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.

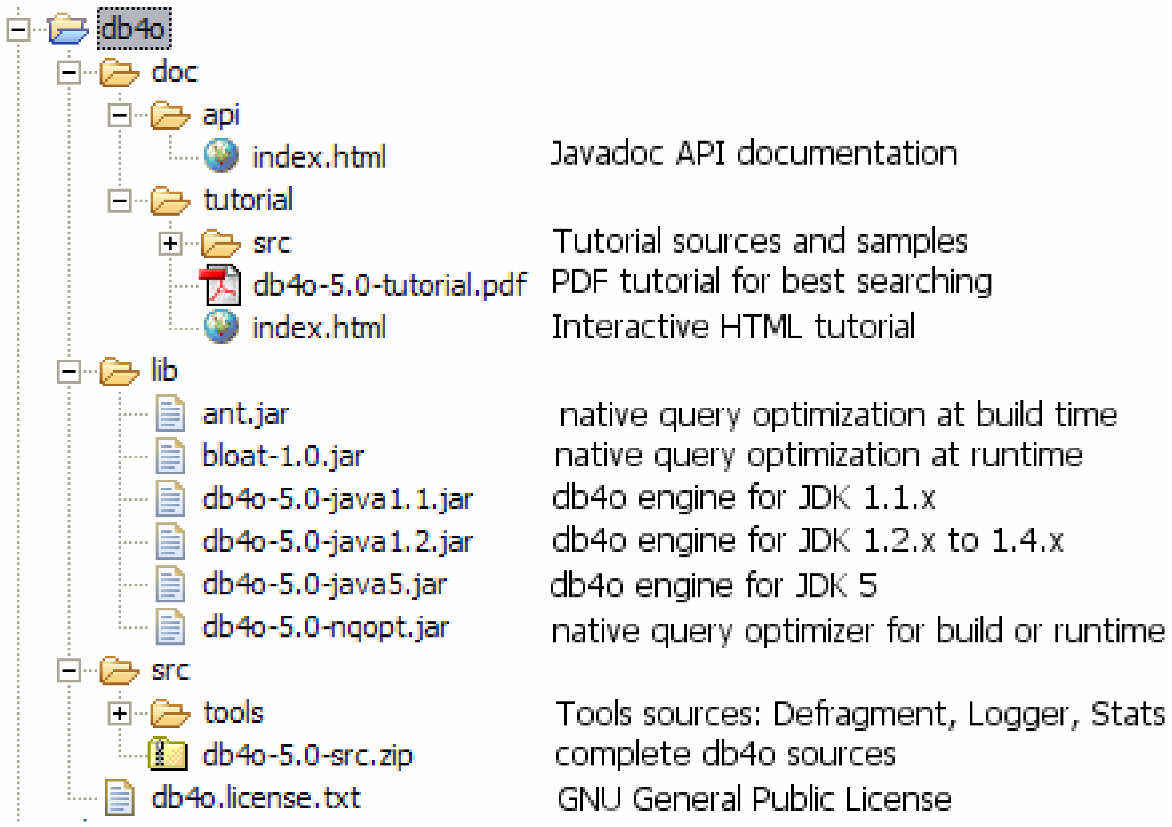
You are invited to join the db4o community in the public db4o forums to ask for help at any time. You may also find the db4o knowledgebase helpful for keyword searches.

Java, .NET and Mono

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

Download Contents

The db4o Java distribution comes as one zip file, db4o-5.0-java.zip. When you unzip this file, you get the following directory structure:



```
db4o
  doc
    api
      index.html              Javadoc API documentation
    tutorial
      src                     Tutorial sources and samples
      db4o-5.0-tutorial.pdf   PDF tutorial for best searching
      index.html              Interactive HTML tutorial
  lib
    ant.jar                   native query optimization at build time
    bloat-1.0.jar             native query optimization at runtime
    db4o-5.0-java1.1.jar      db4o engine for JDK 1.1.x
    db4o-5.0-java1.2.jar      db4o engine for JDK 1.2.x to 1.4.x
    db4o-5.0-java5.jar        db4o engine for JDK 5
    db4o-5.0-nqopt.jar        native query optimizer for build or runtime
  src
    tools                     Tools sources: Defragment, Logger, Stats
    db4o-5.0-src.zip          complete db4o sources
  db4o.license.txt            GNU General Public License
```

db4o-5.0/doc/tutorial/index.html

This is the interactive HTML tutorial. Examples can be run "live" against a db4o database from within the browser. In order to use the interactive functionality a Java JRE 1.3 or above needs to be installed and integrated into the browser. Java security settings have to allow applets to be run.

db4o-5.0/doc/tutorial/db4o-5.0-tutorial.pdf

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

db4o-5.0/doc/api/index.html

The API documentation for db4o is supplied as JavaDocs HTML files. While you read through this tutorial it may be helpful to look into the API documentation occasionaly.

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 jar file. This is all that you need to program against. The versions supplied with the distribution can be found in /db4o-5.0/lib/.

db4o-5.0-java1.1.jar

will run with most Java JDKs that supply JDK 1.1.x functionality such as reflection and Exception handling. That includes many IBM J9 configurations, Symbian and Savaje.

db4o-5.0-java1.2.jar

is built for all Java JDKs between 1.2 and 1.4.

db4o-5.0-java5.jar

is built for Java JDK 5.

1.2. Installation

If you add one of the above db4o-*.jar files to your CLASSPATH db4o is installed. In case you work with an integrated development environment like Eclipse (We really recommend Eclipse, it's also free.) you would copy the db4o-*.jar to a /lib/ folder under your project and add db4o to your project as a library. (You only need to copy the one jar file for the distribution you are targeting.)

Here is how to add the db4o library to an Eclipse project
- create a folder named "lib" under your project directory, if it doesn't exist yet
- copy db4o-*.jar to this folder
- Right-click on your project in the Package Explorer and choose "refresh"
- Right-click on your project in the Package Explorer again and choose "properties"
- select "Java Build Path" in the treeview on the left
- select the "Libraries" tabpage.
- click "Add Jar"
- the "lib" folder should appear below your project
- choose db4o-*.jar in this folder
- hit OK twice

1.3. db4o Object Manager

db4o Object Manager is a GUI tool to browse and query the contents of any db4o database file. Object Manager has to be downloaded seperately from the main db4o distributions. Please visit the db4o Download Center and choose the installation appropriate for your system. The following distributions are currently available:

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

1.4. API Overview

Do not forget the API documentation while reading through this tutorial. It provides an organized view of the API, looking from a package perspective and you may find related functionality to the theme you are currently reading up on.

For starters, the packages com.db4o and com.db4o.query are all that you need to worry about.

com.db4o

The com.db4o package contains almost all of the functionality you will commonly need when using db4o. Two objects of note are com.db4o.Db4o, and the com.db4o.ObjectContainer interface.

The com.db4o.Db4o factory is your starting point. Static methods in this class allow you to open a database file, start a server, or connect to an existing server. It also lets you configure the db4o environment before opening a database.

The most important interface, and the one that you will be using 99% of the time is com.db4o.ObjectContainer: This is your db4o database.
- An ObjectContainer can either be a database in single-user mode or a client connection 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, and is able to achieve a high level of performance.
- ObjectContainers are intended to be kept open as long as you work against them. When you close an ObjectContainer, all database references to objects in RAM will be discarded.

com.db4o.ext

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 packages , 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.
- It is an example of how a lightweight version of db4o could look.

Every com.db4o.ObjectContainer object is also an com.db4o.ext.ExtObjectContainer. You can cast it to ExtObjectContainer or you can use the to get to the advanced features.

com.db4o.config

The com.db4o.config package contains types and classes necessary to configure db4o. The objects and interfaces within are discussed in the Configuration section.

com.db4o.query

The com.db4o.query package contains the Predicate class to construct Native Queries. The Native Query interface is the primary db4o querying interface and should be preferred over the Query API.

2. First Steps

Let's get started as simple as possible. We are going to demonstrate 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 class to hold our data. It looks like this:

```
package com.db4o.f1.chapter1;

public class Pilot {
    private String name;
    private int points;

    public Pilot(String name,int points) {
        this.name=name;
        this.points=points;
    }

    public int getPoints() {
        return points;
    }

    public void addPoints(int points) {
        this.points+=points;
    }

    public String getName() {
        return name;
    }

    public String toString() {
        return name+"/"+points;
    }
}
```

Notice that this class does not contain any db4o-related code.

2.1. Opening the database

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 represents "The Database", and will be your primary interface to db4o. Closing the container with the #.close() method will close the database file and 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, and stores the reference in a variable named 'db'.

2.2. Storing objects

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

```
[storeFirstPilot]

Pilot pilot1=new Pilot("Michael Schumacher",100);
db.set(pilot1);
System.out.println("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);
System.out.println("Stored "+pilot2);
```

OUTPUT:

```
Stored Rubens Barrichello/99
```

2.3. Retrieving objects

db4o supplies three different quering systems, *Query by Example* (QBE), *Native Queries* (NQ) and the *SODA Query API* (SODA). In this first example we will introduce QBE. Once you are familiar with storing objects, we encourage you to use Native Queries, the main db4o querying interface.

When using Query-By-Example, you create a prototypical object for db4o to use as an example of what you wish to retrieve. db4o will retrieve all objects of the given type that contain the same (non-default) field values as the example. The results will be returned as an ObjectSet instance. We will use a convenience method 'listResult' to display the contents of our results:

```
public static void listResult(ObjectSet result) {
    System.out.println(result.size());
    while(result.hasNext()) {
        System.out.println(result.next());
    }
}
```

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

```
[retrieveAllPilotQBE]

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

OUTPUT:

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

Note that we specify 0 points, but our results were not constrained to only those Pilots with 0 points; 0 is the default value for int fields.

db4o also supplies a shortcut to retrieve all instances of a class:

```
[retrieveAllPilots]

ObjectSet result=db.get(Pilot.class);
listResult(result);
```

OUTPUT:

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

For JDK 5 there also is a generics shortcut, using the query method:

```
List <Pilot> pilots = db.query(Pilot.class);
```

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
```

And to query for Pilots with a specific number of 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. They will be covered in more depth in later chapters.

2.4. Updating objects

Updating objects is just as easy as storing them. In fact, you use the same set() method to update your objects: just call set() again after modifying any object.

```
[updatePilot]

ObjectSet result=db.get(new Pilot("Michael Schumacher",0));
Pilot found=(Pilot)result.next();
found.addPoints(11);
db.set(found);
System.out.println("Added 11 points for "+found);
retrieveAllPilots(db);
```

OUTPUT:

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

Notice that we query for the object first. This is an importaint point. When you call set() to modify a stored object, if the object is not 'known' (having been previously stored or retrieved during the current session), db4o will insert a new object. db4o does this because it does not automatically match up objects to be stored, with objects previously stored. It assumes you are inserting a second object which happens to have the same field values.

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

2.5. 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);
System.out.println("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);
System.out.println("Deleted "+found);
retrieveAllPilots(db);
```

OUTPUT:

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

Please check the deletion with the retrieval examples above.

As with updating objects, 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.

2.6. 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 .

2.7. Full source

```java
package com.db4o.f1.chapter1;



import java.io.File;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.f1.Util;



public class FirstStepsExample extends Util {
    public static void main(String[] args) {
        new File(Util.YAPFILENAME).delete();
        accessDb4o();
        new File(Util.YAPFILENAME).delete();
        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);
        System.out.println("Stored "+pilot1);
    }

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

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

    public static void retrieveAllPilots(ObjectContainer db) {
        ObjectSet result=db.get(Pilot.class);
        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);
        System.out.println("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);
        System.out.println("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);
        System.out.println("Deleted "+found);
        retrieveAllPilots(db);
    }
}
```

3. Querying

db4o supplies three querying systems, Query-By-Example (QBE) Native Queries (NQ), and the SODA API. In the previous chapter, you were briefly introduced to *Query By Example*(QBE).

Query-By-Example (QBE) is appropriate as a quick start for users who are still acclimating to storing and retrieving objects with db4o.

Native Queries (NQ) are the main db4o query interface, recommended for general use.

SODA is the underlying internal API. It is provided for backward compatibility and it can be useful for dynamic generation of queries, where NQ are too strongly typed.

3.1. Query by Example (QBE)

When using *Query By Example* (QBE) you provide db4o with a template object. db4o will return all of the objects which match all non-default field values. This is done via reflecting all of the fields and building a query expression where all non-default-value fields are combined with AND expressions. Here's an example from the previous chapter:

```
[retrievePilotByName]

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

Querying this way has some obvious limitations:
- db4o must reflect all members of your example object.
- You cannot perform advanced query expressions. (AND, OR, NOT, etc.)
- You cannot constrain on values like 0 (integers), "" (empty strings), or nulls (reference types) because they would be interpreted as unconstrained.
- You need to be able to create objects without initialized fields. That means you can not initialize fields where they are declared. You can not enforce contracts that objects of a class are only allowed in a well-defined initialized state.
- You need a constructor to create objects without initialized fields.

To get around all of these constraints, db4o provides the Native Query (NQ) system.

3.2. Native Queries

Wouldn't it be nice to pose queries in the programming language that you are using? Wouldn't it be nice if all your query code was 100% typesafe, 100% compile-time checked and 100% refactorable? Wouldn't it be nice if the full power of object-orientation could be used by calling methods from within queries? Enter Native Queries.

Native queries are the main db4o query interface and they are the recommended way to query databases from your application. Because native queries simply use the semantics of your programming language, they are perfectly standardized and a safe choice for the future.

Native Queries are available for all platforms supported by db4o.

3.2.1. Concept

The concept of native queries is taken from the following two papers:

- Cook/Rosenberger, Native Queries for Persistent Objects, A Design White Paper
- Cook/Rai, Safe Query Objects: Statically Typed Objects as Remotely Executable Queries

3.2.2. Principle

Native Queries provide the ability to run one or more lines of code against all instances of a class. Native query expressions should return true to mark specific instances as part of the result set. db4o will attempt to optimize native query expressions and run them against indexes and without instantiating actual objects, where this is possible.

3.2.3. Simple Example

Let's look at how a simple native query will look like in some of the programming languages and dialects that db4o supports:

C# .NET 2.0

```
IList <Pilot> pilots = db.Query <Pilot> (delegate(Pilot pilot) {
    return pilot.Points == 100;
});
```

Java JDK 5

```
List <Pilot> pilots = db.query(new Predicate<Pilot>() {
    public boolean match(Pilot pilot) {
        return pilot.getPoints() == 100;
    }
});
```

Java JDK 1.2 to 1.4

```
List pilots = db.query(new Predicate() {
    public boolean match(Pilot pilot) {
        return pilot.getPoints() == 100;
    }
});
```

Java JDK 1.1

```
ObjectSet pilots = db.query(new PilotHundredPoints());

public static class PilotHundredPoints extends Predicate {
    public boolean match(Pilot pilot) {
        return pilot.getPoints() == 100;
    }
}
```

C# .NET 1.1

```
IList pilots = db.Query(new PilotHundredPoints());

public class PilotHundredPoints : Predicate {
    public boolean Match(Pilot pilot) {
        return pilot.Points == 100;
    }
}
```

VB .NET 1.1

```
Dim pilots As IList = db.Query(new PilotHundredPoints())

Public Class PilotHundredPoints
    Inherits Predicate
    Public Function Match (pilot As Pilot) as Boolean
        If pilot.Points = 100 Then
            Return True
        Else
            Return False
    End Function
End Class
```

A side note on the above syntax:
For all dialects without support for generics, Native Queries work by convention. A class that extends the com.db4o.Predicate class is expected to have a boolean #match() or #Match() method with one parameter to describe the class extent:

```
boolean match(Pilot candidate);
```

When using native queries, don't forget that modern integrated development environments (IDEs) can do all the typing work around the native query expression for you, if you use templates and autocompletion.

