Welcome

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

This tutorial was written to get you started with db4o as quickly as possible. Before you start, please make sure that you have downloaded the latest db4o distribution from the db4objects website.

developer.db4o.com

You are invited to join the db4o community in the public db4o forums to ask for help at any time.

Please also try out the keyword search functionality on the db4o knowledgebase.

Links

Here are some further links on developer.db4o.com that you may find useful:

All Downloads

Release Note Blog

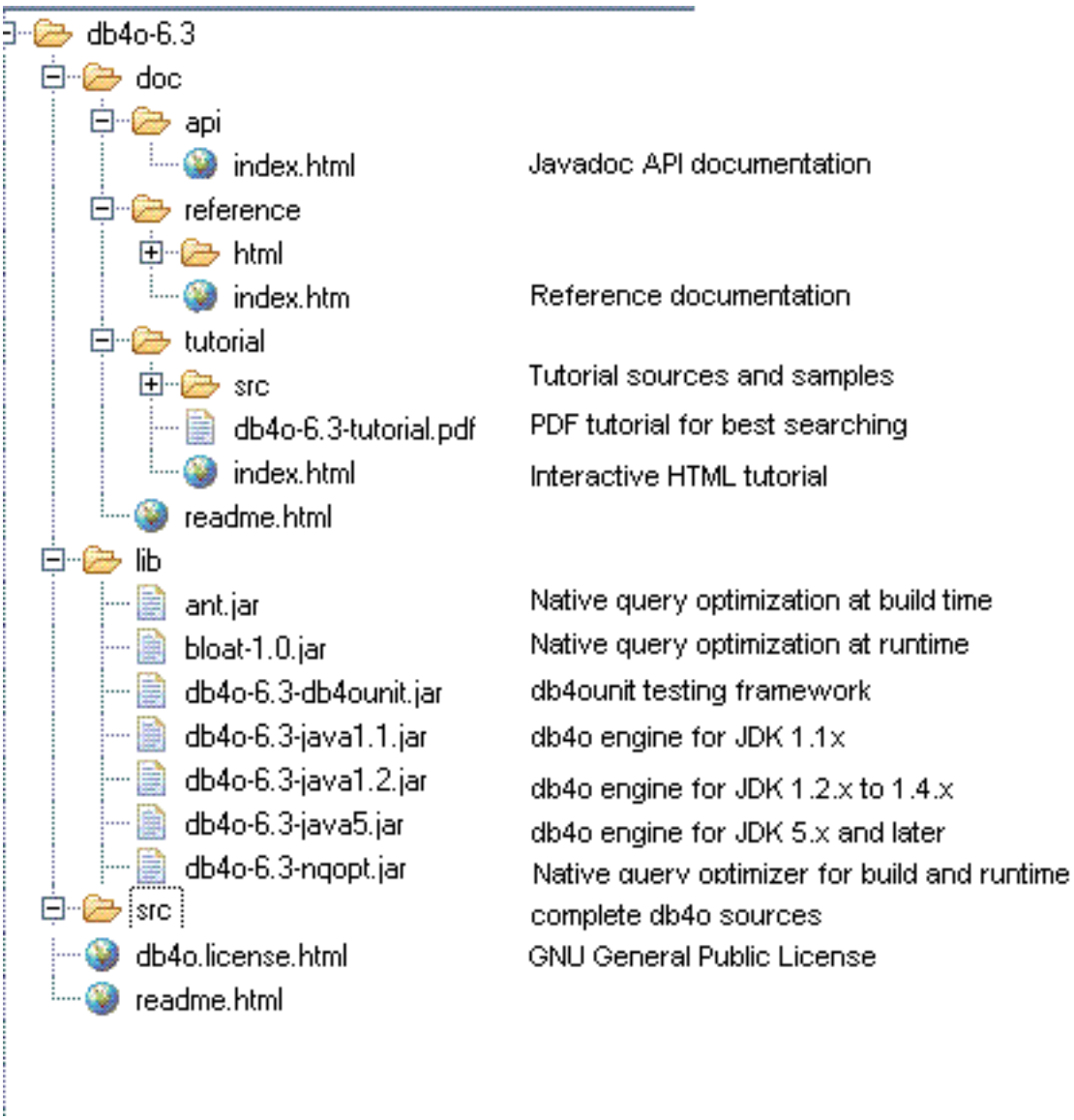
SVN Access

Design Wiki

Community Projects

Download Contents

The db4o Java distribution comes as one zip file, db4o-6.3-java.zip. When you unzip this file, you get the following directory structure:



