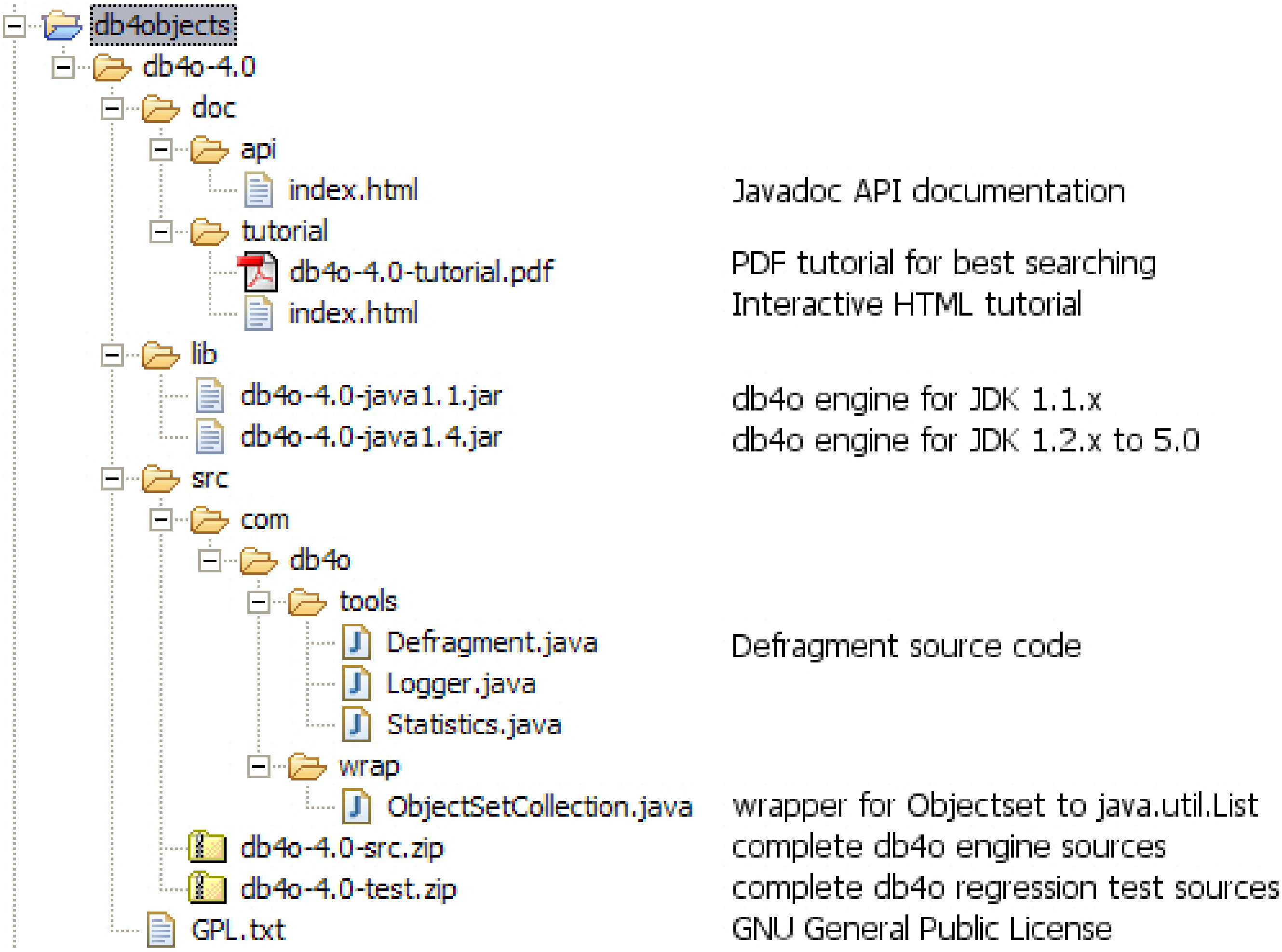
Welcome

db4o is the native Java and .NET object database for embedded applications.