Here is how to configure a Native Query template with Eclipse 3.1:
From the menu, choose Window + Preferences + Java + Editor + Templates + New
As the name type "nq". Make sure that "java" is selected as the context on the right. Paste the following into the pattern field:

```
List <${extent}> list = db.query(new Predicate <${extent}> () {
    public boolean match(${extent} candidate){
        return true;
    }
});
```

Now you can create a native query with three keys: n + q + Control-Space.
Similar features are available in most modern IDEs.

3.2.4. Advanced Example

For complex queries, the native syntax is very precise and quick to write. Let's compare to a SODA query that finds all pilots with a given name or a score within a given range:

```
[retrieveComplexSODA]

Query query=db.query();
query.constrain(Pilot.class);
Query pointQuery=query.descend("points");
query.descend("name").constrain("Rubens Barrichello")
    .or(pointQuery.constrain(new Integer(99)).greater()
        .and(pointQuery.constrain(new Integer(199)).smaller()));
ObjectSet result=query.execute();
listResult(result);
```

OUTPUT:

```
0
```

Here is how the same query will look like with native query syntax, fully accessible to autocompletion, refactoring and other IDE features, fully checked at compile time:

C# .NET 2.0

```
IList <Pilot> result = db.Query<Pilot> (delegate(Pilot pilot) {
    return pilot.Points > 99
        && pilot.Points < 199
        || pilot.Name == "Rubens Barrichello";
});
```

Java JDK 5

```
List <Pilot> result = db.query(new Predicate<Pilot>() {
    public boolean match(Pilot pilot) {
        return pilot.getPoints() > 99
            && pilot.getPoints() < 199
            || pilot.getName().equals("Rubens Barrichello");
   }
});
```

3.2.5. Arbitrary Code

Basically that's all there is to know about native queries to be able to use them efficiently. In principle you can run arbitrary code as native queries, you just have to be very careful with side effects - especially those that might affect persistent objects.

Let's run an example that involves some more of the language features available.

```
[retrieveArbitraryCodeNQ]

final int[] points={1,100};
ObjectSet result=db.query(new Predicate() {
    public boolean match(Pilot pilot) {
        for(int i=0;i<points.length;i++) {
            if(pilot.getPoints()==points[i]) {
                return true;
            }
        }
        return pilot.getName().startsWith("Rubens");
    }
});
listResult(result);
```

OUTPUT:

```
0
```

3.2.6. Native Query Performance

One drawback of native queries has to be pointed out: Under the hood db4o tries to analyze native queries to convert them to SODA. This is not possible for all queries. For some queries it is very difficult to analyze the flowgraph. In this case db4o will have to instantiate some of the persistent objects to actually run the native query code. db4o will try to analyze parts of native query expressions to keep object instantiation to the minimum.

The development of the native query optimization processor will be an ongoing process in a close dialog with the db4o community. Feel free to contribute your results and your needs by providing feedback to our db4o forums.
The current state of the query optimization process is detailed in the chapter on Native Query Optimization

With the current implementation, all above examples will run optimized, except for the "Arbitrary Code" example - we are working on it.

3.2.7. Full source

```
package com.db4o.f1.chapter1;

import com.db4o.*;
import com.db4o.f1.*;
import com.db4o.query.*;

public class NQExample extends Util {

    public static void main(String[] args) {
        ObjectContainer db=Db4o.openFile(Util.YAPFILENAME);
        try {
            storePilots(db);
            retrieveComplexSODA(db);
            retrieveComplexNQ(db);
            retrieveArbitraryCodeNQ(db);
            clearDatabase(db);
        }
        finally {
            db.close();
        }
    }

    public static void storePilots(ObjectContainer db) {
        db.set(new Pilot("Michael Schumacher",100));
        db.set(new Pilot("Rubens Barrichello",99));
    }

    public static void retrieveComplexSODA(ObjectContainer db) {
        Query query=db.query();
        query.constrain(Pilot.class);
        Query pointQuery=query.descend("points");
        query.descend("name").constrain("Rubens Barrichello")
            .or(pointQuery.constrain(new Integer(99)).greater()
                .and(pointQuery.constrain(new
Integer(199)).smaller()));
        ObjectSet result=query.execute();
        listResult(result);
    }

    public static void retrieveComplexNQ(ObjectContainer db) {
        ObjectSet result=db.query(new Predicate() {
            public boolean match(Pilot pilot) {
                return pilot.getPoints()>99
                    && pilot.getPoints()<199
                    || pilot.getName().equals("Rubens Barrichello");
            }
        });
        listResult(result);
    }

    public static void retrieveArbitraryCodeNQ(ObjectContainer db) {
        final int[] points={1,100};
        ObjectSet result=db.query(new Predicate() {
            public boolean match(Pilot pilot) {
                for(int i=0;i<points.length;i++) {
                    if(pilot.getPoints()==points[i]) {
                        return true;
                    }
                }
                return pilot.getName().startsWith("Rubens");
            }
        });
        listResult(result);
    }

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

3.3. SODA Query API

The SODA query API is db4o's low level querying API, allowing direct access to nodes of query graphs. Since SODA uses strings to identify fields, it is neither perfectly typesafe nor compile-time checked and it also is quite verbose to write.

For most applications Native Queries will be the better querying interface.

However there can be applications where dynamic generation of queries is required, that's why SODA is explained here.

First of all we need some pilots in our database again.

```
[storeFirstPilot]

Pilot pilot1=new Pilot("Michael Schumacher",100);
db.set(pilot1);
System.out.println("Stored "+pilot1);
```

OUTPUT:

```
Stored Michael Schumacher/100
```

```
[storeSecondPilot]

Pilot pilot2=new Pilot("Rubens Barrichello",99);
db.set(pilot2);
System.out.println("Stored "+pilot2);
```

OUTPUT:

```
Stored Rubens Barrichello/99
```

3.3.1. Simple queries

Let's see how our familiar QBE queries are expressed with SODA. A new Query object is created through the #query() method of the ObjectContainer and we can add Constraint instances to it. To find all Pilot instances, we constrain the query with the Pilot class object.

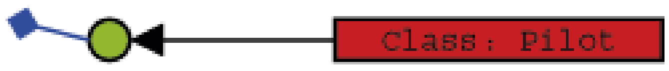
```
[retrieveAllPilots]

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

OUTPUT:

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

Basically, we are 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.

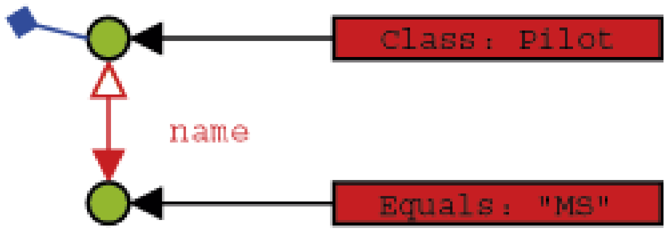
To retrieve a pilot by name, we have to further constrain the candidate pilots by descending to their name field and constraining this with the respective candidate String.

```
[retrievePilotByName]

Query query=db.query();
query.constrain(Pilot.class);
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(Pilot.class);
query.descend("points").constrain(new Integer(100));
ObjectSet result=query.execute();
listResult(result);
```

OUTPUT:

```
1
Michael Schumacher/100
```

3.3.2. 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(Pilot.class);
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(Pilot.class);
Constraint constr=query.descend("name")
        .constrain("Michael Schumacher");
query.descend("points")
        .constrain(new Integer(99)).and(constr);
ObjectSet result=query.execute();
listResult(result);
```

OUTPUT:

```
0
```

```
[retrieveByDisjunction]

Query query=db.query();
query.constrain(Pilot.class);
Constraint constr=query.descend("name")
        .constrain("Michael Schumacher");
query.descend("points")
        .constrain(new Integer(99)).or(constr);
ObjectSet result=query.execute();
listResult(result);
```

OUTPUT:

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

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

```
[retrieveByComparison]

Query query=db.query();
query.constrain(Pilot.class);
query.descend("points")
        .constrain(new Integer(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(Pilot.class);
query.descend("points").constrain(new Integer(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(Pilot.class);
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. There still may be cases left where the predefined query API constraints may not be sufficient - don't worry, you can always let db4o run any arbitrary code that you provide in an Evaluation. Evaluations will be discussed in a later chapter.

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

```
[clearDatabase]

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

OUTPUT:

3.3.3. Conclusion

Now you have been provided with three alternative approaches to query db4o databases: Query-By-Example, Native Queries, SODA.

Which one is the best to use? Some hints:
- Native queries are targetted to be the primary interface for db4o, so they should be preferred.
- With the current state of the native query optimizer there may be queries that will execute faster in SODA style, so it can be used to tune applications. SODA can also be more convenient for constructing dynamic queries at runtime.
- Query-By-Example is nice for simple one-liners, but restricted in functionality. If you like this approach, use it as long as it suits your application's needs.

Of course you can mix these strategies as needed.

We have finished our walkthrough and seen the various ways db4o provides to pose 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 .

3.3.4. Full source

```
package com.db4o.f1.chapter1;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.f1.Util;
import com.db4o.query.Constraint;
import com.db4o.query.Query;


public class QueryExample extends 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);
        System.out.println("Stored "+pilot1);
    }

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

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

    public static void retrievePilotByName(ObjectContainer db) {
        Query query=db.query();
        query.constrain(Pilot.class);
        query.descend("name").constrain("Michael Schumacher");
        ObjectSet result=query.execute();
```

```
        listResult(result);
    }

    public static void retrievePilotByExactPoints(
            ObjectContainer db) {
        Query query=db.query();
        query.constrain(Pilot.class);
        query.descend("points").constrain(new Integer(100));
        ObjectSet result=query.execute();
        listResult(result);
    }

    public static void retrieveByNegation(ObjectContainer db) {
        Query query=db.query();
        query.constrain(Pilot.class);
        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(Pilot.class);
        Constraint constr=query.descend("name")
                .constrain("Michael Schumacher");
        query.descend("points")
                .constrain(new Integer(99)).and(constr);
        ObjectSet result=query.execute();
        listResult(result);
    }

    public static void retrieveByDisjunction(ObjectContainer db) {
        Query query=db.query();
        query.constrain(Pilot.class);
        Constraint constr=query.descend("name")
                .constrain("Michael Schumacher");
        query.descend("points")
                .constrain(new Integer(99)).or(constr);
        ObjectSet result=query.execute();
        listResult(result);
    }
    public static void retrieveByComparison(ObjectContainer db) {
        Query query=db.query();
        query.constrain(Pilot.class);
        query.descend("points")
                .constrain(new Integer(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(Pilot.class);
        query.descend("points").constrain(new Integer(0));
        ObjectSet result=query.execute();
        listResult(result);
        db.delete(somebody);
    }

    public static void retrieveSorted(ObjectContainer db) {
        Query query=db.query();
        query.constrain(Pilot.class);
        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(Pilot.class);
        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.

```
package com.db4o.f1.chapter2;

public class Car {
    private String model;
    private Pilot pilot;

    public Car(String model) {
        this.model=model;
        this.pilot=null;
    }

    public Pilot getPilot() {
        return pilot;
    }

    public void setPilot(Pilot pilot) {
        this.pilot = pilot;
    }

    public String getModel() {
        return model;
    }

    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.setPilot(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.setPilot(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
Michael Schumacher/100
Rubens Barrichello/99
```

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.setPilot(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 access the pilot field directly.

4.2.2. Native Queries

Using native queries with constraints on deep structured objects is straightforward, you can do it just like you would in plain other code.

Let's constrain our query to only those cars driven by a Pilot with a specific name:

```
[retrieveCarsByPilotNameNative]

final String pilotName = "Rubens Barrichello";
ObjectSet results = db.query(new Predicate() {
    public boolean match(Car car){
        return car.getPilot().getName().equals(pilotName);
    }
});
listResult(results);
```

OUTPUT:

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

4.2.3. SODA Query API

In order to use SODA for querying for a car given its pilot's name we have to descend two levels into our query.

```
[retrieveCarByPilotNameQuery]


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

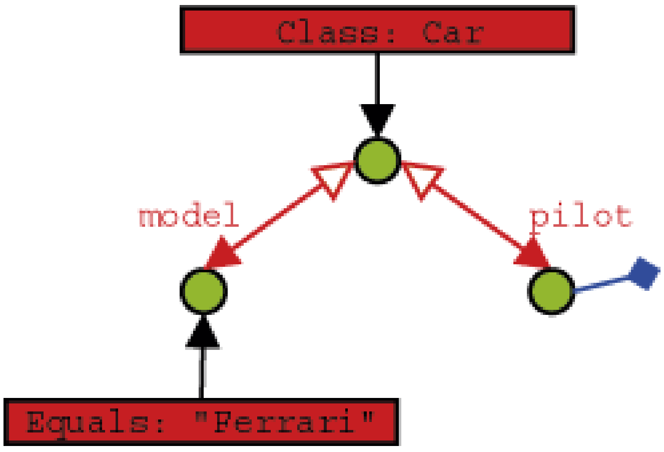
OUTPUT:

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

We have seen that descending into a query provides us with another query. Starting out from a query root we can descend in multiple directions. In practice this is the same as ascending from one child to a parent and descending to another child. We can conclude that queries turn one-directional references in our objects into true relations. Here is an example that queries for "a Pilot that is being referenced by a Car, where the Car model is 'Ferrari'":

```
[retrievePilotByCarModelQuery]

Query carquery=db.query();
carquery.constrain(Car.class);
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.query(new Predicate() {
    public boolean match(Car car){
        return car.getModel().equals("Ferrari");
    }
});
Car found=(Car)result.next();
found.setPilot(new Pilot("Somebody else",0));
db.set(found);
result=db.query(new Predicate() {
    public boolean match(Car car){
        return car.getModel().equals("Ferrari");
    }
});
listResult(result);
```

OUTPUT:

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

Let's modify the pilot, too.

```
[updatePilotSingleSession]

ObjectSet result=db.query(new Predicate() {
    public boolean match(Car car){
        return car.getModel().equals("Ferrari");
    }
});
Car found=(Car)result.next();
found.getPilot().addPoints(1);
db.set(found);
result=db.query(new Predicate() {
    public boolean match(Car car){
        return car.getModel().equals("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:

```
[updatePilotSeparateSessionsPart1]

ObjectSet result=db.query(new Predicate() {
    public boolean match(Car car){
        return car.getModel().equals("Ferrari");
    }
});
Car found=(Car)result.next();
found.getPilot().addPoints(1);
db.set(found);
```

And in the second, we'll double-check our modification:

```
[updatePilotSeparateSessionsPart2]

ObjectSet result=db.query(new Predicate() {
    public boolean match(Car car){
        return car.getModel().equals("Ferrari");
    }
});
listResult(result);
```

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

Looks like we're in trouble: Why did the Pilot's points not change? 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.

