Introduction

db4o is an object database management system developed and distributed by Versant Corporation. db4o is an open-source product and is available under dual license.

Overview

- To install db4o read the getting started manual.
- For information about the database operations with db4o read about the basic operations. Use native queries to query for object in your database. There are also other alternative query methods: query by example and SODA.
- Take a look at basic db4o concepts like transaction handling and ACID properties and object identity .
- To get an idea when db4o loads object into memory read about the activation concept. And learn you update and delete objects with db4o.
- For good query performance you should create indexes.
- Read about all the db4o configuration options available to you. How you can change the storage system, the activation and update depth and how you can configure an individual class.
- db4o also support a client server mode.
- Take a look at advanced features like session containers, unique constrains, backups, db4o ids and UUIDs and meta-information.
- Callbacks allow you to take action on certain database events.
- You should defragment the database from time to time for the best performance.
- Read about the refactoring support in db4o.
- Take a look at the best practices and pitfalls for db4o.
- db4o supports on many platforms and those have some platform specific limitations. Like the pitfalls in a web environment, and Android.
- For replication between db4o, VOD and Hibernate, read the db4o Replication System documentation.

Join The db4o Community

Join the db4o community for additional help, tips and tricks. Ask for help in the db4o forums at any time. If you want to stay informed, subscribe to our blogs.

Product Philosophy

The db4o database is sponsored and supported by Versant Corporation, a publicly-held company (NASD:VSNT) based in Redwood City, California. Versant is a leading developer of object database technology supporting both open source and commercial database initiatives.

Versant's commercial object database technology, targeting extreme scale systems, is powering some of the world's most demanding applications for fortune 1000 companies in industries including:

- Telecommunications: Alcatel-Lucent, Deutsche Telecom, France Telecom, Ericsson, NEC, Nortel, Orange, Samsung, and more.
- Finance: Financial Times, New York Stock Exchange, Dow Jones, Reuters, London Clearing House, Bank of America, and more,
- Transportation: Sabre, GE Railways, BNS Railways, Galileo, and more.
- Defense: Raytheon, Northrop Grumman, Lockheed, and more.
- BioInformatics: Mayo Clinic, St. Jude medical, Eidogen, Science Factory, and more.

db4o has users and customers coming from 170 different countries, from Albania to Zimbabwe, and ranging from world class leaders like Boeing, Bosch, Intel, Ricohand Seagate to a wide range of highly innovative start-up companies.

It is Versant's and the db4o team's mission to give developers a choice differentiated from relational approaches when it comes to object persistence and thus make their life a lot easier. There is no mapping! No mapping annotations or XML mapping meta data. The db4o database is designed to be a universal, affordable product platform that is easy to learn and use. Versant's open source dual-license business model combines the power of an open source development community with servicing commercial customers' needs for a predictable product roadmap, indemnification, single point of contact and full tech support with fast response times. For those requiring the super scale database capabilities, you can find the same easy to learn and implement solution in Versant's commercial products which have been in development for over a decade. This technology is also far more affordable that traditional relational database systems such as Oracle, Sybase, SQL Server, etc and to boot users also enjoy overall reductions in system footprints by as much as 50% due to less indexing data, simpler design, zero mapping.

Data Persistence

Software programs using different data persistence technologies are an integral part of contemporary informational space. More than often such systems are implemented with the help of object-oriented programming language (Java, C#, etc.) and a relational database management system (Oracle, MySQL, etc.). This implementation originally contains a mismatch between relational and object worlds, which is often called object/relational impedance mismatch. The essence of the problem is in the way the systems are designed. Object systems consist of objects and are characterized by identity, state, behavior, encapsulation. The relational model consists of tables, columns, rows and foreign keys and is described by relations, attributes and tuples.

The object-relational mismatch has become enormously significant with the total adoption of OO technology. This resulted in the rapid development of object-relational mappers (ORM), such as Hibernate, EclipseLink or Entity Framework. This solution "cures" the symptoms of the object relational mismatch by adding a layer into the software stack that automates the tedious task of linking objects to tables.

However, this approach creates a huge drain on system performance, drives up software complexity and increases the burden on software maintenance, thus resulting in higher cost of ownership. While the mapper solution may be feasible in large, administered datacenter environments, it is prohibitive in distributed and zero-administration architectures such as those required for embedded databases in client software, mobile devices, middleware or real-time systems.

Significant side effects of the object relational mismatch manifest themselves in unnecessary system overhead with bloated footprints and runtime performance issues. Of course, there is also overall time to market delays due to poor developer productivity. The overhead still exists in ORM because under the covers, the runtime is still query driven. And, despite improvements in productivity for developers, incremental changes to your object models reek havoc during ORM schema evolution pitfalls. The more complicated your models are, the more problematic keeping changes in sync with the internal mapping.

Primary performance issues come from the fact that despite being called a "relational database", an **RDBMS**[1] does not store direct relations. Relations are resolved at runtime by performing set based operations on primary-foreign key pairs. This means the application has to constantly re-discover data relationships at runtime resulting in immense CPU consumption for something that should be an inherent part of your application model. Further, because discovering these relations over and over again requires continual access to index structures and data to perform the set operations, contention is much higher within database internals leading to poor scalability of individual database processes.

Further, lack of direct storage of relations cause the application design to become query driven instead of object modeling driven. Using an object database, the relations are a fundamental part of the storage architecture. So, application design is model driven. You do not have to suffer any performance overhead for discovering an M-M relationship. The relationships are just there and immediately accessible to the requesting thread. This makes the internal structures much simpler and therefore less contention exists with data requests being isolated to data of interest instead of leveraging indexes or sequential scans. The result, individual processes become more scalable under concurrency.

Technology is ever changing and today there is a whole world of object database experts in the software community. Anyone who is an expert in ORM technology is an expert in object database technology. All of the concepts found in object life cycle management within ORM technologies were invented by the object database community in the early 90's. All of the tuning concepts of closure, fetch configurations, value vs reference types, light weight transactions - are concepts created by and applicable to object database technologies. Now with the growing popularity of object based design and the proliferation of ORM tools, thousands of developers are becoming experts in the object database API.

OODBMS

The emergence of distributed data architectures - in networks, on clients and embedded in "smart" products such as cars or medical devices - is causing companies in an array of industries to look beyond traditional **RDBMS**[2] technology and ORM for an improved way to deal with object persistence.

They are searching for a solution that can handle an enormous number of often complex objects, offer powerful replication and query capabilities, reduce development and maintenance costs and require minimum to zero administration overhead.

---

[1] Relational Database Management System
[2] Relational Database Management System

These requirements can be fulfilled by using an Object Oriented Database Management System (**OODBMS**[1]). OODBMS provides an ideal match with object oriented environments like Java and .NET reducing the cost of development, support and versioning and hence overall system costs.

Using OODBMS in software projects also better supports modern Agile software engineering practices like:

- Continuous refactoring.
- Agile modeling.
- Continuous regression testing.
- Configuration management.
- Developer "sandboxes".

For more information about OODBMS technology refer to the ODBMS.ORG website.

db4o Position

The db4o database came to the market in 2004 with a goal to become the mainstream persistence architecture for embedded applications (in which the database is invisible to the end user) in general, and for mobile and embedded devices running on Java or .NET, in particular. Versant's vision for db4o is to become the affordable, dominant, open source persistence solution of object oriented developers of Java and .NET. In a very short time, the db4o team has achieved mainstream adoption with a fast growing user community currently boasting over 60,000 members. Community adoption is continually driven by db4o's efficient innovative technology, native queries, deployment in Java and .NET and its open source dual licensing business model.

The target environments for db4o are persistence architectures where there is no database administrator present and no **RDBMS**[2] legacy, i.e. primarily on equipment, mobile and desktop clients, and in the middleware. Typical industries of db4o customers include transportation, communication, automation, medical sciences, industrial,, consumer and financial applications, among many others.

Existing customers range from world-class leaders like Boeing, Bosch, Intel, Ricoh, and Seagateto a broad range of highly innovative start-up companies - in the Americas, EMEA, and Asia-Pacific.

As a client-side, embeddable database, db4o is particularly suited to be deployed in devices with embedded software.

For deployments requiring a highly scalable client/server database solution, Versant's commercial product line can deliver a solution with equal ease of use at a surprisingly low cost compared to relational database solutions.

Open Source

db4o database technology uses the now-established, open source dual license business model as pioneered by MySQL, one of the world's most popular relational databases. In this model, db4o is available as open source under the GPL and the dOCL , and as a commercial product under the commercial license. Any developer wishing to use the software in an open source product that falls under the GPL or other open-source licenses (Apache, LGPL, BSD, EPL as specified by the dOCL) can use the free open source version. Those developers wishing to embed db4o into a for-profit product can choose the

---

[1] Object Oriented Database Management System
[2] Relational Database Management System

affordable commercial runtime license. Other uses and licenses including those for evaluation, development, and academic application remain free under the GPL, creating a large and lively community around the product at a very low cost to the vendor.

Success Drivers

Open Source platform usage is one of the key factors of db4o success. db4o's openness attracted a vast (60,000 and counting) community of users and contributors. Through the community support db4o gets broad and immediate testing, receives constructive suggestions (from the users actually looking into the code) and invaluable peer exchange of experiences - positive and negative.

Another factor to db4o success is the technology used. As a new-generation object database, native to both Java and .NET, db4o eliminates the traditional trade-off between performance and object-orientation. Recent PolePosition benchmark results show that db4o outperforms object-relational mappers by orders of magnitude, up to 44x in use cases with complex object models.

db4o uniquely offers object persistence with zero-administration, object-oriented querying, replication and browsing capabilities, and a small footprint. Its single library (JAR/DLL) is easily deployed and runs in the same memory process as the application, making it a fully integrated and tunable portion of the developers application.

Customers, analysts, and experts agree that the db4o object database is one of the world's best and most popular choices, because it stores and retrieves objects natively and not only eliminates the overhead and resource consumption of an ORM, but also greatly reduces the product development and maintenance costs, resulting in a lean, fast and easily integratable into an OO development environment persistence solution, far superior in many cases to that of any RDBMS.

db4o Applications

db4o can be used in a wide range of production and educational software. The primary focus is on embedded usage, like mobile systems (phones and handhelds), device electronics (printers, cars, robots), SCADA systems etc. The following table provides many (but not all) possible implementations with an explanations of the benefits of db4o in the selected environment:

Environment	Benefits
Educationalsystems	One-line persistence, Object-oriented model, intuitive programming interface make db4o an ideal educational tool. It is easy to use and it provides a meaningful example of object-oriented world. It is also native to most widely used OO languages: Java and .NET
Prototypes	Using db4o to build a prototype system is much quicker than using an **RDBMS**[1]. In case of db4o you do not need to create a data model. Further there is no need to map your object model to the database. The general persistence mechanism is almost transparent and requires minimum effort to adapt to. Automatic refactoring allows rapid change of classes without the necessity to update the database.
SCADA	Using db4o in SCADA systems allows to achieve high performance in caching and replay of the events. Another benefit is a small footprint and easy integration with Java and .NET programming languages. db4o can also be run as

[1] Relational Database Management System

	a memory database, providing better performance though minimizing disk access.
Mobile applications	Mobile applications can benefit from in-process database, which requires zero-administration. Synchronization with the main server can be done with the help of **dRS**[1]. Automatic refactoring can be another valuable factor, which allows to skip the job of updating the databases when a new version of object model is implemented.
Device applications	Device applications enjoy the same benefits as Mobile applications. In addition, smaller footprints can be achieved by using the minimal Micro edition.
Open-source software	GPL, open-source compatibility licence. Native to Java and .NET. Easily integrates with any Java and .NET open-source products.
Web-applications	open-source, reporting support from several Java open-source reporting frameworks and .NET reporting API

However, other applications might not be well suited for db4o.

For example, in situations where you have increasing amounts of data (over 10 Gigabytes) and high concurrency (over 20 concurrent users/processes) along with your complex models. In these cases, the Versant database is likely a more appropriate choice. Versant's customer applications span a wide range of use including those exhibiting 1000's of current transactions (100's of thousands of concurrent tx per second) to 100's of gigabytes with some Versant customers in the 25T+ sized database. For more information visit http://www.versant.com/

Another case is when you have simple and flat data model, primarily used for reporting. Simple table-like models of tuple records may be better supported by an RDBMS. In this case, adhoc data access would be more important to your application than well defined use cases using an object model. Typically this is complimented with the need let your users to be able to grab one of the plethora of commercial tools to poke at the database in an adhoc fashion.

Scalability

db4o is intended for embedded use with smaller databases around 2-16 GByte and at maximum 256 GByte. As a general rule, if you expect your database to grow beyond 16 gigabytes, you should look at the Versant object database.

db4o is explicitly single-threaded. Concurrent accesses will be synchronized against a global database lock.That means db4o cannot deal with a highly concurrent access, since it blocks on all operations. When you expect a high load and concurrent access, then you should consider a larger database like the Versant object database.

If you want to make sure that your application can grow and scale with db4o database you may take the following steps:

- Write performance tests to accommodate for performance requirements of the growing application. Especially look that you capture the complexity of the model and don't use a simplified flat data model.
- Also check that you can the throughput you need with concurrent access.

---

[1] db4o Replication System

Why Choose db4o

There are many advantages of using "native" object technology over **RDBMS**[1] or RDBMS paired with an OR mapper, and these technological advantages significantly impact an organization's competitiveness and bottom line.

First, object databases not only simplify development by eliminating the resource-consuming OR-mismatch entirely, but they also foster more sophisticated and differentiated product development through gains in flexibility and productivity brought on by "true" object-orientation.

Second, with an object database, the object schema is the same as the data model. Developers can easier update their models to meet changing requirements, or for purposes of debugging or refactoring. db4o lets developers work with object structures almost as if they were "in-memory'" structures. Little additional coding is required to manage object persistence. As a result, companies can add new features to their products faster to stay ahead of the competition.

Third, developers can now use object-oriented and entirely native approaches when it comes to querying, since db4o was the first in the industry to provide Native Queries (**NQ**[2]) with its Version 5.0 launched in November 2005 and since introduction of LINQ in .NET version 3.5. Db4o Native Queries provide an API that uses the programming languages Java or .NET to access the database. No time is spent on learning a separate data manipulation or query language. Unlike incumbent, string-based, non-native APIs (such as SQL, OQL, JDOQL and others) Native Queries and LINQ are 100% type-safe, 100% refactorable and 100% object-oriented, boosting developer productivity by up to 30%. In addition, the sophisticated modern programming development environments can be used to simplify the development and maintenance work even further.

Fourth, db4o's ground-breaking object-oriented replication technology solves problems arising from distributed data architectures. Partially connected devices need to efficiently replicate data with peers or servers. The challenge lies in the creation of "smart" conflict resolution algorithms, when redundant data sets are simultaneously modified and need to be merged. With db4o's OO approach, developers can build smarter and easier synchronization conflict resolution and embed the necessary business logic into the data layer, rather than into the middle-tier or application layer. This creates "smart" objects that can be stored in a distributed fashion, but easily consolidated, as the object itself knows how to resolve synchronization conflicts. It also enables db4o solution on a client to synchronize data with an RDBMS backend server.

As a result, developers can now more consistently persist data on distributed, partially connected clients than ever before, while decreasing bandwidth requirements and increasing the responsiveness and reach of their mobile solutions or smart devices to make products more competitive in the marketplace.

Fifth, db4o also allows for more complex object models than its relational or non-native counterparts do. As the persistence requirements become more complex, db4o's unique design easily handles (or absorbs) the added complexity, so developers can continue to work as though new complexity were never introduced. Complexity means not only taller object trees and extensive use of inheritance, but also dynamically evolving object models, most extremely if development is taking place under runtime conditions (which makes db4o a leading choice for biotech simulation software, for instance). db4o could be referred to as "agnostic to complexity," because it can automatically handle changes to the

---

[1] Relational Database Management System
[2] Native Query

data model, without requiring extra work. No type or amount of complexity will change its behavior or restrain its capabilities, as is the case with RDBMS or non-native technology. With db4o breaking through this complexity, developers are able to write more user friendly and business-appropriate software components without incurring such high costs and modify them as needed, throughout the life cycle of the product with the same low cost.

In sum, db4o's native, object oriented architecture enables its users to build more competitive products with faster update cycles, more natural object models that match more realistically their use cases, and more distributed data architectures to increase the reach of products. db4o is clearly more flexible and powerful for embedded DB applications than any non-native **OODBMS**[1] or RDBMS technologies available.

[1] Object Oriented Database Management System

Basics Operations & Concepts

This topic gives you an overview of the most important and basic features of db4o. First add db4o to your project. db4o doesn't need a complex setup. It's just a library which you can add to your project. See "Getting Started" on page 10

Basic Operations

The basic operations are unsurprisingly, storing, updating, querying and deleting objects. See "The Basic Operations" on page 11

```
ObjectContainer container = Db4oEmbedded.openFile("databaseFile.db4o");
try {
    // store a new pilot
    Pilot pilot = new Pilot("Joe");
    container.store(pilot);

    // query for pilots
    List<Pilot> pilots = container.query(new Predicate<Pilot>() {
        @Override
        public   boolean match(Pilot pilot) {
            return pilot.getName().startsWith("Jo");
        }
    });

    // update pilot
    Pilot toUpdate = pilots.get(0);
    toUpdate.setName("New Name");
    container.store(toUpdate);

    // delete pilot
    container.delete(toUpdate);
} finally {
    container.close();
}
```

Db4oBasics.java: The basic operations

For more information about queries: See "Querying" on page 12

Basic Concepts

There are some basic concepts which are used in db4o. Understanding them helps you to understand how db4o behaves. Some behaviors are common behaviors which you expect from a database, like transactions and ACID properties. See "ACID Properties and Transactions" on page 37

db4o manages objects by identity. This means db4o doesn't need additional id-field on your object. See "Identity Concept" on page 40

Other concepts are more unique to db4o, like the activation concept, which controls which objects are loaded from the storage to memory. See "Activation Concept" on page 43. The same principals are also applied when updating (See "Update Concept" on page 47) or deletion objects (See "Delete Behavior" on page 61

Getting Started

You can start using db4o within a few minutes following these few steps.

1. Download db4o

First download db4o on the official download site. There are different releases available on the web site. The latest production version, beta versions, continues build versions and older stable releases. Use the production version at the top to get started.

For each release there's a ZIP-file which contains db4o, the documentation and the source code.

Download the ZIP-file to your computer and unpack it.

2. Content Of The db4o Distribution

The db4o distribution has following content.

db4o-folder/lib: Contains the db4o database engine and the supporting libraries. There are different versions of the libraries for the Java-versions in this folder.

db4o-folder/doc: Contains all the db4o documentation. There's the complete API-documentation (db4o-folder/doc/api/index.html), the tutorial (db4o-folder/doc/tutorial/index.html) and the reference documentation (db4o-folder/doc/api/index.html)

db4o-folder/ome: Contains the installer for the Object Manager. See "Object Manager Enterprise" on page 230

db4o-folder/src: Contains the full source code of db4o.

3. Adding db4o To Your Project

After you've downloaded and unpacked the db4o distribution, you can start using it. The core of db4o is the single db4o-X.XX-core-javaX-jar and has no additional dependencies. The other jars provide additional functionality See "Dependency Overview" on page 252. You can also use the db4o-X.XX-all-javaX.jar which includes every db4o feature and dependency in one jar.

To use db4o in your project you only need to add the required jars to your project and then you're ready to go.

Here is how to do this with Eclipse:

- Copy the db4o-*.jar to your projects library folder. If you don't have a library-folder yet, create one under your project. For example a folder called 'lib'.
- Right-click on your project in the Package Explorer and choose "Refresh".
- Right-click on your project in the Package Explorer again and choose "Properties".
- Select "Java Build Path" in the tree view on the left.
- Select the "Libraries" tabpage.
- Click "Add JARs".
- Your library folder should appear below your project. Choose the db4o-*.jars in this folder.
- Confirm the changes.
- Expand "Referenced Libraries" branch of your project in Package Explorer.
- Select db4o-* library, right-click and open "Properties".
- Select Javadoc Location in the list and browse to \doc\api folder in your db4o installation.

4. Ready To Go

That's it, now you're ready to go and can use db4o in your project.

The Basic Operations

The object container is the door to the database access. It's the starting point for all database operations.

Accessing a Database

The object container is the interface for accessing the database. To open the database you pass the filename to the object container factory. Normally you should open an object container when the application starts and close it when it is shuts down.

```
ObjectContainer container = Db4oEmbedded.openFile("databaseFile.db4o");
try {
    // use the object container
} finally {
    container.close();
}
```

Db4oBasics.java: Open the object container to use the database

Storing Objects

Storing a object with db4o is extremely easy. Open the object container and pass your object to the store method and db4o will do the rest. There's no mapping required. db4o will read the class meta data, the read the object values with reflection and store the data.

```
ObjectContainer container = Db4oEmbedded.openFile("databaseFile.db4o");
try {
    Pilot pilot = new Pilot("Joe");
    container.store(pilot);
} finally {
    container.close();
}
```

Db4oBasics.java: Store an object

Queries

Querying for objects is also easy. There are different query interfaces available with different benefits. See "Querying" on page 12

The most natural query method is using native queries. Basically you just pass in a instance of a predicate, which filters the objects out you want.

```
ObjectContainer container = Db4oEmbedded.openFile("databaseFile.db4o");
try {
    List<Pilot> pilots = container.query(new Predicate<Pilot>() {
        public    boolean match(Pilot o) {
            return o.getName().equals("Joe");
        }
    });
    for (Pilot pilot : pilots) {
        System.out.println(pilot.getName());
    }
} finally {
    container.close();
}
```

Db4oBasics.java: Query for objects

Update Objects

Updating objects is also easy. First you query for the object which you want to update. Then you change the object and store it again in the database.

```
ObjectContainer container = Db4oEmbedded.openFile("databaseFile.db4o");
try {
    List<Pilot> pilots = container.query(new Predicate<Pilot>() {
        public    boolean match(Pilot o) {
            return o.getName().equals("Joe");
        }
    });
    Pilot aPilot = pilots.get(0);
    aPilot.setName("New Name");
    // update the pilot
    container.store(aPilot);
} finally {
    container.close();
}
```

Db4oBasics.java: Update a pilot

Delete Objects

Use the delete-operation to delete objects.

```
ObjectContainer container = Db4oEmbedded.openFile("databaseFile.db4o");
try {
    List<Pilot> pilots = container.query(new Predicate<Pilot>() {
        public    boolean match(Pilot o) {
            return o.getName().equals("Joe");
        }
    });
    Pilot aPilot = pilots.get(0);
    container.delete(aPilot);
} finally {
    container.close();
}
```

Db4oBasics.java: Delete a object

Querying

db4o supports different query mechanisms.

Native Queries is the main db4o query interface, recommended for general use.

Queries-By-Example are appropriate as a quick start for users who are still acclimating to storing and retrieving objects with db4o, but they are quite restrictive in functionality.

The SODA Query is the underlying internal API. It is provided for backward compatibility and it can be useful for dynamic generation of queries, where Native Queries are too strongly typed. There may be queries that will execute faster in SODA style, so it can be used to tune applications. You can also run snippets of custom query code as part of the SODA query,

Of course, you can mix these strategies as needed.

Native Queries

Wouldn't it be nice to write 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?

Native queries allow you do to this. They bring you a nice query interface which doesn't rely on string literals. 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.

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 where possible and use internal query processor to run them against indexes and without instantiating actual objects.

Dependencies

For a good native query performance you need to add these jars to your project:db4o-X.XX-nqopt-javax.jar, db4o-X.XX-instrumentation-javax.jar, bloat-1.0.jar. See "Dependency Overview" on page 252

Simple Example

Let's look at how a simple native query will look like. See also a collection of example queries.

```
ObjectSet<Pilot> result = container.query(new Predicate<Pilot>() {
    @Override
    public   boolean match(Pilot pilot) {
        return pilot.getName().equals("John");
    }
});
```

NativeQueryExamples.java: Check for equality of the name

Native Query Performance

There's one drawback of native queries: 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 flow-graph. 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 current state of the query optimization process is detailed in the chapter on Native Query Optimization

Native Query Examples

Here's a collection of native query examples. These queries assume that there's a Pilot class with a name and age and a Car class with a pilot and name.

Equality

This query shows you how compare a getter/setter for equality. In this example we compare the name of a person.

```
ObjectSet<Pilot> result = container.query(new Predicate<Pilot>() {
    @Override
    public  boolean match(Pilot pilot) {
        return pilot.getName().equals("John");
    }
});
```

NativeQueryExamples.java: Check for equality of the name

Comparison

You can compare values with the usual comparison operators.

```
ObjectSet<Pilot> result = container.query(new Predicate<Pilot>() {
    @Override
    public  boolean match(Pilot pilot) {
        return pilot.getAge() > 18;
    }
});
```

NativeQueryExamples.java: Compare values to each other

Query For Value Range

Of course you can combine different comparisons. For example you can combine the greater and smaller than operators to check for a range of values.

```
ObjectSet<Pilot> result = container.query(new Predicate<Pilot>() {
    @Override
    public  boolean match(Pilot pilot) {
        return pilot.getAge() > 18 && pilot.getAge()<30;
    }
});
```

NativeQueryExamples.java: Query for a particular rage of values

Combine Check With Logical Operators

Of course you can combine a arbitrary set of conditions with logical operators.

```
ObjectSet<Pilot> result = container.query(new Predicate<Pilot>() {
    @Override
    public  boolean match(Pilot pilot) {
        return (pilot.getAge() > 18 && pilot.getAge()<30)
                || pilot.getName().equals("John");
    }
});
```

NativeQueryExamples.java: Combine different comparisons with the logical operators

Query In Separate Class

You can implement your query in a separate class and then just us it where you need it. This is especially useful when you reuse the same query multiple times. Or you want to give your query a clear name for documentation purposes.