Please take a look at all the supplied documentation formats to choose the one that works best for you:

db4o-6.3/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.

db4o-6.3/doc/reference/index.html

The reference documentation is a complete compilation for experienced db4o users. It is maintained online.

db4o-6.3/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-6.3/doc/tutorial/db4o-6.3-tutorial.pdf

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

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-6.3/lib/. You will only need one of the following libraries, not all of them.

db4o-6.3-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-6.3-java1.2.jar

is built for all Java JDKs between 1.2 and 1.4.

db4o-6.3-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 you would copy the db4o-*.jar to the /lib/ folder under your project and add db4o to your project as a library.

1.3. API Overview

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

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

com.db4o

The com.db4o java 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 java 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 method to get to the advanced features.

com.db4o.config

The com.db4o.config java 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 java 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 Soda 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() and provide the path to your database file as the parameter, to obtain an ObjectContainer instance. ObjectContainer represents "The Database", and will be your primary interface to db4o. Closing the ObjectContainer with the #close() method will close the database file and release all resources associated with it.

```
// accessDb4o

ObjectContainer db=Db4o.openFile(Util.DB4OFILENAME);
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 result ObjectSet :

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

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
Michael Schumacher/100
Rubens Barrichello/99
```

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
Michael Schumacher/111
Rubens Barrichello/99
```

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

```
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.DB4OFILENAME).delete();
        accessDb4o();
        new File(Util.DB4OFILENAME).delete();
        ObjectContainer db=Db4o.openFile(Util.DB4OFILENAME);
        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.DB4OFILENAME);
        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() 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:

```
// storePilots

db.set(new Pilot("Michael Schumacher",100));
db.set(new Pilot("Rubens Barrichello",99));
```

```
// 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:

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

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

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.DB4OFILENAME);
        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.

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.

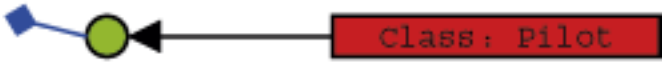
```
// retrieveAllPilots

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

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

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
Michael Schumacher/100
Rubens Barrichello/99
```

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.DB4OFILENAME);
        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
Rubens Barrichello/99
Michael Schumacher/100
```

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

```
// retrieveCarByPilotQBE

Pilot pilotproto=new Pilot("Rubens Barrichello",0);
Car carproto=new Car(null);
carproto.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]
```

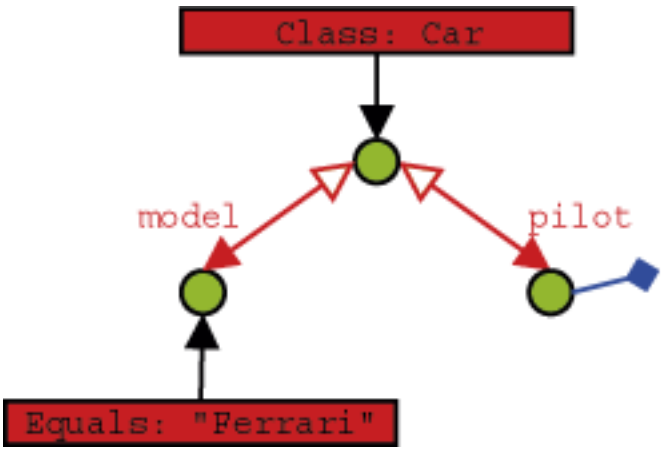
We have seen that descending into a query provides us with another query. Starting out from a query root we can descend in multiple directions. In practice this is the same as ascending from one child to

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

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");
    }
});
if (!result.hasNext()) {
    System.out.println("Pilot not found!");
    return;
}
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.DB4OFILENAME).delete();
        ObjectContainer db=Db4o.openFile(Util.DB4OFILENAME);
        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.DB4OFILENAME);
            updatePilotSeparateSessionsPart2(db);
            db.close();
            updatePilotSeparateSessionsImprovedPart1();
            db=Db4o.openFile(Util.DB4OFILENAME);
            updatePilotSeparateSessionsImprovedPart2(db);
            db.close();
            db=Db4o.openFile(Util.DB4OFILENAME);
            updatePilotSeparateSessionsImprovedPart3(db);
            deleteFlat(db);
            db.close();
            deleteDeepPart1();
            db=Db4o.openFile(Util.DB4OFILENAME);
            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");
            }
```