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

The db4o Java distribution comes as one zip file, db4o-4.0-java.zip. When you unzip this file, you get the following directory structure:



```
db4objects
  db4o-4.0
    doc
      api
        index.html                      Javadoc API documentation
      tutorial
        db4o-4.0-tutorial.pdf           PDF tutorial for best searching
        index.html                      Interactive HTML tutorial
    lib
      db4o-4.0-java1.1.jar              db4o engine for JDK 1.1.x
      db4o-4.0-java1.4.jar              db4o engine for JDK 1.2.x to 5.0
    src
      com
        db4o
          tools
            Defragment.java             Defragment source code
            Logger.java
            Statistics.java
          wrap
            ObjectSetCollection.java    wrapper for Objectset to java.util.List
      db4o-4.0-src.zip                  complete db4o engine sources
      db4o-4.0-test.zip                 complete db4o regression test sources
    GPL.txt                             GNU General Public License
```

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

db4o-4.0/doc/tutorial/index.html

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

db4o-4.0/doc/tutorial/db4o-4.0-tutorial.pdf
The PDF version of the tutorial allows best fulltext search capabilities.

Java and .NET

db4o is available for Java and for .NET. This tutorial was written for Java . The structure of the .NET distribution is slightly different, so please use the tutorial for the version that you plan to experiment with first.

1. First Glance

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

1.1. The db4o engine...

The db4o object database engine consists of one single jar file. This is all that you need to program against. The versions supplied with the distribution can be found in /db4o-4.0/lib/.☐

db4o-4.0-java1.1.jar
will run with most Java JDKs that supply JDK 1.1.x functionality such as reflection and Exception handling. That includes many IBM J9 configurations, Symbian and Savaje.

db4o-4.0-java1.4.jar
is built for all Java JDKs between 1.2 and 5.0.

1.2. Installation

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

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

- the "lib" folder should appear below your project
- choose db4o-*.jar in this folder
- hit OK twice

1.3. API

The API documentation for db4o is supplied as JavaDocs in db4o-4.0/doc/api/index.html. While you read through this tutorial it may be helpful to look into the API documentation occasionaly. For the start, the packages com.db4o and com.db4o.query are all that you need to worry about.

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

```
com.db4o.ObjectContainer
```

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

In case you wonder why you only see very few methods in an ObjectContainer, here is why: The db4o interface is supplied in two steps in two 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.
- We hint how a very-light-version of db4o should look like.

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

2. First Steps

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

First we create a native class such as:

```
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;
    }
}
```

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

2.1. Storing objects

To store an object, we simply open a db4o ObjectContainer and call set(), passing the object as a parameter.

```
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.

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

OUTPUT:

```
Stored Rubens Barrichello/99
```

Closing the ObjectContainer will release all resources associated with it.

2.2. Retrieving objects

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

```
public static void listResult{
    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:

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

OUTPUT:

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

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

To query for a pilot by name:

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

OUTPUT:

```
1
Michael Schumacher/100
```

Let's retrieve a pilot by exact points:

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

OUTPUT:

```
1
Michael Schumacher/100
```

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

2.3. Updating objects

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

```
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
```

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

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

2.4. Deleting objects

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

```
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.

```
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.

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

2.5. Conclusion

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

2.6. Full source

```
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 {
    public static void main(String[] args) {
        new File(Util.YAPFILENAME).delete();
        ObjectContainer db=Db4o.openFile(Util.YAPFILENAME);
        try {
            storeFirstPilot(db);
            storeSecondPilot(db);
            retrieveAllPilots(db);
            retrievePilotByName(db);
            retrievePilotByExactPoints(db);
            updatePilot(db);
            deleteFirstPilotByName(db);
            deleteSecondPilotByName(db);
        }
        finally {
            db.close();
        }
    }

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

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

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

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

- There are queries that simply cannot be expressed with QBE: Retrieve all pilots with more than 100 points, for example.
- Creating a prototype object may have unwanted side effects.
- We may want to query for field default values.

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

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

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

OUTPUT:

```
Stored Michael Schumacher/100
```

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

OUTPUT:

```
Stored Rubens Barrichello/99
```

3.1. Simple queries

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

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

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

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.