First write your class:

```
class AllJohns extends Predicate<Pilot> {
    @Override
    public  boolean match(Pilot pilot) {
        return pilot.getName().equals("John");
    }
}
```

AllJohns.java: Query as class

And then use it:

```
ObjectSet<Pilot> result = container.query(new AllJohns());
```

NativeQueryExamples.java: Use the predefined query

Arbitrary Code

In principal your query can contain any code and can do the most complex comparisons. However in practice the are limitations. The simple queries are optimized and translated to SODA-queries. This is not possible for complex queries. If the query cannot be optimized, db4o will instantiate all objects and pass it to your query-object. This is a order of magnitude slower than a optimized native query and only feasible for smaller data sets.

```
final List<Integer> allowedAges = Arrays.asList(18,20,33,55);
ObjectSet<Pilot> result = container.query(new Predicate<Pilot>() {
    @Override
    public  boolean match(Pilot pilot) {
        return allowedAges.contains(pilot.getAge()) ||
               pilot.getName().toLowerCase().equals("John");
    }
});
```

NativeQueryExamples.java: Arbitrary code

Native Query Sorting

Native Query syntax allows you to specify a comparator, which will be used to sort the results:

```
final ObjectSet<Pilot> pilots = container.query(new Predicate<Pilot>() {
    @Override
    public  boolean match(Pilot o) {
        return o.getAge() > 18;
    }
}, new QueryComparator<Pilot>() {
    public  int compare(Pilot pilot, Pilot pilot1) {
        return pilot.getName().compareTo(pilot1.getName());
    }
});
```

NativeQueriesSorting.java: Native query with sorting

Native Queries Performance Characteristics

This overview shows which query operations perform well or badly on large datasets. It should give you an idea which operations can be used on large datasets and which operations can only be applied to small datasets.

Native queries are translated to SODA and therefore they share the same basic performance characteristics.

Good Performance Characteristics

For a good query performance fields which are used in a query have to be indexed. Otherwise db4o needs to scan through all objects. With an index these operations should scale logarithmically with the amount of data. The following queries all assume that the fields are indexed.

Equals Operation on Indexed Field

Simple equals operations on indexed fields' perform very well.

```
final String criteria = Item.dataString(rnd.nextInt(NUMBER_OF_ITEMS));
final List<Item> result = container.query(new Predicate<Item>() {
    @Override
    public   boolean match(Item o) {
        return o.getIndexedString().equals(criteria);
    }
});
```

GoodPerformance.java: Equals operation

Not equals operations also do perform well. However a 'not equals' operation tends to return a large result which will slow down the query.

```
final String criteria = Item.dataString(rnd.nextInt(NUMBER_OF_ITEMS));
final List<Item> result = container.query(new Predicate<Item>() {
    @Override
    public   boolean match(Item o) {
        return !o.getIndexedString().equals(criteria);
    }
});
```

GoodPerformance.java: Not equals operation

Navigation Queries on Index Field

Queries which navigate along references are executed also efficiently, as long every field and reference is indexed.

However there's a catch to this: The reference field type has to be a concrete type. If a field type is a generic type, an interface or an object-type, then the query runs slow.

```
final String criteria = Item.dataString(rnd.nextInt(NUMBER_OF_ITEMS));
final List<ItemHolder> result = container.query(new Predicate<ItemHolder>() {
    @Override
    public   boolean match(ItemHolder o) {
        return o.getIndexedReference().getIndexedString().equals(criteria);
    }
});
```

GoodPerformance.java: Navigate across object references

Reference-Queries

Like regular equals operation, comparisons against references also have a good performance.

```
final Item item = loadItemFromDatabase();

final List<ItemHolder> result = container.query(new Predicate<ItemHolder>() {
    @Override
    public   boolean match(ItemHolder o) {
        return o.getIndexedReference()==item;
    }
});
```

GoodPerformance.java: Query by reference

Comparison and Range Queries

Comparison and range queries also perform well.

```
final List<Item> result = container.query(new Predicate<Item>() {
    @Override
    public   boolean match(Item o) {
        return o.getIndexNumber()>criteria;
    }
});
```

GoodPerformance.java: Bigger than

```
final List<Item> result = container.query(new Predicate<Item>() {
    @Override
    public   boolean match(Item o) {
        return o.getIndexNumber()>biggerThanThis && o.getIndexNumber() <smallerThanThis;
    }
});
```

GoodPerformance.java: In between

Date Queries

Simple equals operations on dates are fast. However complex date comparisons are not yet optimized and run extremely slowly. For those you can fallback to SODA.

```
final List<Item> result = container.query(new Predicate<Item>() {
    @Override
    public   boolean match(Item o) {
        return o.getIndexDate().equals(date);
    }
});
```

GoodPerformance.java: Search for a date

Bad Performance Characteristics

Here's an overview of the query operations with bad performances characteristics. The reason is that db4o cannot utilize indexes to perform these queries. Or that the native query optimizer cannot translate the query to SODA. That means the query time grows linearly with the amount of data.

Navigation across Generic/Object/Interface Fields

When your query navigates across a getter which type is a generic parameter, an object or interface then the performance is bad. This is a limitation of the current query system implementation.

```
// The type of the 'indexedReference' is the generic parameter 'T'.
// Due to type type erasure that type is unknown to db4o
final List<GenericItemHolder<Item>> result = container.query(new Predicate<GenericItemHolder<Item>>() {
    @Override
    public   boolean match(GenericItemHolder<Item> o) {
        return o.getIndexedReference().getIndexedString().equals(criteria);
    }
});
```

BadPerformance.java: Navigation non concrete types

String Operations: Like, Contains, StartsWith, Ends With

All string operations beside the simple equals operation cannot use indexes at the moment. Therefore all string operations like contains, like, starts with etc. run slowly. Advanced string operations are not translated to SODA and therefore run even more slowly.

```
final List<Item> result = container.query(new Predicate<Item>() {
    @Override
    public   boolean match(Item o) {
        return o.getIndexedString().contains(criteria);
    }
});
```

BadPerformance.java: Contains and other string operations are slow

Date Comparisons

The native query optimizer doesn't recognize date comparison and therefore such queries run extremely slow. You should fall back to SODA for date queries.

```
final List<Item> result = container.query(new Predicate<Item>() {
    @Override
    public   boolean match(Item o) {
        return o.getIndexDate().after(date);
    }
});
```

BadPerformance.java: Slow date query

Queries on Collections / Arrays

Any query which does contains operations on collections/arrays or navigates across a collection/array field will run slowly. The reason is that db4o cannot index collections. Furthermore the native query optimizer may doesn't recognize such a query and just loads all objects to process the query.

```
final List<CollectionHolder> result = container.query(new Predicate<CollectionHolder>() {
    @Override
    public   boolean match(CollectionHolder o) {
        return o.getItems().contains(item);
    }
});
```

BadPerformance.java: Contains on collection

Computation in Query Expression

When you do a computation in a query expression, then the native query optimizer cannot optimize your query. In that case it will load all objects in order to execute your query.

```
final List<Item> result = container.query(new Predicate<Item>() {
    @Override
    public   boolean match(Item o) {
        return o.getIndexedString().equals("data for " + number);
    }
});
```

BadPerformance.java: Computing expression in query

Therefore you should move any computation outside of the query. Like this:

```
final String criteria = "data for " + number;
final List<Item> result = container.query(new Predicate<Item>() {
    @Override
    public   boolean match(Item o) {
        return o.getIndexedString().equals(criteria);
    }
});
```

BadPerformance.java: Fix computing expression in query

Calling Complex Methods

Calling complex methods in native queries is a bad idea. Most of the time the native query optimizer cannot deal with a complex method and will load all objects to execute the query.

```
final List<Item> result = container.query(new Predicate<Item>() {
    @Override
    public   boolean match(Item o) {
        return o.complexMethod();
    }
});
```

BadPerformance.java: Call complex method

Detect Slow Queries

The best indication that a query is slow is when it cannot use any field index. Install a diagnostic listener and look for the LoadedFromClassIndex message. That message indicates that a query couldn't use any field index for its execution.

For native queries another indication is when the 'NativeQueryNotOptimized' or the 'NativeQueryOptimizerNotLoaded' diagnostic message occurs. Watch out for those as well.

Native Query Optimization

Native queries will run out of the box in any environment. This optimization is turned on by default. Native queries will be converted to SODA where this is possible. This allows db4o to use indexes and optimized internal comparison algorithms. Otherwise native query may be executed by instantiating all objects, using SODA evaluations. Naturally performance will not be as good in this case.

Optimization Theory

For Native Query the bytecode is analyzed to create an AST-like expression tree. Then the flow graph of the expression tree is analyzed and converted to a SODA query graph.

For example:

```
ObjectSet<Pilot> result = container.query(new Predicate<Pilot>() {
    @Override
    public     boolean match(Pilot pilot) {
        return pilot.getName().equals("John");
    }
});
```

NativeQueryExamples.java: Check for equality of the name

First the signature of the given class is analyzed to find out the types. This is used to constrain the type in the SODA-query. Then the bytecode of query is analyzed to find out was it does. When the operations a simple and easy to convert, it will be transformed to complete SODA query:

```
final Query query = container.query();
query.constrain(Pilot.class);
query.descend("name").constrain("John");

final ObjectSet<Object> result = query.execute();
```

SodaQueryExamples.java: A simple constrain on a field

Native Query Optimization

Native query optimization on Java requires db4onqopt.jar and bloat.jar to be present in the classpath. See "Dependency Overview" on page 252. Current optimization supports the following constructs well:

- Compile-time constants.
- Simple member access.
- Primitive comparisons.
- The equals-method on primitive wrappers and strings.
- The contains-,startsWith- and endsWith-method for strings.
- Arithmetic expressions.
- Boolean expressions.
- Static field access.
- Array access for static/predicate fields.
- Arbitrary method calls on static/predicate fields (without candidate based parameters).
- Candidate methods composed of the above.
- Chained combinations of the above.

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

db4o for Java supplies three different possibilities to run optimized native queries. By default native queries are optimized at runtime when the query runs the first time. This is the most convenient way because it doesn't need any preparations.

On certain environments (embedded runtimes, older java releases) this runtime optimization doesn't work. In such cases there are two alternatives. The compile time optimization and the class load time optimization. See "Enhancement Tools" on page 83

For more information on native queries optimization see Monitoring Optimization.

Monitoring Optimization

In order to optimize native queries the bytecode is analyzed and converted into SODA queries. This task isn't easy. If there's any doubt in the correctness of the conversion db4o won't do it. In such cases db4o falls back and instantiates all objects and runs it against the query. This is a order of magnitude slower than optimized queries. Therefore you probably want to monitor the query optimization and be warned when a query isn't optimized. This is possible with the diagnostic listeners.

```
configuration.common().diagnostic().addListener(new DiagnosticListener() {
    @Override
    public  void onDiagnostic(Diagnostic diagnostic) {
        if(diagnostic instanceof NativeQueryNotOptimized){
            System.out.println("Query not optimized"+diagnostic);
        } else  if(diagnostic instanceof NativeQueryOptimizerNotLoaded){
            System.out.println("Missing native query optimisation jars in classpath "+diagnostic);
        }
    }
});
```

NativeQueryDiagnostics.java: Use diagnostics to find unoptimized queries

You can register a diagnostic listener and check for certain messages. There are two messages related to the native query optimization. The first is the **NativeQueryNotOptimized**-message. This tells you that a query couldn't be optimized. Consider simplifying the query. The second is the **NativeQueryOptimizerNotLoaded**-message. This message tells you that db4o couldn't find the libraries needed for the native query optimization. Check that you've included the jars-files you need.

Native Query Optimization At Build Time

If the platform you're running doesn't support optimization at runtime you can use the compile-time optimization. See "Enhancement Tools" on page 83

Create the Enhancement Task

First we define the enhancement-task. This task will process the jar and enhance it.

If you haven't used Ant yet, read more on the official Ant website.

```
<target   name="enhance">
    <db4o-enhance   classtargetdir="${basedir}/bin"
                  jartargetdir="${basedir}/lib"
                  nq="true"   ta="true"
                  collections="true">
        <classpath   refid="project.classpath"/>
        <sources   dir="${basedir}/bin">
            <include   name="**/*.class"/>
        </sources>
    </db4o-enhance>
</target>
```

simple-enhance-example.xml: Define a target which runs the task

And the execute the task after the compilation.

```
<target   name="enhance-nq">
    <db4o-enhance   classtargetdir="${basedir}/bin"
                  jartargetdir="${basedir}/lib"
                  nq="true"   ta="false"
                  collections="false">
        <classpath   refid="project.classpath"/>
        <sources   dir="${basedir}/bin">
            <include   name="**/*.class"/>
        </sources>
    </db4o-enhance>
</target>
```

simple-enhance-example.xml: Only enhance native queries

You can configure Eclipse to run the Ant build with each compile step. Right click on your project and choose 'Properties'. Then switch to 'Builders' and add a new one. Choose the 'Ant Builder'. On the new window choose the build-file which contains the example-code. Switch to the 'Targets'-Tab. There choose the enhance-target for the 'Auto-Build'. Now the enhancer-task will be run by Eclipse automatically. The example project above is configured this way.

Often it's practical to have all persistent classes in a separate project or compile unit. Then the enhancement script runs only for this project. This makes it easy to enhance only the classes for the persistent objects.

There are lot of possibilities to tweak and configure the build-time enhancement so that it fits your requirements.

Query By Example

Query By Example is a very special query method. Basically you pass in a example object to db4o. Then db4o searches the database for all objects which look alike.

The basic principal is like this. It goes through all fields of the example object. If a field doesn't have the default value it is used as constraint. It takes that value and checks for all objects which have an equal value for this field. If more than one field has a value, each value is used as constraint and combined. Here's a simple example.

```
Pilot theExample = new Pilot();
theExample.setName("John");
final ObjectSet<Pilot> result = container.queryByExample(theExample);
```

QueryByExamples.java: Query for John by example

Take a look a the various examples of query by example examples. See "Query By Example Examples" on page 23

Query by Example has fundamental limitations, which limits its use cases. See "Query By Example Limitations" on page 25

Query By Example Examples

Query By Example Basics

For a query you pass in an example object to db4o. db4o will exanimate the object with reflection. Each field which doesn't have the default value will be use as a constrain. This means a number-field which isn't zero, a reference which isn't null or a string which isn't null. In this example we set the name to John. Then db4o will return all pilots with the name John.

```
Pilot theExample = new Pilot();
theExample.setName("John");
final ObjectSet<Pilot> result = container.queryByExample(theExample);
```

QueryByExamples.java: Query for John by example

Or we set the age to 33 and db4o will return all 33 years old pilots.

```
Pilot theExample = new Pilot();
theExample.setAge(33);
final ObjectSet<Pilot> result = container.queryByExample(theExample);
```

QueryByExamples.java: Query for 33 year old pilots

Combine Values

When you set multiple values all will be used as constrain. For example when we set the name to Jo and the age to 29 db4o will return all pilots which are 29 years with the name Jo.

```
Pilot theExample = new Pilot();
theExample.setName("Jo");
theExample.setAge(29);
final ObjectSet<Pilot> result = container.queryByExample(theExample);
```

QueryByExamples.java: Query a 29 years old Jo

All Objects Of A Type

If you pass a empty example db4o will return all objects of that type.

```
Pilot example = new Pilot();
final ObjectSet<Pilot> result = container.queryByExample(example);
```

QueryByExamples.java: All objects of a type by passing a empty example

Alternatively you also can directly pass in the type.

```
final ObjectSet<Pilot> result = container.queryByExample(Pilot.class);
```

QueryByExamples.java: All objects of a type by passing the type

All Objects

When you pass null all objects stored in the database will be returned.

```
final ObjectSet<Pilot> result = container.queryByExample(null);
```

QueryByExamples.java: All objects

Nested Objects

You can also use nested objects as an example. For example with a car and a pilot. We can query for a car which has a pilot with certain constrains. In this example we get the cars which pilot is called Jenny.

```
Pilot pilotExample = new Pilot();
pilotExample.setName("Jenny");

Car carExample = new Car();
carExample.setPilot(pilotExample);
final ObjectSet<Car> result = container.queryByExample(carExample);
```

QueryByExamples.java: Nested objects example

Contains Example

Collections and arrays act a little different. Query by example returns all object which have at least the items in the array or collection from the example. For example it returns the blog post which has the "db4o"-tag in its tag-collection.

```
BlogPost pilotExample = new BlogPost();
pilotExample.addTags("db4o");
final ObjectSet<Car> result = container.queryByExample(pilotExample);
```

QueryByExamples.java: Contains in collections

Structured Contains

You can even check that a item in a collection fulfills certain criteria's. For example we can check that the blog post has an author with the name John in its author-collection.

```
BlogPost pilotExample = new BlogPost();
pilotExample.addAuthors(new Author("John"));
final ObjectSet<Car> result = container.queryByExample(pilotExample);
```

QueryByExamples.java: Structured contains

Query By Example Limitations

Query By Example has by design a lot of limitations:

- You cannot perform advanced query expressions. (OR, NOT, etc.)
- You cannot constrain for default-values like 0 on numbers, empty strings or nulls on references because they would be interpreted as unconstrained.
- You need a constructor to create objects without initialized fields.
- Complex queries by example are not easy to read and interpret what they query for.

SODA Query

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.

SODA is also an underlying db4o querying mechanism, all other query syntaxes are translated to SODA under the hood:

- Query By Example is translated to SODA.
- Native Queries use bytecode analysis to convert to SODA

Understanding SODA will provide you with a better understanding of db4o and will help to write more performant queries and applications.

Take a look at the SODA-examples to get a feel for SODA-API. See "SODA Query Examples" on page 25

Also SODA has special capabilities for certain types like collections etc. See "SODA Special Cases Examples" on page 28

Additionally SODA evaluations can help you implementing queries which go beyond the capabilities of pure SODA queries. See "SODA Evaluations" on page 30

At last, you need a way to sort the results of a query. See "SODA Sorting" on page 32

SODA Query Examples

Here's a collection of SODA-query examples. These queries assume that there's a Pilot class with a name and age, a Car class with a pilot and name and a BlogPost class with list of tags, authors and a Map of meta-data.

There are also a few examples for special cases.

Type Constraint

This is the most basic and most used constraint for SODA-queries. SODA acts like a filter on all stored objects. But usually you're only interested for instances of a certain type. Therefore you need to constrain the type of the result.

```
final Query query = container.query();
query.constrain(Pilot.class);

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

SodaQueryExamples.java: Type constrain for the objects

Field Constraint

You can add constrains on fields. This is done by descending into a field and constrain the value of that field. By default the constrain is an equality comparison.

```
final Query query = container.query();
query.constrain(Pilot.class);
query.descend("name").constrain("John");

final ObjectSet<Object> result = query.execute();
```

SodaQueryExamples.java: A simple constrain on a field

Comparisons

You can do comparison on the field-values. For example to check if something is greater or smaller than something else.

```
Query query = container.query();
query.constrain(Pilot.class);
query.descend("age").constrain(42).greater();

ObjectSet<Object> result = query.execute();
```

SodaQueryExamples.java: A greater than constrain

```
Query query = container.query();
query.constrain(Pilot.class);
query.descend("age").constrain(42).greater().equal();

ObjectSet<Object> result = query.execute();
```

SodaQueryExamples.java: A greater than or equals constrain

Combination of Constraints (AND, OR)

You can combine different constraints with an 'AND' or 'OR' condition. By default all constrains are combined with the 'AND' condition.

```
Query query = container.query();
query.constrain(Pilot.class);
query.descend("age").constrain(42).greater()
        .or(query.descend("age").constrain(30).smaller());

ObjectSet<Object> result = query.execute();
```

SodaQueryExamples.java: Logical combination of constrains

Not-Constrain

You can invert a constrain by calling the not-method.

```
Query query = container.query();
query.constrain(Pilot.class);
query.descend("age").constrain(42).not();

ObjectSet<Object> result = query.execute();
```

SodaQueryExamples.java: Not constrain

String Comparison

There are special compare operations for strings. By default strings are compared by equality and the comparison is case sensitive.

There's the contains-comparison which checks if a field contains a substring. The like-comparison is the case-insensitive version of the contains-comparison.

Also a start-with- and a ends-with-caparison is available for strings. For this you can specify if the comparison is case sensitive or not.

Note that string comparison do not utilize any index in db4o and therefore show bad performance.

```
Query query = container.query();
query.constrain(Pilot.class);
// First strings, you can use the contains operator
query.descend("name").constrain("oh").contains()
        // Or like, which is like .contains(), but case insensitive
        .or(query.descend("name").constrain("AnN").like())
                // The .endsWith and .startWith constrains are also there,
                // the true for case-sensitive, false for case-insensitive
        .or(query.descend("name").constrain("NY").endsWith(false));

ObjectSet<Object> result = query.execute();
```

SodaQueryExamples.java: String comparison

Compare Field With Existing Object

When you have a reference type field, you can compare this field with a certain object. It will compare the field and the object by object identity.

Note that this comparison only works with stored objects. When you use a not yet stored object as constrain, it will use query by example. To force a comparison by object identity, you can add a .Identiy() call.

```
Pilot pilot = container.query(Pilot.class).get(0);

Query query = container.query();
query.constrain(Car.class);
// if the given object is stored, its compared by identity
query.descend("pilot").constrain(pilot);

ObjectSet<Object> carsOfPilot = query.execute();
```

SodaQueryExamples.java: Compare with existing object

Descend Deeper Into Objects

You can descend deeper into the objects by following fields. This allows you to setup complex constraints on nested objects. Note that the deeper you descend into the objects, the more expensive the query is to execute.

```
Query query = container.query();
query.constrain(Car.class);
query.descend("pilot").descend("name").constrain("John");

ObjectSet<Object> result = query.execute();
```

SodaQueryExamples.java: Descend over multiple fields

SODA Special Cases Examples

This topic contains a examples which demonstrate special behavior for some types in SODA. Take also a look at the other SODA examples.

Contains on Collections and Arrays

Collections and arrays have a special behavior in SODA to make them easier to query. For example you can simple use a constrain directly on a collection-field to check if it contains that value.

Note that currently collections cannot be indexed and therefore such a constrain can be slow on a large data set.

```
Query query = container.query();
query.constrain(BlogPost.class);
query.descend("tags").constrain("db4o");

ObjectSet<Object> result = query.execute();
```

SodaQueryExamples.java: Collection contains constrain

Constrains on Collection Members

When you have a collection or array field, you can simply descend further to the collection-member fields. This allows you query for a object, which has a collection and certain objects in that collection.

Note that currently collections cannot be indexed and therefore such a constrain can be slow on a large data set.

```
Query query = container.query();
query.constrain(BlogPost.class);
query.descend("authors").descend("name").constrain("Jenny");

ObjectSet<Object> result = query.execute();
```

SodaQueryExamples.java: Descend into collection members

Contains Key on Maps

You can check a dictionary if it contains a certain key. Similar to collections, you just can directly use a constrain on the collection field. This will compare the value with the keys of the Map.

Note that currently collections cannot be indexed and therefore such a constrain can be slow on a large data set.

```
Query query = container.query();
query.constrain(BlogPost.class);
query.descend("metaData").constrain("source");

ObjectSet<Object> result = query.execute();
```

SodaQueryExamples.java: Map contains a key constrain

Return the Objects of a Field

With SODA you can navigate to a field and return the objects of that field. Note that this only works for reference objects and not for value objects like strings and numbers.

```
Query query = container.query();
query.constrain(Car.class);
query.descend("name").constrain("Mercedes");

// returns the pilot of these cars
ObjectSet<Object> result = query.descend("pilot").execute();
```

SodaQueryExamples.java: Return the object of a field

Mixing With Query By Example

When you have a reference type field, you can also use a query by example constrain for that field. Pass a new object as an example for this.

Note that when you pass a persisted object, it will compare it by object identity and not use it as example. You can force this behavior by adding an explicit by example constrain.

```
Query query = container.query();
query.constrain(Car.class);
// if the given object is not stored,
// it will behave like query by example for the given object
final Pilot examplePilot = new Pilot(null, 42);
query.descend("pilot").constrain(examplePilot);

ObjectSet<Object> carsOfPilot = query.execute();
```

SodaQueryExamples.java: Mix with query by example

Dynamically Typed

SODA is a dynamically query language. By default SODA acts like a filter on all stored objects. You just add constrains which filters the objects to the desired output.

An example for this behavior: You just add an field-constraint without any type-constrain on the object. This will return all objects which have such a field and match the constrain.

Note that such queries do not utilize any index and therefore show bad performance.

```
Query query = container.query();
// You can simple filter objects which have a certain field
query.descend("name").constrain(null).not();

ObjectSet<Object> result = query.execute();
```

SodaQueryExamples.java: Pure field constrains

This also means that you can query for not existing fields. SODA will not complain if a field doesn't exist. Instead it won't return any object, because no object could satisfy the constrain.

```
Query query = container.query();
query.constrain(Pilot.class);
// using not existing fields doesn't throw an exception
// but rather exclude all object which don't use this field
query.descend("notExisting").constrain(null).not();

ObjectSet<Object> result = query.execute();
```

SodaQueryExamples.java: Using not existing fields excludes objects

SODA Evaluations

Sometimes the capabilities of regular SODA-queries is not enough. In such cases you can add evaluations to the SODA-query. A evaluation is a piece of code which runs against objects.

To use a evaluation, you need to pass an instance of the Evaluation-interface as a constrain. db4o will call the match-method of that interface. Implement the match-method of the Evaluation-interface. In the match-method you can get the candidate-object and the object-container. Compare the object and when it matches, pass true to the include-method. Otherwise pass false.

While SODA evaluations are extremely powerful they are also slow. In order to run the evaluation the objects need to be instantiated from the database and then processed by the evaluator. This means that you should use evaluations only when there's no other possibility.

Simple Evaluation

Here's an example for a simple evaluation. This evaluation filters pilots by the age and picks only pilots with an odd-number as age.

First we need to create the evaluation class:

```
class OnlyOddAge implements Evaluation {
    public   void evaluate(Candidate candidate) {
        Pilot pilot = (Pilot) candidate.getObject();
        candidate.include(pilot.getAge()%2!=0);
    }
}
```

SodaEvaluationExamples.java: Simple evaluation which includes only odd aged pilots

After that, you can use the evaluation in the SODA-query. An evaluation is added as a regular constrain.