```
        });
        if (!result.hasNext()) {
            System.out.println("Pilot not found!");
            return;
        }
        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 : 1186835768130 : 0.1,0.2,0.3
BMW[Rubens Barrichello/99]/2 : 1186835768130 : 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 : 1186835768130 : 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 : 1186835768130 : 0.1,0.2,0.3,
BMW[Rubens Barrichello/99]/2 : 1186835768130 : 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 : 1186835768130 : 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]/2 : 1186835768130 : 0.1,0.2,0.3
BMW[Rubens Barrichello/99]/2 : 1186835768130 : 0.2,0.4,0.6
Ferrari[Michael Schumacher/100]/1 : 1186835768678 : 0.1,0.2,0.3
```

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 : 1186835768130 : 0.1,0.2,0.3
BMW[Rubens Barrichello/99]/2 : 1186835768130 : 0.2,0.4,0.6
```

(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.DB4OFILENAME).delete();
        ObjectContainer db=Db4o.openFile(Util.DB4OFILENAME);
        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.DB4OFILENAME);
            updateCarPart2(db);
            updateCollection(db);
            db.close();
            deleteAllPart1();
            db=Db4o.openFile(Util.DB4OFILENAME);
            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 : Sat Aug 11 12:36:08 GMT 2007 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : water
temp : 0.8
```

```
// retrieveAllSensorReadoutsQBE

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

OUTPUT:

```
6
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : oil
pressure : 1.5
```

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 : Sat Aug 11 12:36:08 GMT 2007 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : oil
pressure : 1.5
```

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 : Sat Aug 11 12:36:08 GMT 2007 : oil
```

```
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:08 GMT 2007 : oil
pressure : 1.5
```

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.DB4OFILENAME).delete();
        ObjectContainer db=Db4o.openFile(Util.DB4OFILENAME);
        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 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 0.8
BMW[Rubens Barrichello/99]/4 : Sat Aug 11 12:36:09 GMT 2007 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/4 : Sat Aug 11 12:36:09 GMT 2007 : oil
temp : 0.6000000000000001
BMW[Rubens Barrichello/99]/4 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 1.4000000000000001
BMW[Rubens Barrichello/99]/4 : Sat Aug 11 12:36:09 GMT 2007 : oil
pressure : 2.4
```