So, in our previous update example, we were modifying the Pilot child of a Car object. When we saved the change, we told db4o to save our Car object and asumed that the modified Pilot would be updated. But we were modifying and saving in the same manner as we were in the first update sample, so why did it work before? The first time we made the modification, db4o never actually had to retreive the modified Pilot it returned the same one that was still in memory that we modified, but it never actually updated the database. The fact that we saw the modified value was, in fact, a bug. Restarting the application would show that the value was unchanged.

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("com.db4o.f1.chapter2.Car")
        .cascadeOnUpdate(true);
```

```
[updatePilotSeparateSessionsImprovedPart2]

ObjectSet result=db.query(new Predicate() {
    public boolean match(Car car){
        return car.getModel().equals("Ferrari");
    }
});
Car found=(Car)result.next();
found.getPilot().addPoints(1);
db.set(found);
```

```
[updatePilotSeparateSessionsImprovedPart3]

ObjectSet result=db.query(new Predicate() {
    public boolean match(Car car){
        return car.getModel().equals("Ferrari");
    }
});
listResult(result);
```

OUTPUT:

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

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.query(new Predicate() {
    public boolean match(Car car){
        return car.getModel().equals("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:

```
3
Somebody else/1
Michael Schumacher/100
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("com.db4o.f1.chapter2.Car")
        .cascadeOnDelete(true);
```

```
[deleteDeepPart2]

ObjectSet result=db.query(new Predicate() {
    public boolean match(Car car){
        return car.getModel().equals("BMW");
    }
});
Car found=(Car)result.next();
db.delete(found);
result=db.query(new Predicate() {
    public boolean match(Car car){
        return true;
    }
});
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.

```
[retrieveAllPilots]

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

OUTPUT:
```
2
Somebody else/1
Michael Schumacher/100
```

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.query(new Predicate() {
    public boolean match(Pilot pilot){
        return pilot.getName().equals("Michael Schumacher");
    }
});
Pilot pilot=(Pilot)result.next();
Car car1=new Car("Ferrari");
Car car2=new Car("BMW");
car1.setPilot(pilot);
car2.setPilot(pilot);
db.set(car1);
db.set(car2);
db.delete(car2);
result=db.query(new Predicate() {
    public boolean match(Car car){
        return true;
    }
});
listResult(result);
```

```
OUTPUT:
1
Ferrari[Michael Schumacher/100]
```

```
[retrieveAllPilots]

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

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

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 using this feature.

Let's clear our database for the next chapter.

```
[deleteAll]

ObjectSet result=db.get(new Object());
while(result.hasNext()) {
    db.delete(result.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

```
package com.db4o.f1.chapter2;

import java.io.File;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.f1.Util;
import com.db4o.query.Predicate;
import com.db4o.query.Query;


public class StructuredExample extends Util {
    public static void main(String[] args) {
        new File(Util.YAPFILENAME).delete();
        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=Db4o.openFile(Util.YAPFILENAME);
            updatePilotSeparateSessionsImprovedPart2(db);
            db.close();
            db=Db4o.openFile(Util.YAPFILENAME);
            updatePilotSeparateSessionsImprovedPart3(db);
            deleteFlat(db);
            db.close();
            deleteDeepPart1();
            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.setPilot(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.setPilot(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 retrieveAllPilots(ObjectContainer db) {
        ObjectSet result=db.get(Pilot.class);
        listResult(result);
    }

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

    public static void retrieveCarByPilotNameQuery(
            ObjectContainer db) {
        Query query=db.query();
        query.constrain(Car.class);
        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(Car.class);
        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(Car.class);
        carquery.descend("model").constrain("Ferrari");
        Query pilotquery=carquery.descend("pilot");
        ObjectSet result=pilotquery.execute();
        listResult(result);
    }

    public static void retrieveAllPilotsNative(ObjectContainer db) {
        ObjectSet results = db.query(new Predicate() {
            public boolean match(Pilot pilot){
                return true;
            }
        });
        listResult(results);
    }


    public static void retrieveAllCars(ObjectContainer db) {
        ObjectSet results = db.get(Car.class);
        listResult(results);
    }

    public static void retrieveCarsByPilotNameNative(ObjectContainer
db) {
        final String pilotName = "Rubens Barrichello";
        ObjectSet results = db.query(new Predicate() {
            public boolean match(Car car){
                return car.getPilot().getName().equals(pilotName);
            }
        });
        listResult(results);
    }

    public static void updateCar(ObjectContainer db) {
        ObjectSet result=db.query(new Predicate() {
            public boolean match(Car car){
                return car.getModel().equals("Ferrari");
            }
        });
        Car found=(Car)result.next();
        found.setPilot(new Pilot("Somebody else",0));
        db.set(found);
        result=db.query(new Predicate() {
            public boolean match(Car car){
                return car.getModel().equals("Ferrari");
            }
        });
        listResult(result);
    }

    public static void updatePilotSingleSession(
                ObjectContainer db) {
        ObjectSet result=db.query(new Predicate() {
            public boolean match(Car car){
                return car.getModel().equals("Ferrari");
            }
        });
        Car found=(Car)result.next();
        found.getPilot().addPoints(1);
        db.set(found);
        result=db.query(new Predicate() {
            public boolean match(Car car){
                return car.getModel().equals("Ferrari");
            }
        });
        listResult(result);
    }

    public static void updatePilotSeparateSessionsPart1(
            ObjectContainer db) {
        ObjectSet result=db.query(new Predicate() {
            public boolean match(Car car){
                return car.getModel().equals("Ferrari");
            }
        });
        Car found=(Car)result.next();
        found.getPilot().addPoints(1);
        db.set(found);
    }

    public static void updatePilotSeparateSessionsPart2(
                ObjectContainer db) {
        ObjectSet result=db.query(new Predicate() {
            public boolean match(Car car){
                return car.getModel().equals("Ferrari");
            }
        });
        listResult(result);
    }

    public static void updatePilotSeparateSessionsImprovedPart1() {
        Db4o.configure().objectClass("com.db4o.f1.chapter2.Car")
                .cascadeOnUpdate(true);
    }

    public static void updatePilotSeparateSessionsImprovedPart2(
                ObjectContainer db) {
        ObjectSet result=db.query(new Predicate() {
            public boolean match(Car car){
                return car.getModel().equals("Ferrari");
            }
        });
        Car found=(Car)result.next();
        found.getPilot().addPoints(1);
        db.set(found);
    }

    public static void updatePilotSeparateSessionsImprovedPart3(
                ObjectContainer db) {
        ObjectSet result=db.query(new Predicate() {
            public boolean match(Car car){
                return car.getModel().equals("Ferrari");
            }
        });
        listResult(result);
    }

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

```
    ObjectSet result=db.query(new Predicate() {
        public boolean match(Car car){
            return car.getModel().equals("Ferrari");
        }
    });
    Car found=(Car)result.next();
    db.delete(found);
    result=db.get(new Car(null));
    listResult(result);
}

public static void deleteDeepPart1() {
    Db4o.configure().objectClass("com.db4o.f1.chapter2.Car")
            .cascadeOnDelete(true);
}

public static void deleteDeepPart2(ObjectContainer db) {
    ObjectSet result=db.query(new Predicate() {
        public boolean match(Car car){
            return car.getModel().equals("BMW");
        }
    });
    Car found=(Car)result.next();
    db.delete(found);
    result=db.query(new Predicate() {
        public boolean match(Car car){
            return true;
        }
    });
    listResult(result);
}

public static void deleteDeepRevisited(ObjectContainer db) {
    ObjectSet result=db.query(new Predicate() {
        public boolean match(Pilot pilot){
            return pilot.getName().equals("Michael Schumacher");
        }
    });
    Pilot pilot=(Pilot)result.next();
    Car car1=new Car("Ferrari");
    Car car2=new Car("BMW");
```

```
        car1.setPilot(pilot);
        car2.setPilot(pilot);
        db.set(car1);
        db.set(car2);
        db.delete(car2);
        result=db.query(new Predicate() {
            public boolean match(Car car){
                return true;
            }
        });
        listResult(result);
    }

}
```

5. Collections and Arrays

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

```
package com.db4o.f1.chapter3;

import java.util.*;

public class SensorReadout {
    private double[] values;
    private Date time;
    private Car car;

    public SensorReadout(double[] values,Date time,Car car) {
        this.values=values;
        this.time=time;
        this.car=car;
    }

    public Car getCar() {
        return car;
    }

    public Date getTime() {
        return time;
    }

    public int getNumValues() {
        return values.length;
    }

    public double[] getValues(){
        return values;
    }

    public double getValue(int idx) {
        return values[idx];
    }

    public String toString() {
        StringBuffer str=new StringBuffer();
        str.append(car.toString())
            .append(" : ")
            .append(time.getTime())
            .append(" : ");
        for(int idx=0;idx<values.length;idx++) {
            if(idx>0) {
                str.append(',');
            }
            str.append(values[idx]);
        }
        return str.toString();
    }
}
```

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

```
package com.db4o.f1.chapter3;

import java.util.*;

public class Car {
    private String model;
    private Pilot pilot;
    private List history;

    public Car(String model) {
        this(model,new ArrayList());
    }

    public Car(String model,List history) {
        this.model=model;
        this.pilot=null;
        this.history=history;
    }

    public Pilot getPilot() {
        return pilot;
    }

    public void setPilot(Pilot pilot) {
        this.pilot=pilot;
    }

    public String getModel() {
        return model;
    }

    public List getHistory() {
        return history;
    }

    public void snapshot() {
        history.add(new SensorReadout(poll(),new Date(),this));
    }

    protected double[] poll() {
        int factor=history.size()+1;
        return new double[]{0.1d*factor,0.2d*factor,0.3d*factor};
    }

    public String toString() {
        return model+"["+pilot+"]/"+history.size();
    }
}
```

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.setPilot(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.setPilot(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.

```
[retrieveAllSensorReadout]

SensorReadout proto=new SensorReadout(null,null,null);
ObjectSet results=db.get(proto);
listResult(results);
```

```
OUTPUT:
2
BMW[Rubens Barrichello/99]/2 : 1131922850390 : 0.1,0.2,0.3
BMW[Rubens Barrichello/99]/2 : 1131922850390 : 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},null,null);
ObjectSet results=db.get(proto);
listResult(results);
```

```
OUTPUT:
1
BMW[Rubens Barrichello/99]/2 : 1131922850390 : 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},null,null);
List 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 : 1131922850390 : 0.1,0.2,0.3,
BMW[Rubens Barrichello/99]/2 : 1131922850390 : 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. Native Queries

If we want to use Native Queries to find SensorReadouts with matching values, we simply write this as if we would check every single instance:

```
[retrieveSensorReadoutNative]

ObjectSet results = db.query(new Predicate() {
    public boolean match(SensorReadout candidate){
        return Arrays.binarySearch(candidate.getValues(), 0.3) >= 0
            && Arrays.binarySearch(candidate.getValues(), 1.0) >= 0;
    }
});
listResult(results);
```

OUTPUT:

```
0
```

And here's how we find Cars with matching readout values:

```
[retrieveCarNative]

ObjectSet results = db.query(new Predicate() {
    public boolean match(Car candidate){
        List history = candidate.getHistory();
        for(int i = 0; i < history.size(); i++){
            SensorReadout readout = (SensorReadout)history.get(i);
            if( Arrays.binarySearch(readout.getValues(), 0.6) >= 0 ||
            Arrays.binarySearch(readout.getValues(), 0.2) >= 0)
                return true;
        }
        return false;
    }
});
```

```
listResult(results);
```

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

5.2.3. Query API

Handling of arrays and collections is analogous to the previous example. First, lets retrieve only the SensorReadouts with specific values:

```
[retrieveSensorReadoutQuery]

Query query=db.query();
query.constrain(SensorReadout.class);
Query valuequery=query.descend("values");
valuequery.constrain(new Double(0.3));
valuequery.constrain(new Double(0.1));
ObjectSet result=query.execute();
listResult(result);
```

```
OUTPUT:
1
BMW[Rubens Barrichello/99]/2 : 1131922850390 : 0.1,0.2,0.3
```

Then let's get some Cars with matching Readout values:

```
[retrieveCarQuery]

Query query=db.query();
query.constrain(Car.class);
```

```
Query historyquery=query.descend("history");
historyquery.constrain(SensorReadout.class);
Query valuequery=historyquery.descend("values");
valuequery.constrain(new Double(0.3));
valuequery.constrain(new Double(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(Car.class).cascadeOnUpdate(true);
```

```
[updateCarPart2]

ObjectSet results = db.query(new Predicate() {
    public boolean match(Car candidate){
        return true;
    }
});
Car car=(Car)results.next();
car.snapshot();
db.set(car);
retrieveAllSensorReadoutNative(db);
```

OUTPUT:
```
3
BMW[Rubens Barrichello/99]/3 : 1131922850390 : 0.1,0.2,0.3
BMW[Rubens Barrichello/99]/3 : 1131922850390 : 0.2,0.4,0.6
BMW[Rubens Barrichello/99]/3 : 1131922850953 :
0.30000000000000004,0.6000000000000001,0.8999999999999999
```

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

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

```
[updateCollection]

ObjectSet results = db.query(new Predicate() {
    public boolean match(Car candidate){
        return true;
    }
});
Car car =(Car)results.next();
car.getHistory().remove(0);
db.set(car.getHistory());
results = db.query(new Predicate() {
    public boolean match(Car candidate){
        return true;
    }
});
while(results.hasNext()) {
    car=(Car)results.next();
    for (int idx=0;idx<car.getHistory().size();idx++) {
        System.out.println(car.getHistory().get(idx));
    }
}
```

OUTPUT:
```
BMW[Rubens Barrichello/99]/2 : 1131922850390 : 0.2,0.4,0.6
```

```
BMW[Rubens Barrichello/99]/2 : 1131922850953 :
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(Car.class)
        .cascadeOnDelete(true);
```

```
[deleteAllPart2]

ObjectSet cars = db.query(new Predicate() {
    public boolean match(Car candidate){
        return true;
    }
});
while(cars.hasNext()) {
    db.delete(cars.next());
}
ObjectSet readouts = db.query(new Predicate() {
    public boolean match(SensorReadout candidate){
        return true;
    }
});
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

```
package com.db4o.f1.chapter3;

import java.io.*;
import java.util.*;
import com.db4o.*;
import com.db4o.f1.*;
import com.db4o.query.*;


public class CollectionsExample extends Util {
    public static void main(String[] args) {
        new File(Util.YAPFILENAME).delete();
        ObjectContainer db=Db4o.openFile(Util.YAPFILENAME);
        try {
            storeFirstCar(db);
            storeSecondCar(db);
            retrieveAllSensorReadout(db);
            retrieveSensorReadoutQBE(db);
            retrieveCarQBE(db);
            retrieveCollections(db);
            retrieveArrays(db);
            retrieveAllSensorReadoutNative(db);
            retrieveSensorReadoutNative(db);
            retrieveCarNative(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);
        }
        finally {
            db.close();
        }
    }