```
final Query query = container.query();
query.constrain(Pilot.class);
query.constrain(new OnlyOddAge());

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

SodaEvaluationExamples.java: Simple evaluation

Evaluation on Field

It's also possible to use the evaluation on a certain field. For this you descend into the field on which the evaluation should be applied. After that, specify the evaluation as a constrain on that field.

```
final Query query = container.query();
query.constrain(Car.class);
query.descend("pilot").constrain(new OnlyOddAge());

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

SodaEvaluationExamples.java: Evaluation on field

Regex on Fields

Evaluation also allow you to add very specific additional query capabilities. On of the most useful ones is regular expressions. First create a regular expression evaluation:

```
class RegexConstrain implements Evaluation {
    private   final Pattern pattern;

    public RegexConstrain(String pattern) {
        this.pattern = Pattern.compile(pattern);
    }

    public   void evaluate(Candidate candidate) {
        String stringValue = (String) candidate.getObject();
        candidate.include(pattern.matcher(stringValue).matches());
    }
}
```

SodaEvaluationExamples.java: Regex Evaluator

After that you can use it on any string field:

```
final Query query = container.query();
query.constrain(Pilot.class);
query.descend("name").constrain(new RegexConstrain("J.*nn.*"));
ObjectSet result = query.execute();
```

SodaEvaluationExamples.java: Regex-evaluation on a field

SODA Sorting

You can specify to sort by certain fields in SODA. For this you need to descend to the field and use the appropriate order ascending or order descending method.

In cases where this is not enough, you can use a special comparator.

Sorting by Field

To sort by a field navigate to the field and call a order ascending or descending method. Note that this only works for fields which have natural sortable values, such as strings and numbers.

```
final Query query = container.query();
query.constrain(Pilot.class);
query.descend("name").orderAscending();

final ObjectSet<Object> result = query.execute();
```

SodaSorting.java: Order by a field

Sort by Multiple Fields

You can sort by multiple fields. Add a order constrain for each field. The first order statement has the highest priority and last added the lowest.

```
final Query query = container.query();
query.constrain(Pilot.class);
// order first by age, then by name
query.descend("age").orderAscending();
query.descend("name").orderAscending();

final ObjectSet<Object> result = query.execute();
```

SodaSorting.java: Order by multiple fields

Sort With Your Own Comperator

In cases where you have more complex sorting requirements, you can specify your own comparator. It is used like a regular Java-comparator.

```
Query query = container.query();
query.constrain(Pilot.class);
query.sortBy(new QueryComparator<Pilot>() {
    public   int compare(Pilot o, Pilot o1) {
        // sort by string-length
        return (int)Math.signum(o.getName().length() - o1.getName().length());
    }
});

final ObjectSet<Object> result = query.execute();
```

SodaSorting.java: Order by your comparator

SODA Performance Characteristics

This overview shows which query operations perform well or badly on large datasets. It should give you an idea which operations can be used on large datasets and which operations can only be applied for small datasets.

Good Performance Characteristics

For a good query performance fields which are used in a query have to be indexed. Otherwise db4o needs to scan through all objects. With an index these operations should scale logarithmically with the amount of data. The following queries all assume that the fields are indexed.

Equals Operation on Indexed Field

Simple equals operations on indexed fields' perform very well.

```
final Query query = container.query();
query.constrain(Item.class);
query.descend("indexedString")
        .constrain(criteria);
```

GoodPerformance.java: Equals on indexed field

Not equals operations also do perform well. However a 'not equals' operation tends to return a large result which will slow down the query.

```
final Query query = container.query();
query.constrain(Item.class);
query.descend("indexedString")
        .constrain(criteria).not();
```

GoodPerformance.java: Not equals on indexed field

Navigation Queries on Index Fields

Queries which navigate along references are executed also efficiently, as long every field and reference is indexed.

However there's a catch to this: The reference field type has to be a concrete type. If a field type is a generic type, an interface or an object-type, then the query runs slow.

```
// Note that the type of the 'indexedReference' has to the specific type
// which holds the 'indexedString'
final Query query = container.query();
query.constrain(ItemHolder.class);
query.descend("indexedReference").descend("indexedString")
        .constrain(criteria);
```

GoodPerformance.java: Equals across indexed fields

Reference-Queries

Like regular equals operation, comparisons against references also have a good performance.

```
Item item = loadItemFromDatabase();

final Query query = container.query();
query.constrain(ItemHolder.class);
query.descend("indexedReference")
        .constrain(item);
```

GoodPerformance.java: Query by reference

Comparison and Range Queries

Comparison and range queries also perform well.

```
final Query query = container.query();
query.constrain(Item.class);
query.descend("indexNumber")
        .constrain(criteria).greater();
```

GoodPerformance.java: Bigger than

```
final Query query = container.query();
query.constrain(Item.class);
query.descend("indexNumber")
        .constrain(biggerThanThis).greater().and(
        query.descend("indexNumber").constrain(smallerThanThis).smaller());
```

GoodPerformance.java: In between

Date Queries

Comparisons on dates also run fast:

```
final Query query = container.query();
query.constrain(Item.class);
query.descend("indexDate")
        .constrain(date);
```

GoodPerformance.java: Date comparisons are also fast

```
final Query query = container.query();
query.constrain(Item.class);
query.descend("indexDate")
        .constrain(date).greater();
```

GoodPerformance.java: Find a newer date

Bad Performance Characteristics

Here's an overview of the query operations with bad performances characteristics. The reason is that db4o cannot utilize indexes to perform these queries.That means the query time grows linearly with the amount of data.

Since SODA is the low level query API all other query API will also perform badly for these operations.

Navigation across Generic/Object/Interface Fields

When your query navigates across a field which type is a generic parameter, an object or interface then the performance is bad. The reason is that the query engine cannot be sure which objects potentially can be referenced by that field and therefore cannot use the index.

This is not true when the field has a concrete type.

```
// The type of the 'indexedReference' is the generic parameter 'T'.
// Due to type type erasure that type is unknown to db4o
final Query query = container.query();
query.constrain(GenericItemHolder.class);
query.descend("indexedReference").descend("indexedString")
        .constrain(criteria);
```

BadPerformance.java: Navigation across non concrete typed fields

String Operations: Like, Contains, StartsWith, Ends With

All string operations beside the simple equals operation cannot use indexes at the moment. Therefore all string operations like contains, like, starts with etc. run slowly.

```
final Query query = container.query();
query.constrain(Item.class);
query.descend("indexedString")
        .constrain(criteria).contains();
```

BadPerformance.java: Contains is slow

```
final Query query = container.query();
query.constrain(Item.class);
query.descend("indexedString")
        .constrain(criteria).like();
```

BadPerformance.java: Like is slow

Queries on Collections / Arrays

Any query which does contains operations on collections/arrays or navigates across a collection/array field will run slowly. The reason is that db4o cannot index collections.

```
final Query query = container.query();
query.constrain(CollectionHolder.class);
query.descend("items")
        .constrain(itemToQueryFor);
```

BadPerformance.java: Contains on collection

```
final Query query = container.query();
query.constrain(CollectionHolder.class);
query.descend("items")
        .descend("indexedString").constrain(criteria);
```

BadPerformance.java: Navigate into collection

Sorting

db4o does not use indexes for sorting operations. Therefore sorting is not a fast operation. However in most cases a query result is small enough so that the sorting time doesn't consume too much time.

```
final Query query = container.query();
query.constrain(Item.class);
query.descend("indexedString").orderAscending();
```

BadPerformance.java: Sorting a huge result set

Evaluations

Evaluations cannot use indexes and will run slowly.

```
final Query query = container.query();
query.constrain(Item.class);
query.descend("indexedString").constrain(new Evaluation() {
    @Override
    public   void evaluate(Candidate candidate) {
        if (candidate.getObject() instanceof String) {
            String value = (String) candidate.getObject();
            if (value.matches("abc")) {
                candidate.include(true);
            }
        }
    }
});
```

BadPerformance.java: Evaluations

Detect Slow Queries

The best indication that a query is slow is when it cannot use any field index. Install a diagnostic listener and look for the LoadedFromClassIndex message. That message indicates that a query couldn't use any field index for its execution.

SODA Processing

The SODA processing runs in two stages.

First Stage: Index-Lookups

First SODA tries to find the best fitting index for the query. It looks up all available field- and class-indexes. Then it goes through all possible indexes and chooses the index with the smallest candidate result set.

In practice this means that SODA first uses the field-indexes. If no suitable field index is found it falls back to the class-index, which contains all objects of a certain type.

With this index a first set of candidates is created. This means SODA looks up in the index the objects and returns the IDs of the candidate-objects.

Second Stage: Constrain Evaluation

The second stage uses the ids from the first stage and runs all those objects against the given constrains. If the constrains are regular SODA-constrains, it directly uses the values in the database to compare it. If additional evaluations are present, SODA will instantiate the candidate-objects and pass it to the evaluation-function.

Finally SODA will apply the sorting and then return all IDs of objects which match the query criteria.

ACID Properties and Transactions

The ACID properties are one of the oldest and most important concepts of database theory. It sets out the requirements for the database reliability:

- **Atomicity**: This means that must follow an "all or nothing" rule. Each transaction is either successfully completed or in case of failure the state of the database isn't changed at all.
  For example in a bank transfer transaction there are two steps: debit and credit. If the debit operation was successful, but the credit failed, the whole transaction should fail and the system should remain in the initial state.
- **Consistency**: Consistency ensures that the database stays always in a consistent state. Each transaction takes the database from one consistent state to the next consistent state.
- **Isolation**: Isolation means that different operations cannot access modified data from another transaction that has not yet completed. There are different isolation-models. See "Isolation" on page 39
- **Durability**: This just refers to the real goal of any data store. It just means that the data should be persistently stored.

db4o fulfills the ACID properties. Each object container has its own transaction. Each transaction is a unit of work and ensures the ACID properties. This means, that a db4o transaction is an atomic operation. Either all changes of the db4o transactions are committed and made persistent. Or in case of a failure or rollback no state is changed. The database is kept consistent even on application or database crashes. And db4o transactions are isolated from each other. See "db4o Transactions" on page 37

db4o Transactions

All db4o operations are transactional and there's always a transaction running. Each object container has its own transaction running. The transaction is started implicitly.

You can commit the transaction at any time. When the commit-call returns, all changes are made persistent.

Commit A Transactions

In order to commit a transaction, you need to call the commit-method. This will make all changes of the current transaction persistent. When the commit call is finished, everything is safely stored. If something goes wrong during the commit-operation or the commit-operation is interrupted (power-off, crash etc) the database has the state of either before or after the commit-call.

```
container.store(new Pilot("John"));
container.store(new Pilot("Joanna"));

container.commit();
```

Transactions.java: Commit changes

Rollback a Transaction

Of course you also can rollback a transaction. Just call rollback on the object container.

```
container.store(new Pilot("John"));
container.store(new Pilot("Joanna"));

container.rollback();
```

Transactions.java: Rollback changes

Note that when you rollback the changes, db4o won't rollback the objects in memory. All objects in memory will keep the state. If you want to make sure that objects in memory have the same state as in the database, you need to refresh the objects.

```
final Pilot pilot = container.query(Pilot.class).get(0);
pilot.setName("New Name");
container.store(pilot);
container.rollback();

// use refresh to return the in memory objects back
// to the state in the database.
container.ext().refresh(pilot,Integer.MAX_VALUE);
```

Transactions.java: Refresh objects after rollback

Implicit Commits

db4o commits implicitly when you close the object-container. The assumption is that normally you want to make the changes persistent when you close the object container. That's why it commits automatically. When you want to prevent this you should rollback the transaction before closing the container,

Multiple Concurrent Transactions

db4o transactions are always bound to their object container. When you want multiple concurrent transactions, you need to open multiple object containers. You can easily do this with the open session method. See "Session Containers" on page 66

Note that in this mode, db4o uses the read committed isolation. See "Isolation" on page 39

```
ObjectContainer rootContainer = Db4oEmbedded.openFile(DATABASE_FILE_NAME);

// open the db4o-session. For example at the beginning for a web-request
ObjectContainer session = rootContainer.ext().openSession();
try {
    // do the operations on the session-container
    session.store(new Person("Joe"));
} finally {
    // close the container. For example when the request ends
    session.close();
}
```

Db4oSessions.java: Session object container

Isolation

Isolation imposes rules which ensure that transactions do not interfere with each other even if they are executed at the same time. Read more about isolation levels on Wikipedia.

db4o uses the read committed isolation level, on an object level. That's means db4o has a very weak isolation. It ensures that you do not see uncommitted objects of other transactions. However it does not guarantee any consistency across different objects.

Here's an example to demonstrate the isolation level issues. We have two bank accounts. One transaction lists the two bank accounts and sums up the total.

```
long moneyInOurAccounts = 0;
List<BankAccount> bankAccounts = container.query(BankAccount.class);
for (BankAccount account : bankAccounts) {
    System.out.println("This account has "+account.money());
    moneyInOurAccounts +=account.money();
    moveMoneyTransactionFinishes();
}
// We get the wrong answer here
System.out.println("The money total is "+moneyInOurAccounts
        +". Expected is "+INITIAL_MONEY_ON_ONE_ACCOUNT*bankAccounts.size());
```

InconsistentStateRead.java: We list the bank accounts and sum up the money

During that operation another transaction finishes a money transfer from one account to another and commits.

```
List<BankAccount> bankAccounts = container.query(BankAccount.class);
final BankAccount debitAccount = bankAccounts.get(0);
final BankAccount creditAccount = bankAccounts.get(1);

int moneyToTransfer = 200;
creditAccount.withdraw(moneyToTransfer);
debitAccount.deposit(moneyToTransfer);

container.store(debitAccount);
container.store(creditAccount);
container.commit();
```

InconsistentStateRead.java: Meanwhile we transfer money.

Now the other transaction sees one bank account previous transfer, the other account is in the last committed state. Therefore it sees an inconsistent view across these two objects.

Dangerous Practices

db4o lets you configure a lot of low level details. It even lets you configure things which endanger the safety of the database integrity and transaction integrity. You should avoid these settings and only use them in very special cases.

One dangerous setting is disabling file-flushes. When you add the non-flushing decorator you get better performance. However due to the missing file-flushes, the ACID-properties cannot be guaranteed.

```
Storage fileStorage = new FileStorage();
configuration.file().storage(new NonFlushingStorage(fileStorage));
```

DangerousPractises.java: Using the non-flushing storage weakens the ACID-properties

Another setting which endangers the ACID properties is disabling the commit-recovery. This setting should only be used in emergency situations after consulting db4o support. The ACID flow of the commit can be re-enabled after restoring the original configuration.

```
configuration.file().disableCommitRecovery();
```

DangerousPractises.java: Disabling commit-recovery weakens the ACID-properties

db4o's Commit Process

How does db4o make transactions safe, so that it can recover failures? Here's the short overview of the transaction-phases db4o uses.

Phase	In Case Of A Crash
1. During the transactions. New and updated objects are written to a new Slot in the database-file. The id-mapping and freespace changes are kept in the transaction.	The changes are lost, because the id-mapping and freespace changes weren't persisted. Therefore the changes are invisible to the database. The transaction is rolled back.
2. Committing starts: The id-changes and free-space changes are written to a new slot, without damaging the old information.	The changes are lost, because the id-mapping and freespace changes haven't been completely stored. The transaction is rolled back.
3. Write the location of latest id-records, and free-space changes to the first location with and additional checksum.	If the record write was completed, the transaction is resumed and completed. If not, the old information is used.
4. Write the of latest id-recods, and free-space changes to the backup location with and additional checksum.	If the record wasn't completely written, the transaction is resumed.

Of course you don't need to worry about this. db4o ensures that a transaction either completes or is rolled back. Whenever you call commit and the call succeeds, all changes are persisted. If your application or db4o crashes before a successfully commit-call, all changes are undone.

Identity Concept

You've maybe noticed that you don't need to add an identifier to your objects in order to store them with db4o. So how does db4o manage objects? db4o uses the object-identity to identify objects. db4o

ensures that each stored object in the database has only one in memory representation per object container. If you load an object in different ways db4o will always return the same object. Or as rule of thumb: The objects in the database behave like objects in memory. When you run multiple queries or retrieve objects in another way, the same object in the database will always represented by the same object in memory.

```
final Car theCar = container.query(Car.class).get(0);
final Pilot thePilot = container.query(Pilot.class).get(0);
Pilot pilotViaCar = theCar.getPilot();
assertTrue(thePilot == pilotViaCar);
```

IdentityConcepts.java: db4o ensures reference equality

In order to implement this behavior each object container keeps a mapping between the objects in memory and the stored object representation. When you load the same object with multiple object-containers (for example with session-containers or in client-server-mode), it will have different in memory-identity. db4o ensures the same identity only for a single object-container.

```
final Car loadedWithContainer1 = container1.query(Car.class).get(0);
final Car loadedWithContainer2 = container2.query(Car.class).get(0);
assertFalse(loadedWithContainer1 == loadedWithContainer2);
```

IdentityConcepts.java: Loading with different object container results in different objects

This also means that an object should always be processed with the same object-container. When you load a object in one container and store it with another container db4o cannot recognize the object and will store it as a completely new object. Therefore you need to use the same container to store and load objects.

```
final Car loadedWithContainer1 = container1.query(Car.class).get(0);
container2.store(loadedWithContainer1);
// Now the car is store twice.
// Because the container2 cannot recognize objects from other containers
// Therefore always use the same container to store and load objects
printAll(container2.query(Car.class));
```

IdentityConcepts.java: Don't use different object-container for the same object.

The identity concept works really well for desktop and embedded applications where you can have a single object container and keep that container open while the application is running. In such a case the behavior is just like you would work with regular objects. However this behavior doesn't work where you need to serialize objects, for example in web-applications. In such scenarios you need to do some extra work. See "Disconnected Objects" on page 182

Further Information

Maybe your wondering why db4o manages object by identity. Why not by equality? There are good reasons why this is the case. See "Identity Vs Equality" on page 41

In order to manage objects by identity db4o has a reference cache which contains all loaded objects. See "The Reference Cache" on page 42

Identity Vs Equality

One of the most common questions is why db4o doesn't allow to use equals and hash code to identify objects in the database. From the first glance it seems like a very attractive idea to let the developer decide what should be the base for comparing objects and making them unique in the database. For

example if the database identity is based on the object's field values it will prevent duplicate objects from being stored to the database, as they will automatically be considered one object.

Yes, it looks attractive, but there is a huge pitfall: When we deal with objects, we deal with their references to each other comprising a unique object graph, which can be very complex. Preserving these references becomes a task of storing many-to-many relationships. This task can only be solved by providing unique identification to each object in memory and not only in the database, which means that it can't depend on the information stored in the object (like an aggregate of field values).

To see it clearly, let's look at an example. Suppose we have a Pilot and a Car class and their equals-method is based on comparing field values:

1. Store a pilot with the name 'Joe' and a car with that pilot in the database
2. Retrieve the pilot.
3. Change the pilot-name from 'Joe' to 'John'. Note that though it is the same object from the run-time point of view, these are two different objects for the database based on equals comparison.
4. Now what happens when we load the pilot. Should it return a pilot with the original name 'Joe'. Or the update pilot with the 'John'? What happens if there are hundreds of pilots witch had a pilot with the name 'Joe'. Do all those cars return the new Pilot name? Or the old one? How do you update only the name of a Pilot for only one car?

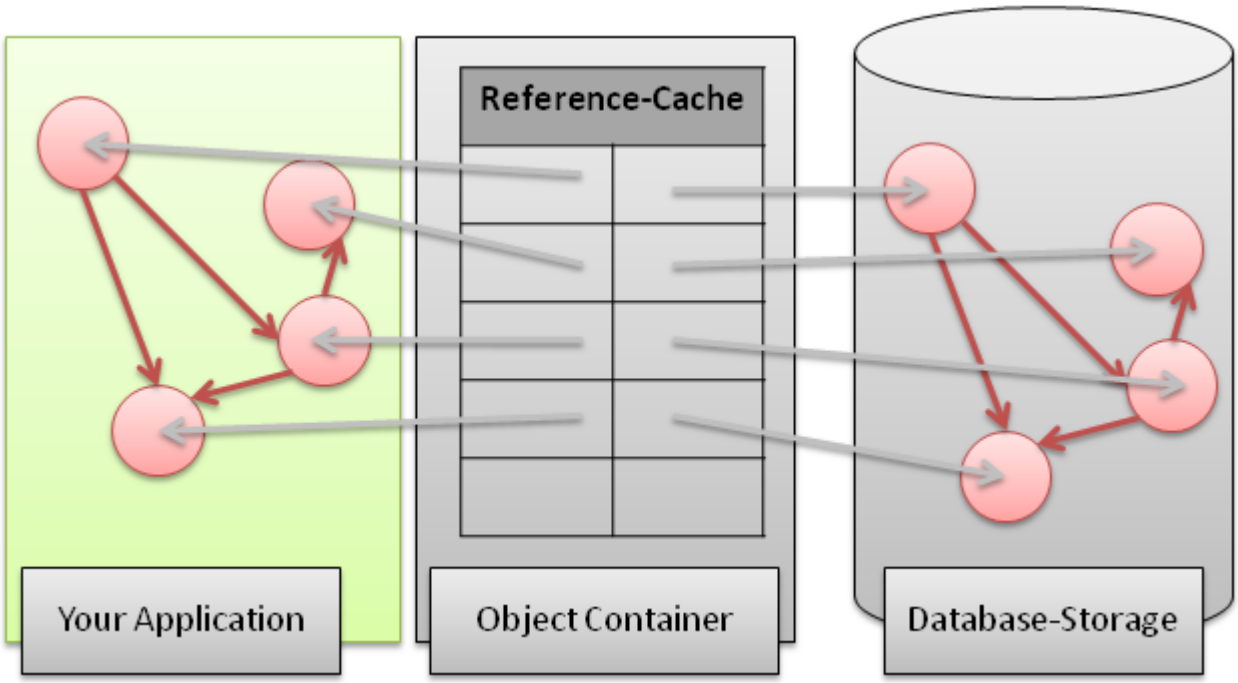
This questions shows that the update-issue is not solvable when the database manages objects by equality. Objects without identity also make Transparent Persistence and Activation impossible, as there will be no way to decide which instance is the right one for update or activation.

So unique identification of database objects in memory is unavoidable and identity based on an object reference is the most straightforward way to get this identification.

The Reference Cache

We know that db4o manages objects by identity. But how does db4o recognize objects? How does it know if it needs to update a object? To archive this, db4o has a reference-cache. This is a table which maps objects in memory to their internal id. The internal id is used to find the object on disk.

Since this table has a reference to the object in memory it also acts as cache. Therefore it's called reference cache. When you load objects, db4o will fist lookup in the reference cache to get objects from there. This avoids loading the data from the disk and also returns the local state of the object. If object isn't in the reference cache, db4o will load it from disk.



Weak Reference

By default db4o uses weak references in the reference cache. While your application has at least one references to an object, the reference cache has reference to it. But as soon as your application has no reference to the object anymore, it can be collected by the garbage collector. db4o will never prevent any object from being garbage collected. In the end persisted objects are garbage collected like any other objects.

To keep the cache clean, db4o does periodically remove all empty weak references. You can configure that clean-up interval. See "Weak Reference Collection Interval" on page 142

You even can disable the weak reference. See "Disable Weak References" on page 142. Then db4o holds regular references to your objects.This prevents the objects from being garbage collected. This means that you need to remove object from the reference cache manually. Or only use short living object containers. See "Session Containers" on page 66

Manually Remove A Object From The Reference Cache

You can manually remove a object from the reference cache. This only required when you have disabled the weak-references.

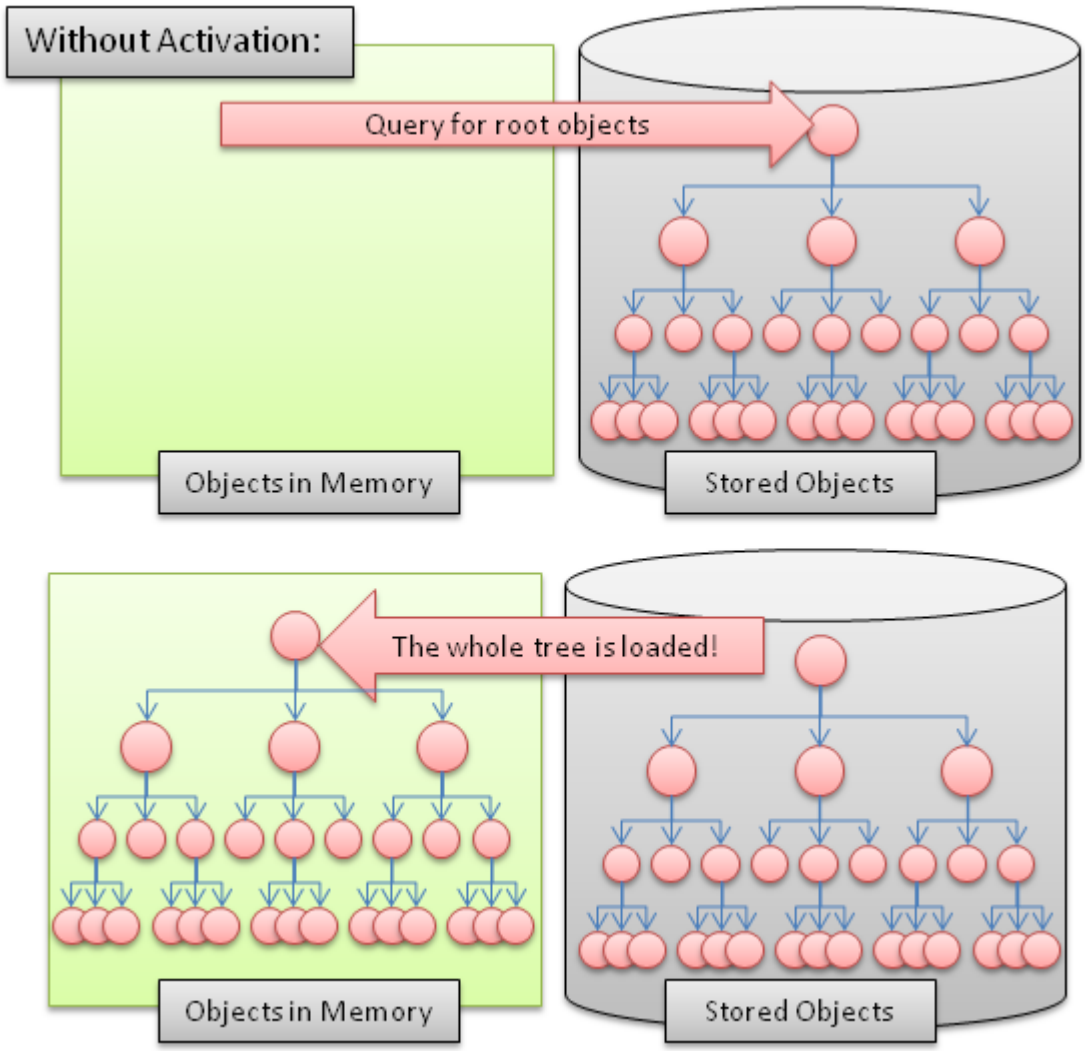
```
final Car theCar = container.query(Car.class).get(0);
container.ext().purge(theCar);
```

IdentityConcepts.java: With purge you can remove objects from the reference cache

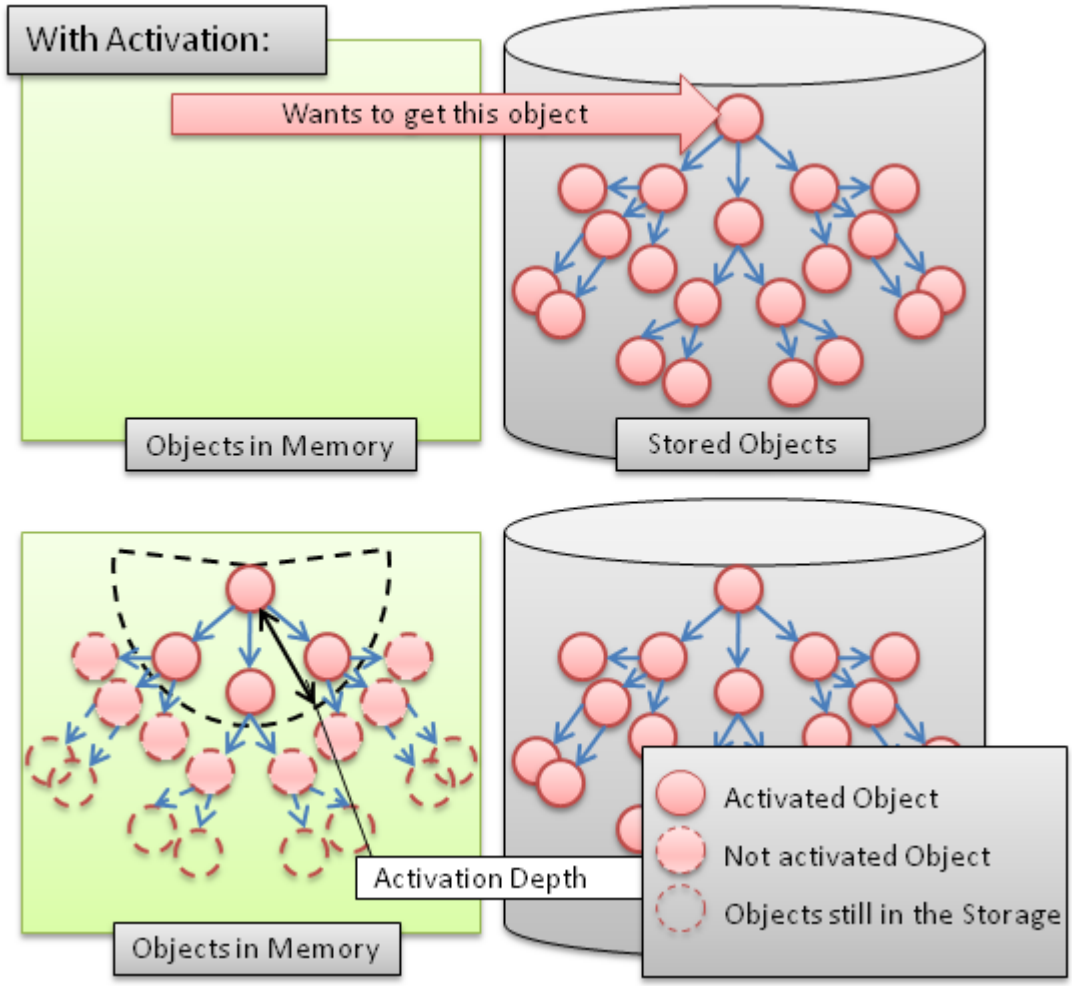
Activation Concept

Activation is a db4o mechanism which controls object instantiation. Why is it necessary? Let's look at an example of a stored tree structure. There's one root object, which has a bunch nodes. Each node has again a few subnodes and so on and so forth. What happens when you run a query and retrieve the root

object? All the sub-objects will have to be created in the memory! If the tree is very large, this will fill up your memory.



Luckily db4o does not behave like this. When a query returns objects, they are loaded into memory (activated in db4o terms) only to a certain activation depth. In this case depth means "number of member references away from the original object". All the fields beyond the activation depth are set to null or to default values. So db4o does not load the whole object graph. Instead, db4o loads only the parts of the object graph you are interested in.



Activation occurs in the following cases:

1. When you iterate over query results.
2. Object is activated explicitly with the object containers activate method.
3. Collections members are activated automatically, when the collection is activated, using at least depth 1 for lists and depth 2 for maps.

For a concrete example of the activation process: See "Activation In Action" on page 45

If you want to automate the activation process: See "Transparent Activation" on page 47

Activation In Action

Let's see db4o's activation in action. To see activation you need a deep object-graph. To keep this example simple we create a person class with a mother-field. This allows us to simply create a very deep object graph.

First the Person class:

```
class Person {
    private Person mother;
    private String name;
    public Person(String name) {
        this.mother = mother;
        this.name = name;
    }

    public Person(Person mother, String name) {
        this.mother = mother;
        this.name = name;
    }

    public Person mother() {
        return mother;
    }

    public String getName() {
        return name;
    }
}
```

Person.java: Person with a reference to the mother

After that store a deep hierarchy of persons, for example a hierarchy of seven people. Then query for it and traverse this object graph. When you hit the sixth person, that object won't be activated, because it's outside the activation depth. That object will have all fields set to null.

```
final Person jodie = queryForJodie(container);
Person julia = jodie.mother().mother().mother().mother().mother();
// This will print null
// Because julia is not activated
// and therefore all fields are not set
System.out.println(julia.getName());
// This will throw a NullPointerException.
// Because julia is not activated
// and therefore all fields are not set
String joannaName = julia.mother().getName();
```

ActivationDepthPitfall.java: Run into not activated objects

Use Explicit Activation

When you traverse deep object graphs, you might run into not activated objects. Therefore you can activate objects explicitly.

```
Person julia = jodie.mother().mother().mother().mother().mother();
container.activate(julia,5);

System.out.println(julia.getName());
String joannaName = julia.mother().getName();
System.out.println(joannaName);
```

ActivationDepthPitfall.java: Fix with explicit activation

Configure Activation

You can configure db4o to increase the activation depth. You can increase it globally or for certain classes. Or you can cascade activate certain objects.

However remember that activation is there to improve the performance and save memory. If you set the activation depth to high it will hurt the performance.

Transparent Activation

If you have a very complex model or don't want to deal with all the activation hassle then transparent activation is the best option. Transparent activation will manage the activation for you. See "Transparent Activation" on page 47

Deactivation

It's also possible to deactivate an object. When you deactivate an object db4o sets all its fields back to their default value and considers it as deactivated. Deactivation is useful in rare cases where you want to return to the inactivated state for some reason.

```
System.out.println(jodie.getName());
container.deactivate(jodie,5);
// Now all fields will be null or 0
// The same applies for all references objects up to a depth of 5
System.out.println(jodie.getName());
```

ActivationDepthPitfall.java: Deactivate an object

Transparent Activation

Activation is a db4o-specific mechanism, which controls object instantiation in a query result. Activation works in several modes and is configurable on a database, object or field level. For more information see Activation.

Using activation in a project with deep object hierarchies and many cross-references on different levels can make activation strategy complex and difficult to maintain. Transparent activation (**TA**[1]) project was started to eliminate this problem and make activation automatic in the same time preserving the best performance and the lowest memory consumption.

With transparent activation enabled, objects are fetched on demand and only those that are used are being loaded.

Go to the transparent activation / persistence chapter to learn more about it.

Update Concept

Updating objects in db4o is as easy as storing them. You just call then store-method again to update a object. How does a update work? There are two main questions. First, how does db4o recognize a object so that it knows whenever it should update a object or store it as a new object? And what's the scope of updates? All all objects updated? Or just the objects you explicitly store?

---

[1] Transparent Activation

Object Recognition

How does db4o know which object needs to be updated and which object has to be stored as new object? Well db4o uses the object-identity and looks up it if has loaded this object. If the object was loaded by db4o, it is an existing object and will be updated. Otherwise it has to be a new object and is stored as a new object.

Update Depth

When you update of a object, db4o only stores the changes to a certain depth. This update depth avoids that db4o needs to go through the whole object graph and find out which objects have changed.

By default this update depth is one. This means when you update a object, only the changes on that object are stored. Changes on other objects are not included. When you want to store changes of multiple objects you need either to increase the update-depth, store each object individually or use transparent persistence.

Take a look at this concrete example to see how the update-depth affects you're operations. See "Update Depth In Action" on page 48

Since collection are regular object in db4o the update depth also applies to collections. See "Updating Collections" on page 49

If you want to automate the update process of object: See "Transparent Persistence" on page 50

Update Depth In Action

Let's see db4o's update depth in action. We store a few cars with their pilots in the database. Then we update a car and its driver and store the car. The we reopen the database and check if everything was updated. To our surprise the car-name was updated, but the driver isn't. This is the direct result of db4o's update depth policy. It only updates object to a certain update-depth.

```
ObjectContainer container = Db4oEmbedded.openFile(DATABASE_FILE);
try {
    Car car = queryForCar(container);
    car.setCarName("New Mercedes");
    car.getDriver().setName("New Driver Name");

    // With the default-update depth of one, only the changes
    // on the car-object are stored, but not the changes on
    // the person
    container.store(car);
} finally {
    container.close();
}
container = Db4oEmbedded.openFile(DATABASE_FILE);
try {
    Car car = queryForCar(container);
    System.out.println("Car-Name:"+car.getCarName());
    System.out.println("Driver-Name:"+car.getDriver().getName());
} finally {
    container.close();
}
```

UpdateDepthPitfall.java: Update depth limits what is store when updating objects

Explicitly Store The Driver

One solution to this issue is to store updated object explicitly, except value objects. So in our case we would store the car and the pilot. This works fine for simple models. However as the model gets more complex this is probably not a feasible solution.

```
Car car = queryForCar(container);
car.setCarName("New Mercedes");
car.getDriver().setName("New Driver Name");

// Explicitly store the driver to ensure that those changes are also in the database
container.store(car);
container.store(car.getDriver());
```

UpdateDepthPitfall.java: Explicitly store changes on the driver

There also a variation of this. You can use the store method of the extended container and explicitly state the update depth for the store operation.

```
Car car = queryForCar(container);
car.setCarName("New Mercedes");
car.getDriver().setName("New Driver Name");

// Explicitly state the update depth
container.ext().store(car,2);
```

UpdateDepthPitfall.java: Explicitly use the update depth

Configure Update Depth

As alternative you can configure the update depth. You can increase it globally or for certain classes. It's also possible to enable cascading updates for certain classes or fields.

Transparent Persistence

You can get rid of all the update depth troubles by using transparent persistence. In this mode db4o tracks all changes and stores them. See "Transparent Persistence" on page 50

Updating Collections

From the db4o perspective collections behave like ordinary objects. This means that the update-depth also applies to collections. When you change a collection and store the object which contains it, the changes are not stored by default.

```
ObjectContainer container = Db4oEmbedded.openFile(DATABASE_FILE);
try {
    Person jodie = queryForJodie(container);
    jodie.add(new Person("Jamie"));
    // Remember that a collection is also a regular object
    // so with the default-update depth of one, only the changes
    // on the person-object are stored, but not the changes on
    // the friend-list.
    container.store(jodie);
} finally {
    container.close();
}
container = Db4oEmbedded.openFile(DATABASE_FILE);
try {
    Person jodie = queryForJodie(container);
    for (Person person : jodie.getFriends()) {
        // the added friend is gone, because the update-depth is to low
        System.out.println("Friend="+person.getName());
    }
} finally {
    container.close();
}
```

UpdateDepthPitfall.java: Update doesn't work on collection

For collections the same rules and settings work as for regular objects. For example when you increase the update depth to two, you can store the parent object and the changes of the collection are persisted as well.

```
EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
config.common().updateDepth(2);
ObjectContainer container = Db4oEmbedded.openFile(config,DATABASE_FILE);
```

UpdateDepthPitfall.java: A higher update depth fixes the issue

Transparent Persistence

One of db4o's goals is to make the database transparent to the application logic. Wouldn't it be nice to initially register an object with a single store()-call and then let the database manage all future object modifications? Transparent Persistence does exactly that.It keeps track of changes and stores all modified objects automatically when committing.

This has several benefits:

- Clean an refactorable code. The code doesn't depend on the right activation-depth.
- Performance-benefits. Only the objects which are needed are loaded into memory. And only modified objects are stored.
- No changes can be lost due to miss configured update-depth.

To get started go to the transparent activation/persistence topics.

Transparent Activation/Persistence

One of the challenges of db4o is to manage the activation of objects. Also updating changed objects can be a challenge. The transparent activation/persistence framework solves these issues.

Transparent activation manages the activation of objects. It activates objects as soon as you access them. That means that you don't have to worry about the activation-depth or cascaded activation. As soon as you use transparent activation you don't need to configure any activation depth or cascading updates.

Transparent persistence builds on top of transparent activation and also manages updating objects. It monitors stored objects and memorizes changed objects. When you commit it automatically stores all updated objects. When you use transparent persistence you don't need any update depth nor cascade update configuration.

Transparent activation and persistence are very similar to implement. Therefore the same steps need to be taken to enable them.

To get started take a look at an example. See "Transparent Persistence Enhancement Example" on page 84

For the learning the implementation details take a look at a manual implementation. See "Implementation" on page 51

Collections need special attention in transparent activation / persistence. See "Collections" on page 58

Also take a look at rollback strategies, which allows db4o to rollback in memory objects. See "Automatic Rollback" on page 60

And finally take a look some well known pitfalls: See "Transparent Activation Pitfalls" on page 60

Implementation

This topic explains how transparent activation/persistence works and how you can implement the required interfaces manually. In practice you should use the provided enhancer's instead of implementing the interfaces yourself. See "Transparent Persistence Enhancement Example" on page 84

Adding the Configuration

We need to explicitly configure transparent activation/persistence.

Transparent Activation

This adds transparent activation, which automatically activates objects.

```
final EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().add(new TransparentActivationSupport());
ObjectContainer container = Db4oEmbedded.openFile(configuration, DATABASE_FILE_NAME);
```

TransparentPersistence.java: Add transparent activation

Transparent Persistence

This adds transparent persistence, which automatically activates objects and manages changes on objects. This includes implicitly the transparent activation support.

Optionally you can specify how rollbacks are handled.

```
final EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().add(new TransparentPersistenceSupport(new DeactivatingRollbackStrategy()));
ObjectContainer container = Db4oEmbedded.openFile(configuration, DATABASE_FILE_NAME);
```

TransparentPersistence.java: Add transparent persistence

Implementing the Activatable Interface

In order to support Transparent Activation, the objects which are stored in the database need to implement the Activatable-interface.

An object which implements the Activatable-interface is responsible for activating itself. For this purpose the class needs a field to keep its activator. This field is only used by the transparent activation/persistence. Therefore it's marked as transient, to avoid that it is stored in the database.

```
import com.db4o.activation.ActivationPurpose;
import com.db4o.activation.Activator;
import com.db4o.ta.Activatable;

public    class Person implements Activatable{

    private    transient Activator activator;
```

Person.java: Implement the required activatable interface and add activator

The implementation of the two methods of the Activatable-interface is straight forward. The bind-method binds an activator to the object. It's called by the transparent activation framework. The activate-method needs to be called before any read or write operation on the object. Since these two methods are always the same, you can move the implementation to a common super class or to a static utility class.

```
public    void bind(Activator activator) {
    if (this.activator == activator) {
        return;
    }
    if (activator != null && null != this.activator) {
        throw    new IllegalStateException("Object can only be bound to one activator");
    }
    this.activator = activator;
}

public    void activate(ActivationPurpose activationPurpose) {
    if(null!=activator){
        activator.activate(activationPurpose);
    }
}
```

Person.java: Implement the activatable interface methods

Now to the most important part. Every time a field of the class is accessed you need to call the activate-method with the purpose. This needs to be done in every getter/setter and method. Probably the best way is to use only getter/setter even within the class to access fields. And the getter/setter ensures that the activate-method is called.

```
public    void setName(String name) {
    activate(ActivationPurpose.WRITE);
    this.name = name;
}

public String getName()  {
    activate(ActivationPurpose.READ);
    return name;
}

public String toString()  {
    // use the getter/setter withing the class,
    // to ensure the activate-method is called
    return getName();
}
```

Person.java: Call the activate method on every field access

Implementing the Activatable-interface manually for every class is repetitive and error prone. That's why this process can be automated.See "Transparent Persistence Enhancement Example" on page 84

After transparent activation/persistence is enabled you can navigate into object-graph as deeply as you want. The transparent activation will load the objects from the database as you need them. When you've enabled transparent persistence updates are also done transparently.

```
{
    ObjectContainer container = openDatabaseWithTA();
    Person person = Person.personWithHistory();
    container.store(person);
    container.close();
}
{
    ObjectContainer container = openDatabaseWithTA();
    Person person = queryByName(container,"Joanna the 10");
    Person beginOfDynasty = person.getMother();

    // With transparent activation enabled, you can navigate deeply
    // nested object graphs. db4o will ensure that the objects
    // are loaded from the database.
    while(null!=beginOfDynasty.getMother()){
        beginOfDynasty = beginOfDynasty.getMother();
    }
    System.out.println(beginOfDynasty.getName());

    container.close();
}
```

TransparentActivationExamples.java: Transparent activation in action

Behavior in Mixed Mode

In some environments there are both, objects which implement the Activatable-interface on other which don't. What's the behavior in this scenario? Then the behavior is this:

- Objects which implement the Activatable-interface are activated when they are used the first time.
- Objects which do not implement the Activatable-interface are always fully loaded.

That behavior ensures that you never run into not activated objects in the transparent activation / persistence mode.

However objects which do not implement the Activatable-interface are not updated with transparent persistence. You have to do that yourself or enhance those classes as well.

Transparent Persistence Enhancement Example

You can inject transparent persistence awareness in your persisted classes without modifying their original code. This is done by enhancing the class-files at build time.

Required jars

For transparent activation/persistence you need following dependencies at compile time. (see also the dependency overview)

- bloat-1.0.jar
- db4o-X.XX-instrumentation.jar
- db4o-X.XX-taj.jar
- db4o-X.XX-tools.jar

Enhance Persistent Classes

The first step is to enhance the persisted classes. One possibility is to introduce an Annotation to mark your persisted classes.

By the way, their are alternative way to select the enhanced classes. See here.

```
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Target(ElementType.TYPE)
public @interface TransparentPersisted {
}
```

TransparentPersisted.java: Annotation to mark persisted classes

This Annotationis then used to mark all persisted classes.

```
@TransparentPersisted
public  class Person {
```

Person.java: Mark your domain model with the annotations

The next step is to create a class filter which reports all classes which should be enhanced. There filter checks for the presence of the annotation.

```
public  final  class AnnotationFilter implements ClassFilter {

    public  boolean accept(Class<?> aClass) {
        if(null==aClass || aClass.equals(Object.class)){
            return  false;
        }
        return hasAnnotation(aClass)
                || accept(aClass.getSuperclass());
    }

    private  boolean hasAnnotation(Class<?> aClass) {
        // We compare by name, to be class-loader independent
        Annotation[] annotations = aClass.getAnnotations();
        for (Annotation annotation : annotations) {
            if(annotation.annotationType().getName()
                    .equals(TransparentPersisted.class.getName())){
                return  true;
            }
        }
        return  false;
    }

}
```

AnnotationFilter.java: Build a filter

Enhancing Classes Using Ant

This enhancement step injects the required bytecode into the domain classes to support transparent activation/persistence.

```
<target  name="enhance">
    <!-- Change these according to your project -->
    <property  name="target"  value="./target/classes/"/>
    <property  name="libraries"  value="./lib/"/>

    <path  id="project.classpath">
        <pathelement  path="${target}"/>
        <fileset  dir="${libraries}">
            <include  name="*.jar"/>
        </fileset>
    </path>

    <!-- We enhance with an additional Ant-run step. You can put this also in an extra file -->
    <typedef  resource="instrumentation-def.properties"
             classpathref="project.classpath"
             loaderRef="instrumentation.loader"/>

    <!-- We filter by our annotation -->
    <typedef  name="annotation-filter"
             classname="com.db4odoc.tp.enhancement.AnnotationFilter"
             classpathref="project.classpath"
             loaderRef="instrumentation.loader"/>

    <db4o-instrument  classTargetDir="${target}"
                  verbose="true">
        <classpath  refid="project.classpath"/>
        <sources  dir="${target}">
            <include  name="**/*.class"/>
        </sources>

        <transparent-activation-step>
            <annotation-filter/>
        </transparent-activation-step>
    </db4o-instrument>
</target>
```

enhance-with-annotation.xml: Ant target for enhancing your classes after building them

Configure Eclipse to Run Ant Target

You can configure Eclipse to run the Ant build with each compile step. Right click on your project and choose 'Properties'. Then switch to 'Builders' and add a new one. Choose the 'Ant Builder'. On the new window choose the build-file which contains the example-code. Switch to the 'Targets'-Tab. There choose the enhance-target for the 'Auto-Build'. Now the enhancer-task will be run by Eclipse automatically. The example project above is configured this way.

Enhancing Classes Using Maven

It's also possible to enhance with Maven by using the Ant plugin.

```
<plugin>
    <artifactId>maven-antrun-plugin</artifactId>
    <version>1.6</version>
    <dependencies>
        <!-- We need the db4o tooling for enhancing stuff -->
        <dependency>
            <groupId>com.db4o</groupId>
            <artifactId>db4o-tools-java5</artifactId>
            <version>8.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
    <executions>
        <execution>
            <phase>compile</phase>
            <configuration>
                <target>
                    <!-- We enhance with an additional Ant-run step. You can put this also in an extra file -->
                    <typedef  resource="instrumentation-def.properties"
                             classpathref="maven.compile.classpath"/>

                    <!-- We filter by our annotation -->
                    <typedef  name="annotation-filter"
                             classname="com.db4odoc.tp.enhancement.AnnotationFilter"
                             classpathref="maven.compile.classpath"/>

                    <db4o-instrument  classTargetDir="target/classes">
                        <classpath  refid="maven.compile.classpath"/>
                        <sources  dir="target/classes">
                            <include  name="**/*.class"/>
                        </sources>

                        <transparent-activation-step>
                            <annotation-filter/>
                        </transparent-activation-step>
                    </db4o-instrument>
                </target>
            </configuration>
            <goals>
                <goal>run</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

pom.xml: Enhance persisted classes during the build

Check Enhancement

You can check if the enhancement worked correctly by checking for the activation interface. Such a check should be part of your test-suite to ensure that everything works correctly.

```
if (!Activatable.class.isAssignableFrom(Person.class)) {
    throw  new AssertionError("Expect that the " + Person.class + " implements " + Activatable.class);
}
```

TransparentPersistence.java: Check for enhancement

Using Transparent Activation

Configure the transparent activation in order to use it.

```
final EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().add(new TransparentActivationSupport());
ObjectContainer container = Db4oEmbedded.openFile(configuration, DATABASE_FILE_NAME);
```

TransparentPersistence.java: Add transparent activation

After that transparent activation is working properly you can transverse along the object graph and don't have to worry about not activated objects:

```
Person person = queryByName(container, "Joanna the 10");
Person beginOfDynasty = person.getMother();

// With transparent activation enabled, you can navigate deeply
// nested object graphs. db4o will ensure that the objects
// are loaded from the database.
while (null != beginOfDynasty.getMother()) {
    beginOfDynasty = beginOfDynasty.getMother();
}
System.out.println(beginOfDynasty.getName());
```

TransparentPersistence.java: Activation just works

Using Transparent Persistence

Transparent persistence not only manages the activation, but also manages updating the objects. Configure transparent persistence in order to use it:

```
final EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().add(new TransparentPersistenceSupport(new DeactivatingRollbackStrategy()));
ObjectContainer container = Db4oEmbedded.openFile(configuration, DATABASE_FILE_NAME);
```

TransparentPersistence.java: Add transparent persistence

After that updated objects are automatically stored every time you commit.

```
Person person = queryByName(container, "Joanna the 10");
Person mother = person.getMother();
mother.setName("New Name");
// Just commit the transaction. All modified objects are stored
container.commit();
```

TransparentPersistence.java: Just update and commit. Transparent persistence manages all updates

Collections

In order to support transparent persistence properly a class needs to implement the Activatable-interface. For your domain classes this is easy to archive. But what about the Java-collections? Wouldn't it be nice when collections also work together the transparent activation framework?

db4o brings special, transparent persistence aware collections with it. This collections load the content only when the collections is actually used.

These collections are currently implemented:

- ActivatableArrayList<T>: An activatable version of the ArrayList<T>
- ActivatableHashMap<T>: An activatable version of the HashMap<T>
- ActivatableHashSet<T>: An activatable version of the HashSet<T>
- ActivatableHashtable<T>: An activatable version of the HashHashtable<T>

- ActivatableLinkedList<T>: An activatable version of the LinkedList<T>
- ActivatableStack<T>: An activatable version of the Stack<T>
- ActivatableTreeSet<T>: An activatable version of the TreeSet<T>

It's recommended to use the collection-interfaces wherever possible instead of the concrete classes. This avoids unnecessary direct dependencies on the implementations and makes it easy to exchange the implementations.

