Snapshot();
    db.set(car);
    retrieveAllSensorReadouts(db);
```

```
OUTPUT:
3
BMW[Rubens Barrichello/99]/3 : 1115341840857 : 0.1,0.2,0.3
BMW[Rubens Barrichello/99]/3 : 1115341841044 :
0.30000000000000004,0.6000000000000001,0.8999999999999999
BMW[Rubens Barrichello/99]/3 : 1115341840857 : 0.2,0.4,0.6
```

There's nothing special about deleting arrays and collections, too.

Deleting an object from a collection is an update, too, of course.

```
[updateCollection]

Query query = db.query();
```

```
    query.constrain(typeof(Car));
    ObjectSet result = query.descend("_history").execute();
    IList coll = (IList)result.next();
    coll.RemoveAt(0);
    db.set(coll);
    Car proto = new Car(null, null);
    result = db.get(proto);
    while (result.hasNext())
    {
        Car car = (Car)result.next();
        foreach (object readout in car.GetHistory())
        {
            Console.WriteLine(readout);
        }
    }
```

**OUTPUT:**

```
BMW[Rubens Barrichello/99]/2 : 1115341840857 : 0.2,0.4,0.6
BMW[Rubens Barrichello/99]/2 : 1115341841044 :
0.30000000000000004,0.6000000000000001,0.8999999999999999
```

(This example also shows that with db4o it is quite easy to access object internals we were never meant to see. Please keep this always in mind and be careful.)

We will delete all cars from the database again to prepare for the next chapter.

```
[deleteAllPart1]

Db4o.configure().objectClass(typeof(Car)).cascadeOnDelete(true);
```

```
[deleteAllPart2]

ObjectSet result = db.get(new Car(null, null));
```

```
    while (result.hasNext())
    {
        db.delete(result.next());
    }
    ObjectSet readouts = db.get(new SensorReadout(null,
DateTime.MinValue, null));
    while(readouts.hasNext())
    {
        db.delete(readouts.next());
    }
```

### 5.4. Conclusion

Ok, collections are just objects. But why did we have to specify the concrete ArrayList type all the way? Was that necessary? How does db4o handle inheritance? We will cover that in the next chapter .

### 5.5. Full source

```
namespace com.db4o.f1.chapter3
{
    using System;
    using System.Collections;
    using System.IO;
    using com.db4o;
    using com.db4o.query;

    public class CollectionsExample : Util
    {
        public static void Main(string[] args)
        {
            File.Delete(Util.YapFileName);
            ObjectContainer db = Db4o.openFile(Util.YapFileName);
            try
            {
                storeFirstCar(db);
                storeSecondCar(db);
                retrieveAllSensorReadouts(db);
```

```
            retrieveSensorReadoutQBE(db);
            retrieveCarQBE(db);
            retrieveCollections(db);
            retrieveArrays(db);
            retrieveSensorReadoutQuery(db);
            retrieveCarQuery(db);
            db.close();
            updateCarPart1();
            db = Db4o.openFile(Util.YapFileName);
            updateCarPart2(db);
            updateCollection(db);
            db.close();
            deleteAllPart1();
            db=Db4o.openFile(Util.YapFileName);
            deleteAllPart2(db);
            retrieveAllSensorReadouts(db);
        }
        finally
        {
            db.close();
        }
    }

    public static void storeFirstCar(ObjectContainer db)
    {
        Car car1 = new Car("Ferrari");
        Pilot pilot1 = new Pilot("Michael Schumacher", 100);
        car1.Pilot = pilot1;
        db.set(car1);
    }

    public static void storeSecondCar(ObjectContainer db)
    {
        Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
        Car car2 = new Car("BMW");
        car2.Pilot = pilot2;
        car2.Snapshot();
        car2.Snapshot();
        db.set(car2);
    }
```

```
        public static void retrieveAllSensorReadouts(ObjectContainer
db)
        {
            SensorReadout proto = new SensorReadout(null,
DateTime.MinValue, null);
            ObjectSet result = db.get(proto);
            listResult(result);
        }

        public static void retrieveSensorReadoutQBE(ObjectContainer
db)
        {
            SensorReadout proto = new SensorReadout(new double[] {
0.3, 0.1 }, DateTime.MinValue, null);
            ObjectSet result = db.get(proto);
            listResult(result);
        }

        public static void retrieveCarQBE(ObjectContainer db)
        {
            SensorReadout protoreadout = new SensorReadout(new
double[] { 0.6, 0.2 }, DateTime.MinValue, null);
            IList protohistory = new ArrayList();
            protohistory.Add(protoreadout);
            Car protocar = new Car(null, protohistory);
            ObjectSet result = db.get(protocar);
            listResult(result);
        }

        public static void retrieveCollections(ObjectContainer db)
        {
            ObjectSet result = db.get(new ArrayList());
            listResult(result);
        }

        public static void retrieveArrays(ObjectContainer db)
        {
            ObjectSet result = db.get(new double[] { 0.6, 0.4 });
            listResult(result);
        }
```

```
        public static void retrieveSensorReadoutQuery(ObjectContainer
db)
        {
            Query query = db.query();
            query.constrain(typeof(SensorReadout));
            Query valuequery = query.descend("_values");
            valuequery.constrain(0.3);
            valuequery.constrain(0.1);
            ObjectSet result = query.execute();
            listResult(result);
        }

        public static void retrieveCarQuery(ObjectContainer db)
        {
            Query query = db.query();
            query.constrain(typeof(Car));
            Query historyquery = query.descend("_history");
            historyquery.constrain(typeof(SensorReadout));
            Query valuequery = historyquery.descend("_values");
            valuequery.constrain(0.3);
            valuequery.constrain(0.1);
            ObjectSet result = query.execute();
            listResult(result);
        }

        public static void updateCarPart1()
        {
Db4o.configure().objectClass(typeof(Car)).cascadeOnUpdate(true);
   }

        public static void updateCarPart2(ObjectContainer db)
        {
            ObjectSet result = db.get(new Car("BMW", null));
            Car car = (Car)result.next();
            car.Snapshot();
            db.set(car);
            retrieveAllSensorReadouts(db);
        }

        public static void updateCollection(ObjectContainer db)
        {
```

```
            Query query = db.query();
            query.constrain(typeof(Car));
            ObjectSet result = query.descend("_history").execute();
            IList coll = (IList)result.next();
            coll.RemoveAt(0);
            db.set(coll);
            Car proto = new Car(null, null);
            result = db.get(proto);
            while (result.hasNext())
            {
                Car car = (Car)result.next();
                foreach (object readout in car.GetHistory())
                {
                    Console.WriteLine(readout);
                }
            }
        }

        public static void deleteAllPart1()
        {
Db4o.configure().objectClass(typeof(Car)).cascadeOnDelete(true);
   }

        public static void deleteAllPart2(ObjectContainer db)
        {
            ObjectSet result = db.get(new Car(null, null));
            while (result.hasNext())
            {
                db.delete(result.next());
            }
            ObjectSet readouts = db.get(new SensorReadout(null,
DateTime.MinValue, null));
            while(readouts.hasNext())
            {
                db.delete(readouts.next());
            }
        }
    }
}
```

## 6. Inheritance

So far we have always been working with the concrete (i.e. most specific type of an object. What about subclassing and interfaces?

To explore this, we will differentiate between different kinds of sensors.

```
namespace com.db4o.f1.chapter4
{
    using System;

    public class SensorReadout
    {
        DateTime _time;
        Car _car;
        string _description;

        public SensorReadout(DateTime time, Car car, string
description)
        {
            _time = time;
            _car = car;
            _description = description;
        }

        public Car Car
        {
            get
            {
                return _car;
            }
        }

        public DateTime Time
        {
            get
            {
                return _time;
```

```
            }
        }

        public string Description
        {
            get
            {
                return _description;
            }
        }

        override public string ToString()
        {
            return _car + ":" + _time + " : " + _description;
        }
    }
}
```

```
namespace com.db4o.f1.chapter4
{
    using System;

    public class TemperatureSensorReadout : SensorReadout
    {
        double _temperature;

        public TemperatureSensorReadout(DateTime time, Car car,
string description, double temperature)
            : base(time, car, description)
        {
            _temperature = temperature;
        }

        public double Temperature
        {
            get
            {
```

```
                return _temperature;
            }
        }

        override public string ToString()
        {
            return base.ToString() + " temp: " + _temperature;
        }
    }
}
```

```
namespace com.db4o.f1.chapter4
{
    using System;

    public class PressureSensorReadout : SensorReadout
    {
        double _pressure;

        public PressureSensorReadout(DateTime time, Car car, string
description, double pressure)
            : base(time, car, description)
        {
            _pressure = pressure;
        }

        public double Pressure
        {
            get
            {
                return _pressure;
            }
        }

        override public string ToString()
        {
            return base.ToString() + " pressure : " + _pressure;
```

```
        }
    }
}
```

Our car's snapshot mechanism is changed accordingly.

```
namespace com.db4o.f1.chapter4
{
    using System;
    using System.Collections;

    public class Car
    {
        string _model;
        Pilot _pilot;
        IList _history;

        public Car(string model)
        {
            _model = model;
            _pilot = null;
            _history = new ArrayList();
        }

        public Pilot Pilot
        {
            get
            {
                return _pilot;
            }

            set
            {
                _pilot = value;
            }
        }
```

```
        public string Model
        {
            get
            {
                return _model;
            }
        }

        public SensorReadout[] GetHistory()
        {
            SensorReadout[] history = new
SensorReadout[_history.Count];
            _history.CopyTo(history, 0);
            return history;
        }

        public void Snapshot()
        {
            _history.Add(new TemperatureSensorReadout(DateTime.Now,
this, "oil", PollOilTemperature()));
            _history.Add(new TemperatureSensorReadout(DateTime.Now,
this, "water", PollWaterTemperature()));
            _history.Add(new PressureSensorReadout(DateTime.Now,
this, "oil", PollOilPressure()));
        }

        protected double PollOilTemperature()
        {
            return 0.1*_history.Count;
        }

        protected double PollWaterTemperature()
        {
            return 0.2*_history.Count;
        }

        protected double PollOilPressure()
        {
            return 0.3*_history.Count;
        }
```

```
        override public string ToString()
        {
            return _model + "[" + _pilot + "]/" + _history.Count;
        }
    }
}
```

### 6.1. Storing

Our setup code has not changed at all, just the internal workings of a snapshot.

```
[storeFirstCar]

Car car1 = new Car("Ferrari");
    Pilot pilot1 = new Pilot("Michael Schumacher", 100);
    car1.Pilot = pilot1;
    db.set(car1);
```

```
[storeSecondCar]

Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
    Car car2 = new Car("BMW");
    car2.Pilot = pilot2;
    car2.Snapshot();
    car2.Snapshot();
    db.set(car2);
```

### 6.2. Retrieving

db4o will provide us with all objects of the given type. To collect all instances of a given class, no matter whether they are subclass members or direct instances, we just provide a corresponding prototype.

```
[retrieveTemperatureReadoutsQBE]

SensorReadout proto = new TemperatureSensorReadout(DateTime.MinValue,
null, null, 0.0);
    ObjectSet result = db.get(proto);
    listResult(result);
```

**OUTPUT:**

```
4
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.30000000000000004
```

```
[retrieveAllSensorReadoutsQBE]

SensorReadout proto = new SensorReadout(DateTime.MinValue, null,
null);
    ObjectSet result = db.get(proto);
    listResult(result);
```

**OUTPUT:**

```
6
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
```

```
pressure : 0.6
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.30000000000000004
```

This is one more situation where QBE might not be applicable: What if the given type is an interface or an abstract class? Well, there's a little DWIM trick to the rescue: Class objects receive special handling with QBE.

```
[retrieveAllSensorReadoutsQBEAlternative]

ObjectSet result = db.get(typeof(SensorReadout));
    listResult(result);
```

**OUTPUT:**

```
6
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.30000000000000004
```

And of course there's our query API to the rescue.

```
[retrieveAllSensorReadoutsQuery]

Query query = db.query();
    query.constrain(typeof(SensorReadout));
    ObjectSet result = query.execute();
    listResult(result);
```

**OUTPUT:**

```
6
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.30000000000000004
```

This procedure applies to all first class objects. We can simply query for all objects present in the database, for example.

```
[retrieveAllObjects]

ObjectSet result = db.get(new object());
    listResult(result);
```

**OUTPUT:**

```
12
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : water
```

```
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.30000000000000004
[BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.0, BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST
2005 : water temp : 0.2, BMW[Rubens Barrichello/99]/6 : Fri May 06
03:10:41 CEST 2005 : oil pressure : 0.6, BMW[Rubens Barrichello/99]/6
: Fri May 06 03:10:41 CEST 2005 : oil temp : 0.30000000000000004,
BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.8, BMW[Rubens Barrichello/99]/6 : Fri May 06 03:10:41 CEST
2005 : oil pressure : 1.5]
[]
BMW[Rubens Barrichello/99]/6
Ferrari[Michael Schumacher/100]/0
Rubens Barrichello/99
Michael Schumacher/100
```

### 6.3. Updating and deleting

is just the same for all objects, no matter where they are situated in the inheritance tree.

Just like we retrieved all objects from the database above, we can delete all stored objects to prepare for the next chapter.

