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:
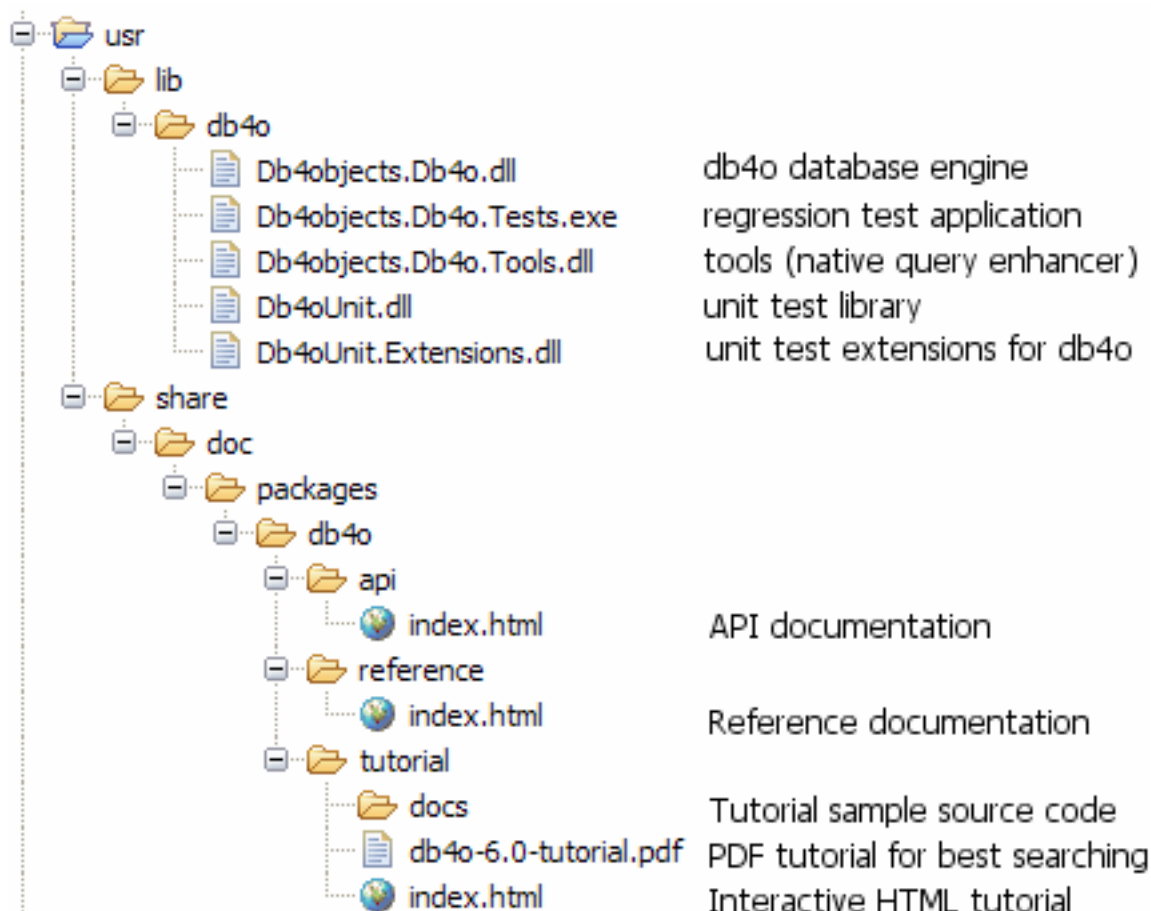
All Downloads

Release Note Blog

SVN Access

Design Wiki

Community Projects

Download Contents

The db4o Mono distribution is available from the db4o download center in 3 versions: *db4o-6.0-mono.tar.gz* (binary and source tarball), *db4o-6.0-mono.noarch.rpm* (binary RPRM), *db4o-6.0-mono.src.rpm* (source RPM). The noarch RPM can be built from the source RPM. After installing/unzipping you will find the following directory structure on your machine:

db4o-6.0-mono.noarch.rpm

```
usr
   lib
      db4o
         Db4objects.Db4o.dll            db4o database engine
         Db4objects.Db4o.Tests.exe      regression test application
         Db4objects.Db4o.Tools.dll      tools (native query enhancer)
         Db4oUnit.dll                   unit test library
         Db4oUnit.Extensions.dll        unit test extensions for db4o
   share
      doc
         packages
            db4o
               api
                  index.html            API documentation
               reference
                  index.html            Reference documentation
               tutorial
                  docs                  Tutorial sample source code
                  db4o-6.0-tutorial.pdf PDF tutorial for best searching
                  index.html            Interactive HTML tutorial
```

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

/usr/share/doc/package/db4o/api/index.html

The API documentation for db4o is supplied as a set of HTML pages. While you read through this tutorial, it may be helpful to look into the API documentation occasionaly.

/usr/share/doc/package/db4o/reference/index.html

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

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

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

/usr/share/doc/packages/db4o/tutorial/db4o-6.x-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 DLL. This is all that you need to program against. The version supplied with the distribution can be found in /usr/lib/db4o/.

/usr/lib/db4o/db4o.dll

The standard db4o engine for the Mono environment.

1.2. Installation

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

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 namespace perspective and you may find related functionality to the theme you are currently reading up on.

For starters, the Db4objects.Db4o and Db4objects.Db4o.Query namespaces are all that you need to worry about.

Db4objects.Db4o

The Db4objects.Db4o namespace contains most of the functionality you will commonly need when you work with db4o. Two classes of special interest are Db4objects.Db4o.Db4oFactory and Db4objects.Db4o.IObjectContainer.

The Db4oFactory 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 IObjectContainer: This is your db4o database.
- An IObjectContainer can either be a database in single-user mode or a client connection to a db4o server.
- Every IObjectContainer owns one transaction. All work is transactional. When you open an IObjectContainer, you are in a transaction, when you commit() or rollback(), the next transaction is started immediately.
- Every IObjectContainer 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.
- IObjectContainers are intended to be kept open as long as you work against them. When you close an IObjectContainer, all database references to objects in RAM will be discarded.

Db4objects.Db4o.Ext

In case you wonder why you only see very few methods in an IObjectContainer, here is why: The db4o interface is supplied in two steps in two namespaces, Db4objects.Db4o and Db4objects.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 IObjectContainer object is also an IExtObjectContainer. You can cast the IObjectContainer to IExtObjectContainer or you can use the .Ext() method to access advanced features.

Db4objects.Db4o.Config

The Db4objects.Db4o.Config namespace contains types necessary to configure db4o. The objects and interfaces within are discussed in the Configuration section.

Db4objects.Db4o.Query

The Db4objects.Db4o.Query namespace 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:

```
namespace Db4objects.Db4o.Tutorial.F1.Chapter1
{
    public class Pilot
    {
        string _name;
        int _points;

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

        public string Name
        {
            get
            {
                return _name;
            }
        }

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

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


        override public string ToString()
        {
            return string.Format("{0}/{1}", _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 Db4oFactory.OpenFile() and provide the path to your database file as the parameter, to obtain an IObjectContainer instance. IObjectContainer represents "The Database", and will be your primary interface to db4o. Closing the IObjectContainer with the #Close() method will close the database file and release all resources associated with it.

```
// accessDb4o


IObjectContainer db = Db4oFactory.OpenFile(Util.YapFileName);
    try
    {
        // do something with db4o
    }
    finally
    {
        db.Close();
    }
```

For the following examples we will assume that our environment takes care of opening and closing the IObjectContainer 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);
    Console.WriteLine("Stored {0}", pilot1);
```

OUTPUT:

Stored Michael Schumacher/100

We'll need a second pilot, too.

```
// storeSecondPilot

Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
    db.Set(pilot2);
    Console.WriteLine("Stored {0}", 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 IObjectSet instance. We will use a convenience method #ListResult() to display the contents of our result IObjectSet :

```
public static void ListResult(IObjectSet result)
    {
        Console.WriteLine(result.Count);
        foreach (object item in result)
        {
            Console.WriteLine(item);
        }
    }
```

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

```
// retrieveAllPilotQBE

Pilot proto = new Pilot(null, 0);
    IObjectSet 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


IObjectSet result = db.Get(typeof(Pilot));
    ListResult(result);
```

OUTPUT:
2
Michael Schumacher/100
Rubens Barrichello/99

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

To query for a pilot by name:

```
// retrievePilotByName


Pilot proto = new Pilot("Michael Schumacher", 0);
    IObjectSet 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);
    IObjectSet 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

IObjectSet result = db.Get(new Pilot("Michael Schumacher", 0));
    Pilot found = (Pilot)result.Next();
    found.AddPoints(11);
    db.Set(found);
    Console.WriteLine("Added 11 points for {0}", 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

IObjectSet result = db.Get(new Pilot("Michael Schumacher", 0));
    Pilot found = (Pilot)result.Next();
    db.Delete(found);
    Console.WriteLine("Deleted {0}", found);
    RetrieveAllPilots(db);
```

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

Let's delete the other one, too.

```
// deleteSecondPilotByName

IObjectSet result = db.Get(new Pilot("Rubens Barrichello", 0));
    Pilot found = (Pilot)result.Next();
    db.Delete(found);
    Console.WriteLine("Deleted {0}", found);
    RetrieveAllPilots(db);
```

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

```
using System;
using System.IO;

using Db4objects.Db4o.Query;
using Db4objects.Db4o.Tutorial;

namespace Db4objects.Db4o.Tutorial.F1.Chapter1
{
    public class FirstStepsExample : Util
    {
        public static void Main(string[] args)
        {
            File.Delete(Util.YapFileName);
            AccessDb4o();
            File.Delete(Util.YapFileName);
            IObjectContainer db =
Db4oFactory.OpenFile(Util.YapFileName);
            try
            {
```

```csharp
            StoreFirstPilot(db);

            StoreSecondPilot(db);

            RetrieveAllPilots(db);

            RetrievePilotByName(db);

            RetrievePilotByExactPoints(db);

            UpdatePilot(db);

            DeleteFirstPilotByName(db);

            DeleteSecondPilotByName(db);

        }
        finally
        {
            db.Close();
        }
    }


    public static void AccessDb4o()
    {
        IObjectContainer db =
Db4oFactory.OpenFile(Util.YapFileName);
        try
        {
            // do something with db4o
        }
        finally
        {
            db.Close();
        }
    }


    public static void StoreFirstPilot(IObjectContainer db)
    {
        Pilot pilot1 = new Pilot("Michael Schumacher", 100);
        db.Set(pilot1);
        Console.WriteLine("Stored {0}", pilot1);
    }


    public static void StoreSecondPilot(IObjectContainer db)
    {
        Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
        db.Set(pilot2);
        Console.WriteLine("Stored {0}", pilot2);
```