```
Query query=db.query();
query.constrain(Pilot.class);
query.descend("name").constrain("Michael Schumacher");
ObjectSet result=query.execute();
Util.listResult(result);
```

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

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.

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

OUTPUT:

```
1
Michael Schumacher/100
```

3.2. Advanced queries

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

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

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

OUTPUT:

```
1
Rubens Barrichello/99
```

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

```
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();
Util.listResult(result);
```

OUTPUT:

```
0
```

```
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();
Util.listResult(result);
```

OUTPUT:

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

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

```
Query query=db.query();
query.constrain(Pilot.class);
query.descend("points")
        .constrain(new Integer(99)).greater();
ObjectSet result=query.execute();
Util.listResult(result);
```

OUTPUT:

```
1
Michael Schumacher/100
```

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

```
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();
Util.listResult(result);
db.delete(somebody);
```

OUTPUT:

```
1
Somebody else/0
```

It is also possible to have db4o sort the results.

```
Query query=db.query();
query.constrain(Pilot.class);
query.descend("name").orderAscending();
ObjectSet result=query.execute();
Util.listResult(result);
query.descend("name").orderDescending();
result=query.execute();
Util.listResult(result);
```

OUTPUT:

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

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

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

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

OUTPUT:

3.3. Conclusion

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

3.4. Full source

```
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 {
    public static void main(String[] args) {
        ObjectContainer db=Db4o.openFile(Util.YAPFILENAME);
        try {
            storeFirstPilot(db);
            storeSecondPilot(db);
            retrieveAllPilots(db);
            retrievePilotByName(db);
            retrievePilotByExactPoints(db);
            retrieveByNegation(db);
            retrieveByConjunction(db);
            retrieveByDisjunction(db);
            retrieveByComparison(db);
            retrieveByDefaultFieldValue(db);
            retrieveSorted(db);
            clearDatabase(db);
        }
        finally {
            db.close();
        }
    }

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

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

```
public static void retrieveAllPilots(ObjectContainer db) {
    Query query=db.query();
    query.constrain(Pilot.class);
    ObjectSet result=query.execute();
    Util.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();
    Util.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();
    Util.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();
    Util.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();
    Util.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();
    Util.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();
    Util.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();
    Util.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();
    Util.listResult(result);
    query.descend("name").orderDescending();
    result=query.execute();
    Util.listResult(result);
}

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

4. Structured objects

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

```
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.

```
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.

```
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.

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

OUTPUT:

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

We can also query for all pilots, of course.

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

OUTPUT:

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

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

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

```
ObjectSet result=db.get(carproto);
Util.listResult(result);
```

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

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

4.2.2. Query API

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

```
Query query=db.query();
query.constrain(Car.class);
query.descend("pilot").descend("name")
        .constrain("Rubens Barrichello");
ObjectSet result=query.execute();
Util.listResult(result);
```

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

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

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

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

OUTPUT:

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

4.3. Updating structured objects

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

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

OUTPUT:

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

Let's modify the pilot, too.

```
ObjectSet result=db.get(new Car("Ferrari"));
Car found=(Car)result.next();
found.getPilot().addPoints(1);
```

```
db.set(found);
result=db.get(new Car("Ferrari"));
Util.listResult(result);
```

OUTPUT:

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

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

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

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

OUTPUT:

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

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

4.3.1. Update depth

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

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

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

```
Db4o.configure().objectClass("com.db4o.f1.chapter2.Car")
        .cascadeOnUpdate(true);
```

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

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

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

OUTPUT:

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

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

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

OUTPUT:

```
3
Michael Schumacher/100
Rubens Barrichello/99
Somebody else/1
```

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.

```
Db4o.configure().objectClass("com.db4o.f1.chapter2.Car")
        .cascadeOnDelete(true);
```

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

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

OUTPUT:

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

4.4.2. Recursive deletion revisited

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

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

OUTPUT:

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

```
Pilot proto=new Pilot(null,0);
ObjectSet result=db.get(proto);
Util.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 activating this feature.

Let's clear our database for the next chapter.

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

4.5. Conclusion

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

4.6. Full source