```
[deleteAllObjects]

ObjectSet result=db.get(new object());
    while (result.hasNext())
    {
```

```
        db.delete(result.next());
    }
```

OUTPUT:

### 6.4. Conclusion

Now we have covered all basic OO features and the way they are handled by db4o. We will complete the first part of our db4o walkthrough in the next chapter by looking at deep object graphs, including recursive structures.

### 6.5. Full source

```
namespace com.db4o.f1.chapter4
{
    using System;
    using System.IO;
    using com.db4o;
    using com.db4o.f1;
    using com.db4o.query;

    public class InheritanceExample : Util
    {
        public static void Main(string[] args)
        {
            File.Delete(Util.YapFileName);
            ObjectContainer db = Db4o.openFile(Util.YapFileName);
            try
            {
                storeFirstCar(db);
                storeSecondCar(db);
                retrieveTemperatureReadoutsQBE(db);
                retrieveAllSensorReadoutsQBE(db);
                retrieveAllSensorReadoutsQBEAlternative(db);
                retrieveAllSensorReadoutsQuery(db);
                retrieveAllObjects(db);
```

```
                deleteAllObjects(db);
            }
            finally
            {
                db.close();
            }
        }

        public static void storeFirstCar(ObjectContainer db)
        {
            Car car1 = new Car("Ferrari");
            Pilot pilot1 = new Pilot("Michael Schumacher", 100);
            car1.Pilot = pilot1;
            db.set(car1);
        }

        public static void storeSecondCar(ObjectContainer db)
        {
            Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
            Car car2 = new Car("BMW");
            car2.Pilot = pilot2;
            car2.Snapshot();
            car2.Snapshot();
            db.set(car2);
        }

        public static void
retrieveAllSensorReadoutsQBE(ObjectContainer db)
        {
            SensorReadout proto = new
SensorReadout(DateTime.MinValue, null, null);
            ObjectSet result = db.get(proto);
            listResult(result);
        }

        public static void
retrieveTemperatureReadoutsQBE(ObjectContainer db)
        {
            SensorReadout proto = new
TemperatureSensorReadout(DateTime.MinValue, null, null, 0.0);
            ObjectSet result = db.get(proto);
```

```
            listResult(result);
        }

        public static void
retrieveAllSensorReadoutsQBEAlternative(ObjectContainer db)
        {
            ObjectSet result = db.get(typeof(SensorReadout));
            listResult(result);
        }

        public static void
retrieveAllSensorReadoutsQuery(ObjectContainer db)
        {
            Query query = db.query();
            query.constrain(typeof(SensorReadout));
            ObjectSet result = query.execute();
            listResult(result);
        }

        public static void retrieveAllObjects(ObjectContainer db)
        {
            ObjectSet result = db.get(new object());
            listResult(result);
        }

        public static void deleteAllObjects(ObjectContainer db)
        {
            ObjectSet result=db.get(new object());
            while (result.hasNext())
            {
                db.delete(result.next());
            }
        }
    }
}
```

## 7. Deep graphs

We have already seen how db4o handles object associations, but our running example is still quite flat and simple, compared to real-world domain models. In particular we haven't seen how db4o behaves in the presence of recursive structures. We will emulate such a structure by replacing our history list with a linked list implicitely provided by the SensorReadout class.

```
namespace com.db4o.f1.chapter5
{
    using System;

    public abstract class SensorReadout
    {
        DateTime _time;
        Car _car;
        string _description;
        SensorReadout _next;

        protected SensorReadout(DateTime time, Car car, string
description)
        {
            _time = time;
            _car = car;
            _description = description;
            _next = null;
        }

        public Car Car
        {
            get
            {
                return _car;
            }
        }

        public DateTime Time
        {
            get
```

```
            {
                return _time;
            }
        }

        public SensorReadout Next
        {
            get
            {
                return _next;
            }
        }

        public void Append(SensorReadout sensorReadout)
        {
            if (_next == null)
            {
                _next = sensorReadout;
            }
            else
            {
                _next.Append(sensorReadout);
            }
        }

        public int CountElements()
        {
            return (_next == null ? 1 : _next.CountElements() + 1);
        }

        override public string ToString()
        {
            return _car + " : " + _time + " : " + _description;
        }
    }
}
```

Our car only maintains an association to a 'head' sensor readout now.

```
namespace com.db4o.f1.chapter5
{
    using System;

    public class Car
    {
        string _model;
        Pilot _pilot;
        SensorReadout _history;

        public Car(string model)
        {
            _model = model;
            _pilot = null;
            _history = null;
        }

        public Pilot Pilot
        {
            get
            {
                return _pilot;
            }

            set
            {
                _pilot = value;
            }
        }

        public string Model
        {
            get
            {
                return _model;
            }
        }

        public SensorReadout GetHistory()
```

```
        {
            return _history;
        }

        public void Snapshot()
        {
            AppendToHistory(new TemperatureSensorReadout(
                DateTime.Now, this, "oil", PollOilTemperature()));
            AppendToHistory(new TemperatureSensorReadout(
                DateTime.Now, this, "water",
PollWaterTemperature()));
            AppendToHistory(new PressureSensorReadout(
                DateTime.Now, this, "oil", PollOilPressure()));
        }

        protected double PollOilTemperature()
        {
            return 0.1*CountHistoryElements();
        }

        protected double PollWaterTemperature()
        {
            return 0.2*CountHistoryElements();
        }

        protected double PollOilPressure()
        {
            return 0.3*CountHistoryElements();
        }

        override public string ToString()
        {
            return _model + "[" + _pilot + "]/" +
CountHistoryElements();
        }

        private int CountHistoryElements()
        {
            return (_history == null ? 0 : _history.CountElements());
        }
```

```
        private void AppendToHistory(SensorReadout readout)
        {
            if (_history == null)
            {
                _history = readout;
            }
            else
            {
                _history.Append(readout);
            }
        }
    }
}
```

### 7.1. Storing and updating

No surprises here.

```
[storeCar]

Pilot pilot = new Pilot("Rubens Barrichello", 99);
    Car car = new Car("BMW");
    car.Pilot = pilot;
    db.set(car);
```

Now we would like to build a sensor readout chain. We already know about the update depth trap, so we configure this first.

```
[setCascadeOnUpdate]

Db4o.configure().objectClass(typeof(Car)).cascadeOnUpdate(true);
```

Let's collect a few sensor readouts.

```
[takeManySnapshots]

ObjectSet result = db.get(new Car(null));
    Car car = (Car)result.next();
    for(int i=0; i<5; i++)
    {
        car.Snapshot();
    }
    db.set(car);
```

### 7.2. Retrieving

Now that we have a sufficiently deep structure, we'll retrieve it from the database and traverse it.

First let's verify that we indeed have taken lots of snapshots.

```
[retrieveAllSnapshots]

ObjectSet result = db.get(typeof(SensorReadout));
    while (result.hasNext())
    {
        Console.WriteLine(result.next());
    }
```

**OUTPUT:**

```
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 4.2
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 2.6
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 1.2000000000000002
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 3.3
```

```
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 2.0
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.9
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 2.4
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 1.4000000000000001
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.6000000000000001
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.2
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.8
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.30000000000000004
```

All these readouts belong to one linked list, so we should be able to access them all by just traversing our list structure.

```
[retrieveSnapshotsSequentially]

ObjectSet result = db.get(new Car(null));
    Car car = (Car)result.next();
    SensorReadout readout = car.GetHistory();
    while (readout != null)
    {
        Console.WriteLine(readout);
        readout = readout.Next;
    }
```

OUTPUT:

```
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.2
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.8
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.6000000000000001
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 1.4000000000000001
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 2.4
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.9
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 2.0
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 3.3
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 1.2000000000000002
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 2.6
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 4.2
```

Ouch! What's happening here?

### 7.2.1. Activation depth

Deja vu - this is just the other side of the update depth issue.

db4o cannot track when you are traversing references from objects retrieved from the database. So it would always have to return 'complete' object graphs on retrieval - in the worst case this would boil down to pulling the whole database content into memory for a single query.

This is absolutely undesirable in most situations, so db4o provides a mechanism to give the client fine-grained control over how much he wants to pull out of the database when asking for an object. This mechanism is called *activation depth*and works quite similar to our familiar update depth.

The default activation depth for any object is 5, so our example above runs into nulls after traversing 5 references.

We can dynamically ask objects to activate their member references. This allows us to retrieve each single sensor readout in the list from the database just as needed.

```
[retrieveSnapshotsSequentiallyImproved]

ObjectSet result = db.get(new Car(null));
    Car car = (Car)result.next();
    SensorReadout readout = car.GetHistory();
    while (readout != null)
    {
        db.activate(readout, 1);
        Console.WriteLine(readout);
        readout = readout.Next;
    }
```

**OUTPUT:**

```
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.2
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
```

```
temp : 0.8
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.6000000000000001
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 1.4000000000000001
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 2.4
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.9
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 2.0
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 3.3
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 1.2000000000000002
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 2.6
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 4.2
```

Note that 'cut' references may also influence the behavior of your objects: In this case the length of the list is calculated dynamically, and therefor constrained by activation depth.

Instead of dynamically activating subgraph elements, you can configure activation depth statically, too. We can tell our SensorReadout class objects to cascade activation automatically, for example.

```
[setActivationDepth]

Db4o.configure().objectClass(typeof(TemperatureSensorReadout))
        .cascadeOnActivate(true);
```

OUTPUT:

```
[retrieveSnapshotsSequentially]

ObjectSet result = db.get(new Car(null));
    Car car = (Car)result.next();
    SensorReadout readout = car.GetHistory();
    while (readout != null)
    {
        Console.WriteLine(readout);
        readout = readout.Next;
    }
```

**OUTPUT:**

```
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.2
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 0.8
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.6000000000000001
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 1.4000000000000001
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 2.4
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
temp : 0.9
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 2.0
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 3.3
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
```

```
temp : 1.2000000000000002
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : water
temp : 2.6
BMW[Rubens Barrichello/99]/15 : Fri May 06 03:10:41 CEST 2005 : oil
pressure : 4.2
```

You have to be very careful, though. Activation issues are tricky. Db4o provides a wide range of configuration features to control activation depth at a very fine-grained level. You'll find those triggers in com.db4o.config.Configuration and the associated ObjectClass and ObjectField classes.

Don't forget to clean up the database.

```
[deleteAllObjects]

ObjectSet result = db.get(new object());
    while (result.hasNext())
    {
        db.delete(result.next());
    }
```

**OUTPUT:**

### 7.3. Conclusion

Now we should have the tools at hand to work with arbitrarily complex object graphs. But so far we have only been working forward, hoping that the changes we apply to our precious data pool are correct. What if we have to roll back to a previous state due to some failure? In the next chapter we will introduce the db4o transaction concept.

### 7.4. Full source

```
namespace com.db4o.f1.chapter5
{
```

```
using System;
using System.IO;
using com.db4o;

public class DeepExample : Util
{
    public static void Main(string[] args)
    {
        File.Delete(Util.YapFileName);
        ObjectContainer db = Db4o.openFile(Util.YapFileName);
        try
        {
            storeCar(db);
            db.close();
            setCascadeOnUpdate();
            db = Db4o.openFile(Util.YapFileName);
            takeManySnapshots(db);
            db.close();
            db = Db4o.openFile(Util.YapFileName);
            retrieveAllSnapshots(db);
            db.close();
            db = Db4o.openFile(Util.YapFileName);
            retrieveSnapshotsSequentially(db);
            retrieveSnapshotsSequentiallyImproved(db);
            db.close();
            setActivationDepth();
            db = Db4o.openFile(Util.YapFileName);
            retrieveSnapshotsSequentially(db);
            deleteAllObjects(db);
        }
        finally
        {
            db.close();
        }
    }

    public static void storeCar(ObjectContainer db)
    {
        Pilot pilot = new Pilot("Rubens Barrichello", 99);
        Car car = new Car("BMW");
        car.Pilot = pilot;
```

```
            db.set(car);
        }

        public static void setCascadeOnUpdate()
        {
Db4o.configure().objectClass(typeof(Car)).cascadeOnUpdate(true);
   }

        public static void takeManySnapshots(ObjectContainer db)
        {
            ObjectSet result = db.get(new Car(null));
            Car car = (Car)result.next();
            for(int i=0; i<5; i++)
            {
                car.Snapshot();
            }
            db.set(car);
        }

        public static void retrieveAllSnapshots(ObjectContainer db)
        {
            ObjectSet result = db.get(typeof(SensorReadout));
            while (result.hasNext())
            {
                Console.WriteLine(result.next());
            }
        }

        public static void
retrieveSnapshotsSequentially(ObjectContainer db)
        {
            ObjectSet result = db.get(new Car(null));
            Car car = (Car)result.next();
            SensorReadout readout = car.GetHistory();
            while (readout != null)
            {
                Console.WriteLine(readout);
                readout = readout.Next;
            }
        }
```

```
        public static void
retrieveSnapshotsSequentiallyImproved(ObjectContainer db)
        {
            ObjectSet result = db.get(new Car(null));
            Car car = (Car)result.next();
            SensorReadout readout = car.GetHistory();
            while (readout != null)
            {
                db.activate(readout, 1);
                Console.WriteLine(readout);
                readout = readout.Next;
            }
        }

        public static void setActivationDepth()
        {
Db4o.configure().objectClass(typeof(TemperatureSensorReadout))
         .cascadeOnActivate(true);
        }

        public static void deleteAllObjects(ObjectContainer db)
        {
            ObjectSet result = db.get(new object());
            while (result.hasNext())
            {
                db.delete(result.next());
            }
        }
    }
}
```

# 8. Transactions

Probably you have already wondered how db4o handles concurrent access to a single database. Just as any other DBMS, db4o provides a transaction mechanism. Before we take a look at multiple, perhaps even remote, clients accessing a db4o instance in parallel, we will introduce db4o transaction concepts in isolation.

## 8.1. Commit and rollback

You may not have noticed it, but we have already been working with transactions from the first chapter on. By definition, you are always working inside a transaction when interacting with db4o. A transaction is implicitly started when you open a container, and the current transaction is implicitly committed when you close it again. So the following code snippet to store a car is semantically identical to the ones we have seen before; it just makes the commit explicit.

```
[storeCarCommit]

Pilot pilot = new Pilot("Rubens Barrichello", 99);
    Car car = new Car("BMW");
    car.Pilot = pilot;
    db.set(car);
    db.commit();
```

```
[listAllCars]

ObjectSet result = db.get(new Car(null));
    listResult(result);
```

**OUTPUT:**

```
1
BMW[Rubens Barrichello/99]/0
```

However, we can also rollback the current transaction, resetting the state of our database to the last commit point.

```
[storeCarRollback]

Pilot pilot = new Pilot("Michael Schumacher", 100);
    Car car = new Car("Ferrari");
    car.Pilot = pilot;
    db.set(car);
    db.rollback();
```

```
[listAllCars]

ObjectSet result = db.get(new Car(null));
    listResult(result);
```

```
OUTPUT:
1
BMW[Rubens Barrichello/99]/0
```

### 8.2. Refresh live objects

There's one problem, though: We can roll back our database, but this cannot automagically trigger a rollback for our live objects.