```
BMW[Rubens Barrichello/99]/14 : Sat Aug 11 12:36:09 GMT 2007 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/14 : Sat Aug 11 12:36:09 GMT 2007 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : oil
temp : 0.9
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 2.0
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : oil
pressure : 3.3
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : oil
temp : 1.2000000000000002
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 0.2
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 2.6
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : oil
pressure : 4.2
```

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 : Sat Aug 11 12:36:09 GMT 2007 : oil
```

```
temp : 0.0
BMW[Rubens Barrichello/99]/5 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 0.2
BMW[Rubens Barrichello/99]/5 : Sat Aug 11 12:36:09 GMT 2007 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/5 : Sat Aug 11 12:36:09 GMT 2007 : 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 : Sat Aug 11 12:36:09 GMT 2007 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/5 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 0.2
BMW[Rubens Barrichello/99]/5 : Sat Aug 11 12:36:09 GMT 2007 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/5 : Sat Aug 11 12:36:09 GMT 2007 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 0.8
BMW[Rubens Barrichello/99]/7 : Sat Aug 11 12:36:09 GMT 2007 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/8 : Sat Aug 11 12:36:09 GMT 2007 : oil
temp : 0.6000000000000001
BMW[Rubens Barrichello/99]/9 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 1.4000000000000001
BMW[Rubens Barrichello/99]/10 : Sat Aug 11 12:36:09 GMT 2007 : oil
pressure : 2.4
BMW[Rubens Barrichello/99]/11 : Sat Aug 11 12:36:09 GMT 2007 : oil
temp : 0.9
BMW[Rubens Barrichello/99]/12 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 2.0
BMW[Rubens Barrichello/99]/13 : Sat Aug 11 12:36:09 GMT 2007 : oil
pressure : 3.3
BMW[Rubens Barrichello/99]/14 : Sat Aug 11 12:36:09 GMT 2007 : oil
temp : 1.2000000000000002
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 2.6
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : 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);
```

```
// 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 : Sat Aug 11 12:36:09 GMT 2007 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 0.2
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 0.8
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : oil
temp : 0.6000000000000001
```

```
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 1.4000000000000001
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : oil
pressure : 2.4
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : oil
temp : 0.9
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 2.0
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : oil
pressure : 3.3
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : oil
temp : 1.2000000000000002
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : water
temp : 2.6
BMW[Rubens Barrichello/99]/15 : Sat Aug 11 12:36:09 GMT 2007 : 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.

The new Transparent Activation feature introduced in db4o-6.3 allows you to forget about all these worries providing activation on request. More information can be obtained from Transparent Activation Framework article online or from the offline version of the Reference documentation.

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.DB4OFILENAME).delete();
        ObjectContainer db=Db4o.openFile(Util.DB4OFILENAME);
        try {
            storeCar(db);
            db.close();
            setCascadeOnUpdate();
            db=Db4o.openFile(Util.DB4OFILENAME);
            takeManySnapshots(db);
            db.close();
            db=Db4o.openFile(Util.DB4OFILENAME);
            retrieveAllSnapshots(db);
            db.close();
            db=Db4o.openFile(Util.DB4OFILENAME);
            retrieveSnapshotsSequentially(db);
            retrieveSnapshotsSequentiallyImproved(db);
            db.close();
            setActivationDepth();
            db=Db4o.openFile(Util.DB4OFILENAME);
            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.

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.DB4OFILENAME).delete();
        ObjectContainer db=Db4o.openFile(Util.DB4OFILENAME);
        try {
            storeCarCommit(db);
            db.close();
            db=Db4o.openFile(Util.DB4OFILENAME);
            listAllCars(db);
            storeCarRollback(db);
            db.close();
            db=Db4o.openFile(Util.DB4OFILENAME);
            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.DB4OFILENAME,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
```

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
```

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
```

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.DB4OFILENAME,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:

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

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:

```
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
```

```
// 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:

```
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
```

9.3. Native Queries in Client/Server mode

A quite subtle problem may occur if you're using Native Queries as anonymous (or just non-static) inner classes in Client/Server mode. Anonymous/non-static inner class instances carry a synthetic field referencing their outer class instance - this is just Java's way of implementing inner class access to fields or final method locals of the outer class without introducing any notion of inner classes at all at

the bytecode level. If such a non-static inner class predicate cannot be converted to S.O.D.A. form on the client, it will be wrapped into an evaluation and passed to the server, introducing the same problem already mentioned in the evaluation chapter : db4o will try to transfer the evaluation, using the standard platform serialization mechanism. If this fails, it will just try to pass it to the server via db4o marshalling. However, this may fail, too, for example if the outer class references any local db4o objects like ObjectContainer, etc., resulting in an ObjectNotStorableException.

So to be on the safe side with NQs in C/S mode, you should declare Predicates as top-level or static inner classes only. Alternatively, you could either make sure that the outer classes containing Predicate definitions could principally be persisted to db4o, too, and don't add significant overhead to the predicate (the appropriate value for 'significant'
being your choice) or ensure during development that all predicates used actually can be optimized to S.O.D.A. form.

9.4. 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().clientServer().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 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()
                .clientServer().getMessageSender();

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

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

9.5. 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.db4o";

  /**
   * 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().clientServer().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.

```
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()
                    .clientServer().getMessageSender();

      // send an instance of a StopServer object
```

```
      messageSender.send(new StopServer());

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

9.6. 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. A much more thorough documentation is provided in the reference that you should have also received with the download or online.

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

- You could browse the remaining chapters. They are a selection of themes from the reference that very frequently come up as questions in our http://forums.db4o.com/forums/.

-*(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 Jira or visit our forums at http://forums.db4o.com/forums/.