```csharp
        }

        public static void RetrieveAllPilotQBE(IObjectContainer db)
        {
            Pilot proto = new Pilot(null, 0);
            IObjectSet result = db.Get(proto);
            ListResult(result);
        }

        public static void RetrieveAllPilots(IObjectContainer db)
        {
            IObjectSet result = db.Get(typeof(Pilot));
            ListResult(result);
        }

        public static void RetrievePilotByName(IObjectContainer db)
        {
            Pilot proto = new Pilot("Michael Schumacher", 0);
            IObjectSet result = db.Get(proto);
            ListResult(result);
        }

        public static void
RetrievePilotByExactPoints(IObjectContainer db)
        {
            Pilot proto = new Pilot(null, 100);
            IObjectSet result = db.Get(proto);
            ListResult(result);
        }

        public static void UpdatePilot(IObjectContainer db)
        {
            IObjectSet result = db.Get(new Pilot("Michael
Schumacher", 0));
            Pilot found = (Pilot)result.Next();
            found.AddPoints(11);
            db.Set(found);
            Console.WriteLine("Added 11 points for {0}", found);
            RetrieveAllPilots(db);
        }
```

```
        public static void DeleteFirstPilotByName(IObjectContainer
db)
        {
            IObjectSet result = db.Get(new Pilot("Michael
Schumacher", 0));
            Pilot found = (Pilot)result.Next();
            db.Delete(found);
            Console.WriteLine("Deleted {0}", found);
            RetrieveAllPilots(db);
        }

        public static void DeleteSecondPilotByName(IObjectContainer
db)
        {
            IObjectSet result = db.Get(new Pilot("Rubens
Barrichello", 0));
            Pilot found = (Pilot)result.Next();
            db.Delete(found);
            Console.WriteLine("Deleted {0}", 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);
    IObjectSet 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

```vbnet
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:

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:

```java
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

IQuery query=db.Query();
    query.Constrain(typeof(Pilot));
    IQuery pointQuery=query.Descend("_points");
    query.Descend("_name").Constrain("Rubens Barrichello")
        .Or(pointQuery.Constrain(99).Greater()
        .And(pointQuery.Constrain(199).Smaller()));
    IObjectSet 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.

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.

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

```
using Db4objects.Db4o;

using Db4objects.Db4o.Query;


using Db4objects.Db4o.Tutorial;


namespace Db4objects.Db4o.Tutorial.F1.Chapter1
{
    public class NQExample : Util
    {
        public static void Main(string[] args)
        {
            IObjectContainer db =
Db4oFactory.OpenFile(Util.YapFileName);
            try
            {
                StorePilots(db);

                RetrieveComplexSODA(db);

                RetrieveComplexNQ(db);

                RetrieveArbitraryCodeNQ(db);

                ClearDatabase(db);

            }
            finally
            {
                db.Close();

            }
        }
```

```
    public static void StorePilots(IObjectContainer db)
    {
        db.Set(new Pilot("Michael Schumacher", 100));
        db.Set(new Pilot("Rubens Barrichello", 99));
    }


    public static void RetrieveComplexSODA(IObjectContainer db)
    {
        IQuery query=db.Query();
        query.Constrain(typeof(Pilot));
        IQuery pointQuery=query.Descend("_points");
        query.Descend("_name").Constrain("Rubens Barrichello")
            .Or(pointQuery.Constrain(99).Greater()
            .And(pointQuery.Constrain(199).Smaller()));
        IObjectSet result=query.Execute();
        ListResult(result);
    }


    public static void RetrieveComplexNQ(IObjectContainer db)
    {
        IObjectSet result = db.Query(new ComplexQuery());
        ListResult(result);
    }


    public static void RetrieveArbitraryCodeNQ(IObjectContainer
db)
    {
        IObjectSet result = db.Query(new ArbitraryQuery(new
int[]{1,100}));
        ListResult(result);
    }


    public static void ClearDatabase(IObjectContainer db)
    {
        IObjectSet result = db.Get(typeof(Pilot));
        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.
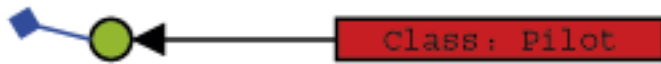
```
// retrieveAllPilots


IQuery query = db.Query();
    query.Constrain(typeof(Pilot));
    IObjectSet 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.
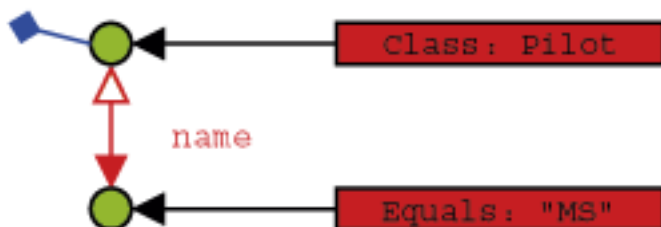
```
// retrievePilotByName

IQuery query = db.Query();
    query.Constrain(typeof(Pilot));
    query.Descend("_name").Constrain("Michael Schumacher");
    IObjectSet 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.

```
// retrievePilotByExactPoints

IQuery query = db.Query();
    query.Constrain(typeof(Pilot));
    query.Descend("_points").Constrain(100);
    IObjectSet 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

IQuery query = db.Query();
    query.Constrain(typeof(Pilot));
    query.Descend("_name").Constrain("Michael Schumacher").Not();
    IObjectSet result = query.Execute();
    ListResult(result);
```

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

```
// retrieveByConjunction

IQuery query = db.Query();
    query.Constrain(typeof(Pilot));
    IConstraint constr = query.Descend("_name")
            .Constrain("Michael Schumacher");
    query.Descend("_points")
            .Constrain(99).And(constr);
    IObjectSet result = query.Execute();
    ListResult(result);
```

```
// retrieveByDisjunction

IQuery query = db.Query();
    query.Constrain(typeof(Pilot));
    IConstraint constr = query.Descend("_name")
            .Constrain("Michael Schumacher");
    query.Descend("_points")
            .Constrain(99).Or(constr);
    IObjectSet result = query.Execute();
    ListResult(result);
```

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

```
// retrieveByComparison

IQuery query = db.Query();
    query.Constrain(typeof(Pilot));
    query.Descend("_points")
            .Constrain(99).Greater();
    IObjectSet result = query.Execute();
    ListResult(result);
```

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

```
// retrieveByDefaultFieldValue

Pilot somebody = new Pilot("Somebody else", 0);
    db.Set(somebody);
    IQuery query = db.Query();
    query.Constrain(typeof(Pilot));
    query.Descend("_points").Constrain(0);
    IObjectSet result = query.Execute();
    ListResult(result);
    db.Delete(somebody);
```

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

It is also possible to have db4o sort the results.

```
// retrieveSorted

IQuery query = db.Query();
    query.Constrain(typeof(Pilot));
    query.Descend("_name").OrderAscending();
    IObjectSet 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

IObjectSet result = db.Get(typeof(Pilot));
    foreach (object item in result)
    {
        db.Delete(item);
    }
```

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

```
using System;

using Db4objects.Db4o;

using Db4objects.Db4o.Query;
```

```csharp
using Db4objects.Db4o.Tutorial;

namespace Db4objects.Db4o.Tutorial.F1.Chapter1
{
    public class QueryExample : Util
    {
        public static void Main(string[] args)
        {
            IObjectContainer db =
Db4oFactory.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(IObjectContainer db)
        {
            Pilot pilot1 = new Pilot("Michael Schumacher", 100);
            db.Set(pilot1);
            Console.WriteLine("Stored {0}", pilot1);
        }

        public static void StoreSecondPilot(IObjectContainer db)
        {
            Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
```

```csharp
        db.Set(pilot2);

        Console.WriteLine("Stored {0}", pilot2);

    }


    public static void RetrieveAllPilots(IObjectContainer db)

    {

        IQuery query = db.Query();

        query.Constrain(typeof(Pilot));

        IObjectSet result = query.Execute();

        ListResult(result);

    }


    public static void RetrievePilotByName(IObjectContainer db)

    {

        IQuery query = db.Query();

        query.Constrain(typeof(Pilot));

        query.Descend("_name").Constrain("Michael Schumacher");

        IObjectSet result = query.Execute();

        ListResult(result);

    }


    public static void
RetrievePilotByExactPoints(IObjectContainer db)

    {

        IQuery query = db.Query();

        query.Constrain(typeof(Pilot));

        query.Descend("_points").Constrain(100);

        IObjectSet result = query.Execute();

        ListResult(result);

    }


    public static void RetrieveByNegation(IObjectContainer db)

    {

        IQuery query = db.Query();

        query.Constrain(typeof(Pilot));

        query.Descend("_name").Constrain("Michael
Schumacher").Not();

        IObjectSet result = query.Execute();

        ListResult(result);

    }
```

```csharp
        public static void RetrieveByConjunction(IObjectContainer db)
        {
            IQuery query = db.Query();
            query.Constrain(typeof(Pilot));
            IConstraint constr = query.Descend("_name")
                    .Constrain("Michael Schumacher");
            query.Descend("_points")
                    .Constrain(99).And(constr);
            IObjectSet result = query.Execute();
            ListResult(result);
        }


        public static void RetrieveByDisjunction(IObjectContainer db)
        {
            IQuery query = db.Query();
            query.Constrain(typeof(Pilot));
            IConstraint constr = query.Descend("_name")
                    .Constrain("Michael Schumacher");
            query.Descend("_points")
                    .Constrain(99).Or(constr);
            IObjectSet result = query.Execute();
            ListResult(result);
        }


        public static void RetrieveByComparison(IObjectContainer db)
        {
            IQuery query = db.Query();
            query.Constrain(typeof(Pilot));
            query.Descend("_points")
                    .Constrain(99).Greater();
            IObjectSet result = query.Execute();
            ListResult(result);
        }


        public static void
RetrieveByDefaultFieldValue(IObjectContainer db)
        {
            Pilot somebody = new Pilot("Somebody else", 0);
            db.Set(somebody);
            IQuery query = db.Query();
            query.Constrain(typeof(Pilot));
```

```csharp
            query.Descend("_points").Constrain(0);
            IObjectSet result = query.Execute();
            ListResult(result);
            db.Delete(somebody);
        }


        public static void RetrieveSorted(IObjectContainer db)
        {
            IQuery query = db.Query();
            query.Constrain(typeof(Pilot));
            query.Descend("_name").OrderAscending();
            IObjectSet result = query.Execute();
            ListResult(result);
            query.Descend("_name").OrderDescending();
            result = query.Execute();
            ListResult(result);
        }


        public static void ClearDatabase(IObjectContainer db)
        {
            IObjectSet result = db.Get(typeof(Pilot));
            foreach (object item in result)
            {
                db.Delete(item);
            }
        }
    }
}
```

4. Structured objects

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

```
namespace Db4objects.Db4o.Tutorial.F1.Chapter2
{
    public class Car
    {
        string _model;
        Pilot _pilot;