```
[carSnapshotRollback]

ObjectSet result = db.get(new Car("BMW"));
    Car car = (Car)result.next();
    car.Snapshot();
    db.set(car);
```

```
    db.rollback();
    Console.WriteLine(car);
```

```
OUTPUT:
BMW[Rubens Barrichello/99]/3
```

We will have to explicitly refresh our live objects when we suspect they may have participated in a rollback transaction.

```
[carSnapshotRollbackRefresh]

ObjectSet result=db.get(new Car("BMW"));
    Car car=(Car)result.next();
    car.Snapshot();
    db.set(car);
    db.rollback();
    db.ext().refresh(car, int.MaxValue);
    Console.WriteLine(car);
```

```
OUTPUT:
BMW[Rubens Barrichello/99]/0
```

What is this ExtObjectContainer construct good for? Well, it provides some functionality that is in itself stable, but the API may still be subject to change. As soon as we are confident that no more changes will occur, *ext* functionality will be transferred to the common ObjectContainer API. We will cover extended functionality in more detail in a later chapter.

Finally, we clean up again.

```
[deleteAllObjects]
```

```
ObjectSet result = db.get(new object());
    while (result.hasNext())
    {
        db.delete(result.next());
    }
```

**OUTPUT:**

### 8.3. Conclusion

We have seen how transactions work for a single client. In the next chapter we will see how the transaction concept extends to multiple clients, whether they are located within the same VM or on a remote machine.

### 8.4. Full source

```
namespace com.db4o.f1.chapter5
{
    using System;
    using System.IO;
    using com.db4o;
    using com.db4o.f1;

    public class TransactionExample : Util
    {
        public static void Main(string[] args)
        {
            File.Delete(Util.YapFileName);
            ObjectContainer db=Db4o.openFile(Util.YapFileName);
            try
            {
                storeCarCommit(db);
                db.close();
                db = Db4o.openFile(Util.YapFileName);
                listAllCars(db);
                storeCarRollback(db);
```

```
            db.close();
            db = Db4o.openFile(Util.YapFileName);
            listAllCars(db);
            carSnapshotRollback(db);
            carSnapshotRollbackRefresh(db);
            deleteAllObjects(db);
        }
        finally
        {
            db.close();
        }
    }

    public static void storeCarCommit(ObjectContainer db)
    {
        Pilot pilot = new Pilot("Rubens Barrichello", 99);
        Car car = new Car("BMW");
        car.Pilot = pilot;
        db.set(car);
        db.commit();
    }

    public static void listAllCars(ObjectContainer db)
    {
        ObjectSet result = db.get(new Car(null));
        listResult(result);
    }

    public static void storeCarRollback(ObjectContainer db)
    {
        Pilot pilot = new Pilot("Michael Schumacher", 100);
        Car car = new Car("Ferrari");
        car.Pilot = pilot;
        db.set(car);
        db.rollback();
    }

    public static void deleteAllObjects(ObjectContainer db)
    {
        ObjectSet result = db.get(new object());
        while (result.hasNext())
```

```
            {
                db.delete(result.next());
            }
        }

        public static void carSnapshotRollback(ObjectContainer db)
        {
            ObjectSet result = db.get(new Car("BMW"));
            Car car = (Car)result.next();
            car.Snapshot();
            db.set(car);
            db.rollback();
            Console.WriteLine(car);
        }

        public static void carSnapshotRollbackRefresh(ObjectContainer
db)
        {
            ObjectSet result=db.get(new Car("BMW"));
            Car car=(Car)result.next();
            car.Snapshot();
            db.set(car);
            db.rollback();
            db.ext().refresh(car, int.MaxValue);
            Console.WriteLine(car);
        }
    }
}
```

## 9. Client/Server

Now that we have seen how transactions work in db4o conceptually, we are prepared to tackle concurrently executing transactions.

We start by preparing our database in the familiar way.

```
[setFirstCar]

Pilot pilot = new Pilot("Rubens Barrichello", 99);
    Car car = new Car("BMW");
    car.Pilot = pilot;
    db.set(car);
```

```
[setSecondCar]

Pilot pilot = new Pilot("Michael Schumacher", 100);
    Car car = new Car("Ferrari");
    car.Pilot = pilot;
    db.set(car);
```

### 9.1. Embedded server

From the API side, there's no real difference between transactions executing concurrently within the same VM and transactions executed against a remote server. To use concurrent transactions within a single VM, we just open a db4o server on our database file, directing it to run on port 0, thereby declaring that no networking will take place.

```
[accessLocalServer]

ObjectServer server = Db4o.openServer(Util.YapFileName, 0);
    try
```

```
    {
        ObjectContainer client = server.openClient();
        // Do something with this client, or open more clients
        client.close();
    }
    finally
    {
        server.close();
    }
```

Again, we will delegate opening and closing the server to our environment to focus on client interactions.

```
[queryLocalServer]

ObjectContainer client = server.openClient();
    listResult(client.get(new Car(null)));
    client.close();
```

**OUTPUT:**

```
2
Ferrari[Michael Schumacher/100]/0
BMW[Rubens Barrichello/99]/0
[db4o 4.5.005   2005-05-06 03:10:41]
 Connection closed by client %.
[db4o 4.5.005   2005-05-06 03:10:41]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula133648.yap' close request
[db4o 4.5.005   2005-05-06 03:10:41]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula133648.yap' closed
```

The transaction level in db4o is *read committed* . However, each client container maintains its own weak reference cache of already known objects. To make all changes committed by other clients immediately, we have to explicitly refresh known objects from the server. We will delegate this task to a specialized version of our listResult() method.

```
public static void listRefreshedResult(ObjectContainer container,
ObjectSet items, int depth)
    {
        Console.WriteLine(items.size());
        while (items.hasNext())
        {
            object item = items.next();
            container.ext().refresh(item, depth);
            Console.WriteLine(item);
        }
    }
```

```
[demonstrateLocalReadCommitted]

ObjectContainer client1 =server.openClient();
    ObjectContainer client2 =server.openClient();
    Pilot pilot = new Pilot("David Coulthard", 98);
    ObjectSet result = client1.get(new Car("BMW"));
    Car car = (Car)result.next();
    car.Pilot = pilot;
    client1.set(car);
    listResult(client1.get(new Car(null)));
    listResult(client2.get(new Car(null)));
    client1.commit();
    listResult(client1.get(typeof(Car)));
    listRefreshedResult(client2, client2.get(typeof(Car)), 2);
    client1.close();
    client2.close();
```

**OUTPUT:**

```
2
Ferrari[Michael Schumacher/100]/0
BMW[David Coulthard/98]/0
2
```

```
Ferrari[Michael Schumacher/100]/0
BMW[Rubens Barrichello/99]/0
2
Ferrari[Michael Schumacher/100]/0
BMW[David Coulthard/98]/0
2
Ferrari[Michael Schumacher/100]/0
BMW[David Coulthard/98]/0
[db4o 4.5.005   2005-05-06 03:10:42]
 Connection closed by client %.
[db4o 4.5.005   2005-05-06 03:10:42]
 Connection closed by client %.
[db4o 4.5.005   2005-05-06 03:10:42]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula133648.yap' close request
[db4o 4.5.005   2005-05-06 03:10:42]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula133648.yap' closed
```

Simple rollbacks just work as you might expect now.

```
[demonstrateLocalRollback]

ObjectContainer client1 = server.openClient();
    ObjectContainer client2 = server.openClient();
    ObjectSet result = client1.get(new Car("BMW"));
    Car car = (Car)result.next();
    car.Pilot = new Pilot("Someone else", 0);
    client1.set(car);
    listResult(client1.get(new Car(null)));
    listResult(client2.get(new Car(null)));
    client1.rollback();
    client1.ext().refresh(car, 2);
    listResult(client1.get(new Car(null)));
    listResult(client2.get(new Car(null)));
    client1.close();
    client2.close();
```

```
OUTPUT:
2
Ferrari[Michael Schumacher/100]/0
BMW[Someone else/0]/0
2
Ferrari[Michael Schumacher/100]/0
BMW[David Coulthard/98]/0
2
Ferrari[Michael Schumacher/100]/0
BMW[David Coulthard/98]/0
2
Ferrari[Michael Schumacher/100]/0
BMW[David Coulthard/98]/0
[db4o 4.5.005   2005-05-06 03:10:42]
 Connection closed by client %.
[db4o 4.5.005   2005-05-06 03:10:42]
 Connection closed by client %.
[db4o 4.5.005   2005-05-06 03:10:42]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula133648.yap' close request
[db4o 4.5.005   2005-05-06 03:10:42]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula133648.yap' closed
```

### 9.2. Networking

From here it's only a small step towards operating db4o over a TCP/IP network. We just specify a port number greater than zero and set up one or more accounts for our client(s).

```
[accessRemoteServer]

ObjectServer server = Db4o.openServer(Util.YapFileName, ServerPort);
    server.grantAccess(ServerUser, ServerPassword);
    try
    {
        ObjectContainer client = Db4o.openClient("localhost",
ServerPort, ServerUser, ServerPassword);
        // Do something with this client, or open more clients
```

```
        client.close();
    }
    finally
    {
        server.close();
    }
```

The client connects providing host, port, user name and password.

```
[queryRemoteServer]

ObjectContainer client = Db4o.openClient("localhost", port, user,
password);
    listResult(client.get(new Car(null)));
    client.close();
```

OUTPUT:

```
[db4o 4.5.005   2005-05-06 03:10:42]
 Server listening on port: '56128'
[db4o 4.5.005   2005-05-06 03:10:42]
 Client 'user' connected.
2
Ferrari[Michael Schumacher/100]/0
BMW[David Coulthard/98]/0
[db4o 4.5.005   2005-05-06 03:10:42]
 Connection closed by client 'user'.
[db4o 4.5.005   2005-05-06 03:10:42]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula133648.yap' close request
[db4o 4.5.005   2005-05-06 03:10:42]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula133648.yap' closed
```

Everything else is absolutely identical to the local server examples above.

```
[demonstrateRemoteReadCommitted]

ObjectContainer client1 = Db4o.openClient("localhost", port, user,
password);
    ObjectContainer client2 = Db4o.openClient("localhost", port,
user, password);
    Pilot pilot = new Pilot("Jenson Button", 97);
    ObjectSet result = client1.get(new Car(null));
    Car car = (Car)result.next();
    car.Pilot = pilot;
    client1.set(car);
    listResult(client1.get(new Car(null)));
    listResult(client2.get(new Car(null)));
    client1.commit();
    listResult(client1.get(new Car(null)));
    listResult(client2.get(new Car(null)));
    client1.close();
    client2.close();
```

OUTPUT:

```
[db4o 4.5.005   2005-05-06 03:10:42]
 Server listening on port: '56128'
[db4o 4.5.005   2005-05-06 03:10:42]
 Client 'user' connected.
[db4o 4.5.005   2005-05-06 03:10:42]
 Client 'user' connected.
2
Ferrari[Jenson Button/97]/0
BMW[David Coulthard/98]/0
2
Ferrari[Michael Schumacher/100]/0
BMW[David Coulthard/98]/0
2
Ferrari[Jenson Button/97]/0
BMW[David Coulthard/98]/0
2
Ferrari[Jenson Button/97]/0
BMW[David Coulthard/98]/0
```

```
[db4o 4.5.005   2005-05-06 03:10:42]
 Connection closed by client 'user'.
[db4o 4.5.005   2005-05-06 03:10:42]
 Connection closed by client 'user'.
[db4o 4.5.005   2005-05-06 03:10:42]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula133648.yap' close request
[db4o 4.5.005   2005-05-06 03:10:42]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula133648.yap' closed
```

```
[demonstrateRemoteRollback]

ObjectContainer client1 = Db4o.openClient("localhost", port, user,
password);
    ObjectContainer client2 = Db4o.openClient("localhost", port,
user, password);
    ObjectSet result = client1.get(new Car(null));
    Car car = (Car)result.next();
    car.Pilot = new Pilot("Someone else", 0);
    client1.set(car);
    listResult(client1.get(new Car(null)));
    listResult(client2.get(new Car(null)));
    client1.rollback();
    client1.ext().refresh(car,2);
    listResult(client1.get(new Car(null)));
    listResult(client2.get(new Car(null)));
    client1.close();
    client2.close();
```

**OUTPUT:**

```
[db4o 4.5.005   2005-05-06 03:10:43]
 Server listening on port: '56128'
[db4o 4.5.005   2005-05-06 03:10:43]
 Client 'user' connected.
[db4o 4.5.005   2005-05-06 03:10:43]
 Client 'user' connected.
2
```

```
Ferrari[Someone else/0]/0
BMW[David Coulthard/98]/0
2
Ferrari[Jenson Button/97]/0
BMW[David Coulthard/98]/0
2
Ferrari[Jenson Button/97]/0
BMW[David Coulthard/98]/0
2
Ferrari[Jenson Button/97]/0
BMW[David Coulthard/98]/0
[db4o 4.5.005   2005-05-06 03:10:43]
 Connection closed by client 'user'.
[db4o 4.5.005   2005-05-06 03:10:43]
 Connection closed by client 'user'.
[db4o 4.5.005   2005-05-06 03:10:43]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula133648.yap' close request
[db4o 4.5.005   2005-05-06 03:10:43]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula133648.yap' closed
```

### 9.3. Out-of-band signalling

Sometimes a client needs to send a special message to a server in order to tell the server to do something.  The server may need to be signalled to perform a defragment or it may need to be signalled to shut itself down gracefully.

This is configured by calling setMessageRecipient(), passing the object that will process client-initiated messages.

```
public void runServer()
    {
        lock(this)
        {
            ObjectServer db4oServer = Db4o.openServer(FILE, PORT);
            db4oServer.grantAccess(USER, PASS);

            // Using the messaging functionality to redirect all
```

```
            // messages to this.processMessage
            db4oServer.ext().configure().setMessageRecipient(this);
            try
            {
                if (! stop)
                {
                    // wait forever for notify() from close()
                    Monitor.Wait(this);
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
            db4oServer.close();
        }
    }
```

The message is received and processed by a processMessage() method:

```
public void processMessage(ObjectContainer con, object message)
    {
        if (message is StopServer)
        {
            close();
        }
    }
```

Db4o allows a client to send an arbitrary signal or message to a server by sending a plain Java object to the server.  The server will receive a callback message, including the object that came from the client. The server can interpret this message however it wants.