9.7. 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.DB4OFILENAME).delete();
        accessLocalServer();
        new File(Util.DB4OFILENAME).delete();
        ObjectContainer db=Db4o.openFile(Util.DB4OFILENAME);
        try {
            setFirstCar(db);
            setSecondCar(db);
        }
        finally {
            db.close();
        }
        configureDb4o();
        ObjectServer server=Db4o.openServer(Util.DB4OFILENAME,0);
        try {
            queryLocalServer(server);
            demonstrateLocalReadCommitted(server);
            demonstrateLocalRollback(server);
        }
        finally {
            server.close();
        }
        accessRemoteServer();
        server=Db4o.openServer(Util.DB4OFILENAME,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.DB4OFILENAME,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.DB4OFILENAME,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: 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.DB4OFILENAME).delete();
    ObjectContainer db=Db4o.openFile(Util.DB4OFILENAME);
    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. 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. A more complete description of Configuration usage and scope can also be obtained from the Reference documentation.

Some hints around using configuration calls:

11.1. 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()
```

A separate configuration instance can be obtained with

```
Configuration config = Db4o.newConfiguration()
```

or cloned from the global configuration with

```
Configuration config = Db4o.cloneConfiguration()
```

Configuration can be submitted when an ObjectContainer/ObjectServer is opened:

```
Db4o.openFile(config, filename)
```

Otherwise the global configuration context will be 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.

11.2. 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.unicode(false);
```

Configuration settings are **not** stored in db4o database files. Accordingly the same configuration has to be submitted **every time** an ObjectContainer/ObjectServer is opened. For using db4o in client/server mode it is recommended to use the same 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 creates an appropriate configuration and to deploy this class both to the server and to all clients.

12. 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 application.

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.

13. 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.

13.1. 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.

13.2. 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(Foo.class).generateUUIDs(true);
```

The respective methods for working with UUIDs are:

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

14. 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:

- compile-time constants
- simple member access
- primitive comparisons
- #equals() on primitive wrappers and Strings
- #contains()/#startsWith()/#endsWith() for Strings
- arithmetic expressions
- boolean expressions
- static field access
- array access for static/predicate fields
- arbitrary method calls on static/predicate fields (without candidate based params)
- candidate methods composed of the above
- chained combinations of the above

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

Note that the current implementation doesn't support polymorphism and multiline methods yet.

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:

14.1. Optimization at query execution time

Note: This will not 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.x-nqopt.jar and bloat-1.x.jar to your CLASSPATH. Optimization can be turned on and off with the following configuration setting:

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

14.2. 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.

14.2.1. 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-6.3-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-6.3-java1.x.jar"
    },
    "my.package"        // optional package prefix
  );
```

14.2.2. 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-6.3-java1.x.jar" />
    <path path="lib/db4o-6.3-nqopt.jar" />
    <path path="lib/bloat-6.3.jar" />
    <path path="lib/db4o-6.3-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-6.3-java1.x.jar" />
    </classpath>
  </db4oenhance>
</target>
```

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

14.3. 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-6.3-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)

14.4. Monitoring optimization

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

```
((ObjectContainerBase)db).getNativeQueryHandler().addListener(new
Db4oQueryExecutionListener() {
  public void notifyQueryExecuted(NQOptimizationInfo info) {
    System.err.println(info);
  }
});
```

The listener will be notified on each native query call and will be passed the Predicate object processed, the optimized expression tree (if
successful) and the success status of the optimization run:

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

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

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

15. License

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

15.1. 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

15.2. Opensource Compatibility license (dOCL)

The db4o Opensource Compatibility License (dOCL) is designed for free/open source projects that want to embed db4o but do not want to (or are not able to) license their derivative work under the GPL in its entirety. This initiative aims to proliferate db4o into many more open source projects by providing compatibility for projects licensed under Apache, LGPL, BSD, EPL, and others, as required by our users. The terms of this license are available here: "dOCL" agreement.

15.3. 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.

15.4. 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.

16. Contacting db4objects Inc.

db4objects Inc.
1900 South Norfolk Street
Suite 350
San Mateo, CA, 94403
USA

Phone
+1 (650) 577-2340

Fax
+1 (650) 240-0431

Sales
Fill out our sales contact form on the db4o website
or
mail to sales@db4o.com

Support
Visit our free Community Forums
or log into your dDN Member Portal (dDN Members Only).

Careers
career@db4o.com

Partnering
partner@db4o.com