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

        public Pilot Pilot
        {
            get
            {
                return _pilot;
            }

            set
            {
                _pilot = value;
            }
        }

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

```
        override public string ToString()
        {
            return string.Format("{0}[{1}]", _model, _pilot);
        }
    }
}
```

4.1. Storing structured objects

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

```
// storeFirstCar

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

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

```
// storeSecondCar

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

4.2. Retrieving structured objects

4.2.1. QBE

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

```
// retrieveAllCarsQBE

Car proto = new Car(null);
    IObjectSet 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);
    IObjectSet result = db.Get(proto);
    ListResult(result);
```

OUTPUT:

2

Michael Schumacher/100

Rubens Barrichello/99

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

```
// retrieveCarByPilotQBE


Pilot pilotproto = new Pilot("Rubens Barrichello",0);

    Car carproto = new Car(null);

    carproto.Pilot = pilotproto;

    IObjectSet 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


string pilotName = "Rubens Barrichello";

    IObjectSet results = db.Query(new

RetrieveCarsByPilotNamePredicate(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

IQuery query = db.Query();
    query.Constrain(typeof(Car));
    query.Descend("_pilot").Descend("_name")
        .Constrain("Rubens Barrichello");
    IObjectSet 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

IQuery query = db.Query();
    query.Constrain(typeof(Car));
    Pilot proto = new Pilot("Rubens Barrichello", 0);
    query.Descend("_pilot").Constrain(proto);
    IObjectSet result = query.Execute();
    ListResult(result);
```

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'":
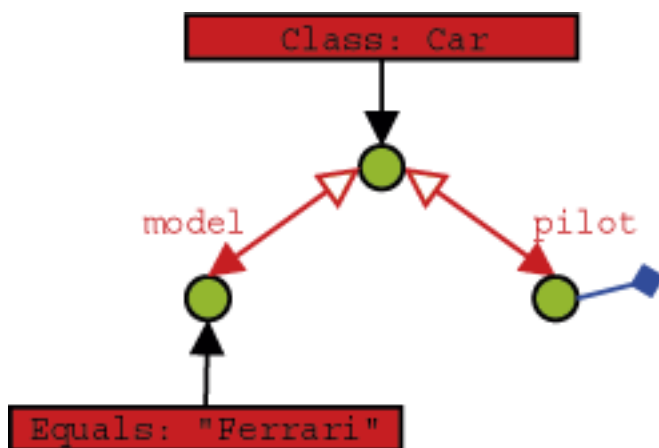
```
// retrievePilotByCarModelQuery

IQuery carQuery = db.Query();
    carQuery.Constrain(typeof(Car));
    carQuery.Descend("_model").Constrain("Ferrari");
    IQuery pilotQuery = carQuery.Descend("_pilot");
    IObjectSet result = pilotQuery.Execute();
    ListResult(result);
```

4.3. Updating structured objects

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

```
// updateCar

IObjectSet result = db.Get(new Car("Ferrari"));
    Car found = (Car)result.Next();
    found.Pilot = new Pilot("Somebody else", 0);
    db.Set(found);
    result = db.Get(new Car("Ferrari"));
    ListResult(result);
```

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

Let's modify the pilot, too.

```
// updatePilotSingleSession

IObjectSet result = db.Get(new Car("Ferrari"));
    Car found = (Car)result.Next();
    found.Pilot.AddPoints(1);
    db.Set(found);
    result = db.Get(new Car("Ferrari"));
    ListResult(result);
```

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

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

```
// updatePilotSeparateSessionsPart1

IObjectSet result = db.Get(new Car("Ferrari"));
    Car found = (Car)result.Next();
    found.Pilot.AddPoints(1);
    db.Set(found);
```

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

```
// updatePilotSeparateSessionsPart2

IObjectSet result = db.Get(new Car("Ferrari"));
    ListResult(result);
```

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

Looks like we're in trouble: Why did the Pilot's points not change? What's happening here and what can we do to fix it?

4.3.1. Update depth

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

So, in our previous update example, we were modifying the Pilot child of a Car object. When we saved

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

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

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

```
// updatePilotSeparateSessionsImprovedPart1


Db4oFactory.Configure().ObjectClass(typeof(Car))
        .CascadeOnUpdate(true);
```

```
// updatePilotSeparateSessionsImprovedPart2


IObjectSet result = db.Get(new Car("Ferrari"));
    Car found = (Car)result.Next();
    found.Pilot.AddPoints(1);
    db.Set(found);
```

```
// updatePilotSeparateSessionsImprovedPart3


IObjectSet result = db.Get(new Car("Ferrari"));
    ListResult(result);
```

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

IObjectSet result = db.Get(new Car("Ferrari"));
    Car found = (Car)result.Next();
    db.Delete(found);
    result = db.Get(new Car(null));
    ListResult(result);
```

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

```
// retrieveAllPilotsQBE
```

```
Pilot proto = new Pilot(null, 0);
    IObjectSet result = db.Get(proto);
    ListResult(result);
```

OUTPUT:
1
Rubens Barrichello/99

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

4.4.1. Recursive deletion

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

```
// deleteDeepPart1

Db4oFactory.Configure().ObjectClass(typeof(Car))
        .CascadeOnDelete(true);
```

```
// deleteDeepPart2

IObjectSet result = db.Get(new Car("BMW"));
    Car found = (Car)result.Next();
    db.Delete(found);
    result = db.Get(new Car(null));
    ListResult(result);
```

OUTPUT:
0

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

Let's have a look at our pilots again.

```
// retrieveAllPilots

Pilot proto = new Pilot(null, 0);
    IObjectSet result = db.Get(proto);
    ListResult(result);
```

OUTPUT:
0

4.4.2. Recursive deletion revisited

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

```
// deleteDeepRevisited

IObjectSet result = db.Get(new Pilot("Michael Schumacher", 0));
    Pilot pilot = (Pilot)result.Next();
    Car car1 = new Car("Ferrari");
    Car car2 = new Car("BMW");
    car1.Pilot = pilot;
    car2.Pilot = pilot;
    db.Set(car1);
    db.Set(car2);
    db.Delete(car2);
    result = db.Get(new Car(null));
    ListResult(result);
```

OUTPUT:

```
Pilot not found!
```

```
// retrieveAllPilots

Pilot proto = new Pilot(null, 0);
    IObjectSet result = db.Get(proto);
    ListResult(result);
```

OUTPUT:

0

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

Let's clear our database for the next chapter.

```
// deleteAll

IObjectSet result = db.Get(typeof(Object));
    foreach (object item in result)
    {
        db.Delete(item);
    }
```

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

```
using System;
using System.IO;

using Db4objects.Db4o;
using Db4objects.Db4o.Query;

namespace Db4objects.Db4o.Tutorial.F1.Chapter2
{
    public class StructuredExample : Util
    {
        public static void Main(String[] args)
        {
            File.Delete(Util.YapFileName);

            IObjectContainer db =
Db4oFactory.OpenFile(Util.YapFileName);
            try
            {
                StoreFirstCar(db);
                StoreSecondCar(db);
                RetrieveAllCarsQBE(db);
                RetrieveAllPilotsQBE(db);
                RetrieveCarByPilotQBE(db);
                RetrieveCarByPilotNameQuery(db);
                RetrieveCarByPilotProtoQuery(db);
                RetrievePilotByCarModelQuery(db);
                UpdateCar(db);
                UpdatePilotSingleSession(db);
                UpdatePilotSeparateSessionsPart1(db);
                db.Close();
                db=Db4oFactory.OpenFile(Util.YapFileName);
                UpdatePilotSeparateSessionsPart2(db);
                db.Close();
                UpdatePilotSeparateSessionsImprovedPart1(db);
                db=Db4oFactory.OpenFile(Util.YapFileName);
                UpdatePilotSeparateSessionsImprovedPart2(db);
                db.Close();
```

```
            db=Db4oFactory.OpenFile(Util.YapFileName);

            UpdatePilotSeparateSessionsImprovedPart3(db);

            DeleteFlat(db);

            db.Close();

            DeleteDeepPart1(db);

            db=Db4oFactory.OpenFile(Util.YapFileName);

            DeleteDeepPart2(db);

            DeleteDeepRevisited(db);

        }
        finally
        {
            db.Close();
        }
    }


    public static void StoreFirstCar(IObjectContainer db)
    {
        Car car1 = new Car("Ferrari");
        Pilot pilot1 = new Pilot("Michael Schumacher", 100);
        car1.Pilot = pilot1;
        db.Set(car1);
    }


    public static void StoreSecondCar(IObjectContainer db)
    {
        Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
        db.Set(pilot2);
        Car car2 = new Car("BMW");
        car2.Pilot = pilot2;
        db.Set(car2);
    }


    public static void RetrieveAllCarsQBE(IObjectContainer db)
    {
        Car proto = new Car(null);
        IObjectSet result = db.Get(proto);
        ListResult(result);
    }


    public static void RetrieveAllPilotsQBE(IObjectContainer db)
    {
```

```
            Pilot proto = new Pilot(null, 0);
            IObjectSet result = db.Get(proto);
            ListResult(result);
        }


        public static void RetrieveCarByPilotQBE(IObjectContainer db)
        {
            Pilot pilotproto = new Pilot("Rubens Barrichello",0);
            Car carproto = new Car(null);
            carproto.Pilot = pilotproto;
            IObjectSet result = db.Get(carproto);
            ListResult(result);
        }


        public static void
RetrieveCarByPilotNameQuery(IObjectContainer db)
        {
            IQuery query = db.Query();
            query.Constrain(typeof(Car));
            query.Descend("_pilot").Descend("_name")
                .Constrain("Rubens Barrichello");
            IObjectSet result = query.Execute();
            ListResult(result);
        }


        public static void
RetrieveCarByPilotProtoQuery(IObjectContainer db)
        {
            IQuery query = db.Query();
            query.Constrain(typeof(Car));
            Pilot proto = new Pilot("Rubens Barrichello", 0);
            query.Descend("_pilot").Constrain(proto);
            IObjectSet result = query.Execute();
            ListResult(result);
        }


        public static void
RetrievePilotByCarModelQuery(IObjectContainer db)
        {
            IQuery carQuery = db.Query();
            carQuery.Constrain(typeof(Car));
```

```csharp
            carQuery.Descend("_model").Constrain("Ferrari");
            IQuery pilotQuery = carQuery.Descend("_pilot");
            IObjectSet result = pilotQuery.Execute();
            ListResult(result);
        }


        public static void RetrieveAllPilots(IObjectContainer db)
        {
            IObjectSet results = db.Get(typeof(Pilot));
            ListResult(results);
        }


        public static void RetrieveAllCars(IObjectContainer db)
        {
            IObjectSet results = db.Get(typeof(Car));
            ListResult(results);
        }


        public class RetrieveCarsByPilotNamePredicate : Predicate
        {
            readonly string _pilotName;


            public RetrieveCarsByPilotNamePredicate(string pilotName)
            {
                _pilotName = pilotName;
            }


            public bool Match(Car candidate)
            {
                return candidate.Pilot.Name == _pilotName;
            }
        }


        public static void
RetrieveCarsByPilotNameNative(IObjectContainer db)
        {
            string pilotName = "Rubens Barrichello";
            IObjectSet results = db.Query(new
RetrieveCarsByPilotNamePredicate(pilotName));
            ListResult(results);
        }
```