```
public static void Main(string[] args)
    {
```

```
        ObjectContainer objectContainer = null;
        try
        {
            // connect to the server
            objectContainer = Db4o.openClient(HOST, PORT, USER,
PASS);
        }
        catch (Exception e)
        {
            Console.WriteLine(e.ToString());
        }

        if (objectContainer != null)
        {
            // get the messageSender for the ObjectContainer
            MessageSender messageSender = objectContainer.ext()
                .configure().getMessageSender();

            // send an instance of a StopServer object
            messageSender.send(new StopServer());

            // close the ObjectContainer
            objectContainer.close();
        }
    }
```

### 9.4. Putting it all together: a simple but complete db4o server

Let's put all of this information together now to implement a simple standalone db4o server with a special client that can tell the server to shut itself down gracefully on demand.

First, both the client and the server need some shared configuration information.  We will provide this using an interface:

```
namespace com.db4o.f1.chapter5
{
    /// <summary>
```

```
    /// Configuration used for StartServer and StopServer.
    /// </summary>
    public class ServerConfiguration
    {
        /// <summary>
        /// the host to be used.
        /// If you want to run the client server examples on two
computers,
        /// enter the computer name of the one that you want to use
as server.
        /// </summary>
        public const string HOST = "localhost";

        /// <summary>
        /// the database file to be used by the server.
        /// </summary>
        public const string FILE = "formula1.yap";

        /// <summary>
        /// the port to be used by the server.
        /// </summary>
        public const int PORT = 4488;

        /// <summary>
        /// the user name for access control.
        /// </summary>
        public const string USER = "db4o";

        /// <summary>
        /// the pasword for access control.
        /// </summary>
        public const string PASS = "db4o";
    }
}
```

Now we'll create the server:

```
using System;
using System.Threading;
using com.db4o;
using com.db4o.messaging;
using j4o.lang;

namespace com.db4o.f1.chapter5
{
    /// <summary>
    /// starts a db4o server with the settings from
ServerConfiguration.
    /// This is a typical setup for a long running server.
    /// The Server may be stopped from a remote location by running
    /// StopServer. The StartServer instance is used as a
MessageRecipient
    /// and reacts to receiving an instance of a StopServer object.
    /// Note that all user classes need to be present on the server
side
    /// and that all possible Db4o.configure() calls to alter the
db4o
    /// configuration need to be executed on the client and on the
server.
    /// </summary>
    public class StartServer : ServerConfiguration, MessageRecipient
    {
        /// <summary>
        /// setting the value to true denotes that the server should
be closed
        /// </summary>
        private bool stop = false;

        /// <summary>
        /// starts a db4o server using the configuration from
        /// ServerConfiguration.
        /// </summary>
        public static void Main(string[] arguments)
        {
            new StartServer().runServer();
        }
```

```
        /// <summary>
        /// opens the ObjectServer, and waits forever until close()
is called
        /// or a StopServer message is being received.
        /// </summary>
        public void runServer()
        {
            lock(this)
            {
                ObjectServer db4oServer = Db4o.openServer(FILE,
PORT);
                db4oServer.grantAccess(USER, PASS);

                // Using the messaging functionality to redirect all
                // messages to this.processMessage
db4oServer.ext().configure().setMessageRecipient(this);
  try
                {
                    if (! stop)
                    {
                        // wait forever for notify() from close()
                        Monitor.Wait(this);
                    }
                }
                catch (Exception e)
                {
                    Console.WriteLine(e.ToString());
                }
                db4oServer.close();
            }
        }

        /// <summary>
        /// messaging callback
        /// see com.db4o.messaging.MessageRecipient#processMessage()
        /// </summary>
        public void processMessage(ObjectContainer con, object
message)
        {
            if (message is StopServer)
            {
```

```
                close();
            }
        }

        /// <summary>
        /// closes this server.
        /// </summary>
        public void close()
        {
            lock(this)
            {
                stop = true;
                Monitor.PulseAll(this);
            }
        }
    }
}
```

And last but not least, the client that stops the server.

```
using System;
using com.db4o;
using com.db4o.messaging;

namespace com.db4o.f1.chapter5
{
    /// <summary>
    /// stops the db4o Server started with StartServer.
    /// This is done by opening a client connection
    /// to the server and by sending a StopServer object as
    /// a message. StartServer will react in it's
    /// processMessage method.
    /// </summary>
    public class StopServer : ServerConfiguration
    {
        /// <summary>
        /// stops a db4o Server started with StartServer.
```

```
        /// </summary>
        /// <exception cref="Exception" />
        public static void Main(string[] args)
        {
            ObjectContainer objectContainer = null;
            try
            {
                // connect to the server
                objectContainer = Db4o.openClient(HOST, PORT, USER,
PASS);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }

            if (objectContainer != null)
            {
                // get the messageSender for the ObjectContainer
                MessageSender messageSender = objectContainer.ext()
                    .configure().getMessageSender();

                // send an instance of a StopServer object
                messageSender.send(new StopServer());

                // close the ObjectContainer
                objectContainer.close();
            }
        }
    }
}
```

## 9.5. Conclusion

That's it, folks. No, of course it isn't. There's much more to db4o we haven't covered yet: schema evolution, custom persistence for your classes, writing your own query objects, etc. The following, more loosely coupled chapters will look into more advanced db4o features.

This tutorial is work in progress. We will successively add chapters and incorporate feedback from the

community into the existing chapters.

We hope that this tutorial has helped to get you started with db4o. How should you continue now?

- Browse the remaining chapters.

-*(Interactive version only)*While this tutorial is basically sequential in nature, try to switch back and forth between the chapters and execute the sample snippets in arbitrary order. You will be working with the same database throughout; sometimes you may just get stuck or even induce exceptions, but you can always reset the database via the console window.

- The examples we've worked through are included in your db4o distribution in full source code. Feel free to experiment with it.

- I you're stuck, see if the FAQ can solve your problem, browse the information on our web site, check if your problem is submitted to Bugzilla or join our newsgroup at news://news.db4odev.com/db4o.users.

### 9.6. Full source

```
namespace com.db4o.f1.chapter5
{
    using System;
    using System.IO;
    using com.db4o;
    using com.db4o.f1;

    public class ClientServerExample : Util
    {
        public static void Main(string[] args)
        {
            File.Delete(Util.YapFileName);
            accessLocalServer();
            File.Delete(Util.YapFileName);
            ObjectContainer db = Db4o.openFile(Util.YapFileName);
            try
            {
                setFirstCar(db);
```

```
            setSecondCar(db);
        }
        finally
        {
            db.close();
        }

        configureDb4o();
        ObjectServer server = Db4o.openServer(Util.YapFileName,
0);
        try
        {
            queryLocalServer(server);
            demonstrateLocalReadCommitted(server);
            demonstrateLocalRollback(server);
        }
        finally
        {
            server.close();
        }

        accessRemoteServer();
        server = Db4o.openServer(Util.YapFileName, ServerPort);
        server.grantAccess(ServerUser, ServerPassword);
        try
        {
            queryRemoteServer(ServerPort, ServerUser,
ServerPassword);
            demonstrateRemoteReadCommitted(ServerPort,
ServerUser, ServerPassword);
            demonstrateRemoteRollback(ServerPort, ServerUser,
ServerPassword);
        }
        finally
        {
            server.close();
        }
    }

    public static void setFirstCar(ObjectContainer db)
    {
```

```
            Pilot pilot = new Pilot("Rubens Barrichello", 99);
            Car car = new Car("BMW");
            car.Pilot = pilot;
            db.set(car);
        }

        public static void setSecondCar(ObjectContainer db)
        {
            Pilot pilot = new Pilot("Michael Schumacher", 100);
            Car car = new Car("Ferrari");
            car.Pilot = pilot;
            db.set(car);
        }

        public static void accessLocalServer()
        {
            ObjectServer server = Db4o.openServer(Util.YapFileName,
0);
            try
            {
                ObjectContainer client = server.openClient();
                // Do something with this client, or open more
clients
                client.close();
            }
            finally
            {
                server.close();
            }
        }

        public static void queryLocalServer(ObjectServer server)
        {
            ObjectContainer client = server.openClient();
            listResult(client.get(new Car(null)));
            client.close();
        }

        public static void configureDb4o()
        {
         Db4o.configure().objectClass(typeof(Car)).updateDepth(3);
```

```
        }

        public static void demonstrateLocalReadCommitted(ObjectServer
server)
        {
            ObjectContainer client1 =server.openClient();
            ObjectContainer client2 =server.openClient();
            Pilot pilot = new Pilot("David Coulthard", 98);
            ObjectSet result = client1.get(new Car("BMW"));
            Car car = (Car)result.next();
            car.Pilot = pilot;
            client1.set(car);
            listResult(client1.get(new Car(null)));
            listResult(client2.get(new Car(null)));
            client1.commit();
            listResult(client1.get(typeof(Car)));
            listRefreshedResult(client2, client2.get(typeof(Car)),
2);
            client1.close();
            client2.close();
        }

        public static void demonstrateLocalRollback(ObjectServer
server)
        {
            ObjectContainer client1 = server.openClient();
            ObjectContainer client2 = server.openClient();
            ObjectSet result = client1.get(new Car("BMW"));
            Car car = (Car)result.next();
            car.Pilot = new Pilot("Someone else", 0);
            client1.set(car);
            listResult(client1.get(new Car(null)));
            listResult(client2.get(new Car(null)));
            client1.rollback();
            client1.ext().refresh(car, 2);
            listResult(client1.get(new Car(null)));
            listResult(client2.get(new Car(null)));
            client1.close();
            client2.close();
        }
```

```
        public static void accessRemoteServer()
        {
            ObjectServer server = Db4o.openServer(Util.YapFileName,
ServerPort);
            server.grantAccess(ServerUser, ServerPassword);
            try
            {
                ObjectContainer client = Db4o.openClient("localhost",
ServerPort, ServerUser, ServerPassword);
                // Do something with this client, or open more
clients
                client.close();
            }
            finally
            {
                server.close();
            }
        }

        public static void queryRemoteServer(int port, string user,
string password)
        {
            ObjectContainer client = Db4o.openClient("localhost",
port, user, password);
            listResult(client.get(new Car(null)));
            client.close();
        }

        public static void demonstrateRemoteReadCommitted(int port,
string user, string password)
        {
            ObjectContainer client1 = Db4o.openClient("localhost",
port, user, password);
            ObjectContainer client2 = Db4o.openClient("localhost",
port, user, password);
            Pilot pilot = new Pilot("Jenson Button", 97);
            ObjectSet result = client1.get(new Car(null));
            Car car = (Car)result.next();
            car.Pilot = pilot;
            client1.set(car);
            listResult(client1.get(new Car(null)));
```

```
            listResult(client2.get(new Car(null)));
            client1.commit();
            listResult(client1.get(new Car(null)));
            listResult(client2.get(new Car(null)));
            client1.close();
            client2.close();
        }

        public static void demonstrateRemoteRollback(int port, string
user, string password)
        {
            ObjectContainer client1 = Db4o.openClient("localhost",
port, user, password);
            ObjectContainer client2 = Db4o.openClient("localhost",
port, user, password);
            ObjectSet result = client1.get(new Car(null));
            Car car = (Car)result.next();
            car.Pilot = new Pilot("Someone else", 0);
            client1.set(car);
            listResult(client1.get(new Car(null)));
            listResult(client2.get(new Car(null)));
            client1.rollback();
            client1.ext().refresh(car,2);
            listResult(client1.get(new Car(null)));
            listResult(client2.get(new Car(null)));
            client1.close();
            client2.close();
        }
    }
}
```

# 10. Constructors

Sometimes you may find that db4o refuses to store instances of certain classes, or appears to store them, but delivers incomplete instances on queries. To understand the problem and the alternative solutions at hand, we'll have to take a look at the way db4o "instantiates" objects when retrieving them from the database.

## 10.1. Instantiating objects

Db4o currently knows three ways of creating and populating an object from the database. The approach to be used can be configured globally and on a per-class basis.

### 10.1.1. Using a constructor

The most obvious way is to call an appropriate constructor. Db4o does *not* require a public or no-args constructor. It can use any constructor that accepts default (null/0) values for all of its arguments without throwing an exception. Db4o will test all available constructors on the class (including private ones) until it finds a suitable one.

What if no such constructor exists?

### 10.1.2. Bypassing the constructor

Db4o can also bypass the constructors declared for this class using platform-specific mechanisms. (For Java, this option is only available on JREs >= 1.4.) This mode allows reinstantiating objects whose class doesn't provide a suitable constructor, However, it will (silently) break classes that rely on the constructor to be executed, for example in order to populate transient members.

*If this option is available in the current runtime environment, it will be the default setting.*

### 10.1.3. Using a translator

If none of the two approaches above is suitable, db4o provides a way to specify in detail how instances of a class should be stored and reinstantiated by implementing the Translator interface and registering this implementation for the offending class.

We'll cover translators in detail in the next chapter .

## 10.2. Configuration

The instantiation mode can be configured globally or on a per class basis.

```
Db4o.configure().callConstructors(true);
```

This will configure db4o to use constructors to reinstantiate any object from the database. (The default is*false*).

```
Db4o.configure().objectClass(Foo.class).callConstructor(true);
```

This will configure db4o to use constructor calls for this class and all its subclasses.

### 10.3. Troubleshooting

At least for development code, it is always a good idea to instruct db4o to check for available constructors at storage time. (If you've configured db4o to use constructors at all.)

```
Db4o.configure().exceptionsOnNotStorable(true);
```

If this setting triggers exceptions in your code, or if instances of a class seem to lose members during storage, check the involved classes (especially their constructors) for problems similar to the ones shown in the following section.

### 10.4. Examples

```
class C1 {
  private String s;

  private C1(String s) {
    this.s=s;
```

```
  }

  public String toString() {
    return s;
  }
}
```

The above class is fine for use with and without callConstructors set.