The enhancement tools can automatically replace the Java-collections with the db4o-equivalent. However there are few rules and limitations. See "Enhance Collections" on page 60

You can use the db4o-collection directly in your code. For example in the case you implement the transparent activation support manually. Take a looks these tips: See "Using Transparent Activatable Collections Directly" on page 59

Using Transparent Activatable Collections Directly

You can use the transparent activation aware collections directly in your code. Their behavior is the same as the Java-collections. Here are a few tips:

- Prefer the collection-interfaces over the concrete classes in your field-, parameter- and return-type declarations. For example declare a field as a List<T> instead of an ArrayList<T>
- Maybe it is useful to create a collection factory in your code. Then you have only one place in your code which decides which implementation is used. That makes it easy to replace the implementations.

An example:

```
public   class Team  extends AbstractActivatable{

    private List<Pilot> pilots = new ActivatableArrayList<Pilot>();

    public   boolean add(Pilot pilot) {
        activate(ActivationPurpose.WRITE);
        return pilots.add(pilot);
    }

    public Collection<Pilot> getPilots(){
        activate(ActivationPurpose.READ);
        return pilots;
    }
}
```

Team.java: Using the activation aware collections

Currently these collections are available:

- ActivatableArrayList<T>: An activatable version of the ArrayList<T>
- ActivatableHashMap<T>: An activatable version of the HashMap<T>
- ActivatableHashSet<T>: An activatable version of the HashSet<T>
- ActivatableHashtable<T>: An activatable version of the HashHashtable<T>
- ActivatableLinkedList<T>: An activatable version of the LinkedList<T>
- ActivatableStack<T>: An activatable version of the Stack<T>
- ActivatableTreeSet<T>: An activatable version of the TreeSet<T>

Enhance Collections

You can use the normal Java-collections in your code and then replace the implementations with the enhancement-tools. See "Transparent Persistence Enhancement Example" on page 84

The enhancement tools will search for instantiations of collections and replace it with an appropriate transparent activation aware collection.

The transparent persistence aware collections are always subclasses of the Java-collections. Therefore all methods are available. Furthermore potential cast or instanceof checks also work without any problem.

Automatic Rollback

When you rollback a transaction only the state of database affected. The state of the objects in memory isn't touched by the rollback method. Wouldn't it be nice when the objects in memory are also rolled back on a transaction rollback? This can be done by providing a rollback strategy for the transparent persistence. For that you need to pass a RollbackStrategy-implementation to the transparent persistence support. The rollback method will be automatically called once per modified object after a rollback. On the rollback method you can deactivate the modified object. This ensures that the object state is cleared and read again from the database. This is a simple but effective strategy and therefore is implemented in the DeactivatingRollbackStrategy.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common()
        .add(new TransparentPersistenceSupport(new DeactivatingRollbackStrategy()));
```

RollbackExample.java: Configure rollback strategy

After adding the rollback strategy objects are rolled back aswell.

```
Pilot pilot = container.query(Pilot.class).get(0);
pilot.setName("NewName");
// Rollback
container.rollback();
// Now the pilot has the old name again
System.out.println(pilot.getName());
```

RollbackExample.java: Rollback with rollback strategy

Note that rollback strategies only work for activatable objects.

Transparent Activation Pitfalls

Transparent Persistence is a powerful feature that can make your development much faster, easier and more error-proof. However it can lead to trouble if used in a wrong way. The aim of this chapter is to point you out to typical pitfalls, which lead to unexpected and undesired results.

Not Activate Call Before Field Access

Before accessing any field you need to call the activate-method. This is true for all getter/setter and also for other methods like the toString-method. The best strategy is to call the activate-method in the getter/setter and then access the field through those even in the class itself.

Or use the enhancement-tools to avoid this issue complete.

Migrating Between Databases

Problem

Transparent Activation is implemented through the Activatable-interface, which binds an object to the current object container. When an object is stored in more than one object container, this logic won't work, as only one binding is allowed per object.

Solution

To allow correct behavior of the object between databases, the object should be unbound before being stored to the another database.Just set the activator to null.

Instrumentation Limitations

Problem

For Java built time enhancement the classes to instrument should be on the classpath.

Solution

Make sure that all classes to be instrumented are available through the classpath

Debugging Instrumented Classes

Debugging instrumented classes may not work 100% correct. Make sure that you are using the debug-flag for the db4otool

You should be able to debug normally anywhere around instrumented bytecode. If you still think that the problem occurs in the instrumented area, please submit a bug report to db4o Jira.

Delete Behavior

Deleting an object is as simple as storing an object. You simply call the delete-method on the container to delete it. By default only the object you pass to the delete method is deleted. All referenced objects are not deleted.

```
Car car = findCar(container);
container.delete(car);
// We've deleted the only care there is
assertEquals(0,allCars(container).size());
// The pilots are still there
assertEquals(1,allPilots(container).size());
```

DeletionExamples.java: Deleting object is as simple as storing

Reference To Deleted Objects

What happens when you delete a object which is still referenced by other objects? Well in such cases that reference is set to null.

```
Pilot pilot = findPilot(container);
container.delete(pilot);
```

DeletionExamples.java: Delete the pilot

```
// Now the car's reference to the car is set to null
Car car = findCar(container);
assertEquals(null,car.getPilot());
```

DeletionExamples.java: Reference is null after deleting

Often you want to ensure that a object isn't referenced anymore, before you can delete it. However such referential integrity isn't supported at the moment. You need to implement your integrity checks manually, for example with callbacks.

Cascading Deletion And Collections.

Additionally you can configure cascading behavior for deletion. See "Cascading Deletion" on page 62

Also collections are treated like regular objects and need to be deleted explicitly. See "Collections and Arrays" on page 62

Cascading Deletion

By default db4o only deletes objects which are passed to the delete-method and doesn't delete referenced objects. You can easily change that. Configure the cascading deletion behavior in the configuration for certain classes or certain fields.

For example we mark that the object in the 'pilot'-field is also deleted:

```
EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
config.common().objectClass(Car.class).objectField("pilot").cascadeOnDelete(true);
ObjectContainer container = Db4oEmbedded.openFile(config,DATABASE_FILE);
```

DeletionExamples.java: Mark field for cascading deletion

When we now delete the car, the pilot of that car is also deleted.

```
Car car = findCar(container);
container.delete(car);
// Now the pilot is also gone
assertEquals(0,allPilots(container).size());
```

DeletionExamples.java: Cascade deletion

Collections and Arrays

Collections and arrays don't have a special behavior in db4o. When you delete a collection, the collection-members are not deleted. The collection and objects are two independent objects for db4o.

Removing From A Collection

To remove object from a collection you can simple use the regular collection-operations and then store that collection.

```
PilotGroup group = findGroup(container);
final Pilot pilot = group.getPilots().get(0);
group.getPilots().remove(pilot);
container.store(group.getPilots());

assertEquals(3,allPilots(container).size());
assertEquals(2,group.getPilots().size());
```

DeletionExamples.java: Removing from a collection doesn't delete the collection-members

Remove And Delete Collection Members

If you want to delete a collection-member, remove it and then delete it.

```
PilotGroup group = findGroup(container);
final Pilot pilot = group.getPilots().get(0);
group.getPilots().remove(pilot);
container.store(group.getPilots());
container.delete(pilot);

assertEquals(2,allPilots(container).size());
assertEquals(2,group.getPilots().size());
```

DeletionExamples.java: Remove and delete

Indexing

db4o supports indexes like most databases do. Indexes are data structures which allow efficient lookup of data. When you enable an index, db4o will add an entry to index for each object. This makes the insert and update operation a little slower. However it makes queries a lot faster. A query which uses an index is a order of magnitude faster that a query which cannot use a index.

You can create a index by enabling it on a field. See "Adding a Field Index" on page 161

```
@Indexed
private String zipCode;
```

City.java: Index a field

As an alternative you can configure a field index externally.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).objectField("name").indexed(true);
```

ObjectFieldConfigurations.java: Index a certain field

When And Where Do I Need An Index

When do you need an index? As a rule of thumb: Add an index all fields which are used in queries. See "When And Where" on page 63

There are different factors which need to be fulfilled to profit from an index.

- Read operations dominate the database operations: When you application only writes objects but rarely query for objects, there's no benefit of faster queries. However in most system reading is the dominate operation and should be fast.
- You're using the field / class in a query: If no query touches the field or class you're have a index on, you have not benefit from the index. The index improves only the query performance, but slows down store and update-operations.
- You actually need a substantial amount of data. The performance gains of an index are negligible for small data sets. When you test indexes use 10'000 and more objects.

When And Where

When do you need an index? And on which fields do I need an index? An index has a costs and benefits. See "Costs And Benefits" on page 64. Therefore you should only add one when you actually benefit from it.

When To Add Queries

Indexes speed up queries but slow down store and delete operations. You only need to add indexes when queries are too slow. There's no real benefit to add a index when your queries are already fast enough.

Also add index only on fields which are used in queries. Indexes on fields which are never used in a query have no benefit.

Where To Add Queries

How do I find the queries which can benefit from indexes? How do I find queries which couldn't utilize indexes? You can use the diagnostic-API find out. Filter for the LoadedFromClassIndex-message. Every time this message arrives the query didn't use a field index. You can add a break-point to the message-output and find out which query is the source and then add the index.

```
configuration.common().diagnostic().addListener(new DiagnosticListener() {
    @Override
    public    void onDiagnostic(Diagnostic diagnostic) {
        if(diagnostic instanceof LoadedFromClassIndex)
        {
            System.out.println("This query couldn't use field indexes "+
                ((LoadedFromClassIndex)diagnostic).reason());
            System.out.println(diagnostic);
        }
    }
});
```

WhereToIndexExample.java: Find queries which cannot use index

Costs And Benefits

Maintaining an index is additional work and therefore it has its costs. Each time you store a objects which has an index on it, db4o needs to look up the a appropriate place in the index and store a entry there. This costs is paid for each store, update or delete operation. Therefore there's a tradeoff when adding indexes.

The costs are mostly the time consumed to maintain the index. Additionally there's some additional space consumed to store the index. The benefits is that queries which run a order of magnitude faster when using an index.

In practice you should measure and benchmark your solution and check if a index is a benefit to your application. Ensure that you have enough test data. The influence of indexes shows better with a lot of objects. (10'000 and more objects). However since most applications do a lot more read operations than store and update operations adding a index brings a huge benefit.

Types And Limitations

There are limitations to the db4o indexing. Not all types can be indexed.

Types Which Can Be Indexed

Basically all primitive types like int, long, double etc can be indexed. Indexes on primitive types work extremely well.

Additionally you can index strings, which are handled by db4o like primitives. Strings can be arbitrary long, so a index on string is usually slower than a index on a primitive value. But it's still fast for straight

lookups. Note also that a string index can only be used for equality comparison. Comparisons like contains, start with etc don't use the index.

The Date DateTime, DateTimeOffset and Guid can also be indexed without any issues.

You also can index any object reference except arrays and strings, which are handled like primitives. This means you can index a field which holds a reference to a object and then look up for objects which have a certain reference.

You can also index BigDecimal and BigNumber objects. Those indexes act like indexes on regular primitive types like int or long. Note that you need to add BigMath-support.

Types Which Cannot Be Indexed

Arrays and collections cannot be indexed. The current db4o index implementation cannot deal with those types. This also means that you cannot do fast look-ups on arrays or collections.

Limitations

Currently the index on strings cannot be used for advanced comparisons like contains, starts with etc.

Check For Existing Indexes

Sometime you may want to know if a index exists on a certain field. You can use the db4o-meta information to find out if a field is indexed.

```
StoredClass metaInfo = container.ext().storedClass(IndexedClass.class);
// list a fields and check if they have a index
for (StoredField field : metaInfo.getStoredFields()) {
    if(field.hasIndex()){
        System.out.println("The field '"+field.getName()+"' is indexed");
    } else{
        System.out.println("The field '"+field.getName()+"' isn't indexed");
    }
}
```

CheckForAndIndex.java: Check for a index

Traversing the Index Values

The db4o query processor uses the index to speed up queries. So you just use the query APIs to benefit from indexes. For special cases you might want to traverse the indexed values yourself. You can do this by accessing the index via the db4o meta data. There you can get the meta-info for a field and then traverse its index. Note that the traverse-method will throw an exception if there is no index available.

```
final StoredField storedField = container.ext()
        .storedClass(Item.class).storedField("data", int.class);
storedField.traverseValues(new Visitor4<Integer>() {
    @Override
    public  void visit(Integer fieldValue) {
        System.out.println("Value "+fieldValue);
    }
});
```

TraverseIndexExample.java: Traverse index

Advanced Topics

This topic is about advanced features of db4o.

If you want to have multiple unit of work or multiple transaction in embedded mode you can take a look at session containers. See "Session Containers" on page 66

db4o support unique constrains on fields. See "Unique Constraints" on page 68

You can also do backups of your database at runtime. See "Backup" on page 68

Implementing the interfaces for transparent activation/persistence is tedious. The enhancement tools can do that job for you. And can also optimize native queries at build time. See "Enhancement Tools" on page 83

The database file can fragment over time. In order to reclaim defragmentet space in the database you need to run the defragmentation. See "Defragment" on page 79

Callbacks allow you to perform additional logic for different database operations. See "Callbacks" on page 91

As some point in time you will change your data model. Then you probably need to also refactor you're stored data in the database. See "Refactoring and Schema Evolution" on page 110

You can access the internal ids or generate UUIDs for objects. See "IDs and UUIDs" on page 68

It's also possible to access the meta information of all stored types in the database. See "db4o Meta-Information" on page 70. And you can access system information about the database.

db4o tried hard to make persisting objects as easy as possible. However storing a object efficient and correctly is quite tricky. Read about db4o's type handling for more information. See "Type Handling" on page 99 Furthermore db4o uses an abstraction layer to encapsulate Java-reflection. This allows you do change how reflection behaves on your objects. See "db4o Reflection API" on page 118

Unfortunately failure happen. db4o communicates failures with exceptions. See "Exception-Handling" on page 116

Session Containers

In an application there are often multiple operations running at the same time. A typical example is a web application which processes multiple requests at the same time. These operations should be isolated from each other. This means that for the database we want to have multiple transactions at the same time. Each transaction does some work and other transactions shouldn't interfere .

db4o supports this scenario with session containers. A session container is a lightweight object-container with its own transaction and reference cache, but shares the resources with its parent container. That means you can commit and rollback changes on such a session container without disturbing the work of other session containers. If you want to implement units of work, you might considers using a session container for each unit. You can create such a container with the open session call.

```
ObjectContainer rootContainer = Db4oEmbedded.openFile(DATABASE_FILE_NAME);

// open the db4o-session. For example at the beginning for a web-request
ObjectContainer session = rootContainer.ext().openSession();
try {
    // do the operations on the session-container
    session.store(new Person("Joe"));
} finally {
    // close the container. For example when the request ends
    session.close();
}
```

Db4oSessions.java: Session object container

Transactions And Isolation

As previously mentioned session-containers are isolated from each other. Each session container has its own transaction and its own reference system. This isolation ensures that the different session container don't interfere witch each other.

They don't share the objects loaded and stored with each other. That means you need to load and store the a object with the same session container. When you try to load a object form one session-container and store it with another, you well end up with two separate copies of that object.

Since the transactions are isolated, changes are only visible for other session containers when you've committed. Before the commit call the changes are not visible to other session containers.

```
session1.store(new Person("Joe"));
session1.store(new Person("Joanna"));

// the second session won't see the changes until the changes are committed
printAll(session2.query(Person.class));

session1.commit();

// new the changes are visiable for the second session
printAll(session2.query(Person.class));
```

Db4oSessions.java: Session are isolated from each other

Note also that sessions also have their own reference cache. So when a object is already loaded, it wont be refreshed if another transaction updates the object. You explicitly need to refresh it.

```
Person personOnSession1 = session1.query(Person.class).get(0);
Person personOnSession2 = session2.query(Person.class).get(0);

personOnSession1.setName("NewName");
session1.store(personOnSession1);
session1.commit();


// the second session still sees the old value, because it was cached
System.out.println(personOnSession2.getName());
// you can explicitly refresh it
session2.ext().refresh(personOnSession2, Integer.MAX_VALUE);
System.out.println(personOnSession2.getName());
```

Db4oSessions.java: Each session does cache the objects

Unique Constraints

Unique constraints allow a user to define a field to be unique across all the objects of a particular class stored to db4o. This means that you cannot save an object where a previously committed object has the same field value for fields marked as unique. A Unique Constraint is checked at commit-time and a constraint violation will cause a UniqueFieldValueConstraintViolationException to be thrown. This functionality is based on Commit-Time Callbacks feature.

How To Use Unique Constraints

First you need to add an index on the field which should be unique. After that you add the UniqueFieldValueConstraint to the configuration for the unique field.

```
config.common().objectClass(UniqueId.class).objectField("id").indexed(true);
config.common().add(new UniqueFieldValueConstraint(UniqueId.class, "id"));
```

UniqueConstrainExample.java: Add the index the field and then the unique constrain

After that, the unique constrain is applied. When you commit a transaction the uniqueness of the field is checked. If there's any violation, the UniqueFieldValueConstraintViolationException will be thrown.

```
container.store(new UniqueId(42));
container.store(new UniqueId(42));
try {
    container.commit();
} catch (UniqueFieldValueConstraintViolationException e) {
    e.printStackTrace();
}
```

UniqueConstrainExample.java: Violating the constrain throws an exception

Client-Server

In client server mode you need to configure the unique constrains on the server side.

Backup

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

```
container.ext().backup("backup.db4o");
```

BackupExample.java: Store a backup while using the database

Maybe you want to use a other storage system for the backup than the main database. You can specify the storage system for the backup directly:

```
container.ext().backup(new FileStorage(),"advanced-backup.db4o");
```

BackupExample.java: Store a backup with storage

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

IDs and UUIDs

The db4o team recommends not to use object IDs where it is not necessary. db4o keeps track of object identities in a transparent way, by identifying "known" objects on updates. See "Identity Concept" on

If you really need to have ids for you're object, take a look at this comparison. See "Comparison Of Different IDs" on page 183

Internal IDs

Each object, except value objects like ints, floats or string, do have an internal id. This id is unique within on db4o database and db4o uses it internally for managing the objects. However you also can access this id or retrieve objects by the internal id.

You can get the internal id of an object from the extended object container.

```
long interalId = container.ext().getID(obj);
```

Db4oInternalIdExample.java: get the db4o internal ids

And you can retrieve an object by the internal id. Note that when you get an object by its internal id that it won't be activated. Therefore you have to explicitly activate the object.

```
long internalId =idForObject;
Object objectForID = container.ext().getByID(internalId);
// getting by id doesn't activate the object
// so you need to do it manually
container.ext().activate(objectForID);
```

Db4oInternalIdExample.java: get an object by db4o internal id

db4o UUIDs

db4o can also generate a UUIDs for each object. The UUIDs main purpose is to enable replication. By default db4o doesn't assign a UUID to each object. You have to enable this globally or for certain types. For example:

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().generateUUIDs(ConfigScope.GLOBALLY);
```

FileConfiguration.java: Enable db4o uuids globally

A db4o UUID consists of two parts. The first part is the database signature which is unique to the database.

The second part a unique id within the object-container for the object. Both parts together represent a unique id.

You can get the db4o uuid from the extended object container.

```
Db4oUUID uuid = container.ext().getObjectInfo(obj).getUUID();
```

Db4oUuidExample.java: get the db4o-uuid

And you can get an object by its UUID. Note that when you get an object by its UUIDthat it won't be activated. Therefore you have to explicitly activate the object.

```
Object objectForId = container.ext().getByUUID(idForObject);
// getting by uuid doesn't activate the object
// so you need to do it manually
container.ext().activate(objectForId);
```

Db4oUuidExample.java: get an object by a db4o-uuid

db4o Meta-Information

Db4o meta information API provides an access to the actual structure of db4o database file. Its primary use is refactoring.

You can access the meta information via extended object container. You can ask the object container for all stored classes or for a specific class. To find the meta information for a specific class you can provide the full name, the class itself or an instance of a particular type.

Note that db4o also returns information about internal db4o instances which have been stored.

```
// Get the information about all stored classes.
StoredClass[] classesInDB = container.ext().storedClasses();
for (StoredClass storedClass : classesInDB) {
    System.out.println(storedClass.getName());
}

// Information for a certain class
StoredClass metaInfo = container.ext().storedClass(Person.class);
```

MetaInfoExample.java: All stored classes

The stored class interface provides all meta information db4o knows about. You can get the name of the class, ask for the instance count, ask for a list of the ids and get the meta info for super classes.

The most important information about the stored classes meta info is the list of the field which are stored. You can get a list of all fields or ask for specific fields. Note that the meta information might return information for fields which don't exist anymore. This is useful for refactoring.

```
StoredClass metaInfoForPerson = container.ext().storedClass(Person.class);
// Access all existing fields
for (StoredField field : metaInfoForPerson.getStoredFields()) {
    System.out.println("Field: "+field.getName());
}
// Accessing the field 'name' of any type.
StoredField nameField = metaInfoForPerson.storedField("name", null);
// Accessing the string field 'name'. Important if this field had another time in previous
// versions of the class model
StoredField ageField = metaInfoForPerson.storedField("age",int.class);

// Check if the field is indexed
boolean isAgeFieldIndexed = ageField.hasIndex();

// Get the type of the field
String fieldType = ageField.getStoredType().getName();
```

MetaInfoExample.java: Accessing stored fields

On a the field meta information you can find out the name, type and if the field has an index. And you also can access the values of a object via the stored field. This allows you to access information which is stored in the database but has been removed from the class model. This is useful for refactoring.

```
StoredClass metaForPerson = container.ext().storedClass(Person.class);
StoredField metaNameField = metaForPerson.storedField("name", null);

ObjectSet<Person> persons = container.query(Person.class);
for (Person person : persons) {
    String name = (String)metaNameField.get(person);
    System.out.println("Name is "+name);
}
```

MetaInfoExample.java: Access via meta data

Consistency Check and File Statistics

Both, the consistency check and the file statistics are part of the db4o-X.XX-optional-java5 jar-file. To run the tool you need to have it and the core jar in the class-path. As alternative use the db4o-X.XX-all-java5 jar, which contains all dependencies.

Consistency Check

To run consistency checks use the com.db4o.consistency.ConsistencyChecker class, like this:

```
java -cp db4oX-X.XX-all-java5.jar com.db4o.consistency.ConsistencyChecker databaseFile.db4o
```

This consistency check doesn't check the content of objects. It only checks if the overall structure of the database file is still intact. Also it doesn't offer any repair functionality. It only tells you if the file is corrupted or not.

File Statistics

To run consistency checks use the com.db4o.filestats.FileUsageStatsCollector class, like this:

```
java -cp db4oX-X.XX-all-java5.jar com.db4o.filestats.FileUsageStatsCollector databaseFile.db4o
```

The file statistics return the space usage of objects, indexes etc in bytes. It also returns the statistics for internal objects, managed by db4o.

System Info

You can ask the object container for basic system information. Following information can be accessed:

Freespace Size

```
long freeSpaceSize = container.ext().systemInfo().freespaceSize();
System.out.println("Freespace in bytes: "+freeSpaceSize);
```

SystemInfoExamples.java: Freespace size info

Returns the freespace size in the database in bytes. When db4o stores modified objects, it allocates a new slot for it. During commit the old slot is freed. Free slots are collected in the freespace manager, so they can be reused for other objects.

This method returns a sum of the size of all free slots in the database file.

To reclaim freespace run Defragment.

Freespace Entry Count

```
int freeSpaceEntries = container.ext().systemInfo().freespaceEntryCount();
System.out.println("Freespace-entries count: "+freeSpaceEntries);
```

SystemInfoExamples.java: Freespace entry count info

Returns the number of entries in the Freespace Manager.A high value for the number of freespace entries is an indication that the database is fragmented and that Defragment should be run.

Total Size

```
long databaseSize = container.ext().systemInfo().totalSize();
System.out.println("Database size: "+databaseSize);
```

SystemInfoExamples.java: Database size info

Returns the total size of the database on disk.

Concurrency and db4o

How should you deal with concurrent access to a db4o database? This chapter gives an overview and guidelines for dealing with that.

Do Not Share an Object Containers Across Threads

There are some basic rules which you should never break, otherwise strange effects due to race-condition can happen:

Never share an object-container instance across threads, nor share the data-objects across threads. That will almost certainly create race-conditions. The reason is that when you change objects, while other threads read them, you will get inconsistent views on the state of your data model.

Now how do you deal with concurrent operations? Well you need to use some kind of synchronization strategy.

Use an Object Containers per Unit of Work

You can avoid synchronization when you are using multiple object containers for different units of work. However you need to be aware to the isolation level db4o guarantees. See "Object Container per Unit of Work" on page 77

This is often used in web applications, where you have an object container per request.

Sharing an Object Container

In a desktop/mobile application you usually want to have one consistent view on your data model. Therefore it makes sense to use the same object container in the whole application. That ensures that we always get the same objects throughout the whole application. As long as you don't load of work to different threads, everything is fine.

However as soon as you start to do manipulations on the data model in different threads you need to synchronize these operations. See "Sharing an Object Container Across Threads" on page 73

Sharing an Object Container Across Threads

You want to share an object container across different threads? Then you automatically also share the stored objects across threads, due to the reference-cache. Therefore you need to synchronize the

access to your object-model.

The basic access-pattern should be like this:

1. Access the lock which protects the data model.
2. Do manipulations on the data model, which may involve operations on the object container.
3. Release the lock.

A Small Example

For example we want to do some back-ground updates, while the rest of the application carries on. As soon as we have this kind pattern we need to protect the access to the data model.

For example this operation starts a background task and carries on doing other things:

```
// Schedule back-ground tasks
final Future<?> task = executor.submit(new Runnable() {
    @Override
    public   void run() {
        updateSomePeople(container);
    }
});
// While doing other work
listAllPeople(container);
```

LockingOperations.java: Schedule back-ground tasks

Unfortunately these two tasks do work on the same data model, which can lead to race conditions. Therefore access to the data model has to be protected.