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

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

    public static void retrieveAllSensorReadout(
                ObjectContainer db) {
        SensorReadout proto=new SensorReadout(null,null,null);
        ObjectSet results=db.get(proto);
        listResult(results);
    }

    public static void retrieveAllSensorReadoutNative(
            ObjectContainer db) {
        ObjectSet results = db.query(new Predicate() {
            public boolean match(SensorReadout candidate){
                return true;
            }
        });
        listResult(results);
    }
    public static void retrieveSensorReadoutQBE(
                ObjectContainer db) {
        SensorReadout proto=new SensorReadout(
                new double[]{0.3,0.1},null,null);
        ObjectSet results=db.get(proto);
        listResult(results);
    }

    public static void retrieveSensorReadoutNative(
            ObjectContainer db) {
        ObjectSet results = db.query(new Predicate() {
            public boolean match(SensorReadout candidate){
                return Arrays.binarySearch(candidate.getValues(),
0.3) >= 0
                    && Arrays.binarySearch(candidate.getValues(),
1.0) >= 0;
            }
        });
        listResult(results);
    }

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

    public static void retrieveCarNative(
            ObjectContainer db) {
        ObjectSet results = db.query(new Predicate() {
            public boolean match(Car candidate){
                List history = candidate.getHistory();
                for(int i = 0; i < history.size(); i++){
                    SensorReadout readout =
(SensorReadout)history.get(i);
                    if( Arrays.binarySearch(readout.getValues(), 0.6)
>= 0 ||
                    Arrays.binarySearch(readout.getValues(), 0.2) >=
0)
                        return true;
                }
                return false;
            }
        });
        listResult(results);
    }

    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(SensorReadout.class);
        Query valuequery=query.descend("values");
        valuequery.constrain(new Double(0.3));
        valuequery.constrain(new Double(0.1));
        ObjectSet result=query.execute();
        listResult(result);
    }

    public static void retrieveCarQuery(ObjectContainer db) {
        Query query=db.query();
        query.constrain(Car.class);
        Query historyquery=query.descend("history");
        historyquery.constrain(SensorReadout.class);
        Query valuequery=historyquery.descend("values");
        valuequery.constrain(new Double(0.3));
        valuequery.constrain(new Double(0.1));
        ObjectSet result=query.execute();
        listResult(result);
    }

    public static void updateCarPart1() {
Db4o.configure().objectClass(Car.class).cascadeOnUpdate(true);      }
  public static void updateCarPart2(ObjectContainer db) {
        ObjectSet results = db.query(new Predicate() {
            public boolean match(Car candidate){
                return true;
            }
        });
        Car car=(Car)results.next();
        car.snapshot();
        db.set(car);
        retrieveAllSensorReadoutNative(db);
    }

    public static void updateCollection(ObjectContainer db) {
        ObjectSet results = db.query(new Predicate() {
            public boolean match(Car candidate){
                return true;
            }
        });
        Car car =(Car)results.next();
        car.getHistory().remove(0);
        db.set(car.getHistory());
        results = db.query(new Predicate() {
            public boolean match(Car candidate){
                return true;
            }
        });
        while(results.hasNext()) {
            car=(Car)results.next();
            for (int idx=0;idx<car.getHistory().size();idx++) {
                System.out.println(car.getHistory().get(idx));
            }
        }
    }

    public static void deleteAllPart1() {
        Db4o.configure().objectClass(Car.class)
```

```
                .cascadeOnDelete(true);
    }

    public static void deleteAllPart2(ObjectContainer db) {
        ObjectSet cars = db.query(new Predicate() {
            public boolean match(Car candidate){
                return true;
            }
        });
        while(cars.hasNext()) {
            db.delete(cars.next());
        }
        ObjectSet readouts = db.query(new Predicate() {
            public boolean match(SensorReadout candidate){
                return true;
            }
        });
        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.

```
package com.db4o.f1.chapter4;

import java.util.*;

public class SensorReadout {
    private Date time;
    private Car car;
    private String description;

    protected SensorReadout(Date time,Car car,String description) {
        this.time=time;
        this.car=car;
        this.description=description;
    }

    public Car getCar() {
        return car;
    }

    public Date getTime() {
        return time;
    }

    public String getDescription() {
        return description;
    }
    public String toString() {
        return car+" : "+time+" : "+description;
    }
}
```

```
package com.db4o.f1.chapter4;

import java.util.*;


public class TemperatureSensorReadout extends SensorReadout {
    private double temperature;

    public TemperatureSensorReadout(
            Date time,Car car,
            String description,double temperature) {
        super(time,car,description);
        this.temperature=temperature;
    }

    public double getTemperature() {
        return temperature;
    }

    public String toString() {
        return super.toString()+" temp : "+temperature;
    }
}
```

```
package com.db4o.f1.chapter4;

import java.util.*;


public class PressureSensorReadout extends SensorReadout {
    private double pressure;

    public PressureSensorReadout(
            Date time,Car car,
            String description,double pressure) {
```

```
        super(time,car,description);
        this.pressure=pressure;
    }

    public double getPressure() {
        return pressure;
    }

    public String toString() {
        return super.toString()+" pressure : "+pressure;
    }
}
```

Our car's snapshot mechanism is changed accordingly.

```
package com.db4o.f1.chapter4;

import java.util.*;

public class Car {
    private String model;
    private Pilot pilot;
    private List history;

    public Car(String model) {
        this.model=model;
        this.pilot=null;
        this.history=new ArrayList();
    }

    public Pilot getPilot() {
        return pilot;
    }

    public void setPilot(Pilot pilot) {
        this.pilot=pilot;
    }
    public String getModel() {
        return model;
    }

    public SensorReadout[] getHistory() {
        return (SensorReadout[])history.toArray(new
SensorReadout[history.size()]);
    }

    public void snapshot() {
        history.add(new TemperatureSensorReadout(
                new Date(),this,"oil",pollOilTemperature()));
        history.add(new TemperatureSensorReadout(
                new Date(),this,"water",pollWaterTemperature()));
        history.add(new PressureSensorReadout(
                new Date(),this,"oil",pollOilPressure()));
    }

    protected double pollOilTemperature() {
        return 0.1*history.size();
    }

    protected double pollWaterTemperature() {
        return 0.2*history.size();
    }

    protected double pollOilPressure() {
        return 0.3*history.size();
    }

    public String toString() {
        return model+"["+pilot+"]/"+history.size();
    }
}
```

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.setPilot(pilot1);
db.set(car1);
```

```
[storeSecondCar]

Pilot pilot2=new Pilot("Rubens Barrichello",99);
Car car2=new Car("BMW");
car2.setPilot(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(null,null,null,0.0);
ObjectSet result=db.get(proto);
listResult(result);
```

OUTPUT:

```
4
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.30000000000000004
```

```
[retrieveAllSensorReadoutsQBE]

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

OUTPUT:

```
6
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 0.6
```

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 trick to keep in mind: Class objects receive special handling with QBE.

```
[retrieveAllSensorReadoutsQBEAlternative]

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

```
OUTPUT:
6
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 0.6
```

And of course there's our SODA API:

```
[retrieveAllSensorReadoutsQuery]

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

```
OUTPUT:
6
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : oil
```

```
pressure : 1.5
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 0.6
```

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.

```
[deleteAll]

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

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

```
package com.db4o.f1.chapter4;

import java.io.*;
import java.util.Arrays;

import com.db4o.*;
import com.db4o.f1.*;
import com.db4o.query.*;


public class InheritanceExample extends Util {
    public static void main(String[] args) {
        new File(Util.YAPFILENAME).delete();
        ObjectContainer db=Db4o.openFile(Util.YAPFILENAME);
        try {
            storeFirstCar(db);
            storeSecondCar(db);
            retrieveTemperatureReadoutsQBE(db);
            retrieveAllSensorReadoutsQBE(db);
            retrieveAllSensorReadoutsQBEAlternative(db);
            retrieveAllSensorReadoutsQuery(db);
            retrieveAllObjectsQBE(db);
        }
        finally {
            db.close();
        }
    }

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

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

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

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

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

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

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

}
```

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.

```
package com.db4o.f1.chapter5;

import java.util.*;

public class SensorReadout {
    private Date time;
    private Car car;
    private String description;
    private SensorReadout next;

    protected SensorReadout(Date time,Car car,String description) {
        this.time=time;
        this.car=car;
        this.description=description;
        this.next=null;
    }

    public Car getCar() {
        return car;
    }

    public Date getTime() {
        return time;
    }

    public String getDescription() {
        return description;
    }

    public SensorReadout getNext() {
        return next;
    }

    public void append(SensorReadout readout) {
        if(next==null) {
            next=readout;
        }
        else {
            next.append(readout);
        }
    }

    public int countElements() {
        return (next==null ? 1 : next.countElements()+1);
    }

    public String toString() {
        return car+" : "+time+" : "+description;
    }
}
```

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

```
package com.db4o.f1.chapter5;

import java.util.*;

public class Car {
    private String model;
    private Pilot pilot;
    private SensorReadout history;

    public Car(String model) {
        this.model=model;
        this.pilot=null;
        this.history=null;
    }
    public Pilot getPilot() {
        return pilot;
    }

    public void setPilot(Pilot pilot) {
        this.pilot=pilot;
    }

    public String getModel() {
        return model;
    }

    public SensorReadout getHistory() {
        return history;
    }

    public void snapshot() {
        appendToHistory(new TemperatureSensorReadout(
                new Date(),this,"oil",pollOilTemperature()));
        appendToHistory(new TemperatureSensorReadout(
                new Date(),this,"water",pollWaterTemperature()));
        appendToHistory(new PressureSensorReadout(
                new Date(),this,"oil",pollOilPressure()));
    }

    protected double pollOilTemperature() {
        return 0.1*countHistoryElements();
    }

    protected double pollWaterTemperature() {
        return 0.2*countHistoryElements();
    }

    protected double pollOilPressure() {
        return 0.3*countHistoryElements();
    }

    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.setPilot(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(Car.class).cascadeOnUpdate(true);
```

Let's collect a few sensor readouts.

```
[takeManySnapshots]

ObjectSet result=db.get(Car.class);
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(SensorReadout.class);
while(result.hasNext()) {
    System.out.println(result.next());
}
```

OUTPUT:

```
BMW[Rubens Barrichello/99]/4 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 4.2
BMW[Rubens Barrichello/99]/4 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 2.6
BMW[Rubens Barrichello/99]/4 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 1.2000000000000002
BMW[Rubens Barrichello/99]/4 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 3.3
BMW[Rubens Barrichello/99]/4 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 2.0
```

```
BMW[Rubens Barrichello/99]/4 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.9
BMW[Rubens Barrichello/99]/4 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 2.4
BMW[Rubens Barrichello/99]/4 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 1.4000000000000001
BMW[Rubens Barrichello/99]/4 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 0.8
BMW[Rubens Barrichello/99]/4 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.6000000000000001
BMW[Rubens Barrichello/99]/4 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 0.2
```

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(Car.class);
Car car=(Car)result.next();
SensorReadout readout=car.getHistory();
while(readout!=null) {
    System.out.println(readout);
    readout=readout.getNext();
}
```

OUTPUT:

```
BMW[Rubens Barrichello/99]/5 : Mon Nov 14 00:00:51 CET 2005 : oil
```

```
temp : 0.0
BMW[Rubens Barrichello/99]/5 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 0.2
BMW[Rubens Barrichello/99]/5 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/5 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.30000000000000004
null : null : null temp : 0.0
```

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(Car.class);
Car car=(Car)result.next();
SensorReadout readout=car.getHistory();
while(readout!=null) {
    db.activate(readout,1);
    System.out.println(readout);
    readout=readout.getNext();
}
```

OUTPUT:

```
BMW[Rubens Barrichello/99]/5 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/5 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 0.2
BMW[Rubens Barrichello/99]/5 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/5 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 0.8
BMW[Rubens Barrichello/99]/7 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/8 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.6000000000000001
BMW[Rubens Barrichello/99]/9 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 1.4000000000000001
BMW[Rubens Barrichello/99]/10 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 2.4
BMW[Rubens Barrichello/99]/11 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.9
BMW[Rubens Barrichello/99]/12 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 2.0
BMW[Rubens Barrichello/99]/13 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 3.3
BMW[Rubens Barrichello/99]/14 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 1.2000000000000002
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 2.6
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 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(TemperatureSensorReadout.class)
        .cascadeOnActivate(true);
```

OUTPUT:

```
[retrieveSnapshotsSequentially]

ObjectSet result=db.get(Car.class);
Car car=(Car)result.next();
SensorReadout readout=car.getHistory();
while(readout!=null) {
    System.out.println(readout);
    readout=readout.getNext();
}
```

OUTPUT:

```
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 0.2
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 0.8
```

```
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.6000000000000001
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 1.4000000000000001
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 2.4
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 0.9
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 2.0
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : oil
pressure : 3.3
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : oil
temp : 1.2000000000000002
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 2005 : water
temp : 2.6
BMW[Rubens Barrichello/99]/15 : Mon Nov 14 00:00:51 CET 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.

```
[deleteAll]

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

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

```
package com.db4o.f1.chapter5;

import java.io.*;
import com.db4o.*;
import com.db4o.f1.*;


public class DeepExample extends Util {
    public static void main(String[] args) {
        new File(Util.YAPFILENAME).delete();
        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);
        }
        finally {
            db.close();
        }
    }

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

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

    public static void takeManySnapshots(ObjectContainer db) {
        ObjectSet result=db.get(Car.class);
        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(SensorReadout.class);
        while(result.hasNext()) {
            System.out.println(result.next());
        }
    }

    public static void retrieveSnapshotsSequentially(
            ObjectContainer db) {
        ObjectSet result=db.get(Car.class);
        Car car=(Car)result.next();
        SensorReadout readout=car.getHistory();
        while(readout!=null) {
            System.out.println(readout);
            readout=readout.getNext();
        }
    }

    public static void retrieveSnapshotsSequentiallyImproved(
            ObjectContainer db) {
        ObjectSet result=db.get(Car.class);
        Car car=(Car)result.next();
        SensorReadout readout=car.getHistory();
        while(readout!=null) {
            db.activate(readout,1);
            System.out.println(readout);
            readout=readout.getNext();
        }
    }

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

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.setPilot(pilot);
db.set(car);
db.commit();
```

```
[listAllCars]

ObjectSet result=db.get(Car.class);
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.setPilot(pilot);
db.set(car);
db.rollback();
```

```
[listAllCars]

ObjectSet result=db.get(Car.class);
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();
System.out.println(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,Integer.MAX_VALUE);
System.out.println(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.

```
[deleteAll]
```