```csharp
        public static void UpdateCar(IObjectContainer db)
        {
            IObjectSet result = db.Get(new Car("Ferrari"));
            Car found = (Car)result.Next();
            found.Pilot = new Pilot("Somebody else", 0);
            db.Set(found);
            result = db.Get(new Car("Ferrari"));
            ListResult(result);
        }


        public static void UpdatePilotSingleSession(IObjectContainer
db)
        {
            IObjectSet result = db.Get(new Car("Ferrari"));
            Car found = (Car)result.Next();
            found.Pilot.AddPoints(1);
            db.Set(found);
            result = db.Get(new Car("Ferrari"));
            ListResult(result);
        }


        public static void
UpdatePilotSeparateSessionsPart1(IObjectContainer db)
        {
            IObjectSet result = db.Get(new Car("Ferrari"));
            Car found = (Car)result.Next();
            found.Pilot.AddPoints(1);
            db.Set(found);
        }


        public static void
UpdatePilotSeparateSessionsPart2(IObjectContainer db)
        {
            IObjectSet result = db.Get(new Car("Ferrari"));
            ListResult(result);
        }


        public static void
UpdatePilotSeparateSessionsImprovedPart1(IObjectContainer db)
        {
```

```csharp
        Db4oFactory.Configure().ObjectClass(typeof(Car))
            .CascadeOnUpdate(true);
    }


    public static void
UpdatePilotSeparateSessionsImprovedPart2(IObjectContainer db)
    {
        IObjectSet result = db.Get(new Car("Ferrari"));
        Car found = (Car)result.Next();
        found.Pilot.AddPoints(1);
        db.Set(found);
    }


    public static void
UpdatePilotSeparateSessionsImprovedPart3(IObjectContainer db)
    {
        IObjectSet result = db.Get(new Car("Ferrari"));
        ListResult(result);
    }


    public static void DeleteFlat(IObjectContainer db)
    {
        IObjectSet result = db.Get(new Car("Ferrari"));
        Car found = (Car)result.Next();
        db.Delete(found);
        result = db.Get(new Car(null));
        ListResult(result);
    }


    public static void DeleteDeepPart1(IObjectContainer db)
    {
        Db4oFactory.Configure().ObjectClass(typeof(Car))
            .CascadeOnDelete(true);
    }


    public static void DeleteDeepPart2(IObjectContainer db)
    {
        IObjectSet result = db.Get(new Car("BMW"));
        Car found = (Car)result.Next();
        db.Delete(found);
        result = db.Get(new Car(null));
```

```
                ListResult(result);
         }


        public static void DeleteDeepRevisited(IObjectContainer db)
        {
                IObjectSet result = db.Get(new Pilot("Michael
Schumacher", 0));
                Pilot pilot = (Pilot)result.Next();
                Car car1 = new Car("Ferrari");
                Car car2 = new Car("BMW");
                car1.Pilot = pilot;
                car2.Pilot = pilot;
                db.Set(car1);
                db.Set(car2);
                db.Delete(car2);
                result = db.Get(new Car(null));
                ListResult(result);
         }
    }
}
```

5. Collections and Arrays

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

```csharp
using System;
using System.Text;

namespace Db4objects.Db4o.Tutorial.F1.Chapter3
{
    public class SensorReadout
    {
        double[] _values;
        DateTime _time;
        Car _car;

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

        public Car Car
        {
            get
            {
                return _car;
            }
        }

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

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


        public double[] Values
        {
            get
            {
                return _values;
            }
        }


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


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

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

```csharp
using System;
using System.Collections;

namespace Db4objects.Db4o.Tutorial.F1.Chapter3
{
    public class Car
    {
        string _model;
        Pilot _pilot;
        IList _history;

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

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

        public Pilot Pilot
        {
            get
            {
                return _pilot;
            }

            set
            {
                _pilot = value;
            }
        }
```

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


        public IList History
        {
            get
            {
                return _history;
            }
        }


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


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


        override public string ToString()
        {
            return string.Format("{0}[{1}]/{2}", _model, _pilot,
_history.Count);
        }
    }
}
```

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

5.1. Storing

This should be familiar by now.

```
// storeFirstCar

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

The second car will take two snapshots immediately at startup.

```
// storeSecondCar

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

5.2. Retrieving

5.2.1. QBE

First let us verify that we indeed have taken snapshots.

```
// retrieveAllSensorReadout

IObjectSet result = db.Get(typeof(SensorReadout));
    ListResult(result);
```

```
OUTPUT:
2
BMW[Rubens Barrichello/99]/2 : 1174951824963 : 0.1,0.2,0.3
BMW[Rubens Barrichello/99]/2 : 1174951824963 : 0.2,0.4,0.6
```

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

```
// retrieveSensorReadoutQBE

SensorReadout proto = new SensorReadout(new double[] { 0.3, 0.1 },
DateTime.MinValue, null);
    IObjectSet result = db.Get(proto);
    ListResult(result);
```

```
OUTPUT:
1
BMW[Rubens Barrichello/99]/2 : 1174951824963 : 0.1,0.2,0.3
```

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

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

```
// retrieveCarQBE

SensorReadout protoReadout = new SensorReadout(new double[] { 0.6,
0.2 }, DateTime.MinValue, null);
    IList protoHistory = new ArrayList();
    protoHistory.Add(protoReadout);
    Car protoCar = new Car(null, protoHistory);
    IObjectSet result = db.Get(protoCar);
```

```
    ListResult(result);
```

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

```
// retrieveCollections

IObjectSet result = db.Get(new ArrayList());
    ListResult(result);
```

This doesn't work with arrays, though.

```
// retrieveArrays

IObjectSet result = db.Get(new double[] { 0.6, 0.4 });
    ListResult(result);
```

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


IObjectSet results = db.Query(new RetrieveSensorReadoutPredicate());
    ListResult(results);
```

OUTPUT:

0

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

```
// retrieveCarNative


IObjectSet results = db.Query(new RetrieveCarPredicate());
    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

IQuery query = db.Query();
    query.Constrain(typeof(SensorReadout));
    IQuery valueQuery = query.Descend("_values");
    valueQuery.Constrain(0.3);
    valueQuery.Constrain(0.1);
    IObjectSet results = query.Execute();
    ListResult(results);
```

OUTPUT:
```
1
BMW[Rubens Barrichello/99]/2 : 1174951824963 : 0.1,0.2,0.3
```

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

```
// retrieveCarQuery

IQuery query = db.Query();
    query.Constrain(typeof(Car));
    IQuery historyQuery = query.Descend("_history");
    historyQuery.Constrain(typeof(SensorReadout));
    IQuery valueQuery = historyQuery.Descend("_values");
    valueQuery.Constrain(0.3);
    valueQuery.Constrain(0.1);
    IObjectSet results = query.Execute();
    ListResult(results);
```

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

Db4oFactory.Configure().ObjectClass(typeof(Car)).CascadeOnUpdate(true
);
```

```
// updateCarPart2

IObjectSet 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]/2 : 1174951824963 : 0.1,0.2,0.3
BMW[Rubens Barrichello/99]/2 : 1174951824963 : 0.2,0.4,0.6
Ferrari[Michael Schumacher/100]/1 : 1174951826070 : 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

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

```
query.Constrain(typeof(Car));

IObjectSet result = query.Descend("_history").Execute();

IList history = (IList)result.Next();

history.RemoveAt(0);

db.Set(history);

Car proto = new Car(null, null);

result = db.Get(proto);

foreach (Car car in result)
{
    foreach (object readout in car.History)
    {
        Console.WriteLine(readout);
    }
}
```

OUTPUT:

```
BMW[Rubens Barrichello/99]/2 : 1174951824963 : 0.1,0.2,0.3
BMW[Rubens Barrichello/99]/2 : 1174951824963 : 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

Db4oFactory.Configure().ObjectClass(typeof(Car)).CascadeOnDelete(true
);
```

```
// deleteAllPart2

IObjectSet result = db.Get(new Car(null, null));
    foreach (object car in result)
```

```
    {
        db.Delete(car);
    }
    IObjectSet readouts = db.Get(new SensorReadout(null,
DateTime.MinValue, null));
    foreach (object readout in readouts)
    {
        db.Delete(readout);
    }
```

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

```
using System;
using System.Collections;
using System.IO;

using Db4objects.Db4o;
using Db4objects.Db4o.Query;

namespace Db4objects.Db4o.Tutorial.F1.Chapter3
{
    public class CollectionsExample : Util
    {
        public static void Main(string[] args)
        {
            File.Delete(Util.YapFileName);
            IObjectContainer db =
Db4oFactory.OpenFile(Util.YapFileName);
            try
            {
                StoreFirstCar(db);
                StoreSecondCar(db);
```

```csharp
            RetrieveAllSensorReadouts(db);

            RetrieveSensorReadoutQBE(db);

            RetrieveCarQBE(db);

            RetrieveCollections(db);

            RetrieveArrays(db);

            RetrieveSensorReadoutQuery(db);

            RetrieveCarQuery(db);

            db.Close();

            UpdateCarPart1();

            db = Db4oFactory.OpenFile(Util.YapFileName);

            UpdateCarPart2(db);

            UpdateCollection(db);

            db.Close();

            DeleteAllPart1();

            db=Db4oFactory.OpenFile(Util.YapFileName);

            DeleteAllPart2(db);

            RetrieveAllSensorReadouts(db);

        }
        finally
        {
            db.Close();

        }
    }


    public static void StoreFirstCar(IObjectContainer db)
    {
        Car car1 = new Car("Ferrari");
        Pilot pilot1 = new Pilot("Michael Schumacher", 100);
        car1.Pilot = pilot1;
        db.Set(car1);
    }


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