```
private   void updateSomePeople(ObjectContainer container) {
    synchronized (dataLock) {
        final ObjectSet<Person> people = container.query(new Predicate<Person>() {
            @Override
            public   boolean match(Person person) {
                return person.getName().equals("Joe");
            }
        });
        for (Person joe : people) {
            joe.setName("New Joe");
            container.store(joe);
        }
    }
}
```

LockingOperations.java: Grab the lock protecting the data

```
private   void listAllPeople(ObjectContainer container) {
    synchronized (dataLock) {
        for (Person person : container.query(Person.class)) {
            System.out.println(person.getName());
        }
    }
}
```

LockingOperations.java: Grab the lock to show the data

Improving the Locking

Of course the locking showed above is very course grained. A simple improvement would be to use read-write locks. In the end you need to adapt the locking strategy to your application.

Creating a Transaction Model

One model which works good for this scenario is to create a transaction abstraction to do your operations on the data model. Then you do all operations in such a transaction. The transaction manages the lock and the db4o transaction. Such an implementation could look like this:

```
public <T> T inTransaction(TransactionFunction<T> transactionClosure) {
    synchronized (lock) {
        try {
            return transactionClosure.inTransaction(database);
        } catch (Exception e) {
            database.rollback();
            throw   new TransactionFailedException(e.getMessage(), e);
        } finally {
            database.commit();
        }
    }
}
```

DatabaseSupport.java: A transaction method

And then we can use this transaction method when accessing our data model.

```
private   void listAllPeople(DatabaseSupport dbSupport) {
    dbSupport.inTransaction(new TransactionAction() {
        @Override
        public   void inTransaction(ObjectContainer container) {
            final ObjectSet<Person> result = container.query(Person.class);
            for (Person person : result) {
                System.out.println(person.getName());
            }
        }
    });
}
```

TransactionOperations.java: Use transaction for reading objects

```
private   void updateAllJoes(DatabaseSupport dbSupport) {
    dbSupport.inTransaction(new TransactionAction() {
        @Override
        public   void inTransaction(ObjectContainer container) {
            final ObjectSet<Person> allJoes = container.query(new Predicate<Person>() {
                @Override
                public   boolean match(Person person) {
                    return person.getName().equals("Joe");
                }
            });
            for (Person joe : allJoes) {
                joe.setName("New Joe");
                container.store(joe);
            }
        }
    });
}
```

TransactionOperations.java: Use transaction to update objects

Remember that this is only an example. You can use other techniques like annotations or aspects to implement the right behavior. And you also can use more sophisticated locks, like read write locks. The only thing which is important is that you synchronize the access your shared data objects.

Alternatives

Alternatively you can avoid sharing data objects and rather use multiple object container to manage concurrent data access.

Read Write Lock Example

This example is the same as used in the shared container topic. Except that it is using read and write locks.

We replace the lock object with a read write lock:

```
private  final ReadWriteLock dataLock = new ReentrantReadWriteLock();
```

ReadWriteLockingOperations.java: Read write lock

And then on reading operations you use the read lock:

```
private  void listAllPeople(ObjectContainer container) {
    dataLock.readLock().lock();
    try{
        for (Person person : container.query(Person.class)) {
            System.out.println(person.getName());
        }

    } finally{
        dataLock.readLock().unlock();
    }
}
```

ReadWriteLockingOperations.java: Grab the read-lock to show the data

On insert and update operations you grab the write lock. Note that you need the write lock every time you change some data.

```
private  void updateSomePeople(ObjectContainer container) {
    dataLock.writeLock().lock();
    try {
        final ObjectSet<Person> people = container.query(new Predicate<Person>() {
            @Override
            public  boolean match(Person person) {
                return person.getName().equals("Joe");
            }
        });
        for (Person joe : people) {
            joe.setName("New Joe");
            container.store(joe);
        }
    } finally {
        dataLock.writeLock().unlock();
    }
}
```

ReadWriteLockingOperations.java: Grab the write-lock to change the data

Read Write Lock Transactions

You can use read write locks also in a transaction abstraction. This example is an extension of transaction abstraction example but with read write locks.

First we introduce the read write lock.

```
private  final ReadWriteLock dataLock = new ReentrantReadWriteLock();
```

DatabaseSupportWithReadWriteLock.java: Read write lock

Then we implement read and write transaction methods:

```
public <T> T inReadTransaction(TransactionFunction<T> transactionClosure) {
    return inTransaction(dataLock.readLock(),transactionClosure);
}
```

DatabaseSupportWithReadWriteLock.java: The read transaction method

```
public <T> T inWriteTransaction(TransactionFunction<T> transactionClosure) {
    return inTransaction(dataLock.writeLock(),transactionClosure);
}
```

DatabaseSupportWithReadWriteLock.java: The write transaction method

```
private <T> T inTransaction(Lock lockToGrab,TransactionFunction<T> transactionClosure) {
    lockToGrab.lock();
    try {
        return transactionClosure.inTransaction(database);
    } catch (Exception e) {
        database.rollback();
        throw  new TransactionFailedException(e.getMessage(), e);
    } finally {
        database.commit();
        lockToGrab.unlock();
    }
}
```

DatabaseSupportWithReadWriteLock.java: The transaction implementation

After that we can use these operations in our code:

```
private  void listAllPeople(DatabaseSupportWithReadWriteLock dbSupport) {
    dbSupport.inReadTransaction(new TransactionAction() {
        @Override
        public  void inTransaction(ObjectContainer container) {
            final ObjectSet<Person> result = container.query(Person.class);
            for (Person person : result) {
                System.out.println(person.getName());
            }
        }
    });
}
```

ReadWriteTransactionOperations.java: Use a read transaction for reading objects

```
private  void updateAllJoes(DatabaseSupportWithReadWriteLock dbSupport) {
    dbSupport.inWriteTransaction(new TransactionAction() {
        @Override
        public  void inTransaction(ObjectContainer container) {
            final ObjectSet<Person> allJoes = container.query(new Predicate<Person>() {
                @Override
                public  boolean match(Person person) {
                    return person.getName().equals("Joe");
                }
            });
            for (Person joe : allJoes) {
                joe.setName("New Joe");
                container.store(joe);
            }
        }
    });
}
```

ReadWriteTransactionOperations.java: Use a write transaction to update objects

Object Container per Unit of Work

One possibility is to use an object container per unit of work and avoid sharing it across threads. A typical example is to use an object container per request. You can create a new session object container at any time.

Let's take a look at an example. This operation starts a background task and carries on doing other things:

```
// Schedule back-ground tasks
final Future<?> task = executor.submit(new Runnable() {
    @Override
    public   void run() {
        updateSomePeople(container);
    }
});
// While doing other work
listAllPeople(container);
```

UnitsOfWork.java: Schedule back-ground tasks

In this example we use an object container for the background work:

```
private   void updateSomePeople(ObjectContainer rootContainer) {
    ObjectContainer container = rootContainer.ext().openSession();
    try {
        final ObjectSet<Person> people = container.query(new Predicate<Person>() {
            @Override
            public   boolean match(Person person) {
                return person.getName().equals("Joe");
            }
        });
        for (Person joe : people) {
            joe.setName("New Joe");
            container.store(joe);
        }
    } finally {
        container.close();
    }
}
```

UnitsOfWork.java: An object container for the background task

And another background container for the list task.

```
private   void listAllPeople(ObjectContainer rootContainer) {
    ObjectContainer container = rootContainer.ext().openSession();
    try {
        for (Person person : container.query(Person.class)) {
            System.out.println(person.getName());
        }
    } finally {
        container.close();
    }
}
```

UnitsOfWork.java: An object container for this unit of work

Be Aware of the Isolation Level

When using multiple object containers you need to be aware of the transaction isolation. db4o has read committed isolation properties. This isolation applies per object level. Object are loaded individually, which means that the different object-states may are from different committed states.

Here's an example to demonstrate the isolation level issues. We have two bank accounts. One transaction lists the two bank accounts and sums up the total.

```
long moneyInOurAccounts = 0;
List<BankAccount> bankAccounts = container.query(BankAccount.class);
for (BankAccount account : bankAccounts) {
    System.out.println("This account has "+account.money());
    moneyInOurAccounts +=account.money();
    moveMoneyTransactionFinishes();
}
// We get the wrong answer here
System.out.println("The money total is "+moneyInOurAccounts
        +". Expected is "+INITIAL_MONEY_ON_ONE_ACCOUNT*bankAccounts.size());
```

InconsistentStateRead.java: We list the bank accounts and sum up the money

During that operation another transaction finishes a money transfer from one account to another and commits.

```
List<BankAccount> bankAccounts = container.query(BankAccount.class);
final BankAccount debitAccount = bankAccounts.get(0);
final BankAccount creditAccount = bankAccounts.get(1);

int moneyToTransfer = 200;
creditAccount.withdraw(moneyToTransfer);
debitAccount.deposit(moneyToTransfer);

container.store(debitAccount);
container.store(creditAccount);
container.commit();
```

InconsistentStateRead.java: Meanwhile we transfer money.

Now the other transaction sees one bank account previous transfer, the other account is in the last committed state. Therefore it sees an inconsistent view across these two objects.

Defragment

A db4o database file is structured as sets of free and occupied slots, similar to a file system. And just like a file system it can be fragmented, resulting in a file that is larger than it needs to be.

The defragmentation API helps to fix this problem. It creates a new database file and copies all objects from the current database file to the new database file. All indexes are recreated. The resulting database file will be smaller and faster. It is recommended to apply defragmentation on a regular basis to achieve better performance.

Take a look how to defragment your database. See "How To Use Defragmentation" on page 79

Take a look at the defragmentation configuration options. See "Defragmentation Configuration" on page 80

You can register a defragmentation listener which notifies you for certain issues: See "Tracking Defragmentation Errors" on page 83

How To Use Defragmentation

The simplest way to defragment a db4o file would be:

```
Defragment.defrag("database.db4o");
```

DefragmentationExample.java: Simplest possible defragment use case

Ensure that the file is not opened by any other process or db4o instance.

This moves the file filename to filename.backup, then it create a defragmented version of this database at the original position. If the backup file already exists this an IOException is throw and no action will be taken.

You can also specify the backup filename manually:

```
Defragment.defrag("database.db4o", "database.db4o.bak");
```

DefragmentationExample.java: Specify backup file explicitly

For more detailed configuration of the defragmentation process, you can use a DefragmentConfig instance:

```
DefragmentConfig config = new DefragmentConfig("database.db4o");
Defragment.defrag(config);
```

DefragmentationExample.java: Defragment with configuration

It's possible to use a more fine grained configuration for the deframentation process. See "Defragmentation Configuration" on page 80

Defragmentation can throw IOException in the following situations:

- Backup file exists.
- Database file not found.
- Database file is opened by another process.

Defragmentation Configuration

You can configure the defragmentation processes for your needs. This topic discusses the available configuration options.

First you need to configure which database file to defragment. See "Original Database And Backup" on page 80

It's also recommended to use the db4o database configuration for the defragmentation process. This ensures that all low level settings which influence the database-file layout are used. See "Database Configuration" on page 81

When you defragment large database you should configure a commit-frequency to speed up the defragmentation process. See "Commit Frequency" on page 81

If you have refractored your classes you might want to remove old meta data. This is possible with the class filters. See "Class Filter" on page 81

By default the backup file isn't deleted after a successful defragmentation. You can change that. See "Delete The Backup" on page 81

You can force a database update before defragmenting the database-file. See "Upgrade Database File" on page 82

You can disable the read-only more the for backup file. See "Disable Read Only Mode" on page 82

If you want to have different storage implementation for the old database file and the new defragmented database file you can configure a separate storage. See "Configure Storage for Backup File" on page 82

You can change the id-mapping implementation for the defragmentation-process. See "Configure IDMapping" on page 82

Original Database And Backup

The database-file which needs to be defragmented is specified in the configuration. The first constructor parameter is the database file.

```
DefragmentConfig config = new DefragmentConfig("database.db4o");

Defragment.defrag(config);
```

DefragmentationConfigurationExamples.java: Configure the file

The defragmentation process creates a backup of the old database. By default the back-up file has the name of the original database-file with an additional '.backup'-suffix. You can explicitly specify the backup file name with the second constructor parameter.

```
DefragmentConfig config = new DefragmentConfig("database.db4o", "database.db4o.back");

Defragment.defrag(config);
```

DefragmentationConfigurationExamples.java: Configure the file and backup file

Database Configuration

Perhaps you're using low level configuration settings which are file-related. In such cases it's recommended to use the database configuration for the defragmentation process. Especially settings like string-encoding and block-size need to be configured properly for the defragmentation.

```
DefragmentConfig config = new DefragmentConfig("database.db4o");
// It's best to use the very same configuration you use for the regular database
final EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
config.db4oConfig(configuration);

Defragment.defrag(config);
```

DefragmentationConfigurationExamples.java: Use the database-configuration

Commit Frequency

The defragmentation system uses db4o's transactional core to write the data to the new defragmented file. For large databases managing a long transaction can become an issue and can slow down the defragmentation. Therefore you can set a commit frequency. This means after the set number of processed objects the transaction is committed on the new file.

```
DefragmentConfig config = new DefragmentConfig("database.db4o");
config.objectCommitFrequency(10000);

Defragment.defrag(config);
```

DefragmentationConfigurationExamples.java: Set the commit frequency

Class Filter

db4o stores meta data about all classes used in the database . Even when the class doesn't exist anymore the meta-data in db4o is still there. The class filter allows you to remove class-meta data from the defragmented database. You can pass you own implementation of a class filter. Or you can use the built in AvailableClassFilter. This filter removes all meta-data of classes which aren't present anymore.

```
DefragmentConfig config = new DefragmentConfig("database.db4o");
config.storedClassFilter(new AvailableClassFilter());

Defragment.defrag(config);
```

DefragmentationConfigurationExamples.java: Use class filter

Delete The Backup

The defragmentation process copies the data from the old database to a new file. The old file is left as a backup. You can force the defragmentation process to delete the backup after a successful

defragmenting.

```
DefragmentConfig config = new DefragmentConfig("database.db4o");
config.forceBackupDelete(true);

Defragment.defrag(config);
```

DefragmentationConfigurationExamples.java: Delete the backup after the defragmentation process

Upgrade Database File

This option will upgrade first the database file format and then defragment it. You need to specify a folder where the temporary data for this process is stored.

```
DefragmentConfig config = new DefragmentConfig("database.db4o");
config.upgradeFile(System.getProperty("java.io.tmpdir"));

Defragment.defrag(config);
```

DefragmentationConfigurationExamples.java: Upgrade database version

Disable Read Only Mode

By default the back-up database is opened in read only mode. You can disable the read only mode if you want.

```
DefragmentConfig config = new DefragmentConfig("database.db4o");
config.readOnly(false);

Defragment.defrag(config);
```

DefragmentationConfigurationExamples.java: Disable readonly on backup

Configure Storage for Backup File

By default the defragmentation process uses the storage from the db4o configuration for both database files, the backup and the new defragmented database file. If you want to have different storage implementations you can specify a different storage for the backup-file. This way the new defragmented database-file uses the storage from the database-configuration and the backup-file uses the back-up storage.

```
DefragmentConfig config = new DefragmentConfig("database.db4o");
config.backupStorage(new FileStorage());

Defragment.defrag(config);
```

DefragmentationConfigurationExamples.java: Use a separate storage for the backup

Configure IDMapping

When you defragment a database each object is stored at a new location. To keep track which old id maps the which new location a IDMapper is uses. You can change the IDMapper or even implement your own.

By default the InMemoryIdMapping is uses, which is the fastest version, but consumes the most memory. As alterative there's the DatabaseIdMapping available, which stores mapping in a file and therefore uses less memory.

```
IdMapping mapping = new InMemoryIdMapping();
DefragmentConfig config = new DefragmentConfig("database.db4o", "database.db4o.back", mapping);

Defragment.defrag(config);
```

DefragmentationConfigurationExamples.java: Choose a id mapping system

Tracking Defragmentation Errors

You can pass a defragmentation listener to the defragmentation process. This listener will be notified when there's no object for an id in the database. This means that a object has a reference to a non-existing object. This happens when you delete objects which are still referenced by other objects.

```
DefragmentConfig config = new DefragmentConfig("database.db4o");
Defragment.defrag(config,new DefragmentListener() {
    @Override
    public   void notifyDefragmentInfo(DefragmentInfo defragmentInfo) {
        System.out.println(defragmentInfo);
    }
});
```

DefragmentationExample.java: Use a defragmentation listener

Enhancement Tools

Enhancement tools provide a convenient framework for application (jar, dll, exe) or classes modification to support db4o-specific functionality. Enhancement tools can work on a ready application or library and apply the improvements at load or build time.

The tools functionality is provided through bytecode instrumentation. This process inserts special, usually short, sequences of bytecode at designated points in your code. It is typically used for profiling or monitoring, however the range of use of bytecode instrumentation is not limited by this tasks: it can be applied anywhere where a specific functionality should be plugged into the ready built classes.

db4o Enhancement Tools currently have these cases for bytecodeinstrumentation:

- Transparent Activation
- Transparent Persistence
- Native Query Optimization

In transparent activation/persistence case, classes are required to implement Activatable interface to support transparent activation. In many cases you don't want to pollute your classes with some additional interface, or even won't be able to do so if you use a third party classes library. That's where bytecode instrumentation comes handy: Activatableinterface will be implemented on your existing classes by applying bytecode instrumentation. Another advantage of this approach - you can still work on your "clean" classes, just do not forget to run the instrumentation afterwards.

In the native query optimization case bytecode instrumentation is used as a more performant alternative to a run-time optimization. When an native query is optimized the user and compiler-friendly syntax of **NQ**[1] predicate is replaced with a query-processor-friendly code. Obviously, optimization process

[1] Native Query

can take some time, therefore it can be a good choice to use pre-instrumented classes, then to let the optimization be executed each time it is required by application.

The instrumentation can be run at build time, also known as static instrumentation. In this case a special build script calls runs the instrumentation on the classes before packaging them to jar, or on the jar itself. This is the fastest solution as no time is spent on bytecode instrumentation at runtime.

Another method is to use bytecode instrumentation at load time. In this case instrumenting information is inserted into the classes by a specific instrumenting classloader just before they are loaded into the VM.

There are different possibilities to integrate the enhancement tools into a project.

- More details about built time enhancements: See "Build Time Enhancement" on page 88
- It's possible to enhance the classes a runtime: See "Load Time Enhancement"

Transparent Persistence Enhancement Example

You can inject transparent persistence awareness in your persisted classes without modifying their original code. This is done by enhancing the class-files at build time.

Required jars

For transparent activation/persistence you need following dependencies at compile time. (see also the dependency overview)

- bloat-1.0.jar
- db4o-X.XX-instrumentation.jar
- db4o-X.XX-taj.jar
- db4o-X.XX-tools.jar

Enhance Persistent Classes

The first step is to enhance the persisted classes. One possibility is to introduce an Annotation to mark your persisted classes.

By the way, their are alternative way to select the enhanced classes. See here.

```
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Target(ElementType.TYPE)
public @interface TransparentPersisted {
}
```

TransparentPersisted.java: Annotation to mark persisted classes

This Annotationis then used to mark all persisted classes.

```
@TransparentPersisted
public      class Person {
```

Person.java: Mark your domain model with the annotations

The next step is to create a class filter which reports all classes which should be enhanced. There filter checks for the presence of the annotation.

```
public      final      class AnnotationFilter implements ClassFilter {

    public      boolean accept(Class<?> aClass) {
        if(null==aClass || aClass.equals(Object.class)){
            return      false;
        }
        return hasAnnotation(aClass)
                || accept(aClass.getSuperclass());
    }

    private      boolean hasAnnotation(Class<?> aClass) {
        // We compare by name, to be class-loader independent
        Annotation[] annotations = aClass.getAnnotations();
        for (Annotation annotation : annotations) {
            if(annotation.annotationType().getName()
                    .equals(TransparentPersisted.class.getName())){
                return      true;
            }
        }
        return      false;
    }

}
```

AnnotationFilter.java: Build a filter

Enhancing Classes Using Ant

This enhancement step injects the required bytecode into the domain classes to support transparent activation/persistence.

```
<target      name="enhance">
    <!-- Change these according to your project -->
    <property      name="target"      value="./target/classes/"/>
    <property      name="libraries"      value="./lib/"/>

    <path      id="project.classpath">
        <pathelement      path="${target}"/>
        <fileset      dir="${libraries}">
            <include      name="*.jar"/>
        </fileset>
    </path>

    <!-- We enhance with an additional Ant-run step. You can put this also in an extra file -->
    <typedef      resource="instrumentation-def.properties"
             classpathref="project.classpath"
             loaderRef="instrumentation.loader"/>

    <!-- We filter by our annotation -->
    <typedef      name="annotation-filter"
             classname="com.db4odoc.tp.enhancement.AnnotationFilter"
             classpathref="project.classpath"
             loaderRef="instrumentation.loader"/>

    <db4o-instrument      classTargetDir="${target}"
                  verbose="true">
        <classpath      refid="project.classpath"/>
        <sources      dir="${target}">
            <include      name="**/*.class"/>
        </sources>

        <transparent-activation-step>
            <annotation-filter/>
        </transparent-activation-step>
    </db4o-instrument>
</target>
```

enhance-with-annotation.xml: Ant target for enhancing your classes after building them

Configure Eclipse to Run Ant Target

You can configure Eclipse to run the Ant build with each compile step. Right click on your project and choose 'Properties'. Then switch to 'Builders' and add a new one. Choose the 'Ant Builder'. On the new window choose the build-file which contains the example-code. Switch to the 'Targets'-Tab. There choose the enhance-target for the 'Auto-Build'. Now the enhancer-task will be run by Eclipse automatically. The example project above is configured this way.

Enhancing Classes Using Maven

It's also possible to enhance with Maven by using the Ant plugin.

```
<plugin>
    <artifactId>maven-antrun-plugin</artifactId>
    <version>1.6</version>
    <dependencies>
        <!-- We need the db4o tooling for enhancing stuff -->
        <dependency>
            <groupId>com.db4o</groupId>
            <artifactId>db4o-tools-java5</artifactId>
            <version>8.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
    <executions>
        <execution>
            <phase>compile</phase>
            <configuration>
                <target>
                    <!-- We enhance with an additional Ant-run step. You can put this also in an extra file -->
                    <typedef      resource="instrumentation-def.properties"
                             classpathref="maven.compile.classpath"/>

                    <!-- We filter by our annotation -->
                    <typedef      name="annotation-filter"
                             classname="com.db4odoc.tp.enhancement.AnnotationFilter"
                             classpathref="maven.compile.classpath"/>

                    <db4o-instrument      classTargetDir="target/classes">
                        <classpath      refid="maven.compile.classpath"/>
                        <sources      dir="target/classes">
                            <include      name="**/*.class"/>
                        </sources>

                        <transparent-activation-step>
                            <annotation-filter/>
                        </transparent-activation-step>
                    </db4o-instrument>
                </target>
            </configuration>
            <goals>
                <goal>run</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

pom.xml: Enhance persisted classes during the build

Check Enhancement

You can check if the enhancement worked correctly by checking for the activation interface. Such a check should be part of your test-suite to ensure that everything works correctly.

```
if (!Activatable.class.isAssignableFrom(Person.class)) {
    throw      new AssertionError("Expect that the " + Person.class + " implements " + Activatable.class);
}
```

TransparentPersistence.java: Check for enhancement

Using Transparent Activation

Configure the transparent activation in order to use it.

```
final EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().add(new TransparentActivationSupport());
ObjectContainer container = Db4oEmbedded.openFile(configuration, DATABASE_FILE_NAME);
```

TransparentPersistence.java: Add transparent activation

After that transparent activation is working properly you can transverse along the object graph and don't have to worry about not activated objects:

```
Person person = queryByName(container, "Joanna the 10");
Person beginOfDynasty = person.getMother();

// With transparent activation enabled, you can navigate deeply
// nested object graphs. db4o will ensure that the objects
// are loaded from the database.
while (null != beginOfDynasty.getMother()) {
    beginOfDynasty = beginOfDynasty.getMother();
}
System.out.println(beginOfDynasty.getName());
```

TransparentPersistence.java: Activation just works

Using Transparent Persistence

Transparent persistence not only manages the activation, but also manages updating the objects. Configure transparent persistence in order to use it:

```
final EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().add(new TransparentPersistenceSupport(new DeactivatingRollbackStrategy()));
ObjectContainer container = Db4oEmbedded.openFile(configuration, DATABASE_FILE_NAME);
```

TransparentPersistence.java: Add transparent persistence

After that updated objects are automatically stored every time you commit.

```
Person person = queryByName(container, "Joanna the 10");
Person mother = person.getMother();
mother.setName("New Name");
// Just commit the transaction. All modified objects are stored
container.commit();
```

TransparentPersistence.java: Just update and commit. Transparent persistence manages all updates

Build Time Enhancement

You can enhance your classes at build time. The main use for the enhancement process is to add the activate interface implementation to the persisted classes. A load time enhancement example is available here.

This topic explains the individual elements of the ant task.

Dependencies.