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

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

```
package com.db4o.f1.chapter5;

import java.io.*;
import com.db4o.*;
import com.db4o.f1.*;


public class TransactionExample extends Util {
    public static void main(String[] args) {
        new File(Util.YAPFILENAME).delete();
        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);
        }
        finally {
            db.close();
        }
    }

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

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

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

    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();
        System.out.println(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,Integer.MAX_VALUE);
        System.out.println(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.setPilot(pilot);
db.set(car);
```

```
[setSecondCar]

Pilot pilot=new Pilot("Michael Schumacher",100);
Car car=new Car("Ferrari");
car.setPilot(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
BMW[Rubens Barrichello/99]/0
Ferrari[Michael Schumacher/100]/0
[db4o 5.0.010   2005-11-14 00:00:53]
 Connection closed by client %.
[db4o 5.0.010   2005-11-14 00:00:53]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula14534.yap' close request
[db4o 5.0.010   2005-11-14 00:00:53]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula14534.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 result,int depth) {
    System.out.println(result.size());
    while(result.hasNext()) {
        Object obj = result.next();
        container.ext().refresh(obj, depth);
        System.out.println(obj);
    }
}
```

```
[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.setPilot(pilot);
client1.set(car);
listResult(client1.get(new Car(null)));
listResult(client2.get(new Car(null)));
client1.commit();
listResult(client1.get(Car.class));
listRefreshedResult(client2,client2.get(Car.class),2);
client1.close();
client2.close();
```

OUTPUT:

```
2
BMW[David Coulthard/98]/0
Ferrari[Michael Schumacher/100]/0
2
BMW[Rubens Barrichello/99]/0
Ferrari[Michael Schumacher/100]/0
2
```

```
BMW[David Coulthard/98]/0
Ferrari[Michael Schumacher/100]/0
2
BMW[David Coulthard/98]/0
Ferrari[Michael Schumacher/100]/0
[db4o 5.0.010   2005-11-14 00:00:54]
 Connection closed by client %.
[db4o 5.0.010   2005-11-14 00:00:54]
 Connection closed by client %.
[db4o 5.0.010   2005-11-14 00:00:54]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula14534.yap' close request
[db4o 5.0.010   2005-11-14 00:00:54]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula14534.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.setPilot(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
BMW[Someone else/0]/0
```

```
Ferrari[Michael Schumacher/100]/0
2
BMW[David Coulthard/98]/0
Ferrari[Michael Schumacher/100]/0
2
BMW[David Coulthard/98]/0
Ferrari[Michael Schumacher/100]/0
2
BMW[David Coulthard/98]/0
Ferrari[Michael Schumacher/100]/0
[db4o 5.0.010   2005-11-14 00:00:54]
 Connection closed by client %.
[db4o 5.0.010   2005-11-14 00:00:54]
 Connection closed by client %.
[db4o 5.0.010   2005-11-14 00:00:54]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula14534.yap' close request
[db4o 5.0.010   2005-11-14 00:00:54]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula14534.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,PORT);
server.grantAccess(USER,PASSWORD);
try {
    ObjectContainer
client=Db4o.openClient("localhost",PORT,USER,PASSWORD);
    // 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 5.0.010   2005-11-14 00:00:54]
 Server listening on port: '56128'
[db4o 5.0.010   2005-11-14 00:00:54]
 Client 'user' connected.
2
BMW[David Coulthard/98]/0
Ferrari[Michael Schumacher/100]/0
[db4o 5.0.010   2005-11-14 00:00:54]
 Connection closed by client 'user'.
[db4o 5.0.010   2005-11-14 00:00:55]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula14534.yap' close request
[db4o 5.0.010   2005-11-14 00:00:55]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula14534.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.setPilot(pilot);
client1.set(car);
listResult(client1.get(new Car(null)));
listResult(client2.get(new Car(null)));
client1.commit();
listResult(client1.get(new Car(null)));
listRefreshedResult(client2,client2.get(Car.class),2);
client1.close();
client2.close();
```

OUTPUT:

```
[db4o 5.0.010   2005-11-14 00:00:55]
 Server listening on port: '56128'
[db4o 5.0.010   2005-11-14 00:00:55]
 Client 'user' connected.
[db4o 5.0.010   2005-11-14 00:00:55]
 Client 'user' connected.
2
BMW[Jenson Button/97]/0
Ferrari[Michael Schumacher/100]/0
2
BMW[David Coulthard/98]/0
Ferrari[Michael Schumacher/100]/0
2
BMW[Jenson Button/97]/0
Ferrari[Michael Schumacher/100]/0
2
BMW[Jenson Button/97]/0
Ferrari[Michael Schumacher/100]/0
[db4o 5.0.010   2005-11-14 00:00:55]
 Connection closed by client 'user'.
[db4o 5.0.010   2005-11-14 00:00:55]
 Connection closed by client 'user'.
[db4o 5.0.010   2005-11-14 00:00:55]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula14534.yap' close request
[db4o 5.0.010   2005-11-14 00:00:55]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula14534.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.setPilot(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 5.0.010   2005-11-14 00:00:55]
 Server listening on port: '56128'
[db4o 5.0.010   2005-11-14 00:00:55]
 Client 'user' connected.
[db4o 5.0.010   2005-11-14 00:00:55]
 Client 'user' connected.
2
BMW[Someone else/0]/0
Ferrari[Michael Schumacher/100]/0
2
BMW[Jenson Button/97]/0
Ferrari[Michael Schumacher/100]/0
2
BMW[Jenson Button/97]/0
Ferrari[Michael Schumacher/100]/0
2
BMW[Jenson Button/97]/0
```

```
Ferrari[Michael Schumacher/100]/0
[db4o 5.0.010   2005-11-14 00:00:55]
 Connection closed by client 'user'.
[db4o 5.0.010   2005-11-14 00:00:55]
 Connection closed by client 'user'.
[db4o 5.0.010   2005-11-14 00:00:55]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula14534.yap' close request
[db4o 5.0.010   2005-11-14 00:00:55]
 'C:\DOCUME~1\Carl\LOCALS~1\Temp\formula14534.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(){
synchronized(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);

  // to identify the thread in a debugger
  Thread.currentThread().setName(this.getClass().getName());

  // We only need low priority since the db4o server has
  // it's own thread.
  Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
  try {
      if(! stop){
        // wait forever for notify() from close()
        this.wait(Long.MAX_VALUE);
      }
  } catch (Exception e) {
    e.printStackTrace();
  }
  db4oServer.close();
}
  }
```

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

```
public void processMessage(ObjectContainer con, Object message) {
if(message instanceof 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) {
  e.printStackTrace();
}

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:

```
package com.db4o.f1.chapter5;

/**
 * Configuration used for {@link StartServer} and {@link StopServer}.
 */
public interface ServerConfiguration {

  /**
   * the host to be used.
   * <br>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.
   */
  public String   HOST = "localhost";

  /**
```

```
   * the database file to be used by the server.
   */
  public String   FILE = "formula1.yap";

  /**
   * the port to be used by the server.
   */
  public int    PORT = 4488;

  /**
   * the user name for access control.
   */
  public String   USER = "db4o";

  /**
   * the pasword for access control.
   */
  public String   PASS = "db4o";
}
```

Now we'll create the server:

```
package com.db4o.f1.chapter5;

import com.db4o.*;
import com.db4o.messaging.*;

/**
 * starts a db4o server with the settings from {@link
ServerConfiguration}.
 * <br><br>This is a typical setup for a long running server.
 * <br><br>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.
 * <br><br>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.
 */
public class StartServer
    implements ServerConfiguration, MessageRecipient {

  /**
   * setting the value to true denotes that the server should be
closed
   */
  private boolean stop = false;

  /**
   * starts a db4o server using the configuration from
   * {@link ServerConfiguration}.
   */
  public static void main(String[] arguments) {
    new StartServer().runServer();
  }

  /**
   * opens the ObjectServer, and waits forever until close() is
called
   * or a StopServer message is being received.
   */
  public void runServer(){
    synchronized(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);

      // to identify the thread in a debugger
      Thread.currentThread().setName(this.getClass().getName());

      // We only need low priority since the db4o server has
      // it's own thread.
      Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
      try {
          if(! stop){
            // wait forever for notify() from close()
            this.wait(Long.MAX_VALUE);
          }
      } catch (Exception e) {
        e.printStackTrace();
      }
      db4oServer.close();
    }
  }

  /**
   * messaging callback
   * @see
com.db4o.messaging.MessageRecipient#processMessage(ObjectContainer,
Object)
   */
  public void processMessage(ObjectContainer con, Object message) {
    if(message instanceof StopServer){
      close();
    }
  }

  /**
   * closes this server.
   */
  public void close(){
    synchronized(this){
      stop = true;
      this.notify();
    }
  }
}
```

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

```java
package com.db4o.f1.chapter5;

import com.db4o.*;
import com.db4o.messaging.*;

/**
 * stops the db4o Server started with {@link StartServer}.
 * <br><br>This is done by opening a client connection
 * to the server and by sending a StopServer object as
 * a message. {@link StartServer} will react in it's
 * processMessage method.
 */
public class StopServer implements ServerConfiguration {

  /**
   * stops a db4o Server started with StartServer.
   * @throws 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) {
      e.printStackTrace();
    }

    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 visit our forums at http://forums.db4o.com/forums/.

9.6. Full source

```
package com.db4o.f1.chapter5;

import java.io.*;
import com.db4o.*;
import com.db4o.f1.*;
```

```
public class ClientServerExample extends Util {
    private final static int PORT=0xdb40;
    private final static String USER="user";
    private final static String PASSWORD="password";

    public static void main(String[] args) throws IOException {
        new File(Util.YAPFILENAME).delete();
        accessLocalServer();
        new File(Util.YAPFILENAME).delete();
        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,PORT);
        server.grantAccess(USER,PASSWORD);
        try {
            queryRemoteServer(PORT,USER,PASSWORD);
            demonstrateRemoteReadCommitted(PORT,USER,PASSWORD);
            demonstrateRemoteRollback(PORT,USER,PASSWORD);
        }
        finally {
            server.close();
        }
    }
    public static void setFirstCar(ObjectContainer db) {
        Pilot pilot=new Pilot("Rubens Barrichello",99);
        Car car=new Car("BMW");
        car.setPilot(pilot);
        db.set(car);
    }

    public static void setSecondCar(ObjectContainer db) {
        Pilot pilot=new Pilot("Michael Schumacher",100);
        Car car=new Car("Ferrari");
        car.setPilot(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(Car.class).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.setPilot(pilot);
        client1.set(car);
        listResult(client1.get(new Car(null)));
        listResult(client2.get(new Car(null)));
        client1.commit();
        listResult(client1.get(Car.class));
        listRefreshedResult(client2,client2.get(Car.class),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.setPilot(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() throws IOException {
        ObjectServer server=Db4o.openServer(Util.YAPFILENAME,PORT);
        server.grantAccess(USER,PASSWORD);
        try {
            ObjectContainer
client=Db4o.openClient("localhost",PORT,USER,PASSWORD);
            // Do something with this client, or open more clients
            client.close();
        }
        finally {
            server.close();
        }
    }

    public static void queryRemoteServer(int port,String user,String
password) throws IOException {
        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) throws IOException {
        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.setPilot(pilot);
        client1.set(car);
        listResult(client1.get(new Car(null)));
        listResult(client2.get(new Car(null)));
        client1.commit();
        listResult(client1.get(new Car(null)));
        listRefreshedResult(client2,client2.get(Car.class),2);
        client1.close();
        client2.close();
    }

    public static void demonstrateRemoteRollback(int port,String
user,String password) throws IOException {
        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.setPilot(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. SODA Evaluations

In the SODA API chapter we already mentioned *Evaluations* as a means of providing user-defined custom constraints and as a means to run any arbitrary code in a SODA query. Let's have a closer look.

10.1. Evaluation API

The evaluation API consists of two interfaces, *Evaluation* and *Candidate* . Evaluation implementations are implemented by the user and injected into a query. During a query, they will be called from db4o with a candidate instance in order to decide whether to include it into the current (sub-)result.

The Evaluation interface contains a single method only:

```
public void evaluate(Candidate candidate);
```

This will be called by db4o to check whether the object encapsulated by this candidate should be included into the current candidate set.

The Candidate interface provides three methods:

```
public Object getObject();
public void include(boolean flag);
public ObjectContainer objectContainer();
```

An Evaluation implementation may call getObject() to retrieve the actual object instance to be evaluated, it may call include() to instruct db4o whether or not to include this object in the current candidate set, and finally it may access the current database directly by calling objectContainer().

10.2. Example

For a simple example, let's go back to our Pilot/Car implementation from the Collections chapter. Back then, we kept a history of SensorReadout instances in a List member inside the car. Now imagine that we wanted to retrieve all cars that have assembled an even number of history entries. A quite contrived and seemingly trivial example, however, it gets us into trouble: Collections are transparent to the query API, it just 'looks through' them at their respective members.

So how can we get this done? Let's implement an Evaluation that expects the objects passed in to be instances of type Car and checks their history size.

```
package com.db4o.f1.chapter6;

import com.db4o.f1.chapter3.*;
import com.db4o.query.*;

public class EvenHistoryEvaluation implements Evaluation {
  public void evaluate(Candidate candidate) {
    Car car=(Car)candidate.getObject();
    candidate.include(car.getHistory().size() % 2 == 0);
  }
}
```

To test it, let's add two cars with history sizes of one, respectively two:

```
[storeCars]

Pilot pilot1=new Pilot("Michael Schumacher",100);
    Car car1=new Car("Ferrari");
    car1.setPilot(pilot1);
    car1.snapshot();
    db.set(car1);
    Pilot pilot2=new Pilot("Rubens Barrichello",99);
    Car car2=new Car("BMW");
    car2.setPilot(pilot2);
    car2.snapshot();
    car2.snapshot();
    db.set(car2);
```

and run our evaluation against them:

```
[queryWithEvaluation]

Query query=db.query();
    query.constrain(Car.class);
    query.constrain(new EvenHistoryEvaluation());
    ObjectSet result=query.execute();
    Util.listResult(result);
```

OUTPUT:

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

10.3. Drawbacks

While evaluations offer you another degree of freedom for assembling queries, they come at a certain cost: As you may already have noticed from the example, evaluations work on the fully instantiated objects, while 'normal' queries peek into the database file directly. So there's a certain performance penalty for the object instantiation, which is wasted if the object is not included into the candidate set.

Another restriction is that, while 'normal' queries can bypass encapsulation and access candidates' private members directly, evaluations are bound to use their external API, just as in the language itself.

One last hint for Javaists: Evaluations are expected to be serializable for client/server operation. So be careful when implementing them as (anonymous) inner classes and keep in mind that those will carry an implicit reference to their surrounding class and everything that belongs to it. Best practice is to always implement evaluations as normal top level or static inner classes.

10.4. Conclusion

With the introduction of evaluations we finally completed our query toolbox. Evaluations provide a simple way of assemble arbitrary custom query building blocks, however, they come at a price.

10.5. Full source

```
package com.db4o.f1.chapter6;

import java.io.*;

import com.db4o.*;
import com.db4o.f1.*;
import com.db4o.f1.chapter3.*;
import com.db4o.query.*;

public class EvaluationExample extends Util {
  public static void main(String[] args) {
    new File(Util.YAPFILENAME).delete();
    ObjectContainer db=Db4o.openFile(Util.YAPFILENAME);
    try {
      storeCars(db);
      queryWithEvaluation(db);
    }
    finally {
      db.close();
    }
  }

  public static void storeCars(ObjectContainer db) {
    Pilot pilot1=new Pilot("Michael Schumacher",100);
    Car car1=new Car("Ferrari");
    car1.setPilot(pilot1);
    car1.snapshot();
    db.set(car1);
    Pilot pilot2=new Pilot("Rubens Barrichello",99);
    Car car2=new Car("BMW");
    car2.setPilot(pilot2);
    car2.snapshot();
    car2.snapshot();
    db.set(car2);
  }

  public static void queryWithEvaluation(ObjectContainer db) {
    Query query=db.query();
    query.constrain(Car.class);
    query.constrain(new EvenHistoryEvaluation());
    ObjectSet result=query.execute();
    Util.listResult(result);
  }
}
```

11. 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.

11.1. 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.

11.1.1. 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?

11.1.2. 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.

11.1.3. 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 .

11.2. 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.

11.3. 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.

11.4. 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.

12. 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.

12.1. A 'NotStorable' class

Let's reuse our example from the previous chapter.