```csharp
        public static void RetrieveAllSensorReadouts(IObjectContainer
db)
        {
            IObjectSet result = db.Get(typeof(SensorReadout));
            ListResult(result);
        }


        public static void RetrieveSensorReadoutQBE(IObjectContainer
db)
        {
            SensorReadout proto = new SensorReadout(new double[] {
0.3, 0.1 }, DateTime.MinValue, null);
            IObjectSet result = db.Get(proto);
            ListResult(result);
        }


        public static void RetrieveCarQBE(IObjectContainer db)
        {
            SensorReadout protoReadout = new SensorReadout(new
double[] { 0.6, 0.2 }, DateTime.MinValue, null);
            IList protoHistory = new ArrayList();
            protoHistory.Add(protoReadout);
            Car protoCar = new Car(null, protoHistory);
            IObjectSet result = db.Get(protoCar);
            ListResult(result);
        }


        public static void RetrieveCollections(IObjectContainer db)
        {
            IObjectSet result = db.Get(new ArrayList());
            ListResult(result);
        }


        public static void RetrieveArrays(IObjectContainer db)
        {
            IObjectSet result = db.Get(new double[] { 0.6, 0.4 });
            ListResult(result);
        }

        public static void
```

```
RetrieveSensorReadoutQuery(IObjectContainer db)
        {
                IQuery query = db.Query();
                query.Constrain(typeof(SensorReadout));
                IQuery valueQuery = query.Descend("_values");
                valueQuery.Constrain(0.3);
                valueQuery.Constrain(0.1);
                IObjectSet results = query.Execute();
                ListResult(results);
        }


        public static void RetrieveCarQuery(IObjectContainer db)
        {
                IQuery query = db.Query();
                query.Constrain(typeof(Car));
                IQuery historyQuery = query.Descend("_history");
                historyQuery.Constrain(typeof(SensorReadout));
                IQuery valueQuery = historyQuery.Descend("_values");
                valueQuery.Constrain(0.3);
                valueQuery.Constrain(0.1);
                IObjectSet results = query.Execute();
                ListResult(results);
        }


        public class RetrieveSensorReadoutPredicate : Predicate
        {
                public bool Match(SensorReadout candidate)
                {
                        return Array.IndexOf(candidate.Values, 0.3) > -1
                                && Array.IndexOf(candidate.Values, 0.1) > -1;
                }
        }


        public static void
RetrieveSensorReadoutNative(IObjectContainer db)
        {
                IObjectSet results = db.Query(new
RetrieveSensorReadoutPredicate());
                ListResult(results);
        }
```

```csharp
public class RetrieveCarPredicate : Predicate
{
    public bool Match(Car car)
    {
        foreach (SensorReadout sensor in car.History)
        {
            if (Array.IndexOf(sensor.Values, 0.3) > -1
                && Array.IndexOf(sensor.Values, 0.1) > -1)
            {
                return true;
            }
        }
        return false;
    }
}

public static void RetrieveCarNative(IObjectContainer db)
{
    IObjectSet results = db.Query(new
RetrieveCarPredicate());
    ListResult(results);
}

public static void UpdateCarPart1()
{
Db4oFactory.Configure().ObjectClass(typeof(Car)).CascadeOnUpdate(true
);
}

public static void UpdateCarPart2(IObjectContainer db)
{
    IObjectSet result = db.Get(new Car("BMW", null));
    Car car = (Car)result.Next();
    car.Snapshot();
    db.Set(car);
    RetrieveAllSensorReadouts(db);
}

public static void UpdateCollection(IObjectContainer db)
{
    IQuery query = db.Query();
```

```csharp
            query.Constrain(typeof(Car));
            IObjectSet result = query.Descend("_history").Execute();
            IList history = (IList)result.Next();
            history.RemoveAt(0);
            db.Set(history);
            Car proto = new Car(null, null);
            result = db.Get(proto);
            foreach (Car car in result)
            {
                foreach (object readout in car.History)
                {
                    Console.WriteLine(readout);
                }
            }
        }


        public static void DeleteAllPart1()
        {
Db4oFactory.Configure().ObjectClass(typeof(Car)).CascadeOnDelete(true
);
        }


        public static void DeleteAllPart2(IObjectContainer db)
        {
            IObjectSet result = db.Get(new Car(null, null));
            foreach (object car in result)
            {
                db.Delete(car);
            }
            IObjectSet readouts = db.Get(new SensorReadout(null,
DateTime.MinValue, null));
            foreach (object readout in readouts)
            {
                db.Delete(readout);
            }
        }
    }
}
```

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.

```
using System;

namespace Db4objects.Db4o.Tutorial.F1.Chapter4
{
    public class SensorReadout
    {
        DateTime _time;
        Car _car;
        string _description;

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

        public Car Car
        {
            get
            {
                return _car;
            }
        }

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

```
                }
        }

        public string Description
        {
            get
            {
                return _description;
            }
        }

        override public string ToString()
        {
            return string.Format("{0}:{1}:{2}", _car, _time,
_description);
        }
    }
}
```

```
using System;

namespace Db4objects.Db4o.Tutorial.F1.Chapter4
{
    public class TemperatureSensorReadout : SensorReadout
    {
        double _temperature;

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

        public double Temperature
        {
            get
```

```
        {
            return _temperature;
        }
    }


    override public string ToString()
    {
        return string.Format("{0} temp: {1}", base.ToString(),
_temperature);
    }
    }
}
```

```
using System;

namespace Db4objects.Db4o.Tutorial.F1.Chapter4
{
    public class PressureSensorReadout : SensorReadout
    {
        double _pressure;

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

        public double Pressure
        {
            get
            {
                return _pressure;
            }
        }

        override public string ToString()
```

```
        {
            return string.Format("{0} pressure: {1}",
base.ToString(), _pressure);
        }
    }
}
```

Our car's snapshot mechanism is changed accordingly.

```
using System;
using System.Collections;

namespace Db4objects.Db4o.Tutorial.F1.Chapter4
{
    public class Car
    {
        string _model;
        Pilot _pilot;
        IList _history;

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

        public Pilot Pilot
        {
            get
            {
                return _pilot;
            }

            set
            {
                _pilot = value;
```

```
        }
    }


    public string Model
    {
        get
        {
            return _model;
        }
    }


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


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


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


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


    protected double PollOilPressure()
    {
```

```
            return 0.3*_history.Count;

        }


        override public string ToString()

        {

            return string.Format("{0}[{1}]/{2}", _model, _pilot,

_history.Count);

        }

    }

}
```

6.1. Storing

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

```
// storeFirstCar


Car car1 = new Car("Ferrari");

    Pilot pilot1 = new Pilot("Michael Schumacher", 100);

    car1.Pilot = pilot1;

    db.Set(car1);
```

```
// storeSecondCar


Pilot pilot2 = new Pilot("Rubens Barrichello", 99);

    Car car2 = new Car("BMW");

    car2.Pilot = pilot2;

    car2.Snapshot();

    car2.Snapshot();

    db.Set(car2);
```

6.2. Retrieving

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

```
// retrieveTemperatureReadoutsQBE

SensorReadout proto = new TemperatureSensorReadout(DateTime.MinValue,
null, null, 0.0);
    IObjectSet result = db.Get(proto);
    ListResult(result);
```

```
OUTPUT:
4
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : water
temp : 0.8
```

```
// retrieveAllSensorReadoutsQBE

SensorReadout proto = new SensorReadout(DateTime.MinValue, null,
null);
    IObjectSet result = db.Get(proto);
    ListResult(result);
```

```
OUTPUT:
6
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : oil
```

```
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 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: Type objects receive special handling with QBE.

```
// retrieveAllSensorReadoutsQBEAlternative

IObjectSet result = db.Get(typeof(SensorReadout));
    ListResult(result);
```

```
OUTPUT:
6
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : oil
pressure : 1.5
```

And of course there's our SODA API:

```
// retrieveAllSensorReadoutsQuery

IQuery query = db.Query();
    query.Constrain(typeof(SensorReadout));
    IObjectSet result = query.Execute();
    ListResult(result);
```

```
OUTPUT:
6
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 2007 : water
temp : 0.8
BMW[Rubens Barrichello/99]/6 : Tue Mar 27 01:30:26 CEST 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

IObjectSet result = db.Get(typeof(Object));
```

```
    foreach (object item in result)
    {
        db.Delete(item);
    }
```

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

```
using System;

using System.IO;

using Db4objects.Db4o;

using Db4objects.Db4o.Query;

namespace Db4objects.Db4o.Tutorial.F1.Chapter4
{
    public class InheritanceExample : Util
    {
        public static void Main(string[] args)
        {
            File.Delete(Util.YapFileName);
            IObjectContainer db =
Db4oFactory.OpenFile(Util.YapFileName);
            try
            {
                StoreFirstCar(db);
                StoreSecondCar(db);
                RetrieveTemperatureReadoutsQBE(db);
                RetrieveAllSensorReadoutsQBE(db);
                RetrieveAllSensorReadoutsQBEAlternative(db);
                RetrieveAllSensorReadoutsQuery(db);
                RetrieveAllObjects(db);
```

```
            }
            finally
            {
                db.Close();
            }
        }


        public static void StoreFirstCar(IObjectContainer db)
        {
            Car car1 = new Car("Ferrari");
            Pilot pilot1 = new Pilot("Michael Schumacher", 100);
            car1.Pilot = pilot1;
            db.Set(car1);
        }


        public static void StoreSecondCar(IObjectContainer db)
        {
            Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
            Car car2 = new Car("BMW");
            car2.Pilot = pilot2;
            car2.Snapshot();
            car2.Snapshot();
            db.Set(car2);
        }


        public static void
RetrieveAllSensorReadoutsQBE(IObjectContainer db)
        {
            SensorReadout proto = new
SensorReadout(DateTime.MinValue, null, null);
            IObjectSet result = db.Get(proto);
            ListResult(result);
        }


        public static void
RetrieveTemperatureReadoutsQBE(IObjectContainer db)
        {
            SensorReadout proto = new
TemperatureSensorReadout(DateTime.MinValue, null, null, 0.0);
            IObjectSet result = db.Get(proto);
            ListResult(result);
```

```
        }

        public static void
RetrieveAllSensorReadoutsQBEAlternative(IObjectContainer db)
        {
            IObjectSet result = db.Get(typeof(SensorReadout));
            ListResult(result);
        }

        public static void
RetrieveAllSensorReadoutsQuery(IObjectContainer db)
        {
            IQuery query = db.Query();
            query.Constrain(typeof(SensorReadout));
            IObjectSet result = query.Execute();
            ListResult(result);
        }

        public static void RetrieveAllObjects(IObjectContainer db)
        {
            IObjectSet 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 implicitly provided by the SensorReadout class.

```
using System;

namespace Db4objects.Db4o.Tutorial.F1.Chapter5
{
    public abstract class SensorReadout
    {
        DateTime _time;
        Car _car;
        string _description;
        SensorReadout _next;

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

        public Car Car
        {
            get
            {
                return _car;
            }
        }

        public DateTime Time
        {
            get
```

```csharp
            {
                return _time;
            }
        }


        public SensorReadout Next
        {
            get
            {
                return _next;
            }
        }


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


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


        override public string ToString()
        {
            return string.Format("{0} : {1} : {2}", _car, _time,
_description);
        }
    }
}
```

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

```csharp
using System;

namespace Db4objects.Db4o.Tutorial.F1.Chapter5
{
    public class Car
    {
        string _model;
        Pilot _pilot;
        SensorReadout _history;

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

        public Pilot Pilot
        {
            get
            {
                return _pilot;
            }

            set
            {
                _pilot = value;
            }
        }

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

```csharp
        public SensorReadout GetHistory()
        {
            return _history;
        }


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


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


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


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


        override public string ToString()
        {
            return string.Format("{0}[{1}]/{2}", _model, _pilot,
CountHistoryElements());
        }


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

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

7.1. Storing and updating

No surprises here.

```
// storeCar

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

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

```
// setCascadeOnUpdate

Db4oFactory.Configure().ObjectClass(typeof(Car)).CascadeOnUpdate(true
);
```

Let's collect a few sensor readouts.