For transparent activation/persistence you need following dependencies at compile time. (see also the dependency overview)

- bloat-1.0.jar
- db4o-X.XX-instrumentation.jar

- db4o-X.XX-taj.jar
- db4o-X.XX-tools.jar

The Example Script

This is an example script. Below it each part of it is campaigned and the alternatives to it.

```
<target   name="enhance">
    <!-- Change these according to your project -->
    <property   name="target"   value="./target/classes/"/>
    <property   name="libraries"   value="./lib/"/>

    <path   id="project.classpath">
        <pathelement   path="${target}"/>
        <fileset   dir="${libraries}">
            <include   name="*.jar"/>
        </fileset>
    </path>

    <!-- We enhance with an additional Ant-run step. You can put this also in an extra file -->
    <typedef   resource="instrumentation-def.properties"
             classpathref="project.classpath"
             loaderRef="instrumentation.loader"/>

    <!-- We filter by our annotation -->
    <typedef   name="annotation-filter"
             classname="com.db4odoc.tp.enhancement.AnnotationFilter"
             classpathref="project.classpath"
             loaderRef="instrumentation.loader"/>

    <db4o-instrument   classTargetDir="${target}"
                 verbose="true">
        <classpath   refid="project.classpath"/>
        <sources   dir="${target}">
            <include   name="**/*.class"/>
        </sources>

        <transparent-activation-step>
            <annotation-filter/>
        </transparent-activation-step>
    </db4o-instrument>
</target>
```

enhance-with-annotation.xml: Ant target for enhancing your classes after building them

The Tags

Tag/Property	Explanation
property target	Specifies to which classes the enhancement is applied.
property libraries	Specifies where the libraries are. This is used for setting up the classpath of the enhancement step. The enhancement should have access to all classes, otherwise class not found exceptions might occur.
path project.classpath	This represents the whole project classpath: Your code and all libraries. This classpath is passed to the enhancer step.
typedef instrumentation-def.properties	Loads the predefined tasks from the db4o jars. Make sure that all required db4o dependencies are on the classpath.
typedef annotation-filter	This defines a regular Java class as part of the Ant script. This way you can implement your own filter. The name attribute specifies how the task is named within the Ant script.
db4o-instrument	The db4o-instrument task, which can do all available instrumentations. In the attribute 'classTargetDir' you specify where to put the enhanced result.
db4o-instrument->classpath	Here we need to specify the classpath for the enhancer-step. Refer to classpath which includes the application code and the libraries.
db4o-instrument->sources	Specifies which class-files are enhanced
transparent-activation-step	This tag specifies that the transparent activation/persistence enhancement is applied. This tag expects that you specify one more filters to filter out the classes which need to be enhance. For the example filter take a look at the example.
native-query-step	You also can specify the native query enhancing step. Usually this is done at runtime, but can be done a compile time. Just add it the the db4o-instrument tag.

Simple Enhancer Alternative

As alternative quick way you can use the 'db4o-enhance'-task. There you just can specify the location of the class files which you want to enhance. It works best when you put all your persistent classes in a separate package or project.

```
<target   name="simpleEnhanceStep">
    <db4o-enhance   classtargetdir="${basedir}/bin"
                  jartargetdir="../libs/java/"
                  nq="false"   ta="true"
                  collections="true"
                  verbose="true">
        <classpath   refid="project.classpath"/>
        <sources   dir="${basedir}/bin">
                  <include   name="com/db4odoc/tp/**/*.class"/>
        </sources>
    </db4o-enhance>
</target>
```

enhance-with-annotation.xml: Simple enhancing step

Enhancing Classes Using Maven

As demonstrated in the example you can include a enhancing Ant script in your Maven build. All descriptions above applies. However you can use the built in Maven classpath. Just use the 'maven.compile.classpath' in the Ant script.

Callbacks

Callbacks, also known as events, allow you to be notified on certain database operations. This is useful to trigger additional operations during a database operation. There are two kind of callbacks. First you can certain methods to your objects. When the method matches certain signature it will be called by db4o. See "Object Callbacks" on page 91

Additionally you can register event listener to a object-container which will be called on certain operations. See "Event Registry API" on page 92

Object Callbacks

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

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

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

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

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

Some possible usecases for callback methods:

- Setting default values after refactorings.
- Checking object integrity before storing objects.
- Setting transient fields.
- Restoring connected state (of GUI, files, connections).
- Cascading activation.

- Cascading updates.
- Creating special indexes.

Event Registry API

You can register to events of the db4o-database. You can used these events to implement all kinds of additional functionality. Take a look a few example use-cases. See "Possible Usecases" on page 94

There's an event for each database operation. Most of the time there are two events for an operation. One is fired before the operation starts, the other when the operation ends.

Register to an Event

You can gain access to the events via a event registry. These three steps show how to register to events.

First obtain a EventRegistry-instance from the object container.

```
EventRegistry events = EventRegistryFactory.forObjectContainer(container);
```

EventRegistryExamples.java: Obtain the event-registry

Now you can register your event-handlers on the event registry.

```
events.committing().addListener(new EventListener4<CommitEventArgs>() {
    public     void onEvent(Event4<CommitEventArgs> source,
                        CommitEventArgs arguments) {
        handleCommitting(source,arguments);
    }
});
```

EventRegistryExamples.java: register for a event

Then implement your event handling.

```
private     static     void handleCommitting(Event4<CommitEventArgs> source,
                                    CommitEventArgs commitEventArgs) {
    // handle the event here
}
```

EventRegistryExamples.java: implement your event handling

Cancelable Events

Some events can cancel the operation. All events which have a CancellableObjectEventArgs-parameter can cancel the operation. When you cancel in a event, the operation won't be executed. For example:

```
EventRegistry events = EventRegistryFactory.forObjectContainer(container);
events.creating().addListener(new EventListener4<CancellableObjectEventArgs>() {
    public     void onEvent(Event4<CancellableObjectEventArgs> events,
                        CancellableObjectEventArgs eventArgs) {
        if(eventArgs.object() instanceof Person){
            Person p = (Person) eventArgs.object();
            if(p.getName().equals("Joe Junior")){
                eventArgs.cancel();
            }
        }
    }
});
```

EventRegistryExamples.java: Cancel store operation

Register Events On The Server

When you want to register for the events on the server, you should register it on the server-container.

```
ObjectServer server =
        Db4oClientServer.openServer(DATABASE_FILE_NAME, PORT_NUMBER);
EventRegistry eventsOnServer =
        EventRegistryFactory.forObjectContainer(server.ext().objectContainer());
```

EventRegistryExamples.java: register for events on the server

Commit-Events

Commit-events bring a collection of the added, updated and deleted object with it. You can iterate over these objects. The updated- and added-collections contain LazyObjectReferences, the deleted-event a FrozenObjectInfos. Note that you may cannot get deleted object-instance anymore, but only the meta-info. Furthermore the object doesn't need to be activated. So when you need to read information out if it, ensure that you've activated it first.

```
EventRegistry events = EventRegistryFactory.forObjectContainer(container);
events.committed().addListener(new EventListener4<CommitEventArgs>() {
    public     void onEvent(Event4<CommitEventArgs> events,
                        CommitEventArgs eventArgs) {
        for(Iterator4 it=eventArgs.added().iterator();it.moveNext();){
            LazyObjectReference reference = (LazyObjectReference) it.current();
            System.out.println("Added "+reference.getObject());
        }
        for(Iterator4 it=eventArgs.updated().iterator();it.moveNext();){
            LazyObjectReference reference = (LazyObjectReference) it.current();
            System.out.println("Updated "+reference.getObject());
        }
        for(Iterator4 it=eventArgs.deleted().iterator();it.moveNext();){
            FrozenObjectInfo deletedInfo = (FrozenObjectInfo) it.current();
            // the deleted info might doesn't contain the object anymore and
            // return the null.
            System.out.println("Deleted "+deletedInfo.getObject());
        }
    }
});
```

EventRegistryExamples.java: Commit-info

Pitfalls and Limitations

- All embedded clients-/session share the same event registry. So you need to register the events only on one.
- You cannot call recursively the event-producing operation within the event-handler. For example in the storing-event you cannot call store. In the committing-event you cannot call commit.
- In client-server mode, each client has it's own event-registry, and therefore only sees its own events. Except the committed-event. See "Events In Client Server-Mode" on page 168

Events Overview

This overview shows you all available events. Additionally it shows on which side the event is called in client-server-mode.

Event	Explanation	Cancellable	Client	Server
activating	Fired before a object is activated.	X	X	
activated	Fired after a object is activated.		X	
creating	Fired before a object is stored to first time.	X	X	
creating	Fired after a object is stored to first time.		X	
deleting	Fired before a object is deleted.	X		X
deleted	Fired after a object is deleted.			X
updating	Fired before a object is updated.	X	X	
updated	Fired after a object is updated.		X	
deactivating	Fired before a object is deactivated.	X	X	
deactivated	Fired after a object is deactivated.	X	X	
queryStarted	Fired when a query starts.		X	
queryFinished	Fired when a query has finished.		X	
committing	Fired before a commit.			X
committed	Fired after a commit.		X[1]	X
closing	Fired when the object is closed.		X	
classRegistered	Fired when a new class is stored/loaded.		X	
instantiated	Fired when a object is instantiated.		X	

Possible Usecases

There are many use cases for external callbacks, including:

- Cascaded deletes, updates.
- Referential integrity checks.
- Gathering statistics.
- Auto assigned fields .
- Assigning customary unique IDs for external referencing.

---

[1] This event is asynchronously distributed across all clients

- Delayed deletion (objects are marked for deletion when delete(object) is called and cleaned out of database in a later maintenance operation).
- Ensuring object fields uniqueness within the same class etc.

More Reading:

- Referential Integrity
- Autoincrement

Autoincrement

db4o does not deliver a field auto increment feature, which is common in **RDBMS**[1]. Normally you don't need any additional ids, since db4o manages objects by object-identity. However cases where you have disconnected objects, you need additional ids. One of then possibilities it to use auto incremented ids.

If your application logic requires this feature you can implement it using external callbacks. One of the possible solutions is presented below. Note that this example only works in embedded-mode.

This example assumes that all object which need an auto incremented id are subclasses of the IDHolder class. This class contains the auto-incremented id.

```
private  int id;
public  int getId() {
    return id;
}

public  void setId(int id) {
    this.id = id;
}
```

IDHolder.java: id holder

First create a class which keeps the state of the auto-increment numbers. For example a map which keeps the latest auto incremented id for each class.

```
private  static  class PersistedAutoIncrements {
 private  final Map<Class, Integer> currentHighestIds = new HashMap<Class, Integer>();

    public  int nextNumber(Class forClass) {
        Integer number = currentHighestIds.get(forClass);
        if (null == number) {
            number = 0;
        }
        number += 1;
        currentHighestIds.put(forClass,number);
        return number;
    }
}
```

AutoIncrement.java: persistent auto increment

Then create two methods, which are called later. One which returns the next auto-incremented id for a certain class. Another which stores the current state of the auto-increments.

---

[1] Relational Database Management System

```
public  synchronized  int getNextID(Class forClass) {
       PersistedAutoIncrements incrementState = ensureLoadedIncrements();
       return incrementState.nextNumber(forClass);
}

   public  synchronized  void storeState(){
       if(null!=state){
           container.ext().store(state,2);
       }
   }
```

AutoIncrement.java: getting the next id and storing state

The last part is to ensure that the existing auto-increments are loaded from the database. Or if not existing a new instance is created.

```
private PersistedAutoIncrements ensureLoadedIncrements() {
    if(null==state){
        state = loadOrCreateState();
    }
    return state;

}

private PersistedAutoIncrements loadOrCreateState() {
    ObjectSet<PersistedAutoIncrements> existingState = container.query(PersistedAutoIncrements.class);
    if(0==existingState.size()){
        return  new PersistedAutoIncrements();
    } else  if(1==existingState.size()){
        return existingState.get(0);
    } else{
        throw  new IllegalStateException("Cannot have more than one state stored in database");
    }
}
```

AutoIncrement.java: load the state from the database

Now it's time to use the callbacks. Every time when a new object is created, assign a new id. For this the creating-event is perfect. When commiting also make the auto increment-state persistent, to ensure that no id is used twice.

```
final AutoIncrement increment = new AutoIncrement(container);
EventRegistry eventRegistry = EventRegistryFactory.forObjectContainer(container);
eventRegistry.creating().addListener(new EventListener4<CancellableObjectEventArgs>() {
    public  void onEvent(Event4<CancellableObjectEventArgs> event4,
                         CancellableObjectEventArgs objectArgs) {
        if(objectArgs.object() instanceof IDHolder){
            IDHolder idHolder = (IDHolder) objectArgs.object();
            idHolder.setId(increment.getNextID(idHolder.getClass()));
        }
    }
});
eventRegistry.committing().addListener(new EventListener4<CommitEventArgs>() {
    public  void onEvent(Event4<CommitEventArgs> commitEventArgsEvent4,
                         CommitEventArgs commitEventArgs) {
        increment.storeState();
    }
});
```

AutoIncrementExample.java: use events to assign the ids

Last, don't forget to index the id-field. Otherwise looks-ups will be slow.

```
configuration.common().objectClass(IDHolder.class).objectField("id").indexed(true);
```

AutoIncrementExample.java: index the id-field

Referential Integrity

db4o does not have a built-in referential integrity checking mechanism. Luckily EventRegistry gives you access to all the necessary events to implement it. You will just need to trigger validation on create, update or delete and cancel the action if the integrity is going to be broken.

For example, if Car object is referencing Pilot and the referenced object should exist, this can be ensured with the following handler in deleting() event:

```
final EventRegistry events = EventRegistryFactory.forObjectContainer(container);
events.deleting().addListener(new EventListener4<CancellableObjectEventArgs>() {
    @Override
    public  void onEvent(Event4<CancellableObjectEventArgs> events,
                         CancellableObjectEventArgs eventArgs) {
        final Object toDelete = eventArgs.object();
        if(toDelete instanceof Pilot){
            final ObjectContainer container = eventArgs.objectContainer();
            final ObjectSet<Car> cars = container.query(new Predicate<Car>() {
                @Override
                public  boolean match(Car car) {
                    return car.getPilot() == toDelete;
                }
            });
            if(cars.size()>0){
                eventArgs.cancel();
            }
        }
    }
});
```

CallbackExamples.java: Referential integrity

You can also add handlers for creating() and updating() events for a Car object to make sure that the pilot field is not null.

Note, that in client/server mode deleting event is only raised on the server side, therefore the code above can't be used and will throw an exception.

Committed Event Example

Committed callbacks can be used in various scenarios:

- Backup on commit.
- Client database synchronization.

This example shows you how to refresh objects on a client on commits.

When several clients are working on the same objects it is possible that the data will be outdated on a client. You can use the committed-event refresh object on each commit.

When a client commit will trigger a committed event on all clients. In order to refresh the object, register for the committed event. In the commit-event-handler, refresh the object which have been modified.

```
EventRegistry events = EventRegistryFactory.forObjectContainer(container);
events.committed().addListener(new EventListener4<CommitEventArgs>() {
    public   void onEvent(Event4<CommitEventArgs> commitEvent, CommitEventArgs commitEventArgs) {
        for(Iterator4 it = commitEventArgs.updated().iterator();it.moveNext();){
            LazyObjectReference reference = (LazyObjectReference) it.current();
            Object obj = reference.getObject();
            commitEventArgs.objectContainer().ext().refresh(obj,1);
        }
    }
});
```

RefreshingObjects.java: On the updated-event we refresh the objects

You can register such a event-handler for each client. The committed event is transferred to each client. Note that this requires a lot of network-traffic to notify all clients and transfer the changes.

When working with committed events you should remember that the listener is called on a separate thread, which needs to be synchronized with the rest of the application.

Commit-Time Callbacks

Commit-time callbacks allow a user to add some specific behavior just before and just after a transaction is committed.

Typical use-cases for commit-time callbacks:

- Add constraint-violation checking before commit.
- Check application-specific conditions before commit is done.
- Start synchronization or backup after commit.
- Notify other clients/applications about successful/unsuccessful commit.

Commit-time callbacks can be triggered by the following 2 events:

- **Committing**: Event subscribers are notified before the container starts any meaningful commit work and are allowed to cancel the entire operation by throwing an exception; the object container instance is completely blocked while subscribers are being notified which is both a blessing

because subscribers can count on a stable and safe environment and a curse because it prevents any parallelism with the container;

- **Committed**: Event subscribers are notified in a separate thread after the container has completely finished the commit operation; exceptions if any will be ignored.

Type Handling

db4o tries to be simple an easy to use. A big part of this is to transparently store any object without any complex mapping or configuration. Storing an object correctly is a complex process and heavily depends on the type of the object. db4o has different storing strategies for different types.

Supported Types

For most types the regular db4o type handling is sufficient. You can store objects composed of following types:

- All basic numeric types like ints, bytes, floats, doubles etc (and their wrapper types).
- Strings
- Arrays
- Dates
- Enums
- Basic collections like ArrayLists and HashMaps.
- UUIDs and BigMath types after adding the appropriate configuration item.
- Any class type which you've build with yourself with the types above.

Types Which Should Not Be Stored

These kind of types shouldn't be stored without any special treatment:

- Any type from the JDK / .NET Framework, library or framework. As long as you don't have control over the implementation of a type you cannot be sure how and what exactly is stored.
- This is also true for collections with sophisticated behavior like ConcurrentMaps or WeakHashmaps.

Control Persistence

If you need more control on how a object is persisted, then you can do that with object translators and type handlers.

UUID Support

When you are using the Java UUIDs you should turn on the support those. When you this on UUIDs will be handles as value objects like strings and numbers. This has several advantages:

- Queries for UUIDs fields a much more efficient.
- You can properly index UUID fields for faster queries.
- UUIDs are handles 100% correctly.

Add the UUID support like this:

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().add(new UuidSupport());
```

ConfigurationItemsExamples.java: Add proper support for UUIDs

BigMath

If you are dealing with very big numbers, you might be using the BigDecimal or BigInteger classes. These classes are specially designed to allow computations with of arbitrary precision. Internally the values are stored in byte arrays for both types. Now, thinking about it - it should not be a problem for db4o to store such values, as it is just a matter of storing a class with the actual value in a byte array field. However, a deeper consideration uncovers the following problems:

- BigInteger/BigDecimal representation is different in different Java versions, which can cause problems re-instantiating the objects from a database created with a different Java version.
- BigDecimal relies on transient field setup in the constructor, which means that constructor use is compulsory
- db4o would store instances of these classes as full object graphs: A BigDecimal contains a BigInteger which contains a byte array, plus some other fields. This graph would faithfully be persisted into the database file and it would have to be read and reconstructed on access - activation depth applies.
- Querying and indexing will essentially be broken due to the above limitations.

In order to solve the above mentioned problems db4o implements special type handlers for BigInteger and BigDecimal, which allow to treat them as normal value types.So that BigDecimal and BigInteger behave the same way as long and double. These typehandlers are implemented in db4o optional jar and should be added to the configuration before opening the file with the following method:

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().add(new BigMathSupport());
```

ConfigurationItemsExamples.java: Add support for BigDecimal and BigInteger

Object Construction

How does db4o construct its objects when it loads it from the database?

Bypassing the Constructor

By default db4o creates the object and bypasses any constructor. It does that by using the available methods on the platform. After that it uses reflection of fill the fields of the object with the content. After that object creating process is finished and the object is ready to be used.

Using the Constructor

Optionally db4o can also create objects using the constructor. In that case db4o tries to find a constructor which is can use. db4o starts to tries out all constructors, starting with the constructor with the least arguments. As soon a constructor can be called without throwing an exception, it is used. When a constructor has arguments db4o calls it with default values for all arguments. For reference-types that's null, for number zero and for other values types the appropriate default. When no constructor can be found, db4o will throw an appropriate exception.

Special Construction

The last possibility for db4o to construct objects is by special handlers. Either with translators or type handlers.

Blobs

In some cases user has to deal with large binary objects (BLOBs) such as images, video, music, which should be stored in a structured way, and retrieved/queried easily. There are several challenges associated with this task:

- Storage location.
- Loading into Memory.
- Querying interface.
- Objects' modification.
- Information backup.
- Client/Server processing.

db4o provides you with a flexibility of using 2 different solutions for this case:

1. The db4o blob-type.
2. Byte[] arrays stored inside the database file

These two solutions' main features in comparison are represented below:

Blob

1. Every Blob gets it's own file.
2. Special code is necessary to store and load .
3. No concerns about activation depth.

byte[] array

1. Data in the same file
2. Transparent handling without special concerns.
3. Control over activation depth may be necessary

Storing data in a byte[] array works just as storing usual objects, but this method is not always applicable/desirable. First of all, the size of the db4o file can grow over the limit (256 GB) due to the BLOB data added. Secondly, object activation and client/server transferring logic can be an additional load for your application.

Db4o Blob Implementation

Built-in db4o blob type helps you to get rid of the problems of byte[] array, though it has its own drawbacks.

1. Every Blob gets it's own file:
   + Main database file stays a lot smaller.
   + Backups are possible over individual files.
   +The BLOBs are accessible without db4o.
   - Multiple files need to be managed .

1. Special code is necessary to store and load.

   - It is more difficult to move objects between db4o database files.

2. No concerns about activation depth

   + Big objects won't be loaded into memory as part of the activation process.

Configuration

First, the blob storage location should be defined. If that value is not defined, db4o will use the default folder "blobs" in user directory.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
try {
    configuration.file().blobPath("myBlobDirectory");
} catch (IOException e) {
    e.printStackTrace();
}
```

FileConfiguration.java: Configure the blob-path

Using The db4o-Blob

There are two important operations on the blob type. The first one write a file into the db4o blob:

```
blob.readFrom(fileToStore);
```

BlobStorage.java: Store the file as a db4o-blob

And then there's the operation which .loads the db4o blob into a new file.

```
blob.writeTo(file);
```

BlobStorage.java: Load a blob from a db4o-blob

The db4o blob-type has a status attached to it. This status tells you if the blob-file all ready has been transferred:

```
while (blob.getStatus() > Status.COMPLETED){
    try {
        Thread.sleep(50);
    } catch (InterruptedException ex) {
        Thread.currentThread().interrupt();
    }
}
```

BlobStorage.java: wait until the operation is done

Collections

Unfortunately this implementation is not very efficient for searches/updates of a certain value in a collection, as the whole collection needs to be instantiated to access any of its elements.

db4o brings some special collections with it. There are collections which support transparent persistence. See "Collections" on page 58

When you have a need for a huge collection, you might run into some performance bottleneck, since collections are always stored and retrieved as complete unit. You can use db4o special big-set to improve performance. See "Big Set" on page 103

You might wonder what is better to use, collection or arrays. Most of times it doesn't matter. See "Collections or Arrays?" on page 103

Collections or Arrays?

If you are planning an application with db4o, you may be asking yourself, what is better to use: collections or arrays? From db4o's point of view it doesn't make a big difference. You can base your solution on the overall system design, entrusting db4o to handle the internals efficiently in both cases.

However you need to consider that collections are more flexible and convenient than arrays for most operations. Additionally, collections can be **TA**[1]/**TP**[2] aware by using db4o-collections, while arrays are always fully activated. See "Collections" on page 58

Big Set

When you need to store large sets, you can use db4o's big set. This big-set operates directly on top of B-trees, which are also used for indexes. The big-set doesn't need to activate all items to perform its operations. For example when you check if the set already contains a member, the big-set can do that without activating all its items. Especially lookup-operation like contains perform much faster with a big set.

Not that currently the big set implementation only works in embedded-mode, but not in client-server mode.

You can create a new big-set with the CollectionFactory:

```
Set<Person> citizen= CollectionFactory.forObjectContainer(container).newBigSet();
// now you can use the big-set like a normal set:
citizen.add(new Person("Citizen Kane"));
```

BigSetExample.java: Crate a big-set instance

After that, the big-set behaves just like an ordinary set. Except that the big-set used the object-identity instead of the object-equality to compare the items. So when you add a equal object with a different identity, it will be added to the set. You can add, remove and iterate over the items or check if an item is already in the set. The items will be loaded and activated on demand, for example when you iterate over the set.

```
Person aCitizen = city.citizen().iterator().next();
System.out.println("The big-set uses the identity, not equality of an object");
System.out.println("Therefore it .contains() on the same person-object is "
        +city.citizen().contains(aCitizen));
Person equalPerson = new Person(aCitizen.getName());
System.out.println("Therefore it .contains() on a equal person-object is "
        +city.citizen().contains(equalPerson));
```

BigSetExample.java: Note that the big-set compares by identity, not by equality

Static Fields And Enums

How to deal with static fields and enumerations? Do they belong to your application code or to the database? Let's have a look.

---

[1] Transparent Activation
[2] Transparent Persistence

More Reading:

- Static fields API

Storing Static Fields

By default db4o does not persist static fields. This is not necessary because static values are set for a class, not for an object. However you can set up db4o to store static fields if you want to implement constants or enumeration: See "Persist Static Fields" on page 160

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).persistStaticFieldValues();
```

ObjectConfigurationExamples.java: Persist also the static fields

When this setting is enabled, all non-primitive-typed static fields are stored the first time an instance of the class is stored. The values are restored every time a database file is opened afterwards, after the class meta information is loaded for this class (when the class objects are retrieved with a query, for example).

Use this option with caution. This option means that static fields are stored in the database. When you change the value of this field, you need to store it explicitly again. Furthermore, db4o will replace the static value at runtime, which can lead to very subtle bugs in your application.

This option does not have any effect on primitive types like ints, longs, floats etc.

Enum Class Use case

One of the use-cases is when you have an enumeration class which you want to store. In fact, Java enums implement this enumeration class idiom and db4o persist the static fields or all enums. For example we have a color class, which also has some static colors.

```
public    final    class Color {
    public    final    static Color BLACK = new Color(0,0,0);
    public    final    static Color WHITE = new Color(255,255,255);
    public    final    static Color RED = new Color(255,0,0);
    public    final    static Color GREEN = new Color(0,255,0);
    public    final    static Color BLUE = new Color(0,0,255);

    private    final    int red;
    private    final    int green;
    private    final    int blue;

    private Color(int red, int green, int blue) {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }

    public    int getRed() {
        return red;
    }

    public    int getGreen() {
        return green;
    }

    public    int getBlue() {
        return blue;
    }

    @Override
    public    boolean equals(Object o) {
        if (this == o) return    true;
        if (o == null || getClass() != o.getClass()) return    false;

        Color color = (Color) o;

        if (blue != color.blue) return    false;
        if (green != color.green) return    false;
        if (red != color.red) return    false;

        return    true;
    }

    @Override
    public    int hashCode() {
        int result = red;
        result = 31 * result + green;
        result = 31 * result + blue;
        return result;
    }

    @Override
    public String toString() {
        return    "Color{" +
                "red=" + red +
                ", green=" + green +
                ", blue=" + blue +
                '}';
    }
```

```
}
```

Color.java: Class as enumeration

We want to ensure reference equality on colors so that you easily can check for a certain color. But when we load the colors from the database you get new instances and not the same instance as in the static field. This means that comparing the references will fail.

```
// When you enable persist static field values, you can compare by reference
// because db4o stores the static field
if(car.getColor()== Color.BLACK){
    System.out.println("Black cars are boring");
} else    if(car.getColor()== Color.RED){
    System.out.println("Fire engine?");
}
```

StoringStaticFields.java: Compare by reference

When you enable the persist static fields option, the static fields are stored. This means that the object referenced in the static fields are loaded from the database and therefore the same instance. And the comparing the references works again.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Color.class).persistStaticFieldValues();
```

StoringStaticFields.java: Enable storing static fields for our color class

Translators

Sometimes objects cannot be stored in db4o. For example because the objects have references to other parts of the system other where never intended to be stored. This is especially for objects from third party libraries.

Now the db4o object translators is a simple mechanism which allows you to manually handle the persistence of an object. There are two important interfaces for this. The ObjectTranslator-interface and the ObjectConstructor. The first interface lets you take the control over storing and activation of the object. The second interface also allows you to control the instantiation of the object.

If you register a ObjectTranslator-instance for a certain type, it will also be applied to sub-types. This doesn't apply to ObjectConstructor-instances, because those need to create the right instance and therefore cannot handle subtypes.

Creating a Translator

First you need to create a translator for your types. Let's take a look at this example. There three distinct tasks a translator has to do. The first task is to convert the not storable object into another, storable object. Another task of the translator is to take care of the activation of the object. There it need to copy the values from the stored object into a instance of the original type. The third task it to create instances of the object. There you create a instance of the original type. And for some types you maybe also read the data at this point in time.

As an alternative you can use the predefined translators.