```
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.Query;


public class StructuredExample {
    private final static String FILENAME="f1.yap";

    public static void main(String[] args) {
        new File(FILENAME).delete();
        ObjectContainer db=Db4o.openFile(FILENAME);
        try {
            storeFirstCar(db);
            storeSecondCar(db);
            retrieveAllCarsQBE(db);
            retrieveAllPilotsQBE(db);
            retrieveCarByPilotQBE(db);
            retrieveCarByPilotNameQuery(db);
            retrieveCarByPilotProtoQuery(db);
            updateCar(db);
            updatePilotSingleSession(db);
            updatePilotSeparateSessionsPart1(db);
            db.close();
            db=Db4o.openFile(FILENAME);
            updatePilotSeparateSessionsPart2(db);
            db.close();
            updatePilotSeparateSessionsImprovedPart1();
            db=Db4o.openFile(FILENAME);
            updatePilotSeparateSessionsImprovedPart2(db);
            db.close();
            db=Db4o.openFile(FILENAME);
            updatePilotSeparateSessionsImprovedPart3(db);
            deleteFlat(db);
            db.close();
            deleteDeepPart1();
            db=Db4o.openFile(FILENAME);
            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);
        Util.listResult(result);
    }

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

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

public static void updatePilotSingleSession(
            ObjectContainer db) {
    ObjectSet result=db.get(new Car("Ferrari"));
    Car found=(Car)result.next();
    found.getPilot().addPoints(1);
```

```
        db.set(found);
        result=db.get(new Car("Ferrari"));
        Util.listResult(result);
    }

    public static void updatePilotSeparateSessionsPart1(
                ObjectContainer db) {
     ObjectSet result=db.get(new Car("Ferrari"));
        Car found=(Car)result.next();
        found.getPilot().addPoints(1);
        db.set(found);
    }

    public static void updatePilotSeparateSessionsPart2(
                ObjectContainer db) {
        ObjectSet result=db.get(new Car("Ferrari"));
        Util.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.get(new Car("Ferrari"));
        Car found=(Car)result.next();
        found.getPilot().addPoints(1);
        db.set(found);
    }

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

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

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

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

5. Collections and Arrays

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

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

    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.

```
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.

```
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.

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

OUTPUT:

```
2
```

```
BMW[Rubens Barrichello/99]/2 : 1099751294337 : 0.2,0.4,0.6
BMW[Rubens Barrichello/99]/2 : 1099751294337 : 0.1,0.2,0.3
```

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

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

OUTPUT:

```
1
BMW[Rubens Barrichello/99]/2 : 1099751294337 : 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.

```
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);
Util.listResult(result);
```

OUTPUT:

```
1
```

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

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

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

```
OUTPUT:
2
[]
[BMW[Rubens Barrichello/99]/2 : 1099751294337 : 0.1,0.2,0.3,
BMW[Rubens Barrichello/99]/2 : 1099751294337 : 0.2,0.4,0.6]
```

This doesn't work with arrays, though.

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

```
OUTPUT:
0
```

5.2.2. Query API

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

```
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();
Util.listResult(result);
```

OUTPUT:

```
1
BMW[Rubens Barrichello/99]/2 : 1099751294337 : 0.1,0.2,0.3
```

```
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();
Util.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 .

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

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

OUTPUT:

```
3
BMW[Rubens Barrichello/99]/3 : 1099751294337 : 0.2,0.4,0.6
BMW[Rubens Barrichello/99]/3 : 1099751294337 : 0.1,0.2,0.3
BMW[Rubens Barrichello/99]/3 : 1099751294446 :
0.30000000000000004,0.6000000000000001,0.8999999999999999
```

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

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

```
Query query=db.query();
query.constrain(Car.class);
ObjectSet result=query.descend("history").execute();
List coll=(List)result.next();
coll.remove(0);
db.set(coll);
Car proto=new Car(null,null);
result=db.get(proto);
while(result.hasNext()) {
    Car car=(Car)result.next();
    for (int idx=0;idx<car.getHistory().length;idx++) {
        System.out.println(car.getHistory()[idx]);
    }
}
```

```
OUTPUT:
BMW[Rubens Barrichello/99]/2 : 1099751294337 : 0.2,0.4,0.6
BMW[Rubens Barrichello/99]/2 : 1099751294446 :
0.30000000000000004,0.6000000000000001,0.8999999999999999
```

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

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

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

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

5.4. db4o custom collections

db4o also provides customized collection implementations, tweaked for use with db4o. We will get to that in a later chapter when we have finished our first walkthrough.

5.5. 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?