```
// takeManySnapshots


IObjectSet result = db.Get(typeof(Car));
    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


IObjectSet result = db.Get(typeof(SensorReadout));
    while (result.HasNext())
    {
        Console.WriteLine(result.Next());
    }
```

OUTPUT:
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
temp : 0.6000000000000001
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water
temp : 1.4000000000000001
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
pressure : 2.4
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil

```
pressure : 4.2

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil

pressure : 0.6

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil

temp : 0.0

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water

temp : 0.2

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil

temp : 0.30000000000000004

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil

temp : 0.9

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water

temp : 2.0

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil

pressure : 3.3

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil

temp : 1.2000000000000002

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water

temp : 2.6

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water

temp : 0.8

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil

pressure : 1.5
```

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

```
// retrieveSnapshotsSequentially

IObjectSet result = db.Get(typeof(Car));
    Car car = (Car)result.Next();
    SensorReadout readout = car.GetHistory();
    while (readout != null)
    {
        Console.WriteLine(readout);
        readout = readout.Next;
    }
```

```
OUTPUT:

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
temp : 0.0

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water
temp : 0.2

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
pressure : 0.6

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
temp : 0.30000000000000004

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water
temp : 0.8

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
pressure : 1.5

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
temp : 0.6000000000000001

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water
temp : 1.4000000000000001

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
pressure : 2.4

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
temp : 0.9

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water
temp : 2.0

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
pressure : 3.3

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
temp : 1.2000000000000002

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water
temp : 2.6

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
pressure : 4.2
```

Ouch! What's happening here?

7.2.1. Activation depth

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

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

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

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

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

```
// retrieveSnapshotsSequentiallyImproved

IObjectSet result = db.Get(typeof(Car));
    Car car = (Car)result.Next();
    SensorReadout readout = car.GetHistory();
    while (readout != null)
    {
        db.Activate(readout, 1);
        Console.WriteLine(readout);
        readout = readout.Next;
    }
```

OUTPUT:
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil temp : 0.0
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water temp : 0.2
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil pressure : 0.6
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water

```
temp : 0.8

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil

pressure : 1.5

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil

temp : 0.6000000000000001

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water

temp : 1.4000000000000001

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil

pressure : 2.4

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil

temp : 0.9

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water

temp : 2.0

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil

pressure : 3.3

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil

temp : 1.2000000000000002

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water

temp : 2.6

BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 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

Db4oFactory.Configure().ObjectClass(typeof(TemperatureSensorReadout))
        .CascadeOnActivate(true);
```

```
// retrieveSnapshotsSequentially
```

```
IObjectSet result = db.Get(typeof(Car));

    Car car = (Car)result.Next();

    SensorReadout readout = car.GetHistory();

    while (readout != null)

    {

        Console.WriteLine(readout);

        readout = readout.Next;

    }
```

OUTPUT:

```
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
temp : 0.0
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water
temp : 0.2
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
pressure : 0.6
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water
temp : 0.8
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
pressure : 1.5
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
temp : 0.6000000000000001
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water
temp : 1.4000000000000001
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
pressure : 2.4
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
temp : 0.9
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water
temp : 2.0
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
pressure : 3.3
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : oil
temp : 1.2000000000000002
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 2007 : water
temp : 2.6
```

```
BMW[Rubens Barrichello/99]/15 : Tue Mar 27 01:30:28 CEST 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 Db4objects.Db4o.Config.Configuration and the associated IObjectClass and IObjectField classes.

Don't forget to clean up the database.

```
// deleteAll

IObjectSet result = db.Get(typeof(Object));
    foreach (object item in result)
    {
        db.Delete(item);
    }
```

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

```
using System;
using System.IO;
using Db4objects.Db4o;

namespace Db4objects.Db4o.Tutorial.F1.Chapter5
{
    public class DeepExample : Util
    {
```

```csharp
        public static void Main(string[] args)
        {
            File.Delete(Util.YapFileName);
            IObjectContainer db =
Db4oFactory.OpenFile(Util.YapFileName);
            try
            {
                StoreCar(db);
                db.Close();
                SetCascadeOnUpdate();
                db = Db4oFactory.OpenFile(Util.YapFileName);
                TakeManySnapshots(db);
                db.Close();
                db = Db4oFactory.OpenFile(Util.YapFileName);
                RetrieveAllSnapshots(db);
                db.Close();
                db = Db4oFactory.OpenFile(Util.YapFileName);
                RetrieveSnapshotsSequentially(db);
                RetrieveSnapshotsSequentiallyImproved(db);
                db.Close();
                SetActivationDepth();
                db = Db4oFactory.OpenFile(Util.YapFileName);
                RetrieveSnapshotsSequentially(db);
            }
            finally
            {
                db.Close();
            }
        }


        public static void StoreCar(IObjectContainer db)
        {
            Pilot pilot = new Pilot("Rubens Barrichello", 99);
            Car car = new Car("BMW");
            car.Pilot = pilot;
            db.Set(car);
        }


        public static void SetCascadeOnUpdate()
        {
Db4oFactory.Configure().ObjectClass(typeof(Car)).CascadeOnUpdate(true
```

```
);
    }

    public static void TakeManySnapshots(IObjectContainer db)
    {
        IObjectSet result = db.Get(typeof(Car));
        Car car = (Car)result.Next();
        for (int i=0; i<5; i++)
        {
            car.Snapshot();
        }
        db.Set(car);
    }

    public static void RetrieveAllSnapshots(IObjectContainer db)
    {
        IObjectSet result = db.Get(typeof(SensorReadout));
        while (result.HasNext())
        {
            Console.WriteLine(result.Next());
        }
    }

    public static void
RetrieveSnapshotsSequentially(IObjectContainer db)
    {
        IObjectSet result = db.Get(typeof(Car));
        Car car = (Car)result.Next();
        SensorReadout readout = car.GetHistory();
        while (readout != null)
        {
            Console.WriteLine(readout);
            readout = readout.Next;
        }
    }

    public static void
RetrieveSnapshotsSequentiallyImproved(IObjectContainer db)
    {
        IObjectSet result = db.Get(typeof(Car));
        Car car = (Car)result.Next();
```

```
            SensorReadout readout = car.GetHistory();

            while (readout != null)

            {

                db.Activate(readout, 1);

                Console.WriteLine(readout);

                readout = readout.Next;

            }

        }


        public static void SetActivationDepth()

        {

Db4oFactory.Configure().ObjectClass(typeof(TemperatureSensorReadout))

                .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.Pilot = pilot;
    db.Set(car);
    db.Commit();
```

```
// listAllCars

IObjectSet result = db.Get(typeof(Car));
    ListResult(result);
```

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

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

```
// storeCarRollback

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

```
// listAllCars

IObjectSet result = db.Get(typeof(Car));
    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

IObjectSet result = db.Get(new Car("BMW"));
    Car car = (Car)result.Next();
    car.Snapshot();
    db.Set(car);
```

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

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

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

```
// carSnapshotRollbackRefresh

IObjectSet result=db.Get(new Car("BMW"));
    Car car=(Car)result.Next();
    car.Snapshot();
    db.Set(car);
    db.Rollback();
    db.Ext().Refresh(car, int.MaxValue);
    Console.WriteLine(car);
```

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

What is this IExtObjectContainer 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 IObjectContainer API.

Finally, we clean up again.

```
// deleteAll

IObjectSet result = db.Get(typeof(Object));
```

```
    foreach (object item in result)
    {
        db.Delete(item);
    }
```

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

```
using System;
using System.IO;
using Db4objects.Db4o;


namespace Db4objects.Db4o.Tutorial.F1.Chapter5
{
    public class TransactionExample : Util
    {
        public static void Main(string[] args)
        {
            File.Delete(Util.YapFileName);
            IObjectContainer
db=Db4oFactory.OpenFile(Util.YapFileName);
            try
            {
                StoreCarCommit(db);
                db.Close();
                db = Db4oFactory.OpenFile(Util.YapFileName);
                ListAllCars(db);
                StoreCarRollback(db);
                db.Close();
                db = Db4oFactory.OpenFile(Util.YapFileName);
                ListAllCars(db);
                CarSnapshotRollback(db);
```

```
            CarSnapshotRollbackRefresh(db);
        }
        finally
        {
            db.Close();
        }
    }


    public static void StoreCarCommit(IObjectContainer db)
    {
        Pilot pilot = new Pilot("Rubens Barrichello", 99);
        Car car = new Car("BMW");
        car.Pilot = pilot;
        db.Set(car);
        db.Commit();
    }


    public static void ListAllCars(IObjectContainer db)
    {
        IObjectSet result = db.Get(typeof(Car));
        ListResult(result);
    }


    public static void StoreCarRollback(IObjectContainer db)
    {
        Pilot pilot = new Pilot("Michael Schumacher", 100);
        Car car = new Car("Ferrari");
        car.Pilot = pilot;
        db.Set(car);
        db.Rollback();
    }


    public static void CarSnapshotRollback(IObjectContainer db)
    {
        IObjectSet result = db.Get(new Car("BMW"));
        Car car = (Car)result.Next();
        car.Snapshot();
        db.Set(car);
        db.Rollback();
        Console.WriteLine(car);
    }
```

```
        public static void
CarSnapshotRollbackRefresh(IObjectContainer db)
        {
            IObjectSet result=db.Get(new Car("BMW"));
            Car car=(Car)result.Next();
            car.Snapshot();
            db.Set(car);
            db.Rollback();
            db.Ext().Refresh(car, int.MaxValue);
            Console.WriteLine(car);
        }
    }
}
```

9. Client/Server

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

We start by preparing our database in the familiar way.