```
package com.db4o.f1.chapter6;

public class NotStorable {
  private int id;
  private String name;
  private transient int length;

  public NotStorable(int id,String name) {
    this.id=id;
    this.name=name;
    this.length=name.length();
  }

  public int getId() {
    return id;
  }

  public String getName() {
    return name;
  }

  public int getLength() {
    return length;
  }

  public String toString() {
    return id+"/"+name+": "+length;
  }
}
```

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(YAPFILENAME);
try {
  NotStorable notStorable = new NotStorable(42,"Test");
  System.out.println("ORIGINAL: "+notStorable);
  db.set(notStorable);
}
catch(Exception exc) {
  System.out.println(exc.toString());
  return;
}
finally {
  db.close();
}
db=Db4o.openFile(YAPFILENAME);
try {
  ObjectSet result=db.get(NotStorable.class);
  while(result.hasNext()) {
    NotStorable notStorable=(NotStorable)result.next();
    System.out.println("RETRIEVED: "+notStorable);
    db.delete(notStorable);
  }
}
finally {
  db.close();
}
  }
```

12.1.1. Using the constructor

12.1.2. Bypassing the constructor

```
[tryStoreWithoutCallConstructors]

Db4o.configure().exceptionsOnNotStorable(false);
Db4o.configure().objectClass(NotStorable.class)
    .callConstructor(false);
tryStoreAndRetrieve();
```

OUTPUT:

```
ORIGINAL: 42/Test: 4
RETRIEVED: 42/Test: 0
```

In this case our object seems to be nicely stored and retrieved, however, it has forgotten about its length, since db4o doesn't store transient members and the constructor code that sets it is not executed.

```
[tryStoreWithCallConstructors]

Db4o.configure().exceptionsOnNotStorable(true);
    Db4o.configure().objectClass(NotStorable.class)
.callConstructor(true);
    tryStoreAndRetrieve();
```

OUTPUT:

```
ORIGINAL: 42/Test: 4
com.db4o.ext.ObjectNotStorableException: Add a constructor that won't
throw exceptions, configure constructor calls, or provide a
translator to class 'com.db4o.f1.chapter6.NotStorable'.
```

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.)

12.2. 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.

12.2.1. 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.

12.2.2. ObjectConstructor

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

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.

12.3. A translator implementation

To translate NotStorable instances, we will pack their id and name values into an Object array to be stored and retrieve it from there again. Note that we don't have to do any work in onActivate(), since object reinstantiation is already fully completed in onInstantiate().

```
package com.db4o.f1.chapter6;

import com.db4o.*;
import com.db4o.config.*;

public class NotStorableTranslator
    implements ObjectConstructor {
  public Object onStore(ObjectContainer container,
      Object applicationObject) {
    System.out.println("onStore for "+applicationObject);
    NotStorable notStorable=(NotStorable)applicationObject;
    return new Object[]{new Integer(notStorable.getId()),
        notStorable.getName()};
  }

  public Object onInstantiate(ObjectContainer container,
      Object storedObject) {
    System.out.println("onInstantiate for "+storedObject);
    Object[] raw=(Object[])storedObject;
    int id=((Integer)raw[0]).intValue();
    String name=(String)raw[1];
    return new NotStorable(id,name);
  }

  public void onActivate(ObjectContainer container,
      Object applicationObject, Object storedObject) {
    System.out.println("onActivate for "+applicationObject
        +" / "+storedObject);
  }

  public Class storedClass() {
    return Object[].class;
  }
}
```

Let's try it out:

```
[storeWithTranslator]

Db4o.configure().objectClass(NotStorable.class)
.translate(new NotStorableTranslator());
    tryStoreAndRetrieve();
```

OUTPUT:

```
ORIGINAL: 42/Test: 4
onStore for 42/Test: 4
onInstantiate for [Ljava.lang.Object;@15dbaab
onActivate for 42/Test: 4 / [Ljava.lang.Object;@a9fd96
RETRIEVED: 42/Test: 4
```

12.4. 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.

12.5. Full source

```
package com.db4o.f1.chapter6;

import com.db4o.*;
import com.db4o.f1.*;

public class TranslatorExample extends Util {
  public static void main(String[] args) {
    tryStoreWithoutCallConstructors();
    tryStoreWithCallConstructors();
    storeWithTranslator();
  }

  public static void tryStoreWithoutCallConstructors() {
        Db4o.configure().exceptionsOnNotStorable(false);
        Db4o.configure().objectClass(NotStorable.class)
            .callConstructor(false);
        tryStoreAndRetrieve();
  }

  public static void tryStoreWithCallConstructors() {
    Db4o.configure().exceptionsOnNotStorable(true);
    Db4o.configure().objectClass(NotStorable.class)
        .callConstructor(true);
    tryStoreAndRetrieve();
  }

  public static void storeWithTranslator() {
    Db4o.configure().objectClass(NotStorable.class)
        .translate(new NotStorableTranslator());
    tryStoreAndRetrieve();
  }
  public static void tryStoreAndRetrieve() {
    ObjectContainer db=Db4o.openFile(YAPFILENAME);
    try {
      NotStorable notStorable = new NotStorable(42,"Test");
      System.out.println("ORIGINAL: "+notStorable);
      db.set(notStorable);
    }
    catch(Exception exc) {
      System.out.println(exc.toString());
      return;
    }
    finally {
      db.close();
    }
    db=Db4o.openFile(YAPFILENAME);
    try {
      ObjectSet result=db.get(NotStorable.class);
      while(result.hasNext()) {
        NotStorable notStorable=(NotStorable)result.next();
        System.out.println("RETRIEVED: "+notStorable);
        db.delete(notStorable);
      }
    }
    finally {
      db.close();
    }
  }
}
```

13. 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:

13.1. 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.

13.2. 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.

13.3. Further reading

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

Tuning

Indexes

14. 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(fa
lse);
```

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.

15. 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.

15.1. 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 .

15.2. 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);
```

16. 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

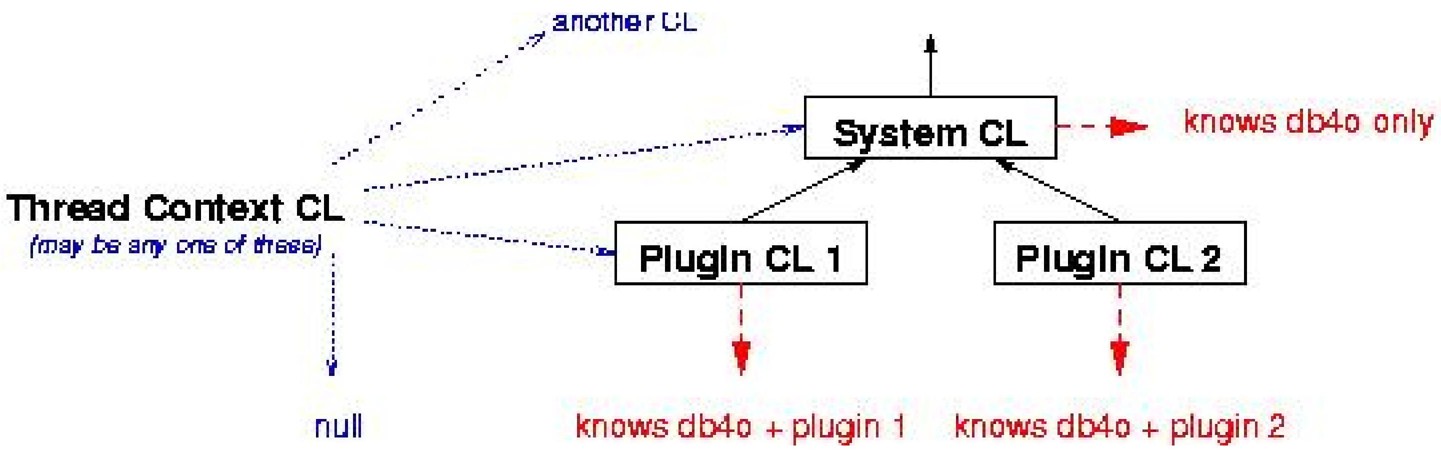
17. Classloader issues

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.

17.1. 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

17.2. 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.

17.3. 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.

17.3.1. 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

17.3.2. 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.

17.4. 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.

18. Servlets

Running db4o as the persistence layer of a Java web application is easy. There is no installation procedure - db4o is just another library in your application. There are only two issues that make web applications distinct from standalone programs from a db4o point of view. One is the more complex classloader environment - db4o needs to know itself (of course) and the classes to be persisted. Please refer to the classloader chapter for more information.

The other issue is configuring, starting and shutting down the db4o server correctly. This can be done at the servlet API layer or within the web application framework you are using.

On the servlet API layer, you could bind db4o server handling to the servlet context via an appropriate listener. A very basic sketch might look like this:

```
public class Db4oServletContextListener
    implements ServletContextListener {
  public static final String KEY_DB4O_FILE_NAME = "db4oFileName";
  public static final String KEY_DB4O_SERVER = "db4oServer";
  private ObjectServer server=null;

  public void contextInitialized(ServletContextEvent event) {
    close();
    ServletContext context=event.getServletContext();
    String filePath=context.getRealPath(
        "WEB-INF/db/"+context.getInitParameter(KEY_DB4O_FILE_NAME));
    server=Db4o.openServer(filePath,0);
    context.setAttribute(KEY_DB4O_SERVER,server);
    context.log("db4o startup on "+filePath);
  }

  public void contextDestroyed(ServletContextEvent event) {
    ServletContext context = event.getServletContext();
    context.removeAttribute(KEY_DB4O_SERVER);
    close();
    context.log("db4o shutdown");
  }

  private void close() {
    if(server!=null) {
      server.close();
    }
    server=null;
  }
}
```

This listener just has to be registered in the web.xml.

```
<context-param>
  <param-name>db4oFileName</param-name>
  <param-value>db4oweb.yap</param-value>
</context-param>
<listener>
  <listener-class>
    com.db4o.sample.web.Db4oServletContextListener
  </listener-class>
</listener>
```

Now db4o should be available to your application classes.

```
ObjectServer server=
    (ObjectServer)context.getAttribute("db4oServer");
```

A more complex and 'old school' example without using context listeners comes with the samples section of the db4o3 distribution that's still available from our web site.

However, We strongly suggest that you use the features provided by your framework and that you consider not exposing db4o directly to your application logic. (There is nothing db4o-specific about these recommendentations, we would vote for this in the presence of any persistence layer.)

19. 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 as an example
- write your own IoAdapter implementation that delegates raw file access to another adapter using the GoF decorator pattern.
- Implement 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());
```

However, you'll have to keep in mind that db4o will write partial udates. For example, it may write a full object and then only modify one field entry later one. Therefore it is not sufficient to en-/decrypt each access in isolation. You'll rather have to make up a tiling structure that defines the data chunks that have to be en-/decrypted together.

Another method to inject encryption capabilities into db4o for instances of specific classes only is to implement and configure an en-/decrypting translator.

20. Refactoring and "Schema Evolution"

20.1. Introduction

By Refactoring, we mean anything that will change the shape of the class as it is stored on the disk. In the context of other databases (where the database structure is said to be a "Schema"), refactoring is also called Schema Evolution.

20.2. What db4objects does today automatically or easily

Db4o automaticaly handles the most common refactoring or schema evolution changes for you.

- Changing the interface or API to a class
- Adding a field
- Removing a field

In addition, Db4o can rename classes and fields simply:

- Db4o.configure().objectClass("package.class").rename("newPackage.newClass");
- Db4o.configure().objectClass("package.class").objectField("oldField").rename("newField");

If you modify a field's type, db4o will internally create a new field of the same name, but with the new type. The values of the old typed field will still be present, but be hidden. If you change the type back to the old type the old values will still be there.

20.3. What is not yet automated

If you want to move a class to a new place in the inheritence hierarchy, you will need to follow the following procedure:

- Create the new class in the proper location using a temporary name
- Write manual code to create objects of the new class from the old ones
- Delete the old objects
- Run Defragment
- Rename the temporary class name back to the correct name

21. 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.

21.1. 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.

21.2. 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.

21.3. 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.

21.4. 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.

21.5. 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)

21.6. 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);
```

21.7. 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);
```

21.8. 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)
```

21.9. 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()
```

21.10. Increasing the maximum database file size

```
Db4o.configure().blockSize(newBlockSize);
Defragment.main(new String[] {"mydb.yap"});
```

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

Effect
By default db4o databases can have a maximum size of 2GB. By increasing the block size that db4o should internally use, the upper limit for database files sizes can be raised to multiples of 2GB. Any value between 1 byte (2GB) to 127 bytes (254GB) can be chosen as the block size.

Because of possible padding for objects that are not exact multiples in length of the block size, database files will naturally tend to be bigger if a higher value is chosen. Because of less file access cache hits a higher value will also have a negative effect on performance.

A very good choice for this value is 8 bytes, because that corresponds to the slot length of the pointers (address + length) that db4o internally uses.

Alternate Strategies
It can also be very efficient to use multiple ObjectContainers instead of one big one. Objects can be freely moved, copied and replicated between Objectcontainers.

22. Native Query Optimization

Native Queries will run out of the box in any environment. If an optimizer is present in the CLASSPATH and if optimisation is turned on, Native Queries will be converted to SODA queries where this is possible, allowing db4o to use indexes and optimized internal comparison algorithms.

If no optimizer is found in the CLASSPATH or if optimization is turned off, Native Quer may be executed by instantiating all objects, using SODA Evaluations. Naturally performance will not be as good in this case.

The Native Query optimizer is still under development to eventually "understand" all Java constructs. Current optimization supports the following constructs well:

- constants
- simple member access
- simple getter methods
- chaining of member access/simple getter methods
- primitive comparisons
- String#equals()
- arithmetic expressions
- boolean expressions

This list will constantly grow with the latest versions of db4o.

db4o for Java supplies three different possibilities to run optimized native queries, optimization at
(1) query execution time
(2) deployment time
(3) class loading time

The three options are described in the following:

22.1. Optimization at query execution time

Note: This will work with JDK1.1.

To enable code analysis and optimization of native query expressions at query query execution time, you just have to add db4o-5.0-nqopt.jar and bloat-1.0.jar to your CLASSPATH. Optimization can be turned on and off with the following configuration setting:

```
Db4o.configure().optimizeNativeQueries(boolean optimizeNQ);
```

22.2. Instrumenting class files

Note: Instrumented optimized classes will work with JDK1.1, but the optimization process itself requires at least JDK 1.2.

File instrumentation can be done either programmatically or during an Ant build.

22.2.1. Programmatic Instrumentation

To instrument all predicate classes in directory 'orig' whose package name starts with 'my.package' and store the modified files below directory 'instrumented', ensure that db4o-5.0-nqopt.jar and bloat-1.0.jar are in your CLASSPATH and use code like the following:

```
new com.db4o.nativequery.main.Db4oFileEnhancer().enhance(
    "orig",             // source directory
    "instrumented",     // target directory
    new String[]{       // class path
      "lib/my_application.jar",
      "lib/db4o-5.0-java1.x.jar"
    },
    "my.package"        // optional package prefix
  );
```

22.2.2. Ant Instrumentation

An equivalent Ant target might look like this:

```
<taskdef name="db4ooptimize"
classname="com.db4o.nativequery.main.Db4oFileEnhancerAntTask">    <clas
```

```
spath>
    <path path="lib/db4o-5.0-java1.x.jar" />
    <path path="lib/db4o-5.0-nqopt.jar" />
    <path path="lib/bloat-1.0.jar" />
    <path path="lib/db4o-5.0-java1.x.jar" />
  </classpath>
</taskdef>

<target name="optimize">
  <db4ooptimize
      srcdir="orig"
      targetdir="instrumented"
      packagefilter="my.package">
    <classpath>
      <path location="lib/my_application.jar" />
      <path path="lib/db4o-5.0-java1.x.jar" />
    </classpath>
  </db4oenhance>
</target>
```

All non-Predicate classes will just be copied to the target directory without modification.

22.3. Instrumenting classes at load time

Note: This will not work with JDK1.1.

If classes of an existing application are to be instrumented when they are loaded, a special ClassLoader needs to be used to run your application, com.db4o.nativequery.main.Db4oEnhancingClassLoader. Again db4o-5.0-nqopt.jar and bloat-1.0.jar need to be in the CLASSPATH.

All the native query code of your application would need to run in this ClassLoader. If we assume that you have a static starting method "goNative" in a class named "my.StarterClass", here is how you could run this method within the special native query ClassLoader:

```
ClassLoader loader=
  new com.db4o.nativequery.main.Db4oEnhancingClassLoader();
Class clazz=loader.loadClass("my.StarterClass");
```

```
Method method=clazz.getMethod("goNative",new Class[]{});
method.invoke(null,new Object[]{});
```

To start a full application in optimized mode, you can use the Db4oRunner utility class. If you would normally start your application like this:

```
$> java my.StarterClass some arguments
```

start Db4oRunner with the fully qualified name of your main class as the first argument and the actual arguments appended:

```
$> java com.db4o.nativequery.main.Db4oRunner my.StarterClass some
arguments
```

Further options:

- Setting the system class loader
  (-Djava.system.class.loader=com.db4o.nativequery.main.Db4oEnhancingClassLoader)
- Configuring Tomcat to use the optimizing class loader
  (Tomcat server.xml <Loader/> section)

22.4. Monitoring optimization

This feature still is quite basic but it will soon be improved. Currently you can only attach a listener to the ObjectContainer:

```
((YapStream)db).addListener(new Db4oQueryExecutionListener() {
  public void notifyQueryExecuted(Predicate pred, String msg) {
    System.err.println(msg);
  }
});
```

The listener will be notified on each native query call and will be passed the Predicate object processed and the success status of the optimization run:

YapStream.UNOPTIMIZED ("UNOPTIMIZED")
  if the predicate could not be optimized and is run in unoptimized mode

YapStream.PREOPTIMIZED ("PREOPTIMIZED")
  if the predicate already was optimized (due to class file or load time
  instrumentation)

YapStream.DYNOPTIMIZED ("DYNOPTIMIZED")
  if the predicate was optimized at query execution time

23. 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:

23.1. 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.

23.2. 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.

24. 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)

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.

24.1. 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".

24.2. 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.

25. Semaphores

db4o Semaphores are named flags that can only be owned by one client/transaction at one time. They are supplied to be used as locks for exclusive access to code blocks in applications and to signal states from one client to the server and to all other clients.

The naming of semaphores is up to the application. Any string can be used.

Semaphores are freed automatically when a client disconnects correctly or when a clients presence is no longer detected by the server, that constantly monitors all client connections.

Semaphores can be set and released with the following two methods:

```
ExtObjectContainer#setSemaphore(String name, int waitMilliSeconds);
ExtObjectContainer#releaseSemaphore(String name);
```

The concept of db4o semaphores is very similar to the concept of synchronization in OO programming languages:

Java

```
synchronized(monitorObject){

    // exclusive code block here

}
```

C#

```
lock(monitorObject){

    // exclusive code block here

}
```

db4o semaphore

```
if(objectContainer.ext().setSemaphore(semaphoreName, 1000){

    // exclusive code block here

    objectContainer.ext().releaseSemaphore(semaphoreName);
}
```

Although the principle of semaphores is very simple they are powerful enough for a wide range of usecases:

25.1. Locking objects

```
import com.db4o.*;
import com.db4o.ext.*;

/**
 * This class demonstrates a very rudimentary implementation
 * of virtual "locks" on objects with db4o. All code that is
 * intended to obey these locks will have to call lock() and
 * unlock().
 */
public class LockManager {

    private final String SEMAPHORE_NAME = "locked: ";
    private final int WAIT_FOR_AVAILABILITY = 300; // 300
milliseconds

    private final ExtObjectContainer _objectContainer;

    public LockManager(ObjectContainer objectContainer){
        _objectContainer = objectContainer.ext();
    }

    public boolean lock(Object obj){
        long id = _objectContainer.getID(obj);
        return _objectContainer.setSemaphore(SEMAPHORE_NAME + id,
WAIT_FOR_AVAILABILITY);
    }

    public void unlock(Object obj){
        long id = _objectContainer.getID(obj);
        _objectContainer.releaseSemaphore(SEMAPHORE_NAME + id);
    }
}
```

25.2. Ensuring Singletons

```
import com.db4o.*;
import com.db4o.query.*;

/**
 * This class demonstrates the use of a semaphore to ensure that only
 * one instance of a certain class is stored to an ObjectContainer.
 *
 * Caution !!! The getSingleton method contains a commit() call.
 */
public class Singleton {

    /**
     * returns a singleton object of one class for an
ObjectContainer.
     * <br><b>Caution !!! This method contains a commit() call.</b>
     */
    public static Object getSingleton(ObjectContainer
objectContainer, Class clazz) {

        Object obj = queryForSingletonClass(objectContainer, clazz);
        if (obj != null) {
            return obj;
        }
        String semaphore = "Singleton#getSingleton_" +
clazz.getName();

        if (!objectContainer.ext().setSemaphore(semaphore, 10000)) {
            throw new RuntimeException("Blocked semaphore " +
semaphore);
        }

        obj = queryForSingletonClass(objectContainer, clazz);

        if (obj == null) {

            try {
                obj = clazz.newInstance();
            } catch (InstantiationException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            }

            objectContainer.set(obj);

            /* !!! CAUTION !!!
             * There is a commit call here.
             *
             * The commit call is necessary, so other transactions
             * can see the new inserted object.
             */
            objectContainer.commit();

        }

        objectContainer.ext().releaseSemaphore(semaphore);

        return obj;
    }

    private static Object queryForSingletonClass(ObjectContainer
objectContainer, Class clazz) {
        Query q = objectContainer.query();
        q.constrain(clazz);
        ObjectSet objectSet = q.execute();
        if (objectSet.size() == 1) {
            return objectSet.next();
        }
        if (objectSet.size() > 1) {
            throw new RuntimeException(
                "Singleton problem. Multiple instances of: " +
clazz.getName());
        }
        return null;
    }

}
```

25.3. Limiting the number of users

```
import java.io.*;
import com.db4o.*;

/**
 * This class demonstrates the use of semaphores to limit the
 * number of logins to a server.
 */
public class LimitLogins {

    static final String HOST = "localhost";
    static final int PORT = 4455;
    static final String USER = "db4o";
    static final String PASSWORD = "db4o";

    static final int MAXIMUM_USERS = 10;

    public static ObjectContainer login(){

        ObjectContainer objectContainer;
        try {
            objectContainer = Db4o.openClient(HOST, PORT, USER,
PASSWORD);
```

```
        } catch (IOException e) {
            return null;
        }

        boolean allowedToLogin = false;

        for (int i = 0; i < MAXIMUM_USERS; i++) {
            if(objectContainer.ext().setSemaphore("max_user_check_" + 
i, 0)){
                allowedToLogin = true;
                break;
            }
        }

        if(! allowedToLogin){
            objectContainer.close();
            return null;
        }

        return objectContainer;
    }
}
```

25.4. Controlling log-in information

```
import java.util.*;
import com.db4o.*;
import com.db4o.config.*;
import com.db4o.ext.*;
import com.db4o.query.*;

/**
 * This class demonstrates how semaphores can be used
 * to rule out race conditions when providing exact and
 * up-to-date information about all connected clients
 * on a server. The class also can be used to make sure
 * that only one login is possible with a give user name
 * and ipAddress combination.
 */
public class ConnectedUser {

    static final String SEMAPHORE_CONNECTED = "ConnectedUser_";
    static final String SEMAPHORE_LOCK_ACCESS = 
"ConnectedUser_Lock_";

    static final int TIMEOUT = 10000;  // concurrent access timeout 
10 seconds

    String userName;
    String ipAddress;

    public ConnectedUser(String userName, String ipAddress){
        this.userName = userName;
        this.ipAddress = ipAddress;
    }

    // make sure to call this on the server before opening the 
database
    // to improve querying speed
    public static void configure(){
        ObjectClass objectClass = 
Db4o.configure().objectClass(ConnectedUser.class);
        objectClass.objectField("userName").indexed(true);
        objectClass.objectField("ipAddress").indexed(true);
    }

    // call this on the client to ensure to have a ConnectedUser 
record
    // in the database file and the semaphore set
    public static void login(ObjectContainer client, String userName, 
String ipAddress){
        if(! client.ext().setSemaphore(SEMAPHORE_LOCK_ACCESS, 
TIMEOUT)){
            throw new RuntimeException("Timeout trying to get access 
to ConnectedUser lock");
        }
        Query q = client.query();
        q.constrain(ConnectedUser.class);
        q.descend("userName").constrain(userName);
        q.descend("ipAddress").constrain(ipAddress);
        if(q.execute().size() == 0){
            client.set(new ConnectedUser(userName, ipAddress));
            client.commit();
        }
        String connectedSemaphoreName = SEMAPHORE_CONNECTED +
userName + ipAddress;
        boolean unique =
client.ext().setSemaphore(connectedSemaphoreName, 0);
        client.ext().releaseSemaphore(SEMAPHORE_LOCK_ACCESS);
        if(! unique){
            throw new RuntimeException("Two clients with same
userName and ipAddress");
        }
    }

    // here is your list of all connected users, callable on the
server
    public static List connectedUsers(ObjectServer server){
        ExtObjectContainer serverObjectContainer =
server.ext().objectContainer().ext();
        if(serverObjectContainer.setSemaphore(SEMAPHORE_LOCK_ACCESS,
TIMEOUT)){
            throw new RuntimeException("Timeout trying to get access
to ConnectedUser lock");
        }
        List list = new ArrayList();
        Query q = serverObjectContainer.query();
        q.constrain(ConnectedUser.class);
        ObjectSet objectSet = q.execute();
        while(objectSet.hasNext()){
            ConnectedUser connectedUser =
(ConnectedUser)objectSet.next();
            String connectedSemaphoreName =
                SEMAPHORE_CONNECTED +
                connectedUser.userName +
                connectedUser.ipAddress;
if(serverObjectContainer.setSemaphore(connectedSemaphoreName, TIMEOUT
)){
                serverObjectContainer.delete(connectedUser);
            }else{
                list.add(connectedUser);
            }
        }
        serverObjectContainer.commit();
serverObjectContainer.releaseSemaphore(SEMAPHORE_LOCK_ACCESS);
 return list;
    }
}
```

26. Messaging

In client/server mode the TCP connection between the client and the server can also be used to send messages from the client to the server.

Possible usecases could be:
- shutting down and restarting the server
- triggering server backup
- using a customized login strategy to restrict the number of allowed client connections

Here is some example code how this can be done.

First we need to decide on a class that we want to use as the message. Any object that is storable in db4o can be used as a message, but strings and other simple types will not be carried (as they are not storable in db4o on their own). Let's create a dedicated class:

```
class MyClientServerMessage {

    private String info;

    public MyClientServerMessage(String info){
        this.info = info;
    }

    public Strint toString(){
        return "MyClientServerMessage: " + info;
    }

}
```

Now we have to add some code to the server to react to arriving messages. This can be done by configuring a MessageRecipient object on the server. Let's simply print out all objects that arrive as messages. For the following we assume that we already have an ObjectServer object, opened with Db4o.openServer() .

```
objectServer.ext().configure().setMessageRecipient(

    new MessageRecipient() {

        public void processMessage(
            ObjectContainer objectContainer,
            Object message) {

            // message objects will arrive in this code block

            System.out.println(message);

        }

    });
```

Here is what we would do on the client to send the message:

```
MessageSender sender =
  clientObjectContainer.ext().configure().getMessageSender();

sender.send(new MyClientServerMessage("Hello from client.");
```

The MessageSender object on the client can be reused to send multiple messages.

27. Object Manager

The db4o Object Manager is a GUI tool to browse and query the contents of any db4o database file.
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.

27.1. Installation

Object Manager has to be downloaded seperately from the main db4o distributions. Please visit the db4o Download Center and choose the installation appropriate for your system. The following distributions are currently available:
- 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 any location of your choice and unpack the downloaded zip file there.

27.1.1. Running

27.1.1.1. 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.

27.1.1.2. 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.

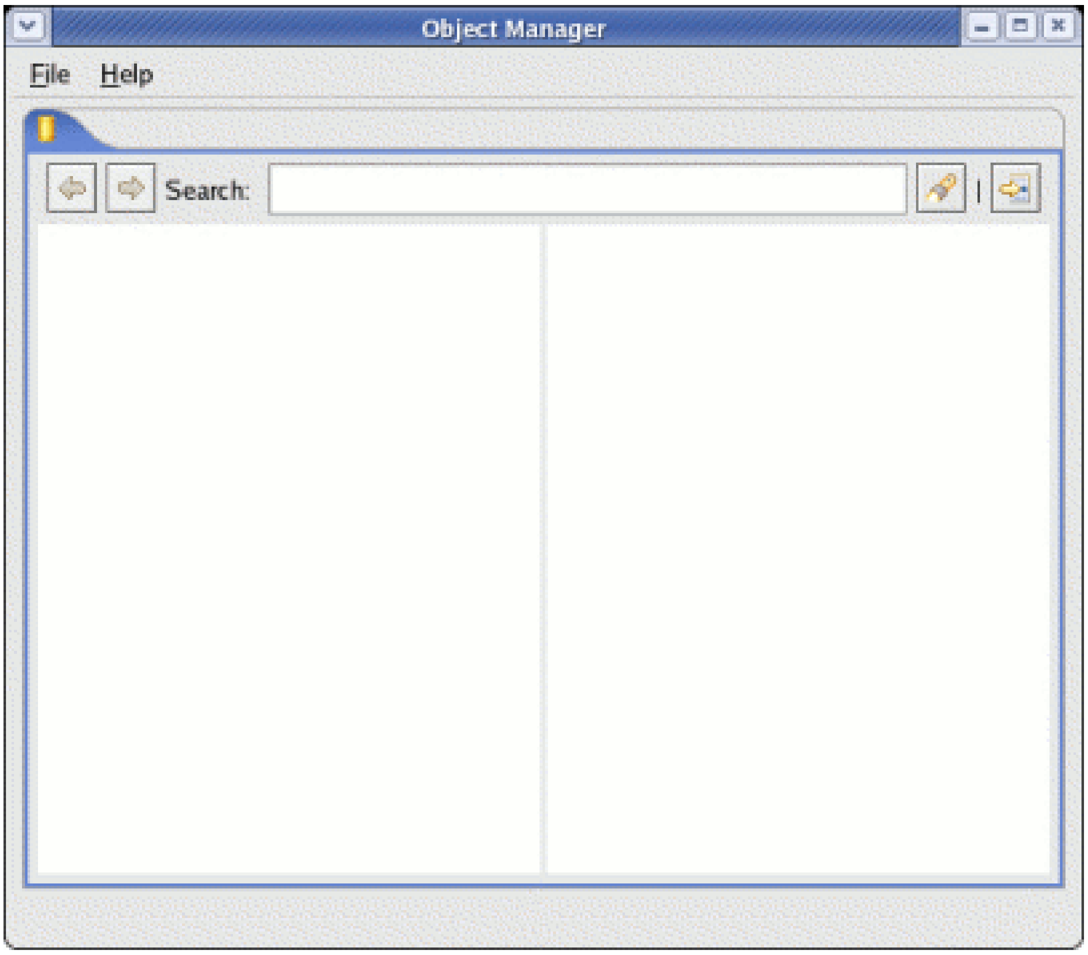
27.1.1.3. Linux

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.

27.2. Object Manager Tour

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



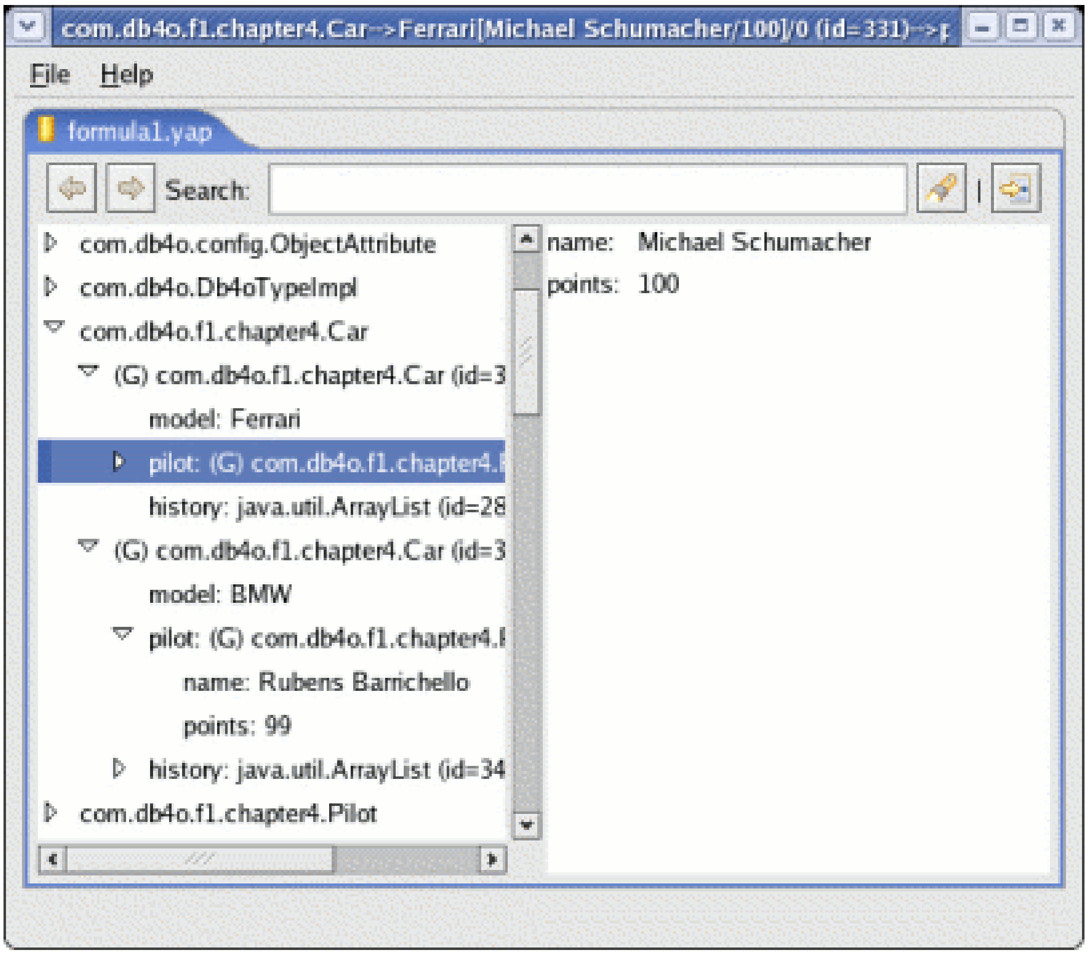
At this point, you can either
- Open a db4o database file
- Open a connection to a db4o database server

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:


com.db4o.f1.chapter4.Car->Ferrari[Michael Schumacher/100]/0 (id=331)->
File
Help
formula1.yap
Search:

```
▷ com.db4o.config.ObjectAttribute
▷ com.db4o.Db4oTypeImpl
▽ com.db4o.f1.chapter4.Car
    ▽ (G) com.db4o.f1.chapter4.Car (id=3
        model: Ferrari
      ▷ pilot: (G) com.db4o.f1.chapter4.
        history: java.util.ArrayList (id=28
    ▽ (G) com.db4o.f1.chapter4.Car (id=3
        model: BMW
      ▽ pilot: (G) com.db4o.f1.chapter4.
          name: Rubens Barrichello
          points: 99
      ▷ history: java.util.ArrayList (id=34
▷ com.db4o.f1.chapter4.Pilot
```

```
name:   Michael Schumacher
points: 100
```

27.2.1. 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).

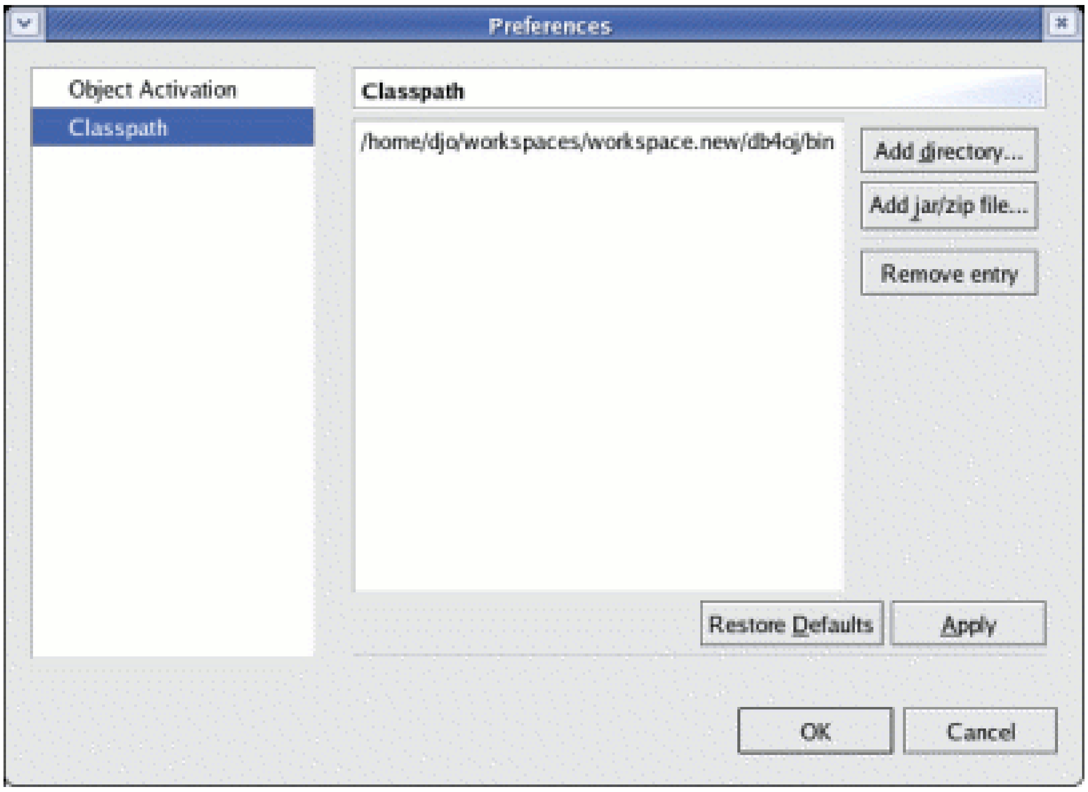
The Object Manager can display some data types more naturally if it has the class definitions available, so it provides a way to extend its classpath at runtime so that it can find the Java or .NET class definitions. This is done through the preferences dialog, accessed through "File | Preferences..."

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:



Classpath

/home/djo/workspaces/workspace.new/db4oj/bin

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.

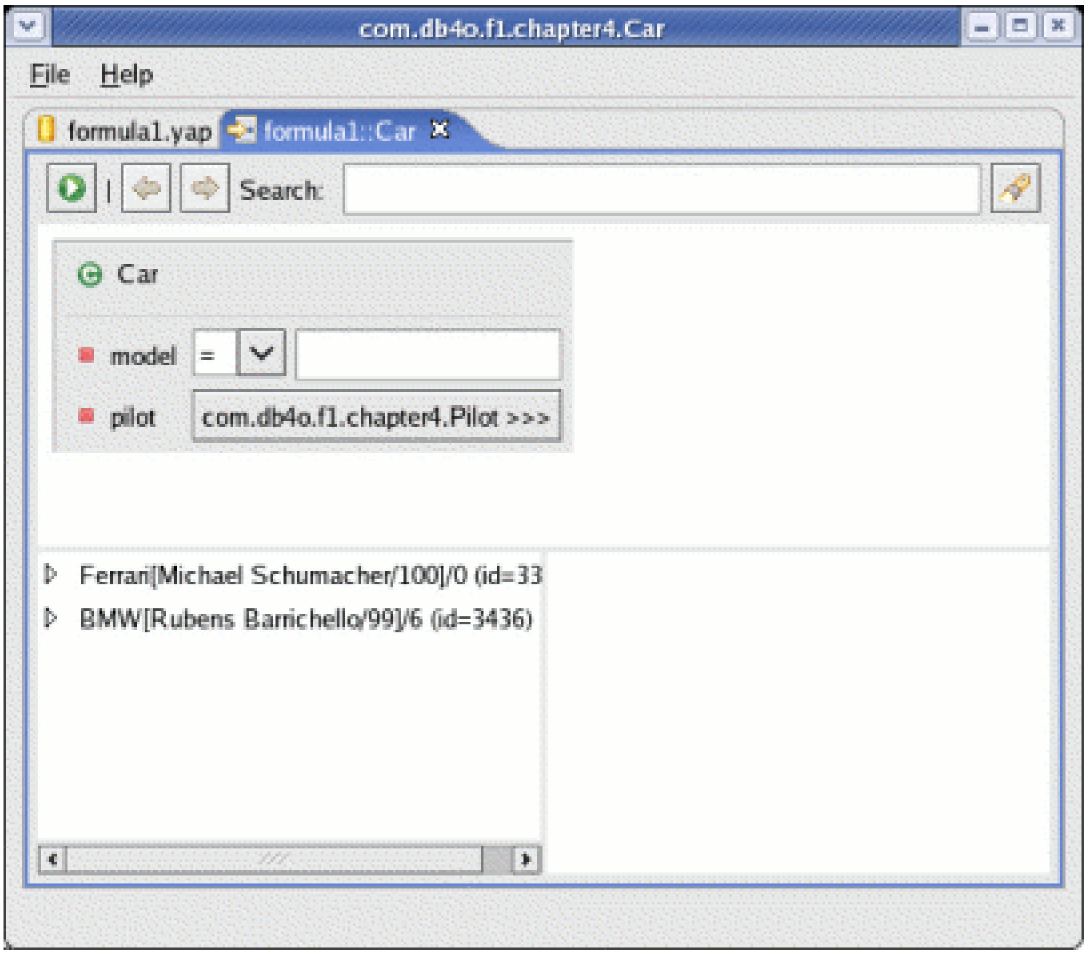
27.2.2. Querying for objects

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:



```
Ferrari[Michael Schumacher/100]/0 (id=33
BMW[Rubens Barrichello/99]/6 (id=3436)
```

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.

27.3. 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.

28. RegressionTests

28.1. The built-in db4o regression tests

Db4o includes a fairly comprehensive suite of regression tests to help prevent errors from creeping into the code over time. Since no regression test suite is perfect, we are documenting how to work with our tests in the hope that customers will benefit from seeing the test suite improve.

28.2. Running the regression tests

Db4o includes its complete source code and test suite packaged in zip files in the src folder of the distribution. The first step in running the test suite is to unpack these zip files in the src folder so that they can be compiled an run. From the command line on Linux, this would look like:

- cd /path/to/db4o-<version>/src
- unzip *src*
- unzip *test*

Users of other platforms can easily substitute their favoriate tools in this procedure.

Next, create an Eclipse project, pointing to the folder where you unpacked db4o. For Eclipse 3.1M7, the procedure is:

- Make sure the Console view is open so you can see the output
- File | New | Project...
- Pick "Java Project" and click "Next"
- Type a project name
- Pick the "Create Project from existing source" radio button
- Enter "/path/to/db4o-<version>" in the box requesting the source directory
- Click "Finish"
- Expand the "src" folder, then the com.db4o.test package
- Right-click "AllTests.java"
- Choose "Run As | Java Application"

28.3. Switching individual tests on or off

Individual tests may be turned on or off by commenting or uncommenting lines corresponding to each individual test in the suite. The test suite is defined by classes beginning with com.db4o.test.AllTestsConf*. Each test class is listed in an array returned by the tests() method in the *Conf* class.

29. License

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

29.1. 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 db4o.license.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

29.2. 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.

29.3. Bundled 3rd Party Licenses

The db4o distribution comes with the following 3rd party libraries:

-Apache Ant(Apache Software License)

Files: lib/ant.jar, lib/ant.license.txt

Ant can be used as a make tool for class file based optimization of native
queries at compile time.

This product includes software developed by the Apache Software Foundation
(http://www.apache.org/).

-BLOAT(GNU LGPL)

Files: lib/bloat-1.0.jar, lib/bloat.license.txt

Bloat is used for bytecode analysis during native queries optimization. It needs to be on the classpath during runtime at load time or query execution time for just-in-time optimization. Preoptimized class files are not dependent on BLOAT at runtime.

30. 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 online forums

http://forums.db4o.com/forums