5.6. Full source

```java
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 {
    private final static String FILENAME="f1.yap";

    public static void main(String[] args) {
        new File(FILENAME).delete();
        ObjectContainer db=Db4o.openFile(FILENAME);
        try {
            storeFirstCar(db);
            storeSecondCar(db);
            retrieveAllSensorReadouts(db);
            retrieveSensorReadoutQBE(db);
            retrieveCarQBE(db);
            retrieveCollections(db);
            retrieveArrays(db);
            retrieveSensorReadoutQuery(db);
            retrieveCarQuery(db);
            db.close();
            updateCarPart1();
            db=Db4o.openFile(FILENAME);
            updateCarPart2(db);
            updateCollection(db);
            db.close();
            deleteAllPart1();
            db=Db4o.openFile(FILENAME);
            deleteAllPart2(db);
            retrieveAllSensorReadouts(db);
        }
        finally {
            db.close();
        }
    }

    public static void storeFirstCar(ObjectContainer db) {
        Car car1=new Car("Ferrari");
        Pilot pilot1=new Pilot("Michael Schumacher",100);
        car1.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 retrieveAllSensorReadouts(
                ObjectContainer db) {
        SensorReadout proto=new SensorReadout(null,null,null);
        ObjectSet result=db.get(proto);
        Util.listResult(result);
    }

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

    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);
        Util.listResult(result);
    }

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

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

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

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

    public static void updateCollection(ObjectContainer db) {
        Query query=db.query();
        query.constrain(Car.class);
        ObjectSet result=query.descend("history").execute();
        List coll=(List)result.next();
        coll.remove(0);
        db.set(coll);
        Car proto=new Car(null,null);
        result=db.get(proto);
        while(result.hasNext()) {
            Car car=(Car)result.next();
            for (int idx=0;idx<car.getHistory().length;idx++) {
                System.out.println(car.getHistory()[idx]);
            }
        }
    }

    public static void deleteAllPart1() {
        Db4o.configure().objectClass(Car.class)
          .cascadeOnDelete(true);
    }
    public static void deleteAllPart2(ObjectContainer db) {
        ObjectSet result=db.get(new Car(null,null));
        while(result.hasNext()) {
            db.delete(result.next());
        }
        ObjectSet readouts=db.get(
                new SensorReadout(null,null,null));
        while(readouts.hasNext()) {
            db.delete(readouts.next());
        }
    }
}
```

6. Inheritance

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

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

```
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.

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

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.

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

OUTPUT:

```
4
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.30000000000000004
```

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

OUTPUT:

```
6
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
```

```
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 0.6
```

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

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

OUTPUT:

```
6
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 0.6
```

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

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

```
OUTPUT:
6
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 0.6
```

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

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

```
OUTPUT:
13
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
```

```
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 0.6
Db4oDatabase: [B@9ba5aa
[]
[BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.0, BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET
2004 : water temp : 0.2, BMW[Rubens Barrichello/99]/6 : Sat Nov 06
15:28:14 CET 2004 : oil pressure : 0.6, BMW[Rubens Barrichello/99]/6
: Sat Nov 06 15:28:14 CET 2004 : oil temp : 0.30000000000000004,
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.8, BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET
2004 : oil pressure : 1.5]
Ferrari[Michael Schumacher/100]/0
BMW[Rubens Barrichello/99]/6
Rubens Barrichello/99
Michael Schumacher/100
```

6.3. Updating and deleting

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

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

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

OUTPUT:

6.4. Conclusion

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

6.5. Full source

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

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


public class InheritanceExample {
    private final static String FILENAME="f1.yap";

    public static void main(String[] args) {
        new File(FILENAME).delete();
        ObjectContainer db=Db4o.openFile(FILENAME);
        try {
            storeFirstCar(db);
            storeSecondCar(db);
            retrieveTemperatureReadoutsQBE(db);
            retrieveAllSensorReadoutsQBE(db);
            retrieveAllSensorReadoutsQBEAlternative(db);
            retrieveAllSensorReadoutsQuery(db);
            retrieveAllObjects(db);
            deleteAllObjects(db);
        }
        finally {
            db.close();
        }
    }

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

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

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

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

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

7. Deep graphs

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

```
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.

```
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.

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

Let's collect a few sensor readouts.

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

7.2. Retrieving

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

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

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

OUTPUT:

```
BMW[Rubens Barrichello/99]/4 : Sat Nov 06 15:28:14 CET 2004 : oil
```

```
pressure : 4.2
BMW[Rubens Barrichello/99]/4 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 2.6
BMW[Rubens Barrichello/99]/4 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 1.2000000000000002
BMW[Rubens Barrichello/99]/4 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 3.3
BMW[Rubens Barrichello/99]/4 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 2.0
BMW[Rubens Barrichello/99]/4 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.9
BMW[Rubens Barrichello/99]/4 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/5 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.2
BMW[Rubens Barrichello/99]/5 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/5 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 2.4
BMW[Rubens Barrichello/99]/5 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 1.4000000000000001
BMW[Rubens Barrichello/99]/5 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.6000000000000001
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.8
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.30000000000000004
```

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

```
ObjectSet result=db.get(new Car(null));
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 Nov 06 15:28:14 CET 2004 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/5 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.2
BMW[Rubens Barrichello/99]/5 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/5 : Sat Nov 06 15:28:14 CET 2004 : 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.

```
ObjectSet result=db.get(new Car(null));
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 Nov 06 15:28:14 CET 2004 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/5 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.2
BMW[Rubens Barrichello/99]/5 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/5 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.8
BMW[Rubens Barrichello/99]/7 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/8 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.6000000000000001
BMW[Rubens Barrichello/99]/9 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 1.4000000000000001
BMW[Rubens Barrichello/99]/10 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 2.4
BMW[Rubens Barrichello/99]/11 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.9
BMW[Rubens Barrichello/99]/12 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 2.0
BMW[Rubens Barrichello/99]/13 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 3.3
BMW[Rubens Barrichello/99]/14 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 1.2000000000000002
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : water
```

```
temp : 2.6
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : 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.

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

OUTPUT:

```
ObjectSet result=db.get(new Car(null));
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 Nov 06 15:28:14 CET 2004 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.2
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : oil
```

```
pressure : 0.6
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 0.8
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.6000000000000001
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 1.4000000000000001
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 2.4
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 0.9
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 2.0
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 3.3
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : oil
temp : 1.2000000000000002
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : water
temp : 2.6
BMW[Rubens Barrichello/99]/15 : Sat Nov 06 15:28:14 CET 2004 : oil
pressure : 4.2
```

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

Don't forget to clean up the database.

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

OUTPUT:

7.3. 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.

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

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

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

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

- I you're stuck, see if the FAQ can solve your problem, browse the information on our web site, check if your problem is submitted to Bugzilla yet or join our newsgroup at news.dbv4odev.com .

7.4. Full source

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

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



public class DeepExample {
    private final static String FILENAME="f1.yap";

    public static void main(String[] args) {
        new File(FILENAME).delete();
        ObjectContainer db=Db4o.openFile(FILENAME);
        try {
            storeCar(db);
            db.close();
            setCascadeOnUpdate();
            db=Db4o.openFile(FILENAME);
            takeManySnapshots(db);
            db.close();
            db=Db4o.openFile(FILENAME);
            retrieveAllSnapshots(db);
            db.close();
            db=Db4o.openFile(FILENAME);
            retrieveSnapshotsSequentially(db);
            retrieveSnapshotsSequentiallyImproved(db);
            db.close();
            setActivationDepth();
            db=Db4o.openFile(FILENAME);
            retrieveSnapshotsSequentially(db);
            deleteAllObjects(db);
        }
        finally {
            db.close();
        }
    }

    public static void storeCar(ObjectContainer db) {
        Pilot pilot=new Pilot("Rubens Barrichello",99);
        Car car=new Car("BMW");
        car.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(new Car(null));
        Car car=(Car)result.next();
        for(int i=0;i<5;i++) {
            car.snapshot();
        }
        db.set(car);
    }

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

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

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

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

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

8. License

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

8.1. General Public License (GPL)

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

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

8.2. Commercial License

For incorporation into own commercial products and for use together with redistributed software that is not placed under the GPL, db4o is also available under a commercial license.

Visit the purchasing area on the db4o website or contact db4o sales for licensing terms and pricing.

9. Contacting db4objects Inc.

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

Phone
+1 (650) 577-2340

Fax
+1 (650) 577-2341

General Enquiries

info@db4o.com

Sales

sales@db4o.com

or

fill out our sales contact form on the db4o website

Careers

career@db4o.com

Partnering

partner@db4o.com

Support

support@db4o.com

or post to our newsgroup

news://news.db4odev.com/db4o.users