```
class C2 {
  private transient String x;
  private String s;

  private C2(String s) {
    this.s=s;
    this.x="x";
  }

  public String toString() {
    return s+x.length();
  }
}
```

The above C2 class needs to have callConstructors set to true. Otherwise, since transient members are not stored and the constructor code is not executed, toString() will potentially run into a NullPointerException on x.length().

```
class C3 {
  private String s;
  private int i;

  private C3(String s) {
    this.s=s;
    this.i=s.length();
```

```
  }

  public String toString() {
    return s+i;
  }
}
```

The above C3 class needs to have callConstructors set to false (the default), since the (only) constructor will throw a NullPointerException when called with a null value.

```
class C4 {
  private String s;
  private transient int i;

  private C4(String s) {
    this.s=s;
    this.i=s.length();
  }

  public String toString() {
    return s+i;
  }
}
```

This class cannot be cleanly reinstantiated by db4o: Both approaches will fail, so one has to resort to configuring a translator.

## 11. Translators

In the last chapter we have covered the alternative configurations db4o offers for object reinstantiation. What's left to see is how we can store objects of a class that can't be cleanly stored with either of these approaches.

### 11.1. An example class

For this example we'll be using a hypothetical LocalizedItemList class which binds together culture information with a list of items.

System.Globalization.CultureInfo is particularly interesting because it internally holds a native pointer to a system structure which in turn cannot be cleanly stored by db4o.

```
namespace com.db4o.f1.chapter6
{
    using System.Globalization;

    /// <summary>
    /// A CultureInfo aware list of objects.
    /// CultureInfo objects hold a native pointer to
    /// a system structure.
    /// </summary>
    public class LocalizedItemList
    {
        CultureInfo _culture;
        string[] _items;

        public LocalizedItemList(CultureInfo culture, string[] items)
        {
            _culture = culture;
            _items = items;
        }

        override public string ToString()
        {
            return string.Join(_culture.TextInfo.ListSeparator + " ",
_items);
```

```
        }
    }
}
```

We'll be using this code to store and retrieve and instance of this class with different configuration settings:

```
public static void tryStoreAndRetrieve()
    {
        ObjectContainer db = Db4o.openFile(Util.YapFileName);
        try
        {
            string[] champs = new string[] { "Ayrton Senna", "Nelson
Piquet" };
            LocalizedItemList LocalizedItemList = new
LocalizedItemList(CultureInfo.CreateSpecificCulture("pt-BR"),
champs);
            System.Console.WriteLine("ORIGINAL: {0}",
LocalizedItemList);
            db.set(LocalizedItemList);
        }
        catch (Exception x)
        {
            System.Console.WriteLine(x);
            return;
        }
        finally
        {
            db.close();
        }
        db = Db4o.openFile(Util.YapFileName);
        try
        {
            ObjectSet result = db.get(typeof(LocalizedItemList));
            while (result.hasNext())
            {
                LocalizedItemList LocalizedItemList =
```

```
(LocalizedItemList)result.next();
                System.Console.WriteLine("RETRIEVED: {0}",
LocalizedItemList);
                db.delete(LocalizedItemList);
            }
        }
        finally
        {
            db.close();
        }
    }
```

Let's verify that both approaches to object reinstantiation will fail for this class.

### 11.1.1. Using the constructor

```
[tryStoreWithCallConstructors]

Db4o.configure().exceptionsOnNotStorable(true);
    Db4o.configure().objectClass(typeof(CultureInfo))
        .callConstructor(true);
    tryStoreAndRetrieve();
```

At storage time, db4o tests the only available constructor with null arguments and runs into a NullPointerException, so it refuses to accept our object.

(Note that this test only occurs when configured with exceptionsOnNotStorable - otherwise db4o will silently fail when trying to reinstantiate the object.)

### 11.1.2. Bypassing the constructor

```
[tryStoreWithoutCallConstructors]

Db4o.configure().objectClass(typeof(CultureInfo))
```

```
        .callConstructor(false);
    // trying to store objects that hold onto
    // system resources can be pretty nasty
    // uncomment the following line to see
    // how nasty it can be
    //tryStoreAndRetrieve();
```

This still does not work for our case because the native pointer will definetely be invalid. In fact this example crashes the Common Language Runtime.

### 11.2. The Translator API

So how do we get our object into the database, now that everything seems to fail? Db4o provides a way to specify a custom way of storing and retrieving objects through the ObjectTranslator and ObjectConstructor interfaces.

#### 11.2.1. ObjectTranslator

The ObjectTranslator API looks like this:

```
public Object onStore(ObjectContainer container,
                      Object applicationObject);
public void onActivate(ObjectContainer container,
                       Object applicationObject,
                       Object storedObject);
public Class storedClass ();
```

The usage is quite simple: When a translator is configured for a class, db4o will call its onStore method with a reference to the database and the instance to be stored as a parameter and will store the object returned. This object's type has to be primitive from a db4o point of view and it has to match the type specification returned by storedClass().

On retrieval, db4o will create a blank object of the target class (using the configured instantiation method) and then pass it on to onActivate() along with the stored object to be set up accordingly.

### 11.2.2. ObjectConstructor

However, this will only work if the application object's class provides some way to recreate its state from the information contained in the stored object, which is not the case for CultureInfo.

For these cases db4o provides an extension to the ObjectTranslator interface, ObjectConstructor, which declares one additional method:

```
public Object onInstantiate(ObjectContainer container,
                            Object storedObject);
```

If db4o detects a configured translator to be an ObjectConstructor implementation, it will pass the stored class instance to the onInstantiate() method and use the result as a blank application object to be processed by onActivate().

Note that, while in general configured translators are applied to subclasses, too, ObjectConstructor application object instantiation will not be used for subclasses (which wouldn't make much sense, anyway), so ObjectConstructors have to be configured for the concrete classes.

## 11.3. A translator implementation

To translate CultureInfo instances, we will store only their name since this is enough to recreate them later. Note that we don't have to do any work in onActivate(), since object reinstantiation is already fully completed in onInstantiate().

```
namespace com.db4o.f1.chapter6
{
    using System.Globalization;
    using com.db4o;
    using com.db4o.config;

    public class CultureInfoTranslator : ObjectConstructor
    {
        public object onStore(ObjectContainer container, object
applicationObject)
        {
```

```
            System.Console.WriteLine("onStore for {0}",
applicationObject);
            return ((CultureInfo)applicationObject).Name;
        }

        public object onInstantiate(ObjectContainer container, object
storedObject)
        {
            System.Console.WriteLine("onInstantiate for {0}",
storedObject);
            string name = (string)storedObject;
            return CultureInfo.CreateSpecificCulture(name);
        }

        public void onActivate(ObjectContainer container, object
applicationObject, object storedObject)
        {
            System.Console.WriteLine("onActivate for {0}/{1}",
applicationObject, storedObject);
        }

        public j4o.lang.Class storedClass()
        {
            return j4o.lang.Class.getClassForType(typeof(string));
        }
    }
}
```

Let's try it out:

```
[storeWithTranslator]

Db4o.configure().objectClass(typeof(CultureInfo))
        .translate(new CultureInfoTranslator());
    tryStoreAndRetrieve();
    Db4o.configure().objectClass(typeof(CultureInfo))
        .translate(null);
```

```
OUTPUT:
ORIGINAL: 42/Test: 4
onStore for 42/Test: 4
onInstantiate for [Ljava.lang.Object;@99721d
onActivate for 42/Test: 4 / [Ljava.lang.Object;@1fab88b
RETRIEVED: 42/Test: 4
```

### 11.4. Conclusion

For classes that cannot cleanly be stored and retrieved with db4o's standard object instantiation mechanisms, db4o provides an API to specify custom reinstantiation strategies. These also come in two flavors: ObjectTranslators let you reconfigure the state of a 'blank' application object reinstantiated by db4o, ObjectConstructors also take care of instantiating the application object itself.

### 11.5. Full source

```
namespace com.db4o.f1.chapter6
{
    using System;
    using System.Globalization;
    using com.db4o;
    using com.db4o.f1;

    public class TranslatorExample : Util
    {
        public static void Main(string[] args)
        {
            tryStoreWithCallConstructors();
            tryStoreWithoutCallConstructors();
            storeWithTranslator();
        }

        public static void tryStoreWithCallConstructors()
        {
            Db4o.configure().exceptionsOnNotStorable(true);
            Db4o.configure().objectClass(typeof(CultureInfo))
```

```
                .callConstructor(true);
            tryStoreAndRetrieve();
        }

        public static void tryStoreWithoutCallConstructors()
        {
            Db4o.configure().objectClass(typeof(CultureInfo))
                .callConstructor(false);
            // trying to store objects that hold onto
            // system resources can be pretty nasty
            // uncomment the following line to see
            // how nasty it can be
            //tryStoreAndRetrieve();
        }

        public static void storeWithTranslator()
        {
            Db4o.configure().objectClass(typeof(CultureInfo))
                .translate(new CultureInfoTranslator());
            tryStoreAndRetrieve();
            Db4o.configure().objectClass(typeof(CultureInfo))
                .translate(null);
        }

        public static void tryStoreAndRetrieve()
        {
            ObjectContainer db = Db4o.openFile(Util.YapFileName);
            try
            {
                string[] champs = new string[] { "Ayrton Senna",
"Nelson Piquet" };
                LocalizedItemList LocalizedItemList = new
LocalizedItemList(CultureInfo.CreateSpecificCulture("pt-BR"),
champs);
                System.Console.WriteLine("ORIGINAL: {0}",
LocalizedItemList);
                db.set(LocalizedItemList);
            }
            catch (Exception x)
            {
                System.Console.WriteLine(x);
```

```
                return;
            }
            finally
            {
                db.close();
            }
            db = Db4o.openFile(Util.YapFileName);
            try
            {
                ObjectSet result = db.get(typeof(LocalizedItemList));
                while (result.hasNext())
                {
                    LocalizedItemList LocalizedItemList =
(LocalizedItemList)result.next();
                    System.Console.WriteLine("RETRIEVED: {0}",
LocalizedItemList);
                    db.delete(LocalizedItemList);
                }
            }
            finally
            {
                db.close();
            }
        }
    }
}
```

# 12. Configuration

db4o provides a wide range of configuration methods to request special behaviour. For a complete list of all available methods see the API documentation for the com.db4o.config package/namespace.

Some hints around using configuration calls:

## 12.1. Scope

Configuration calls can be issued to a global VM-wide configuration context with

```
Db4o.configure()
```

and to an open ObjectContainer/ObjectServer with

```
objectContainer.ext().configure()
objectServer.ext().configure()
```

When an ObjectContainer/ObjectServer is opened, the global configuration context is cloned and copied into the newly opened ObjectContainer/ObjectServer. Subsequent calls against the global context with Db4o.configure() have no effect on open ObjectContainers/ObjectServers.

## 12.2. Calling Methods

Many configuration methods have to be called before an ObjectContainer/ObjectServer is opened and will be ignored if they are called against open ObjectContainers/ObjectServers. Some examples:

```
Configuration conf = Db4o.configure();
conf.objectClass(Foo.class).objectField("bar").indexed(true);
conf.objectClass(Foo.class).cascadeOnUpdate();
conf.objectClass(Foo.class).cascadeOnDelete();
conf.objectClass(typeof(System.Drawing.Image))
  .translate(new TSerializable());
conf.generateUUIDs(Integer.MAX_VALUE);
conf.generateVersionNumbers(Integer.MAX_VALUE);
conf.automaticShutDown(false);
conf.lockDatabaseFile(false);
conf.singleThreadedClient(true);
```

```
conf.weakReferences(false);
```

Configurations that influence the database file format will have to take place, before a database is created, before the first #openXXX() call. Some examples:

```
Configuration conf = Db4o.configure();
conf.blockSize(8);
conf.encrypt(true);
conf.password("yourEncryptionPasswordHere");
conf.unicode(false);
```

Configuration settings are **not** stored in db4o database files. Accordingly all configuration methods have to be called **every time** before an ObjectContainer/ObjectServer is opened. For using db4o in client/server mode it is recommended to use the same global configuration on the server and on the client. To set this up nicely it makes sense to create one application class with one method that does all the db4o configuration and to deploy this class both to the server and to all clients.

### 12.3. Further reading

Some configuration switches are discussed in more detail in the following chapters:

Tuning

Indexes

## 13. Indexes

db4o allows to index fields to provide maximum querying performance. To request an index to be created, you would issue the following API method call in your global db4o configuration method before you open an ObjectContainer/ObjectServer:

```
// assuming
class Foo{
  String bar;
}

Db4o.configure().objectClass(Foo.class).objectField("bar").indexed(tr
ue);
```

If the configuration is set in this way, an index on the Foo#bar field will be created (if not present already) the next time you open an
ObjectContainer/ObjectServer and you use the Foo class the first time
in your applcation.

Contrary to all other configuration calls indexes - once created - will remain in a database even if the index configuration call is not issued before opening an ObjectContainer/ObjectServer.

To drop an index you would also issue a configuration call in your db4o configuration method:

```
Db4o.configure().objectClass(Foo.class).objectField("bar").indexed(tr
ue);
```

Actually dropping the index will take place the next time the respective class is used.

db4o will tell you when it creates and drops indexes, if you choose a message level of 1 or higher:

```
Db4o.configure().messageLevel(1);
```

For creating and dropping indexes on large amounts of objects there are two possible strategies:
(1) Import all objects with indexing off, configure the index and reopen the ObjectContainer/ObjectServer.
(2) Import all objects with indexing turned on and commit regularly for a fixed amount of objects (~10,000).

(1) will be faster.
(2) will keep memory consumption lower.

## 14. IDs

The db4o team recommends, not to use object IDs where this is not necessary. db4o keeps track of object identities in a transparent way, by identifying "known" objects on updates. The reference system also makes sure that every persistent object is instantiated only once, when a graph of objects is retrieved from the database, no matter which access path is chosen. If an object is accessed by multiple queries or by multiple navigation access paths, db4o will always return the one single object, helping you to put your object graph together exactly the same way as it was when it was stored, without having to use IDs.

The use of IDs does make sense when object and database are disconnected, for instance in stateless applications.

db4o provides two types of ID systems.

### 14.1. Internal IDs

The internal db4o ID is a physical pointer into the database with only one indirection in the file to the actual object so it is the fastest external access to an object db4o provides. The internal ID of an object is available with