```
// setFirstCar

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

```
// setSecondCar

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

9.1. Embedded server

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

```
// accessLocalServer

IObjectServer server = Db4oFactory.OpenServer(Util.YapFileName, 0);
    try
```

```
      {
          IObjectContainer 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

IObjectContainer 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(IObjectContainer container,
IObjectSet items, int depth)
    {
        Console.WriteLine(items.Count);
```

```
        foreach (object item in items)
        {
            container.Ext().Refresh(item, depth);
            Console.WriteLine(item);
        }
    }
```

```
// demonstrateLocalReadCommitted

IObjectContainer client1 =server.OpenClient();
    IObjectContainer client2 =server.OpenClient();
    Pilot pilot = new Pilot("David Coulthard", 98);
    IObjectSet result = client1.Get(new Car("BMW"));
    Car car = (Car)result.Next();
    car.Pilot = pilot;
    client1.Set(car);
    ListResult(client1.Get(new Car(null)));
    ListResult(client2.Get(new Car(null)));
    client1.Commit();
    ListResult(client1.Get(typeof(Car)));
    ListRefreshedResult(client2, client2.Get(typeof(Car)), 2);
    client1.Close();
    client2.Close();
```

OUTPUT:
```
2
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

IObjectContainer client1 = server.OpenClient();
    IObjectContainer client2 = server.OpenClient();
    IObjectSet result = client1.Get(new Car("BMW"));
    Car car = (Car)result.Next();
    car.Pilot = new Pilot("Someone else", 0);
    client1.Set(car);
    ListResult(client1.Get(new Car(null)));
    ListResult(client2.Get(new Car(null)));
    client1.Rollback();
    client1.Ext().Refresh(car, 2);
    ListResult(client1.Get(new Car(null)));
    ListResult(client2.Get(new Car(null)));
    client1.Close();
    client2.Close();
```

OUTPUT:
```
2
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

IObjectServer server = Db4oFactory.OpenServer(Util.YapFileName,
ServerPort);
    server.GrantAccess(ServerUser, ServerPassword);
    try
    {
        IObjectContainer client = Db4oFactory.OpenClient("localhost",
ServerPort, ServerUser, ServerPassword);
        // Do something with this client, or open more clients
        client.Close();
    }
    finally
    {
        server.Close();
    }
```

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

```
// queryRemoteServer

IObjectContainer client = Db4oFactory.OpenClient("localhost", port,
user, password);
    ListResult(client.Get(new Car(null)));
    client.Close();
```

OUTPUT:

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

```
// demonstrateRemoteReadCommitted


IObjectContainer client1 = Db4oFactory.OpenClient("localhost", port,
user, password);
    IObjectContainer client2 = Db4oFactory.OpenClient("localhost",
port, user, password);
    Pilot pilot = new Pilot("Jenson Button", 97);
    IObjectSet result = client1.Get(new Car(null));
    Car car = (Car)result.Next();
    car.Pilot = pilot;
    client1.Set(car);
    ListResult(client1.Get(new Car(null)));
    ListResult(client2.Get(new Car(null)));
    client1.Commit();
    ListResult(client1.Get(new Car(null)));
    ListResult(client2.Get(new Car(null)));
    client1.Close();
    client2.Close();
```

OUTPUT:

```
// demonstrateRemoteRollback


IObjectContainer client1 = Db4oFactory.OpenClient("localhost", port,
user, password);
    IObjectContainer client2 = Db4oFactory.OpenClient("localhost",
port, user, password);
    IObjectSet result = client1.Get(new Car(null));
    Car car = (Car)result.Next();
    car.Pilot = new Pilot("Someone else", 0);
    client1.Set(car);
    ListResult(client1.Get(new Car(null)));
```

```
    ListResult(client2.Get(new Car(null)));

    client1.Rollback();

    client1.Ext().Refresh(car,2);

    ListResult(client1.Get(new Car(null)));

    ListResult(client2.Get(new Car(null)));

    client1.Close();

    client2.Close();
```

OUTPUT:

9.3. Out-of-band signalling

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

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

```
public void RunServer()
    {
        lock(this)
        {
            IObjectServer db4oServer = Db4oFactory.OpenServer(FILE,
PORT);

            db4oServer.GrantAccess(USER, PASS);


            // Using the messaging functionality to redirect all
            // messages to this.processMessage
db4oServer.Ext().Configure().ClientServer().SetMessageRecipient(this)
;

            try
            {
                if (! stop)
```

```
                {
                    // wait forever for Notify() from Close()
                    Monitor.Wait(this);
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
            db4oServer.Close();
        }
    }
```

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

```
public void ProcessMessage(IObjectContainer con, object message)
    {
        if (message is StopServer)
        {
            Close();
        }
    }
```

Db4o allows a client to send an arbitrary signal or message to a server by sending a plain 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)
    {
        IObjectContainer IObjectContainer = null;
        try
        {
            // connect to the server
            IObjectContainer = Db4oFactory.OpenClient(HOST, PORT,
```

```
USER, PASS);
        }
        catch (Exception e)
        {
            Console.WriteLine(e.ToString());
        }

        if (IObjectContainer != null)
        {
            // get the messageSender for the IObjectContainer
            IMessageSender messageSender = IObjectContainer.Ext()
                .Configure().ClientServer().GetMessageSender();

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

            // close the IObjectContainer
            IObjectContainer.Close();
        }
    }
```

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

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

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

```
namespace Db4objects.Db4o.Tutorial.F1.Chapter5
{
    /// <summary>
    /// Configuration used for StartServer and StopServer.
    /// </summary>
    public class ServerConfiguration
    {
        /// <summary>
```

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


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


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


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


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

Now we'll create the server:

```
using System;
using System.Threading;
using Db4objects.Db4o;
using Db4objects.Db4o.Messaging;
```

```csharp
namespace Db4objects.Db4o.Tutorial.F1.Chapter5
{
    /// <summary>
    /// starts a db4o server with the settings from
ServerConfiguration.
    /// This is a typical setup for a long running server.
    /// The Server may be stopped from a remote location by running
    /// StopServer. The StartServer instance is used as a
MessageRecipient
    /// and reacts to receiving an instance of a StopServer object.
    /// Note that all user classes need to be present on the server
side
    /// and that all possible Db4oFactory.Configure() calls to alter
the db4o
    /// configuration need to be executed on the client and on the
server.
    /// </summary>
    public class StartServer : ServerConfiguration, IMessageRecipient
    {
        /// <summary>
        /// setting the value to true denotes that the server should
be closed
        /// </summary>
        private bool stop = false;

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

        /// <summary>
        /// opens the IObjectServer, and waits forever until Close()
is called
        /// or a StopServer message is being received.
        /// </summary>
        public void RunServer()
        {
```

```
        lock(this)
        {
            IObjectServer db4oServer =
Db4oFactory.OpenServer(FILE, PORT);
            db4oServer.GrantAccess(USER, PASS);

            // Using the messaging functionality to redirect all
            // messages to this.processMessage
db4oServer.Ext().Configure().ClientServer().SetMessageRecipient(this)
;

            try
            {
                if (! stop)
                {
                    // wait forever for Notify() from Close()
                    Monitor.Wait(this);
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
            db4oServer.Close();
        }
    }

    /// <summary>
    /// messaging callback
    /// see com.db4o.messaging.MessageRecipient#ProcessMessage()
    /// </summary>
    public void ProcessMessage(IObjectContainer con, object
message)
    {
        if (message is StopServer)
        {
            Close();
        }
    }

    /// <summary>
    /// closes this server.
```

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

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

```
using System;
using Db4objects.Db4o;
using Db4objects.Db4o.Messaging;


namespace Db4objects.Db4o.Tutorial.F1.Chapter5
{
    /// <summary>
    /// stops the db4o Server started with StartServer.
    /// This is done by opening a client connection
    /// to the server and by sending a StopServer object as
    /// a message. StartServer will react in it's
    /// processMessage method.
    /// </summary>
    public class StopServer : ServerConfiguration
    {
        /// <summary>
        /// stops a db4o Server started with StartServer.
        /// </summary>
        /// <exception cref="Exception" />
        public static void Main(string[] args)
        {
            IObjectContainer IObjectContainer = null;
            try
```

```
            {
                // connect to the server
                IObjectContainer = Db4oFactory.OpenClient(HOST, PORT,
USER, PASS);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }

            if (IObjectContainer != null)
            {
                // get the messageSender for the IObjectContainer
                IMessageSender messageSender = IObjectContainer.Ext()
                    .Configure().ClientServer().GetMessageSender();

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

                // close the IObjectContainer
                IObjectContainer.Close();
            }
        }
    }
}
```

9.5. Conclusion

That's it, folks. No, of course it isn't. There's much more to db4o we haven't covered yet: schema evolution, custom persistence for your classes, writing your own query objects, etc. A much more thorough documentation is provided in the reference that you should have also received with the download.

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/.

9.6. Full source

```
using System.IO;
using Db4objects.Db4o;


namespace Db4objects.Db4o.Tutorial.F1.Chapter5
{
    public class ClientServerExample : Util
    {
        public static void Main(string[] args)
        {
            File.Delete(Util.YapFileName);
            AccessLocalServer();
            File.Delete(Util.YapFileName);
            IObjectContainer db =
Db4oFactory.OpenFile(Util.YapFileName);
            try
            {
                SetFirstCar(db);
                SetSecondCar(db);
            }
            finally
            {
                db.Close();
            }

            ConfigureDb4o();
            IObjectServer server =
```

```
Db4oFactory.OpenServer(Util.YapFileName, 0);
        try
        {
            QueryLocalServer(server);
            DemonstrateLocalReadCommitted(server);
            DemonstrateLocalRollback(server);
        }
        finally
        {
            server.Close();
        }

        AccessRemoteServer();
        server = Db4oFactory.OpenServer(Util.YapFileName,
ServerPort);
        server.GrantAccess(ServerUser, ServerPassword);
        try
        {
            QueryRemoteServer(ServerPort, ServerUser,
ServerPassword);
            DemonstrateRemoteReadCommitted(ServerPort,
ServerUser, ServerPassword);
            DemonstrateRemoteRollback(ServerPort, ServerUser,
ServerPassword);
        }
        finally
        {
            server.Close();
        }
    }

    public static void SetFirstCar(IObjectContainer db)
    {
        Pilot pilot = new Pilot("Rubens Barrichello", 99);
        Car car = new Car("BMW");
        car.Pilot = pilot;
        db.Set(car);
    }

    public static void SetSecondCar(IObjectContainer db)
    {
```

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

        public static void AccessLocalServer()
        {
            IObjectServer server =
Db4oFactory.OpenServer(Util.YapFileName, 0);
            try
            {
                IObjectContainer client = server.OpenClient();
                // Do something with this client, or open more
clients
                client.Close();
            }
            finally
            {
                server.Close();
            }
        }

        public static void QueryLocalServer(IObjectServer server)
        {
            IObjectContainer client = server.OpenClient();
            ListResult(client.Get(new Car(null)));
            client.Close();
        }

        public static void ConfigureDb4o()
        {
Db4oFactory.Configure().ObjectClass(typeof(Car)).UpdateDepth(3);
        }

        public static void
DemonstrateLocalReadCommitted(IObjectServer server)
        {
            IObjectContainer client1 =server.OpenClient();
            IObjectContainer client2 =server.OpenClient();
            Pilot pilot = new Pilot("David Coulthard", 98);
```