```
objectContainer.ext().getID(object);
```

To get an object for an internal ID use

```
objectContainer.ext().getByID(id);
```

Note that #getByID() does not activate objects. If you want to work with objects that you get with #getByID(), your code would have to make sure the object is activated by calling

```
objectContainer.activate(object, depth);
```

db4o assigns internal IDs to any stored first class object. These internal IDs are guaranteed to be unique within one ObjectContainer/ObjectServer and they will stay the same for every object when an ObjectContainer/ObjectServer is closed and reopened. Internal IDs **will change** when an object is moved from one ObjectContainer to another, as it happens during Defragment .

### 14.2. Unique Universal IDs (UUIDs)

For long term external references and to identify an object even after it has been copied or moved to another ObjectContainer, db4o supplies UUIDs. These UUIDs are not generated by default, since they occupy some space and consume some performance for maintaining their index. UUIDs can be turned on globally or for individual classes:

```
Db4o.configure().generateUUIDs(Integer.MAX_VALUE);
Db4o.configure().objectClass(typeof(Foo)).generateUUIDs(true);
```

The respective methods for working with UUIDs are:

```
ExtObjectContainer#getObjectInfo(Object)
ObjectInfo#getUUID();
ExtObjectContainer#getByUUID(Db4oUUID);
```

## 15. Callbacks

Callback methods are automatically called on persistent objects by db4o during certain database events.

For a complete list of the signatures of all available methods see the com.db4o.ext.ObjectCallbacks interface.

You do not have to implement this interface. db4o recognizes the presence of individual methods by their signature, using reflection. You can simply add one or more of the methods to your persistent classes and they will be called.

Returning false to the #objectCanXxxx() methods will prevent the current action from being taken.

In a client/server environment callback methods will be called on the client with two exceptions: objectOnDelete(), objectCanDelete()

Some possible usecases for callback methods:
- setting default values after refactorings
- checking object integrity before storing objects
- setting transient fields
- restoring connected state (of GUI, files, connections)
- cascading activation
- cascading updates
- creating special indexes

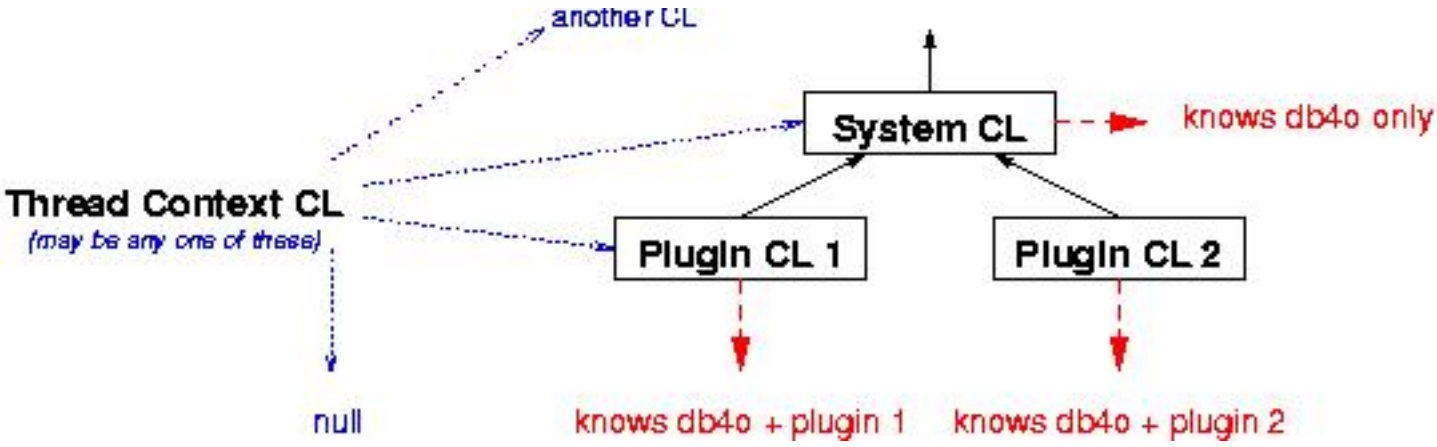
## 16. Classloader issues

Db4o needs to know its own classes, of course, and it needs to know the class definitions of the objects it stores. (In Client/Server mode, both the server and the clients need access to the class definitions.) While this usually is a non-issue with self-contained standalone applications, it can become tricky to ensure this condition when working with plugin frameworks, where one might want to deploy db4o as a shared library for multiple plugins, for example.

### 16.1. Classloader basics

Classloaders are organized in a tree structure, where classloaders deeper down the tree (usually) delegate requests to their parent classloaders and thereby 'share' their parent's knowledge.

A typical situation might look like this:




An in-depth explanation of the workings of classloaders is beyond the scope of this tutorial, of course. Starting points might be found here:

http://www.javaworld.com/javaworld/javaqa/2003-06/01-qa-0606-load.html
http://java.sun.com/developer/technicalArticles/Networking/classloaders/
http://java.sun.com/products/jndi/tutorial/beyond/misc/classloader.html

### 16.2. Configuration

Db4o can be configured to use a user-defined classloader.

```
Db4o.configure().reflectWith(new JdkReflector(classloader));
```

This line will configure db4o to use the provided classloader. Note that, as with most db4o configuration options, this configuration will have to occur before the respective database has been opened.

The usual ways of getting a classloader reference are:

- Using the classloader the class containing the currently executed code was loaded from. (*this.getClass().getClassLoader()*)
- Using the classloader db4o was loaded from. (*Db4o.class.getClassLoader()*)
- Using the classloader your domain classes were loaded from. (*SomeDomainClass.class.getClassLoader()*)
- Using the context classloader that may have been arbitrarily set by the execution environment. (*Thread.currentThread().getContextClassLoader()*).

To choose the right classloader to use, you have to be aware of the classloader hierarchy of your specific execution environment. As a rule of thumb, one should configure db4o to use a classloader as deep/specialized in the tree as possible. In the above example this would be the classloader of the plugin db4o is supposed to work with.

### 16.3. Typical Environments

In your average standalone program you'll probably never have to face these problems, but there are standard framework classics that'll force you to think about these issues.

#### 16.3.1. Servlet container

In a typical servlet container, there will be one or more classloader responsible for internal container classes and shared libraries, and one dedicated classloader per deployed web application. If you deploy db4o within your web application, there should be no problem at all. When used as a shared library db4o has to be configured to use the dedicated web application classloader. This can be done by assigning the classloader of a class that's present in the web application only, or by using the context classloader, since all servlet container implementations we are aware of will set it accordingly.

You will find more detailed information on classloader handling in Tomcat, the reference servlet container implementation, here:

http://jakarta.apache.org/tomcat/tomcat-4.1-doc/class-loader-howto.html

#### 16.3.2. Eclipse

Eclipse uses the system classloader to retrieve its core classes. There is one dedicated classloader per plugin, and the classloader delegation tree will resemble the plugin dependency tree. The context classloader will usually be the system classloader that knows nothing about db4o and your business classes. So the best candidate is the classloader for one of your domain classes within the plugin.

### 16.4. Running without classes

Recently db4o has started to learn to cope with missing class definitions. This is a by-product of the work on our object manager application. However, this feature is still quite restricted (read-only mode, etc.), incomplete and is under heavy development. If you like to play with this feature and help us with your feedback to enhance it, you are welcome, but we strongly recommend not to try to use this for production code of any kind.

## 17. Encryption

db4o provides built-in encryption functionality.

In order to use it, the following two methods have to be called, before a database file is created:

```
Db4o.configure().encrypt(true);
Db4o.configure().password("yourEncryptionPasswordHere");
```

The security standard of the built-in encryption functionality is not very high, not much more advanced than "substract 5 from every byte".

There are 2 reasons for not providing more advanced encryption functionality:
(1) The db4o library is designed to stay small and portable.
(2) The db4o team is determined to avoid problems with U.S. security regulations and export restrictions.

db4o still provides a solution for high-security encryption by allowing any user to choose his own encryption mechanism that he thinks he needs:
The db4o file IO mechanism is pluggable and any fixed-length encryption mechanism can be added. All that needs to be done is to write an IoAdapter plugin for db4o file IO.

This is a lot easier than it sounds. Simply:
- take the sources of com.db4o.io.RandomAccessFileAdapter
- modify the #read() and #write() methods to encrypt and decrypt when bytes are being exchanged with the file
- plug your adapter into db4o with the following method:

```
Db4o.configure().io(new MyEncryptionAdapter());
```

# 18. Tuning

The following is an overview over possible tuning switches that can be set when working with db4o. Users that do not care about performance may like to read this chapter also because it provides a side glance at db4o features with *Alternate Strategies* and some insight on how db4o works.

## 18.1. Discarding Free Space

```
Db4o.configure().discardFreeSpace(byteCount);
```

Recommended settings for byteCount:
- Integer.MAX_VALUE will turn freespace management off
- Moderate range: 10 to 50
- Default built-in setting: 0

*Advantage*
will reduce the RAM memory overhead and the speed loss from maintaining the freespace lists.

*Effect*
When objects are updated or deleted, the space previously occupied in the database file is marked as "free", so it can be reused. db4o maintains two lists in RAM, sorted by address and by size. Adjacent entries are merged. After a large number of updates or deletes have been executed, the lists can become large, causing RAM consumption and performance loss for maintenance. With this method you can specify an upper bound for the byte slot size to discard.

*Alternate Strategies*
Regular defragment will also keep the number of free space slots small. See:

```
com.db4o.tools.Defragment
```

(supplied as source code insrc/com/db4o/tools)
If defragment can be frequently run, it will also reclaim lost space and decrease the database file to the minimum size. Therefore #discardFreeSpace() may be a good tuning mechanism for setups with frequent defragment runs.

### 18.2. Calling constructors

```
Db4o.configure().callConstructors(true);
```

*Advantage*
will configure db4o to use constructors to instantiate objects.

*Effect*
On VMs where this is supported (Sun Java VM > 1.4, .NET, Mono) db4o tries to create instances of objects without calling a constructor. db4o is using reflection for this feature so this may be considerably slower than using a constructor. For the best performance it is recommended to add a public zero-parameter constructor to every persistent class and to turn constructors on.

*Alternate Strategies*
Constructors can also be turned on for individual classes only with

```
Db4o.configure().objectClass(Foo.class).callConstructor(true);
```

There are some classes (e.g. java.util.Calendar) that require a constructor to be called to work. Further details can be found in the chapter on Constructors.

### 18.3. Turning Off Weak References

```
Db4o.configure().weakReferences(false);
```

*Advantage*
will configure db4o to use hard direct references instead of weak references to control instantiated and stored objects.

*Effect*
A db4o database keeps a reference to all persistent objects that are currently held in RAM, whether they were stored to the database in this session or instantiated from the database in this session. This is how db4o can "know" than an object is to be updated: Any "known" object must be an update, any "unknown" object will be stored as "new". (Note that the reference system will only be in place as long as an ObjectContainer is open. Closing and reopening an ObjectContainer will clean the references system of the ObjectContainer and all objects in RAM will be treated as "new" afterwards.) In the

default configuration db4o uses weak references and a dedicated thread to clean them up after objects have been garbage collected by the VM. Weak references need extra ressources and the cleanup thread will have a considerable impact on performance since it has to be synchronized with the normal operations within the ObjectContainer. Turning off weak references will improve speed.
The downside: To prevent memory consumption from growing consistantly, the application has to take care of removing unused objects from the db4o reference system by itself. This can be done by calling

```
ExtObjectContainer.purge(object);
```

*Alternate Strategies*

```
ExtObjectContainer.purge(object);
```

can also be called in normal weak reference operation mode to remove an object from the reference cache. This will help to keep the reference tree as small as possible. After calling #purge(object) an object will be unknown to the ObjectContainer so this feature is also suitable for batch inserts.

### 18.4. Defragment

```
new Defragment().run("db.yap", delete);
```

*Advantage*
It is recommended to run Defragment frequently to reduce the database file size and to remove unused fields and freespace slots.

*Effect*
db4o does not discard fields from the database file that are no longer being used. Within the database file quite a lot of space is used for transactional processing. Objects are always written to a new slot when they are modified. Deleted objects continue to occupy 8 bytes until the next Defragment run. Defragment cleans all this up by writing all objects to a completely new database file. The resulting file will be smaller and faster.

*Alternate Strategies*
Instead of deleting objects it can be an option to mark objects as deleted with a "deleted" boolean field and to clean them out (by not copying them to the new database file) during the Defragment run. Two

advantages: (1) Deleted objects can be restored. (2) In case there are multiple references to a deleted object, none of them would point to null. To clean out objects during the Defragment run, the Defragment source code would have to be modified. com.db4o.tools.Defragment is **only supplied as source code** to encourage embedding maintenance tasks.

### 18.5. No Shutdown Thread

```
Db4o.configure().automaticShutDown(false);
```

*Advantage*
can prevent the creation of a shutdown thread on some platforms.

*Effect*
On some platforms db4o uses a ShutDownHook to cleanly close all database files upon system termination. If a system is terminated without calling ObjectContainer#close() for all open ObjectContainers, these ObjectContainers will still be usable but they will not be able to write back their freespace management system back to the database file. Accordingly database files will be observed to grow.

*Alternate Strategies*
Database files can be reduced to their minimal size with

```
com.db4o.tools.Defragment
```

(supplied as source code in /src/com/db4o/tools)

### 18.6. No callbacks

```
Db4o.configure().callbacks(false);
```

*Advantage*
will prevent db4o from looking for callback methods in all persistent classes on system startup.

*Effect*
Upon system startup, db4o will scan all persistent classes for methods with the same signature as the

methods defined in com.db4o.ext.ObjectCallbacks, even if the interface is not implemented. db4o uses reflection to do so and on constrained environments this can consume quite a bit of time. If callback methods are not used by the application, callbacks can be turned off safely.

*Alternate Strategies*
Class configuration features are a good alternative to callbacks. The most recommended mechanism to cascade updates is:

```
Db4o.configure().objectClass("yourPackage.yourClass").cascadeOnUpdate
(true);
```

## 18.7. No schema changes

```
Db4o.configure().detectSchemaChanges(false);
```

*Advantage*
will prevent db4o from analysing the class structure upon opening a database file.

*Effect*
Upon system startup, db4o will use reflection to scan the structure of all persistent classes. This process can take some time, if a large number of classes are present in the database file. For the best possible startup performance on "warm" database files (all classes already analyzed in a previous startup), this feature can be turned off.

*Alternate Strategies*
Instead of using one database file to store a huge and complex class structure, a system may be more flexible and faster, if multiple database files are used. In a client/server setup, database files can also be switched from the client side with

```
((ExtClient)objectContainer).switchToFile(databaseFile);
```

## 18.8. No lock file thread

```
Db4o.configure().lockDatabaseFile(false);
```

*Advantage*
will prevent the creation of a lock file thread on Java platforms without NIO (< JDK 1.4.1).

*Effect*
If file locking is not available on the system, db4o will regularily write a timestamp lock information to the database file, to prevent other VM sessions from accessing the database file at the same time. Uncontrolled concurrent access would inevitably lead to corruption of the database file. If the application ensures that it can not be started multiple times against the database file, db4o file locking may not be necessary.

*Alternate Strategies*
Database files can safely be opened from multiple sessions in readonly
mode. Use:

```
Db4o.configure().readOnly(true)
```

## 18.9. No test instances

```
Db4o.configure().testConstructors(false);
```

*Advantage*
will prevent db4o from creating a test instance of persistent classes upon opening a database file.

*Effect*
Upon system startup, db4o attempts to create a test instance of all persistent classes, to ensure that a public zero-parameter constructor is present. This process can take some time, if a large number of classes are present in the database file. For the best possible startup performance this feature can be turned off.

*Alternate Strategies*
In any case it's always good practice to create a zero-parameter constructor. If this is not possible because a class from a third party is used, it may be a good idea to write a translator that translates

the third party class to one's own class. The dowload comes with the source code of the preconfigured translators in
src/com/db4o/samples/translators.
The default configuration can be found in the above folder in the file
Default.java/Default.cs
Take a look at the way the built-in translators work to get an idea how to write a translator. It just requires implementing 3 (4 for ObjectConstructors) methods and configuring db4o to use a translator on startup with

```
Db4o.configure().objectClass("yourPackage.yourClass").translate()
```

## 18.10. Increase max database size / block size

```
Db4o.configure().blockSize(newBlockSize);
```

```
Defragment.main(new
```

String[] {"mydb.yap"});

*Effect*
Increasing the block size from the default of 1 to a higher value permits you to store more data in a db4o database.

The block size can be between 1 (the default) and 127, which will give you a maximum of 254 GB of storage.

# 19. Maintenance

db4o is designed to minimize maintenance tasks to the absolute minimum. The stored class schema adapts to the application automatically as it is being developed. db4o "understands" the addition and removal of fields which allows it to continue to work against modified classes without having to reorganize the database file. Internally db4o works with a superset of all class versions previously used.

However there are two recommended maintenance tasks, that can both be fully automated remotely with API calls:

## 19.1. Defragment

Defragment creates a new database file and copies all objects from the current database file to the new database file. All indexes are recreated. The resulting database file will be smaller and faster.

com.db4o.tools.Defragment is **only supplied as source code** to encourage embedding custom maintenance tasks on objects.

## 19.2. Backup

db4o supplies hot backup functionality to backup single-user databases and client-server databases while they are running.

The respective API calls for backups are:

```
objectContainer.ext().backup(String path)
```

```
objectServer.ext().backup(String path)
```

The methods can be called while an ObjectContainer/ObjectServer is open and they will execute with low priority in a dedicated thread, with as little impact on processing performance as possible.

It is recommended to backup the current development state of an application (ideally source code and bytecode) along with the database files since the old code can make it easier to work with the old data.

## 20. Replication

db4o provides replication functionality to periodically synchronize databases that work disconnected from eachother, such as remote autonomous servers or handheld devices synchronizing with central servers.

In order to use replication, the following configuration settings have to be called before a database file is created or opened:

```
Db4o.configure().generateUUIDs(Integer.MAX_VALUE);
Db4o.configure().generateVersionNumbers(Integer.MAX_VALUE);
```

(See the section below on how to enable replication for existing databases that were )

Both settings can also be configured on a per-class basis:

```
Db4o.configure().objectClass(Foo.class).generateUUIDs(true);
Db4o.configure().objectClass(Foo.class).generateVersionNumbers(true);
```

Now suppose we have opened two ObjectContainers from two different databases called "handheld" and "desktop", that we want to replicate. This is how we do it:

```
ReplicationProcess replication =
  desktop.ext().replicationBegin(handheld, new
ReplicationConflictHandler() {
    public Object resolveConflict(
      ReplicationProcess replicationProcess,
      Object a,
      Object b) {
        return a;
      }});
replication.setDirection(desktop, handheld);
```

For conflict resolution the ObjectContainer on which replicationBegin() was called, is treated as container "A", the other one is container "B". Both ObjectContainers are treated equally in all other respects.

db4o replication is bi-directional by default. The setDirection() call above is used to ensure that changes will only be replicated from the "desktop" to the "handheld". In that case, replication is said to be "directed".

A conflict occurs when an object to be replicated has been modified in both ObjectContainers. db4o cannot arbitrarily pick one side, so the ReplicationConflictHandler we passed is called to resolve the conflict. If the ReplicationConflictHandler returns null, no changes are replicated. In the case of directed replication, such as our example above, a conflict also occurs when an object has been modified only in the destination container. In our example, the ReplicationConflictHandler always determines that the object from container "A" (desktop) will "win" the conflict, thus overriding any changes made in container "B" (handheld).

Do all objects always get replicated? No. How do we decide which objects get replicated? Like this:

```
Query q = desktop.query();
 replication.whereModified(q);
 ObjectSet replicationSet = q.execute();
 while (replicationSet.hasNext()) {
    replication.replicate(replicationSet.next());
 }
 replication.commit();
```

That's all there is to it.

We are using a query that will return all objects but we could use any query we like to constrain the objects we want.

Calling whereModified() will add a constraint to the query so that it only returns the objects that have actually been modified since the last replication between both the containers in question.

After replication commit, all modified objects (INCLUDING THE ONES THAT WERE NOT REPLICATED) are considered to be "in sync" and will not show up in future "where modified" queries, unless they are modified again.

## 20.1. Under the Hood

Let's take a look at the necessary configuration calls to tell db4o to generate version numbers and UUIDs:

(1) An object's version number indicates the last time an object was modified. It is the database version at the moment of the modification. The database version starts at zero and is incremented every time a transaction is commited.

(2) UUIDs are object IDs that are unique across all databases created with db4o. That is achieved by having the database's creation timestamp as part of its objects' UUIDs. Manually copying db4o database files can produce duplicate UUIDs, of course.

When the replication process is commited, the lowest database version number among both databases is set to be equal to the highest. After replication commit, therefore, both databases have the same version number and are "in sync".

## 20.2. Replicating Existing Data Files

As we learned in the last sections, Db4o.configure().generateUUIDs() and Db4o.configure().generateVersionNumbers()  (or its objectClass counterparts) must be called before storing any objects to a data file because db4o replication needs object versions and UUIDs to work. This implies that objects in existing data files stored without the correct settings can't be replicated.

Fortunately enabling replication for existing data files is a very simple process:
We just need to use the Defragment tool in com.db4o.tools (source code only) after enabling replication:

```
Db4o.configure().objectClass(Task.class).enableReplication(true);
 new Defragment().run(currentFileName(), true);
```

□
After a successful defragmentation our data files are ready for replication.

# 21. .NET Specific Notes

## 21.1. Enums

When dealing with .NET enumerated types we must face the fact that they are nothing more than glorified integer numbers. In other words, everything we know about how db4o handles integer values also applies to .NET enumerated types: it does not make sense to store, delete or retrieve enumerated values directly but to always use them as fields of other types.

A very important implication of this similarity that might not be entirely obvious is in how db4o will handle enumerated values during a Query By Example. Let's consider the following (WRONG) example:

```
enum DoorState {
  Open,
  Closed
}

class Door {
  DoorState _state;
  public Door(DoorState state) {
    _state = state;
  }
}

class Example {
  public static ObjectSet FindOpenDoors(ObjectContainer container) {
    return container.get(new Door(DoorState.Open));
  }
}
```

The problem is in how the method FindOpenDoors tries to use the enumerated value DoorState.Open to retrieve only the specified subset of Door objects when the query in fact will return ALL Door instances regardless of their _state field value.

This might seem intriguing at first but it's just a matter of understanding two things:

1) QBE works by looking at the given prototype and filtering the system by only those fields which are not set to their default/null/zero value according to its underlying type;

2) the DoorState enum declaration in our first example really means:

```
enum DoorState {
  Open = 0,
  Closed = 1
}
```

which implies that DoorState.Open is the default value for the DoorState type thus excluding it from the filter.

There are two main strategies to cope with this situation:

1) to use the SODA API and recode the Example class as:

```
class Example {
  public static ObjectSet FindOpenDoors(ObjectContainer container) {
    Query query = container.query();
    query.constrain(typeof(Door));
    query.descend("_state").constrain(DoorState.Open);
    return query.execute();
  }
}
```

2) to always reserve the first slot of an enumerate value by either explicitly naming it:

```
enum DoorState {
  Uninitialized,
  Open,
  Closed
}
```

or explicitly setting the value of every element:

```
enum DoorState {
  Open = 1,
  Closed = 2
}
```

## 21.2. Delegates and Events

Db4o rules for delegate fields are very straightforward: **delegates are simply not stored.**

Events and delegates are generally used for binding user interface elements and domain models together. The Db4o team felt that not storing delegate fields by default was more appropriate than opening what could potentially be a very nasty can of worms (just think of a text box bound to a Customer.Changed event).

After careful thought we can easily add delegate persistence to our domain model by either installing translators for the delegate types of interest or reconnecting the necessary objects upon activation using callbacks.

For details see the specific chapters on Translators and Callbacks.

# 22. Object Manager

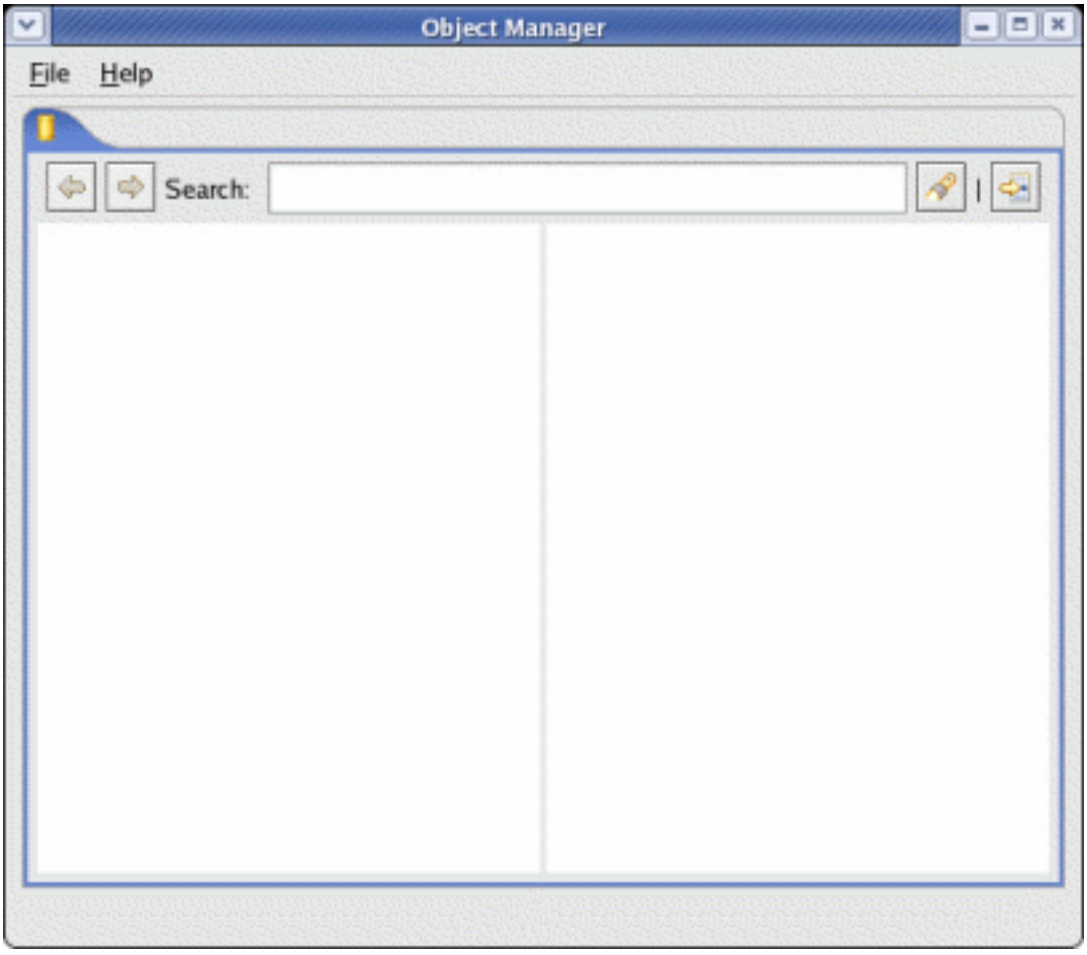
## 22.1. Introduction

The db4o Object Manager provides a simple way to efficiently browse the objects that are stored in your database.  Object Manager currently provides the following features:

- Open either a local database file or a db4o database server
- Browse objects in a database
- Query for objects using a simple graphical query by example.

## 22.2. Object Manager Tour

Upon opening, the Object Manager will look like the following:




At this point, you can either

- Open a db4o database file
- Open a connection to a db4o database server

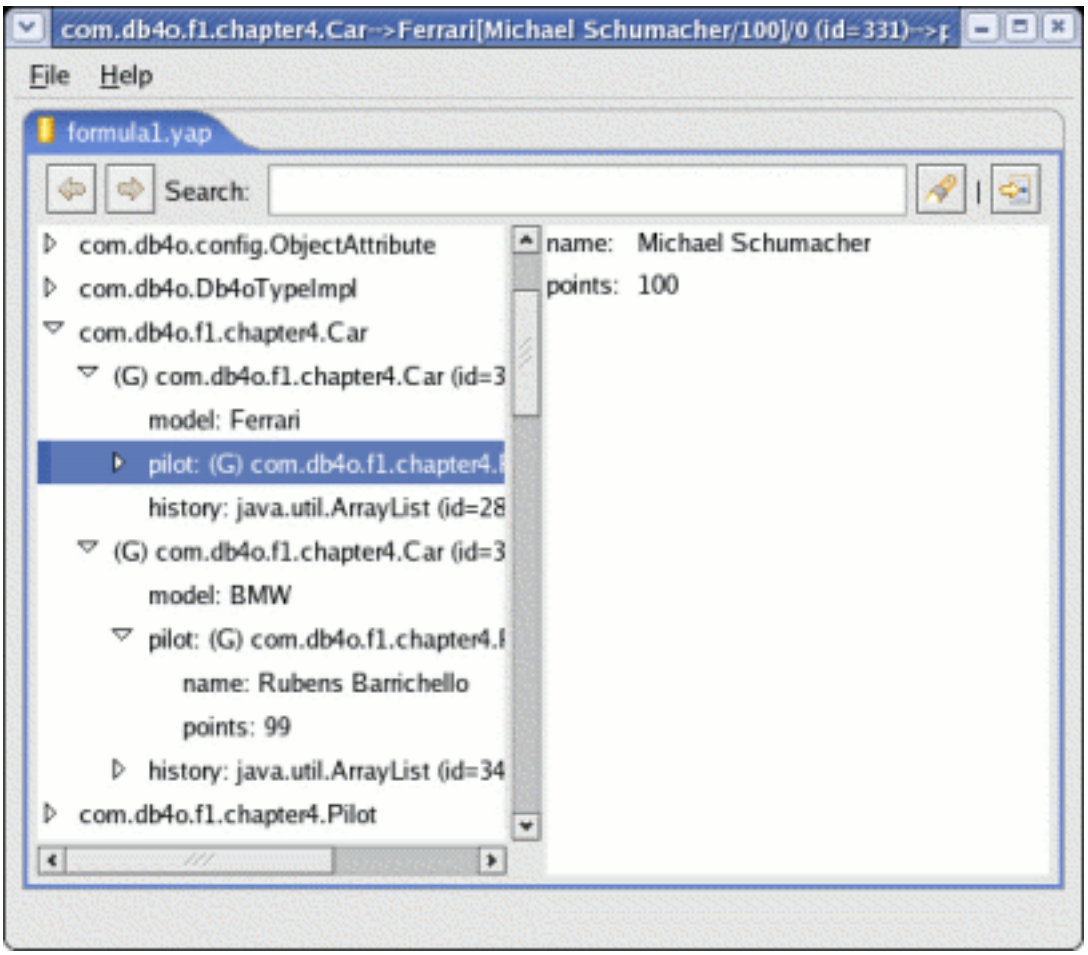
In order to open a db4o database file, simply choose "File | Open..." and choose the db4o database file

to open.

In order to open a connection to a db4o database server, choose "File | Connect to server...", then enter a host name, port number, username, and password into the resulting dialog box.

When the database is open, the Object Manager will list all classes stored in the database in the left-hand tree pane. Expanding a tree item will show instances of objects in the class, then fields within the instance, etc. The right-hand pane will show the next level of detail for the item that is currently selected in the tree.

For example, loading the formula1 database created by chapter 4 of this tutorial produces the following:




### 22.2.1. Generic reflector versus the JDK reflector

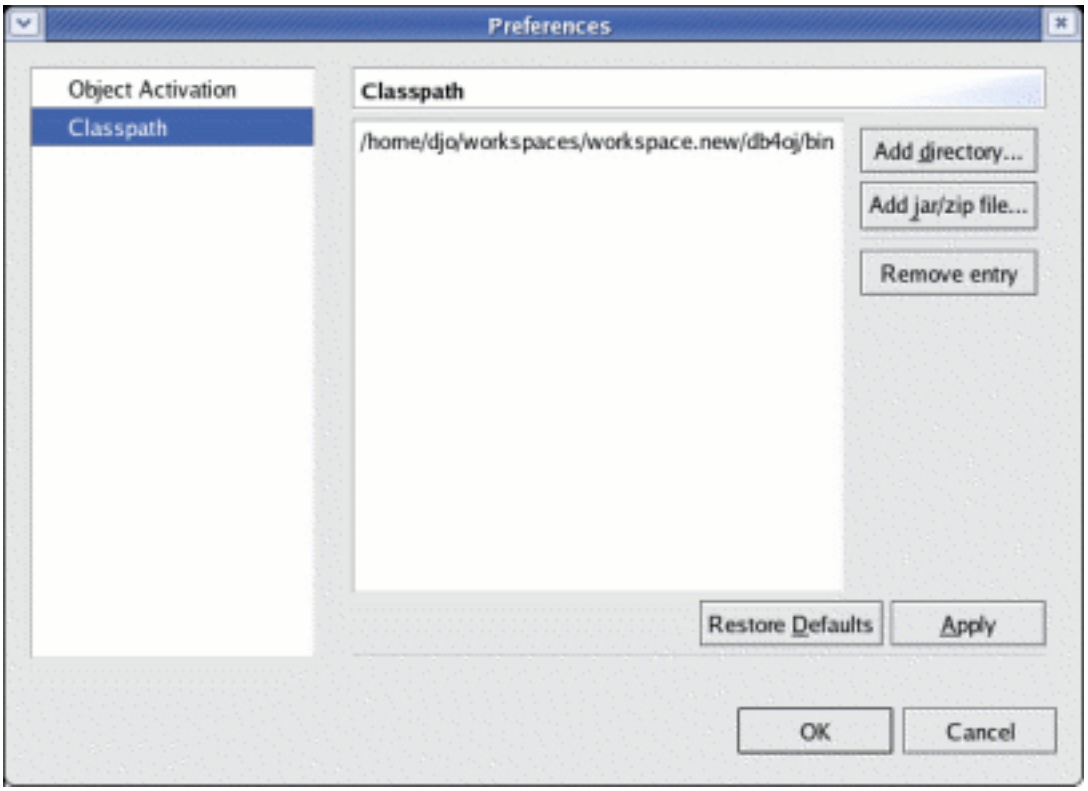
You will notice in the screen shot that all objects have (G) in front of them. This is because the Object Manager does not know the correct classpath in order to load the actual Java classes that these objects represent, so it is displaying them generically (using the generic reflector).

The Object Manager can display some data types more naturally if it has the class definitions available, so it provides a way to extend its classpath at runtime so that it can find the Java or .NET class definitions.  This is done through the preferences dialog, accessed through "File | Preferences..."

Let's add the classpath of the db4o chapter 4 tutorial classes:
- Choose "File | Preferences..."
- Select the "Classpath" page of the preferences dialog box
- Choose the "Add directory..."
- Select the directory you have selected as your output directory for compiling the db4o tutorial files and click OK.

(You can add a Jar or zip file similarly.)

Your preferences dialog should look something like the following when you are done:




Click OK to close the Preferences dialog box.  Now your tree will be refreshed, and you will notice that all of the (G)'s are gone.  All of your preference settings, including your preferred classpath, are stored in a db4o database called .objectmanager.yap in your home directory.

### 22.2.2. Querying for objects

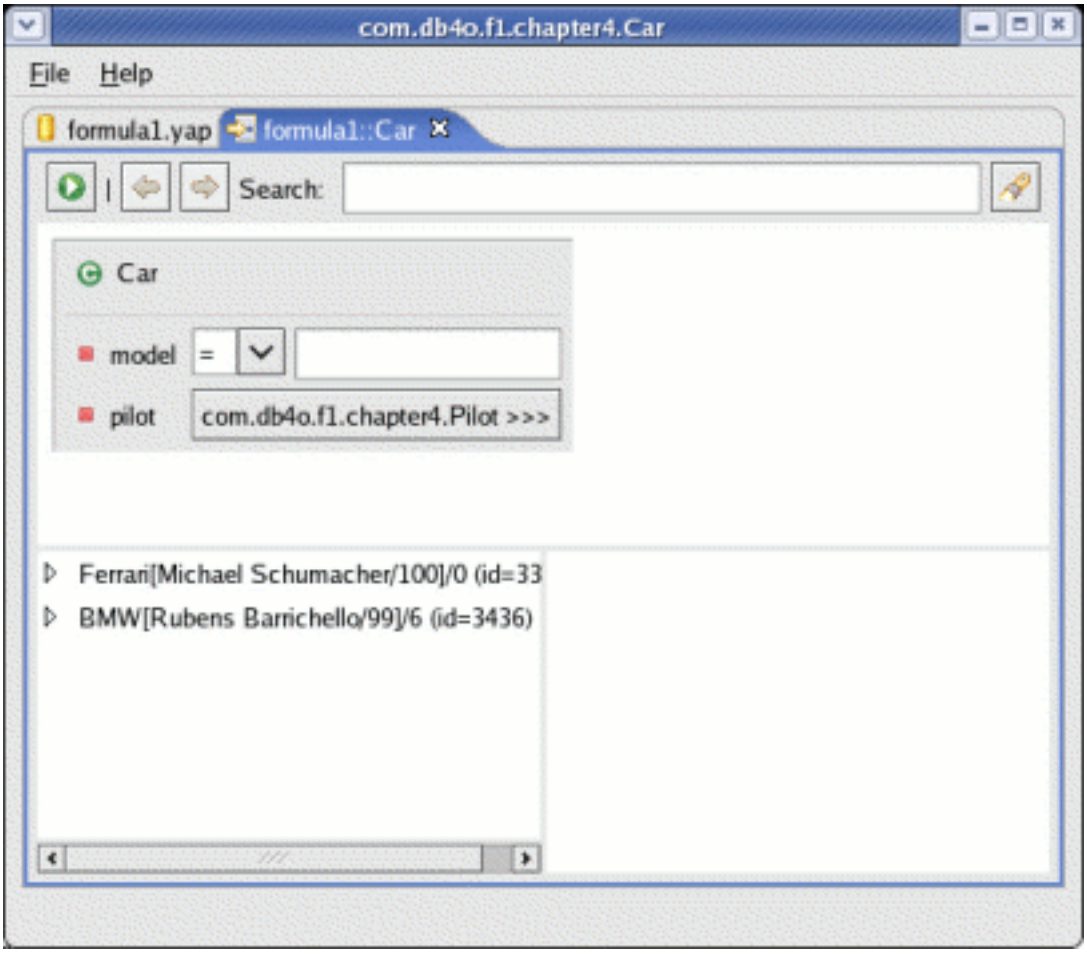
You may now want to query your object database for objects based on some criteria.  This will work

using either the generic or the JDK reflector.

You can open a query editor using any of the following methods:
- Double-click a class name in the tree view.
- Click the query button at the right side of the tool bar.
- Choose "File | Query..." from the menu.

If you choose either of the latter two options, you will then be presented with a dialog box allowing you to choose a class to query.  Select a class and click OK to proceed.  Your new query will open in a new tab.

Let's query for Car objects in our formula1.yap file.  Open a query editor using your favorite of the methods listed above.  By default it will look like the following:




You may now query for a model name, or you can descend on the "pilot" field by clicking the button next to the pilot field to expand the query editor to include a Pilot object.  If you accidentally expand a class that you didn't intend to expand, that isn't a problem.  Just leave all of the fields blank and they will be ignored.

At this point, you can query for any combination of field values simply by filling appropriate values into each field.  You can change the constraint type using the combo boxes next to each field.  And you can run the query using the green "run" button at the top-left of the tool bar.

## 22.3. Known bugs and limitations

The following are known bugs and limitations in Object Manager and their workarounds:
- Object Manager currently operates in read-only mode only.
- If a database includes fields that have been renamed, Object Manager correctly browses both the old and new versions of the fields.  However, queries on fields that have been refactored will randomly select the old or the new version of the field.  The workaround is to browse a backup or defragmented database.
- The Generic Reflector still has trouble expanding certain types of objects. Currently, we recommend running Object Manager with all classes available at all times by using the Classpath preference page to set an appropriate classpath.

If something goes wrong when using Object Manager, there almost certainly will be detailed error messages in the Object Manager log file.  This file is named .objectmanager.log and is stored in your user's home directory.  If you post a question on the db4o newsgroups concerning a crash or failure in Object Manager, please also attach a copy of your Object Manager log file, so that we can efficiently diagnose the problem.  This file is overwritten each time you run Object Manager, so please make a copy of it after an error occurs so that this diagnostic information is not lost.

# 23. License

db4objects Inc. supplies the object database engine db4o under a dual licensing regime:

## 23.1. General Public License (GPL)

db4o is free to be used:
- for development,
- in-house as long as no deployment to third parties takes place,
- together with works that are placed under the GPL themselves.

You should have received a copy of the GPL in the file GPL.txt together with the db4o distribution.

If you have questions about when a commercial license is required, please
read our GPL Interpretation policy for further detail, available at:

http://www.db4o.com/about/company/legalpolicies/gplinterpretation.aspx

## 23.2. Commercial License

For incorporation into own commercial products and for use together with redistributed software that is not placed under the GPL, db4o is also available under a commercial license.

Visit the purchasing area on the db4o website or contact db4o sales for licensing terms and pricing.

## 24. Contacting db4objects Inc.

**db4objects Inc.**
1900 South Norfolk Street
Suite 350
San Mateo, CA, 94403
USA

**Phone**
+1 (650) 577-2340

**Fax**
+1 (650) 577-2341

**General Enquiries**
info@db4o.com

**Sales**
fill out our sales contact form on the db4o website
or
sales@db4o.com

**Careers**
career@db4o.com

**Partnering**
partner@db4o.com

**Support**
support@db4o.com
or post to our newsgroup
news://news.db4odev.com/db4o.users