```csharp
            IObjectSet result = client1.Get(new Car("BMW"));
            Car car = (Car)result.Next();
            car.Pilot = pilot;
            client1.Set(car);
            ListResult(client1.Get(new Car(null)));
            ListResult(client2.Get(new Car(null)));
            client1.Commit();
            ListResult(client1.Get(typeof(Car)));
            ListRefreshedResult(client2, client2.Get(typeof(Car)),
2);
            client1.Close();
            client2.Close();
        }

        public static void DemonstrateLocalRollback(IObjectServer
server)
        {
            IObjectContainer client1 = server.OpenClient();
            IObjectContainer client2 = server.OpenClient();
            IObjectSet result = client1.Get(new Car("BMW"));
            Car car = (Car)result.Next();
            car.Pilot = new Pilot("Someone else", 0);
            client1.Set(car);
            ListResult(client1.Get(new Car(null)));
            ListResult(client2.Get(new Car(null)));
            client1.Rollback();
            client1.Ext().Refresh(car, 2);
            ListResult(client1.Get(new Car(null)));
            ListResult(client2.Get(new Car(null)));
            client1.Close();
            client2.Close();
        }

        public static void AccessRemoteServer()
        {
            IObjectServer server =
Db4oFactory.OpenServer(Util.YapFileName, ServerPort);
            server.GrantAccess(ServerUser, ServerPassword);
            try
            {
                IObjectContainer client =
```

```
Db4oFactory.OpenClient("localhost", ServerPort, ServerUser,
ServerPassword);
                // Do something with this client, or open more
clients
                client.Close();
            }
            finally
            {
                server.Close();
            }
        }

        public static void QueryRemoteServer(int port, string user,
string password)
        {
            IObjectContainer client =
Db4oFactory.OpenClient("localhost", port, user, password);
            ListResult(client.Get(new Car(null)));
            client.Close();
        }

        public static void DemonstrateRemoteReadCommitted(int port,
string user, string password)
        {
            IObjectContainer client1 =
Db4oFactory.OpenClient("localhost", port, user, password);
            IObjectContainer client2 =
Db4oFactory.OpenClient("localhost", port, user, password);
            Pilot pilot = new Pilot("Jenson Button", 97);
            IObjectSet result = client1.Get(new Car(null));
            Car car = (Car)result.Next();
            car.Pilot = pilot;
            client1.Set(car);
            ListResult(client1.Get(new Car(null)));
            ListResult(client2.Get(new Car(null)));
            client1.Commit();
            ListResult(client1.Get(new Car(null)));
            ListResult(client2.Get(new Car(null)));
            client1.Close();
            client2.Close();
        }
```

```
        public static void DemonstrateRemoteRollback(int port, string
user, string password)
        {
            IObjectContainer client1 =
Db4oFactory.OpenClient("localhost", port, user, password);
            IObjectContainer client2 =
Db4oFactory.OpenClient("localhost", port, user, password);
            IObjectSet result = client1.Get(new Car(null));
            Car car = (Car)result.Next();
            car.Pilot = new Pilot("Someone else", 0);
            client1.Set(car);
            ListResult(client1.Get(new Car(null)));
            ListResult(client2.Get(new Car(null)));
            client1.Rollback();
            client1.Ext().Refresh(car,2);
            ListResult(client1.Get(new Car(null)));
            ListResult(client2.Get(new Car(null)));
            client1.Close();
            client2.Close();
        }
    }
}
```

10. 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.

```
using Db4objects.Db4o.Query;

using Db4objects.Db4o.Tutorial.F1.Chapter3;

namespace Db4objects.Db4o.Tutorial.F1.Chapter6
{
    public class EvenHistoryEvaluation : IEvaluation
    {
        public void Evaluate(ICandidate candidate)
        {
            Car car=(Car)candidate.GetObject();
            candidate.Include(car.History.Count % 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.Pilot = pilot1;
    car1.Snapshot();
    db.Set(car1);
```

```
        Pilot pilot2 = new Pilot("Rubens Barrichello", 99);

        Car car2 = new Car("BMW");

        car2.Pilot = pilot2;

        car2.Snapshot();

        car2.Snapshot();

        db.Set(car2);
```

and run our evaluation against them:

```
// queryWithEvaluation

IQuery query = db.Query();
        query.Constrain(typeof (Car));
        query.Constrain(new EvenHistoryEvaluation());
        IObjectSet result = query.Execute();
        Util.ListResult(result);
```

OUTPUT:

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.

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

```
using System.IO;

using Db4objects.Db4o.Query;

using Db4objects.Db4o.Tutorial.F1.Chapter3;

namespace Db4objects.Db4o.Tutorial.F1.Chapter6
{
    public class EvaluationExample : Util
    {
        public static void Main(string[] args)
        {
            File.Delete(Util.YapFileName);
            IObjectContainer db =
Db4oFactory.OpenFile(Util.YapFileName);
            try
            {
                StoreCars(db);
                QueryWithEvaluation(db);
            }
            finally
            {
                db.Close();
            }
        }

        public static void StoreCars(IObjectContainer db)
        {
            Pilot pilot1 = new Pilot("Michael Schumacher", 100);
            Car car1 = new Car("Ferrari");
            car1.Pilot = pilot1;
            car1.Snapshot();
            db.Set(car1);
```

```
            Pilot pilot2 = new Pilot("Rubens Barrichello", 99);

            Car car2 = new Car("BMW");

            car2.Pilot = pilot2;

            car2.Snapshot();

            car2.Snapshot();

            db.Set(car2);

        }


        public static void QueryWithEvaluation(IObjectContainer db)

        {

            IQuery query = db.Query();

            query.Constrain(typeof (Car));

            query.Constrain(new EvenHistoryEvaluation());

            IObjectSet 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.

Some hints around using configuration calls:

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

```
Db4oFactory.Configure()
```

and to an open IObjectContainer/IObjectServer with

```
objectContainer.Ext().Configure()
objectServer.Ext().Configure()
```

When an IObjectContainer/IObjectServer is opened, the global configuration context is cloned and copied into the newly opened IObjectContainer/IObjectServer. Subsequent calls against the global context with Db4oFactory.Configure() have no effect on open IObjectContainers/IObjectServers.

11.2. Calling Methods
Many configuration methods have to be called before an IObjectContainer/IObjectServer is opened and will be ignored if they are called against open IObjectContainers/IObjectServers. Some examples:

```
Configuration conf = Db4oFactory.Configure();
conf.ObjectClass(typeof(Foo)).ObjectField("bar").Indexed(true);
conf.ObjectClass(typeof(Foo)).CascadeOnUpdate();
conf.ObjectClass(typeof(Foo)).CascadeOnDelete();
conf.ObjectClass(typeof(System.Drawing.Image))
   .translate(new TSerializable());
conf.GenerateUUIDs(int.MAX_VALUE);
conf.GenerateVersionNumbers(int.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 = Db4oFactory.Configure();
conf.BlockSize(8);
conf.Unicode(false);
```

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

12. 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 IObjectContainer/IObjectServer:

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

Db4oFactory.Configure().ObjectClass(typeof(Foo)).ObjectField("bar").I
ndexed(true);
```

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
IObjectContainer/IObjectServer 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 IObjectContainer/IObjectServer.

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

```
Db4oFactory.Configure().ObjectClass(typeof(Foo)).ObjectField("bar").I
ndexed(false);
```

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:

```
Db4oFactory.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 IObjectContainer/IObjectServer and they will stay the same for every object when an IObjectContainer/IObjectServer is closed and reopened. Internal IDs **will change** when an object is moved from one IObjectContainer 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 IObjectContainer, 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:

```
Db4oFactory.Configure().GenerateUUIDs(int.MAX_VALUE);

Db4oFactory.Configure().ObjectClass(typeof(Foo)).GenerateUUIDs(true);
```

The respective methods for working with UUIDs are:

```
IExtObjectContainer#GetObjectInfo(Object)

IObjectInfo#GetUUID();

IExtObjectContainer#GetByUUID(Db4oUUID);
```

14. Native Query Optimization

Native Queries will run out of the box in any environment. If optimization is turned on, Native Queries will be converted to SODA queries whenever possible, allowing db4o to use indexes and optimized internal comparison algorithms.

If optimization is turned off or not possible for some reason, a Native Query will 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 valid C# constructs. Current optimization supports the following constructs well:

- compile-time constants
- simple member access
- primitive comparisons
- equality operator
- #Contains()/#StartsWith()/#EndsWith() for Strings
- boolean expressions
- arbitrary method calls (including property accessors) on predicate fields (without any arguments)
- 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 yet.

14.1. Enabling Native Query optimization on the CompactFramework 2.0

Due to some platform limitations, CompactFramework 2.0 users using the more convenient delegate based Native Query syntax that want their queries to be optimized are required to run the Db4oAdmin.exe command line utility on their assemblies prior to deploying them.

The utility which can be found in the /bin folder of this distribution is required because the CompactFramework API does not expose any of the delegate metadata needed by the Native Query optimizer. The tool works by augmenting the bytecode with the necessary delegate metadata and replacing ObjectContainer#Query<Extent> invocations with invocations to a lower level method that makes use of the additional information.

The tool can be easily integrated inside Visual Studio.NET 2005 as a Post Build tool by following the

simple steps below:

- Right click the project you want to enable Native Query optimization for
- Select **'Properties'**
- In the Properties Page select the **'Build Events'** tab
- In the **'Post-build event command line'** text box insert the following text **"<path-to-your-db4o-installation>/bin/Db4oAdmin.exe -cf2-delegates $(TargetPath)"** without the quotes and replacing <path-to-your-db4o-installation> to the correct value for your system.

A complete example can be found in the /src/instrumentation/Db4oAdmin.Example directory of this distribution.

14.2. Build Time Optimization for Native Queries

Db4oAdmin.exe can also be used to pre optimize the Native Queries in a given assembly. This makes it possible to deploy an application without Db4oTools.dll (the assembly where the Native Query runtime optimizer lives) while also possibly reducing Native Query execution time by dropping runtime analysis completely.

Execute the Db4oAdmin.exe command line utility without any arguments and check out the help information.

IMPORTANT: the tool is still in constant development and it currently does not support pre optimization of Native Queries expressed as delegates.

14.3. Monitoring optimization

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

```
NativeQueryHandler handler = ((YapStream)db).GetNativeQueryHandler();
NativeQueryHandler handler =
((YapStream)container).GetNativeQueryHandler();
handler.QueryExecution += OnQueryExecution;
handler.QueryOptimizationFailure += OnQueryOptimizationFailure;
```

15. License

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

15.3. Bundled 3rd Party Licenses

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

-15.4. http://mono-project.com/Cecil Mono.Cecil(MIT/X11)

Used inside Db4objects.Db4o.Tools.dll

Cecil is used to read the CIL code during native queries optimization.

-15.5. http://mono-project.com/Cecil Cecil.FlowAnalysis(MIT/X11)

Used inside Db4objects.Db4o.Tools.dll

Cecil.FlowAnalysis is used to analyse the CIL code during native queries optimization.

-15.6. **http://mono-project.com/ Mono.GetOptions**(MIT/X11)

Used inside Db4oAdmin.exe

Mono.GetOptions is used to parse the command line of the Db4oAdmin.exe tool.

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