```
class ExampleTranslator implements ObjectConstructor {

    // This is called to store the object
    public Object onStore(ObjectContainer objectContainer, Object objToStore) {
        NonStorableType notStorable = (NonStorableType) objToStore;
        return notStorable.getData();
    }

    // This is called when the object is activated
    public   void onActivate(ObjectContainer objectContainer, Object targetObject, Object storedObject) {
        NonStorableType notStorable = (NonStorableType) targetObject;
        notStorable.setData((String)storedObject);
    }

    // Tell db4o which type we use to store the data
    public Class storedClass() {
        return String.class;
    }

    // This method is called when a new instance is needed
    public Object onInstantiate(ObjectContainer objectContainer, Object storedObject) {
        return   new NonStorableType("");
    }
}
```

ExampleTranslator.java: An example translator

Registering a Translator

After that you can register the translator for you're type. If you register a ObjectTranslator-instance it will also be applied to the sub-types. However a ObjectConstructor-instance is only applied for the specific type.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(NonStorableType.class).translate(new ExampleTranslator());
ObjectContainer container = Db4oEmbedded.openFile(configuration, "database.db4o");
```

TranslatorExample.java: Register type translator for the NonStorableType-class

Using The Translator

After that you can store and use the not storable objects like any other persistent objects. db4o will call the translator for each instance when required in order to store the object correctly.

```
container.store(new NonStorableType("TestData"));
```

TranslatorExample.java: Store the non storable type

```
NonStorableType instance = container.query(NonStorableType.class).get(0);
```

TranslatorExample.java: Load the non storable type

Limitations

The object translator mechanism is great for types which couldn't be stored otherwise. However there are serious limitations.

- Queries into the members of a object which was stored with a object translator are extremely slow. The reason is that the object first need to be loaded and instantiated with the translator in order to run the query on it.
- You cannot index types which are translated.

Built-In Translators

db4o supplies some build-in translators, which can be used in general cases. You can use them for your classes if they are not storable or nedd special treatment.

- TTransient: Doesn't store the object at all. Usable when you don't want to store instances of certain type. If makes all instances of that type transient.
- TSerializable: Uses the built in serialisation mechanism to store this object.

There are other built in translators, which are not intended be used directly. Instead they are used by db4o internally. Nevertheless you can use them as example implementation. Look for all classes which implement the ObjectTranslator-interface

TypeHandlers

TypeHandlers are at the lowest level of db4o. A typehandler is responsible to turn a object or parts of an object into a byte stream. db4o brings are whole set of type handlers for different types of object to store them.

For special cases it can make sense to write your own type handler. This allows you to control the persistence details down to the byte level. However keep in mind that type handlers have following problems and limitations:

- There is no versioning support: If you need to change the serialisation scheme of the type your are on your own.
- The typehandler-API is mostly undocumented and not as stable as other db4o APIs.
- Elaborate type handlers need deep knowledge of undocumented, internal implementation details of db4o.
- Mistakes in typehandlers can lead to cryptic error messages and database corruption.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().registerTypeHandler(
        new SingleClassTypeHandlerPredicate(StringBuilder.class), new StringBuilderHandler());
```

TypeHandlerExample.java: Register type handler

Note that type handler are a low level API which doesn't protect you from any mistakes. If you make a mistake in a typehandler you can lose data.

Type handler functionality is best explained on a working example.

Custom Typehandler Example

For a custom typehandler example we will try to write a very simple typehandler for the StringBuilder class. We want to handle a StringBuilder as a value type, therefore we implement the ValueTypeHandler interface. Not that there's a whole collection of interfaces for typehandlers. Take a look at the TypeHandler4 type hierarchy.

To keep it simple we will skip information required for indexing - please look at IndexableTypeHandler in db4o sources to get more information on how to handle indexes.

The first thing should be the write method, which determines how the object is persisted:

```
@Override
public  void write(WriteContext writeContext, Object o) {
    StringBuilder builder = (StringBuilder) o;
    String str = builder.toString();
    final  byte[] bytes = str.getBytes(CHAR_SET);
    writeContext.writeInt(bytes.length);
    writeContext.writeBytes(bytes);
}
```

StringBuilderHandler.java: Write the StringBuilder

As you can see from the code above, there are 3 steps:

1. Get the buffer from WriteContext/I WriteContext
2. Convert the string-content to a byte-array using the UTF8 encoding.
3. Write the length of the resulted byte-array.
4. Write the byte array of the string.

Next step is to read the stored object. It is just opposite to the write method:

```
@Override
    public Object read(ReadContext readContext) {
        final  int length = readContext.readInt();
        byte[] data = new  byte[length];
        readContext.readBytes(data);
        return  new StringBuilder(new String(data,CHAR_SET));
    }
}
```

StringBuilderHandler.java: Read the StringBuilder

Delete is simple - we just reposition the buffer offset to the end of the slot:

```
@Override
public  void delete(DeleteContext deleteContext) throws Db4oIOException {
    skipData(deleteContext);
}

private  void skipData(ReadBuffer deleteContext) {
    int numBytes = deleteContext.readInt();
    deleteContext.seek(deleteContext.offset()+ numBytes);
}
```

StringBuilderHandler.java: Delete the content

The last method left: #defragment. This one only moves the offset to the beginning of the object data, i.e. skips Id and size information (to be compatible to older versions):

```
@Override
public  void defragment(DefragmentContext defragmentContext) {
    skipData(defragmentContext);
}
```

StringBuilderHandler.java: Defragment the content

Now to use this type handler we need to configure db4o. To register a typehandler you have to provide a predicate which decides if a type is handled by the typehandler and the typehandler itself.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().registerTypeHandler(
        new SingleClassTypeHandlerPredicate(StringBuilder.class), new StringBuilderHandler());
```

TypeHandlerExample.java: Register type handler

After that all string builders are handled by you're type handler.

Refactoring and Schema Evolution

Application design is a volatile thing: it changes from version to version, from one customer implementation to another. The database changes together with the application. For relational databases this process is called Schema Evolution, for object databases the term Refactoring is used as more appropriate.

Object database refactoring changes the shape of classes stored on the disk. The main challenge here is to preserve old object information and make it usable with the new classes' design.

Simple cases like adding or removing a field and changing interfaces are handled automatically. See "Automatic Refactoring" on page 110

For renaming classes and fields you can use the renaming API. See "Renaming API" on page 110.

You can change the type of a field any time. However db4o won't migrate the data to the new type. You need to do this explicitly. See "Field Type Change" on page 111 And there are some small limitations when refactoring the field-type. See "Field Refactoring Limitation" on page 112

Unfortunatly db4o doesn't support changing the inheritance hierarchy. See "Refactoring Class Hierarchy" on page 113

Automatic Refactoring

In simple cases db4o handles schema changes automatically:

- When you **add** a new field, db4o automatically starts storing the new data. Older instances of your stored class have the default value in the new field.
- When you **remove** a field, db4o ignores the stored value for that field. The stored value is not removed from the database until you update the object or run a defragmentation. Meanwhile the old values are still accessible with the StoredClass/StoredField API.
- You can **add an interface** to a class any time. The db4o operations are not affected by interfaces.

Renaming API

db4o provides a special API to move classes between packages, rename classes or fields.

Rename a Class

Use the configuration API to rename a class.You need to rename the class before you open the database.

```
configuration.common().objectClass("com.db4odoc.strategies.refactoring.PersonOld")
        .rename("com.db4odoc.strategies.refactoring.PersonNew");
```

RefactoringExamples.java: Rename a class

Rename a Field

Use the configuration API to rename a field. You need to rename the field before you open the database.

```
configuration.common().objectClass("com.db4odoc.strategies.refactoring.PersonOld")
        .objectField("name").rename("sirname");
```

RefactoringExamples.java: Rename field

Rename Step by Step

The safe order of actions for renaming is:

1. Backup your database.
2. Close all open object containers if any.
3. Rename classes or fields in your application.
4. Add the renaming call to on the configuration before opening the database.
5. Pass the configuration with the rename-information to the object container factory. Open the database and you're ready to work with the renamed classes or fields.

Field Type Change

db4o's default policy is to never do any damage to stored data. When you change the type of a field, db4o will not update the data in this field. Instead db4o internally creates a new field of the same name, but with the new type. For existing object, the values of the old typed field are still present, but hidden. Of course you can access the old data. When you want to convert the content from the old field type to the new field type, you have to do it yourself.

You can use the stored-classes API to retrieve the data of the old typed field. An example: We decide that we want to refactor the id-field from a simple int to a special identity class. First we change the field-type:

```
public Identity id = Identity.newId();
//  was an int previously:
//    public int id = new Random().nextInt();
```

Person.java: change type of field

After than read the old value from the old field-type and convert it to the new type:

```
ObjectContainer container = Db4oEmbedded.openFile( "database.db4o");
try{
    // first get all objects which should be updated
    ObjectSet<Person> persons = container.query(Person.class);
    for (Person person : persons) {
        // get the database-metadata about this object-type
        StoredClass dbClass = container.ext().storedClass(person);
        // get the old field which was an int-type
        StoredField oldField = dbClass.storedField("id", int.class);
        if(null!=oldField){
            // Access the old data and copy it to the new field!
            Object oldValue = oldField.get(person);
            if(null!=oldValue){
                person.id = new Identity((Integer)oldValue);
                container.store(person);
            }
        }
    }
} finally {
    container.close();
}
```

RefactoringExamples.java: copying the data from the old field type to the new one

db4o's approach gives you the maximum flexibility for refactoring field types. You can handle the convertion with regular code, which means it can be as complex as needed. Furthermore you can decide when you convert the values. You can update all objects in one operation, you can dynamically update and covert when you access a object or even decide not to convert the old values.

Field Refactoring Limitation

For most cases changing the field type isn't an issue. db4o keeps the old values around and you can access the old values without issues. See "Field Type Change" on page 111

However there's one limitation to this mechanism. You cannot change the type of a field to its array-type and vice versa. This only applies if it's the same array-type. For example:

- You cannot change a string field to a string array field and vice versa.
- You can change a string field to an int-, object-, etc array. Every type is possible except a string-array.
- You can change a string-array to an int-, object etc. Every type is possible except a string.

Refactoring To An Array-Field Step by Step

When you change the type of a field to its array-type equivalent, you can do this only by copying the old data to a new class. In this example we have a Person class which has its name in a string field. Now we want to change that to a string array to support multiple names.

1. Create a copy of the Person class with a new name.
2. Do the refactoring on the new Person class
3. Query for old instances of the old Person class and copy the values over to the new class.

```
List<PersonOld> oldPersons = container.query(PersonOld.class);
for (PersonOld old : oldPersons) {
    PersonNew newPerson = new PersonNew();
    newPerson.setName(new String[]{old.getName()});
    container.store(newPerson);
    container.delete(old);
}
```

ChangeArrayType.java: Copy the string-field to the new string-array field

Note that this example doesn't change existing references from the old instances to the new ones. You need to do this manually as well.

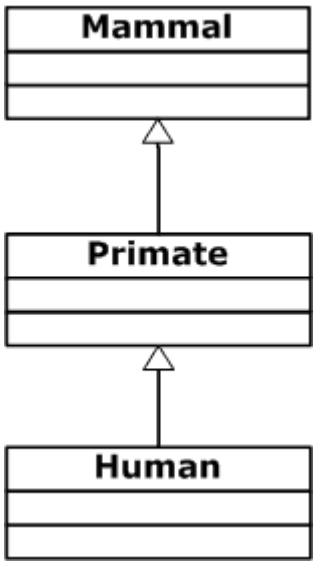
Refactoring Class Hierarchy

db4o does not directly support the following two refactorings:

- Inserting classes into an inheritance hierarchy.
- Removing classes from inheritance hierarchy.

Note that interfaces don't influence the inheritance-hierarchy and can be added and removed at any time.

For example we've following classes:



In this example you cannot introduce a 'Animal' class above the 'Mammal' or add another class between 'Mammal' and 'Primate'. Also you shouldn't remove an class form the inheritance-hierarchy.

Currently the only possible solution for this refactoring is this.

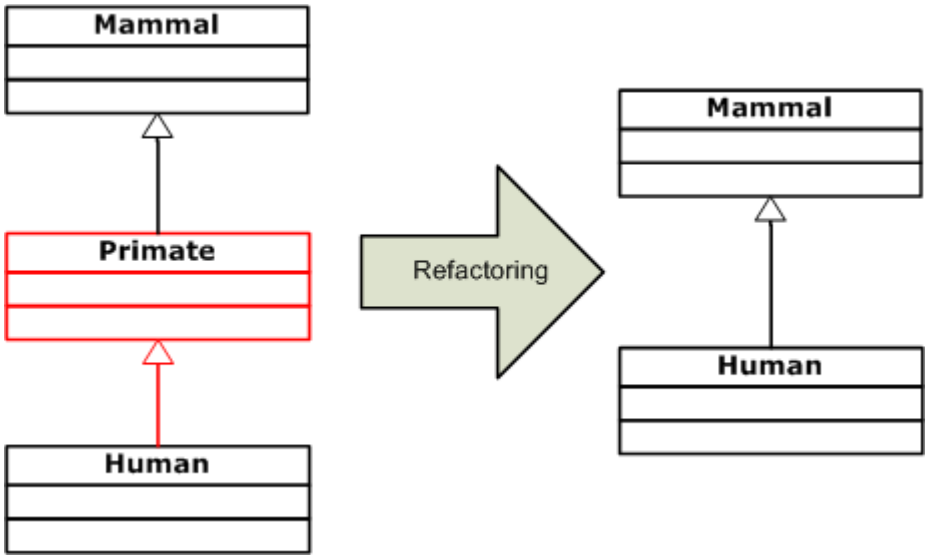
1. Create the new hierarchy with different names, preferably in a new package
2. Copy all values from the old classes to the new classes.
3. Redirect all references from existing objects to the new classes.

Take a look at the example to how to add a class into the hierarchy. See "Inserting Class Into A Hierarchy" on page 115.

Or how you can remove a class from the inheritance hierarchy. See "Removing Class From A Hierarchy" on page 114

Removing Class From A Hierarchy

In this example we have a Human class which inherits from the Primate class. Now we want to remove the Primate class and let the Human class inherit directly from the Mammal class.



Unfortunately db4o doesn't support this kind of refactoring. We need to use a work-around. Basically we create a copy of the Human class with the new Inheritance-hierarchy and the copy the existing data over.

Step by Step

1. Create a copy of the Human class, for example HumanNew!
2. Change the inheritance of the HumanNew class to inherit directly from the Mammal class.
3. After that, load all existing Human-instances, copy the values over to HumanNew-instances. Store the HumanNew-instance and delete the old Human-instances

Now the objects have the new inheritance hierarchy. You can delete the old Human class.

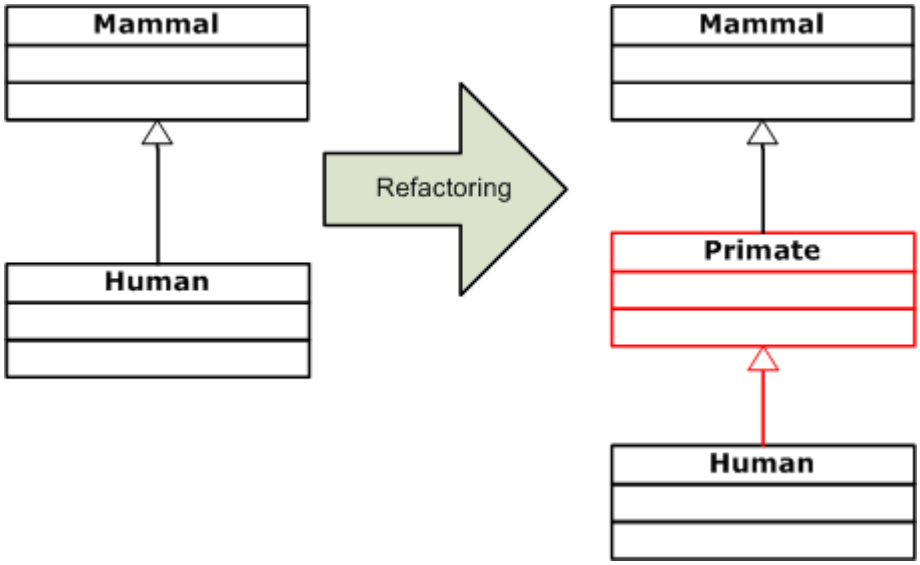
```
ObjectSet<Human> allMammals = container.query(Human.class);
for (Human oldHuman : allMammals) {
    HumanNew newHuman = new HumanNew("");
    newHuman.setBodyTemperature(oldHuman.getBodyTemperature());
    newHuman.setIq(oldHuman.getIq());
    newHuman.setName(oldHuman.getName());

    container.store(newHuman);
    container.delete(oldHuman);
}
```

RemoveClassFromHierarchy.java: copy the data from the old type to the new one

Note that this example doesn't change existing references from the old instances to the new ones. You need to do this manually as well.

Inserting Class Into A Hierarchy

In this example we have a Human class witch inherits from the Mammal class. Now we want to introduce a new Primate class and let the Human class inherit from it.



Unfortunately db4o doesn't support this kind of refactoring. We need to use a work-around. Basically we create a copy of the Human class with the new Inheritance-hierarchy and the copy the existing data over.

Step by Step

1. Create the new Primate class
2. Create a copy of the Human class, for example HumanNew!
3. Change the HumanNew class to inherit from the new Primate class instead of the Mammal class.
4. After that, load all existing Human-instances, copy the values over to HumanNew instances. Store the HumanNew instance and delete the old Human-instances

Now the objects have the new inheritance hierarchy. You can delete the old Human class.

```
ObjectSet<Human> allMammals = container.query(Human.class);
for (Human oldHuman : allMammals) {
    HumanNew newHuman = new HumanNew("");
    newHuman.setBodyTemperature(oldHuman.getBodyTemperature());
    newHuman.setIq(oldHuman.getIq());
    newHuman.setName(oldHuman.getName());

    container.store(newHuman);
    container.delete(oldHuman);
}
```

AddClassToHierarchy.java: copy the data from the old type to the new one

Note that this example doesn't change existing references from the old instances to the new ones. You need to do this manually as well.

Exception-Handling

A part of the db4o operations is handling possible exceptions. There are two fundamental different exception-types for db4o. The recoverable exceptions and the fatal exceptions.

Recoverable Exceptions

The recoverable exceptions are all exceptions which don't endanger the consistency of the database. For example if a event callback throws an exception. After a recoverable exception you can continue to work with the object container.

Typical recoverable exceptions are:

- Exceptions in callbacks.
- Passing invalid ids to the object container.
- Unsupported schema-changes.
- Constraint-violations.

Fatal Exceptions

A fatal exception will immediately shut down the object container without committing anything. This tries to protect the database from damaging itself. Any exception which happens in the db4o core and is not expected and handled is considered as a fatal exception. Because when an exception happens in the db4o-core, it could cause a invalid state in the db4o-core and then cause further errors and lead to database corruption. That's why the policy it the stop immediately any operation after a fatal exception.

Typical fatal recoverable exceptions are:

- Exceptions related to runtime, like OutOfMemory-exceptions
- Exceptions related to corrupted database-files.

Handle-Exceptions

Take a look a the list of the most common db4o related exceptions. See "Exception Types" on page 116 Handling db4o exceptions is nothing special and complies with regular exception handling.See "How To Work With db4o Exceptions" on page 117

Exception Types

Using db4o you will have to deal with db4o-specific exceptions and system exceptions thrown directly out of db4o.

db4o-specific exceptions are unchecked exceptions, which all inherit from a single root class Db4oException.

In Java Unchecked exceptions are inherited from RuntimeExceptions class, while in .NET all exceptions are unchecked.

db4o-exceptions are chained; you can get the cause of the exception using:

Java:

```
db4oException.getCause();
```

In order to see all db4o-specific exceptions you can examine the hierarchy of Db4oException class. Currently the following exceptions are available:

Db4oException - db4o exception wrapper: Exceptions occurring during internal processing will be proliferated to the client calling code encapsulated in an exception of this type.

BackupInProgressException - An exception to be thrown when another process is already busy with the backup.

ConstraintViolationException - Base class for all constraint violations.

UniqueFieldValueConstraintViolationException - An exception which will be thrown when the unique constrain is violated.

DatabaseClosedException - An exception to be thrown when the database was closed or failed to open.

DatabaseFileLockedException - This exception is thrown during any of db4o open calls if the database file is locked by another process.

DatabaseMaximumSizeReachedException - This exception is thrown if the database size is bigger than possible. See "Increasing the Maximum Database File Size" on page 227

DatabaseReadOnlyException - This exception is thrown when a write operation was attempted on a database in read-only mode.

GlobalOnlyConfigException - This exception is thrown when you try to change a setting on a open object container, but this setting cannot be changed at runtime.

IncompatibleFileFormatException - An exception to be thrown when an open operation is attempted on a file(database), which format is incompatible with the current version of db4o.

InvalidIDException - an exception to be thrown when an ID format supplied to #bind or #getById methods is incorrect.

InvalidPasswordException - This exception is thrown when a client tries to connect to a server with the wrong password.

EventException - This exception is thrown when a exception is thrown in a event callback.

OldFormatException - An exception to be thrown when an old file format was detected and the file could not be open.

ReflectException - An exception to be thrown when a class can not be stored or instantiated by current db4o reflector.

ReplicationConflictException - an exception to be thrown when a conflict occurs and no ReplicationEventListener is specified.

How To Work With db4o Exceptions

Appropriate exception handling will help you to create easy to support systems, saving your time and efforts in the future. The following hints identify important places for exception handling. Take also a look at the list of common db4o exceptions.

1. Opening a database file can throw a DatabaseFileLockedException.

```
try{
    ObjectContainer container = Db4oEmbedded.openFile("database.db4o");
} catch (DatabaseFileLockedException e){
    // Database is already open!
    // Use another database-file or handle this case gracefully
}
```

ImportantExceptionCases.java: If the database is already open

2. Opening a client connection can throw IOException.

```
try{
    final ObjectContainer container = Db4oClientServer.openClient("localhost", 1337, "sa", "sa");

} catch(Db4oIOException e){
    // Couldn't connect to the server.
    // Ask for new connection-settings or handle this case gracefully
}
```

ImportantExceptionCases.java: Cannot connect to the server

3. Working with db4o-unique constraints the commit may throw exceptions when the constraints are violated.

```
container.store(new UniqueId(42));
container.store(new UniqueId(42));
try{
    container.commit();
} catch (UniqueFieldValueConstraintViolationException e){
    // Violated the unique-constraint!
    // Retry with a new value or handle this gracefully
    container.rollback();
}
```

ImportantExceptionCases.java: Violation of the unique constraint

db4o Reflection API

Reflection gives your code access to internal information for classes loaded into the JVM. It allows you to explore the structure of objects at runtime. In the case of reflection metadata is the description of classes and objects within the JVM, including their fields, methods and constructors. It allows the programmer to select target classes at runtime, create new objects, call their methods and operate with the fields.

In order to persist object db4o uses the reflection to read object and store their values. You can exchange this reflector layer in the configuration.

By default the JdkReflector is used. This reflector also allows you to specify the right class-loader.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().reflectWith(
        new JdkReflector(Thread.currentThread().getContextClassLoader()));
```

CommonConfigurationExamples.java: Change the reflector

It's also possible to use very special class resolving strategy by implementing the JdkLoader-interface. For example when you want to look up classes in multiple class loaders.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();

JdkLoader classLookUp = new ClassLoaderLookup(
        Thread.currentThread().getContextClassLoader(),
        new URLClassLoader(new URL[]{new URL("file://./some/other/location")}));
configuration.common().reflectWith(new JdkReflector(classLookUp));

ObjectContainer container = Db4oEmbedded.openFile("database.db4o");
```

CommonConfigurationExamples.java: Complex class loader scenario

```
public     class ClassLoaderLookup implements JdkLoader {
    private     final List<ClassLoader> classLoaders;


    ClassLoaderLookup(ClassLoader...classLoaders) {
        this.classLoaders = Arrays.asList(classLoaders);
    }

    ClassLoaderLookup(Collection<ClassLoader> classLoaders) {
        this.classLoaders = new ArrayList<ClassLoader>(classLoaders);
    }

    @Override
    public Class loadClass(String className) {
        for (ClassLoader loader : classLoaders) {
            Class<?> theClass = null;
            try {
                theClass = loader.loadClass(className);
                return theClass;
            } catch (ClassNotFoundException e) {
                // first check the other loaders
            }
        }
        throw     new RuntimeException(new ClassNotFoundException(className));
    }

    @Override
    public Object deepClone(Object o) {
        return     new ClassLoaderLookup(classLoaders);
    }
}
```

ClassLoaderLookup.java: Complex class loader scenario

If you need some special treatment you can create you're own reflector implementation. See "Creating your own reflector" on page 119

db4o has also a generic reflector which can deal with stored objects without using the original class. See "GenericReflector" on page 121

Creating your own reflector

By default db4o uses the JdkReflector . As alternative you can create your own reflector and use it with db4o. In order to do so you need to implement the Reflector interface. And then pass an instance of your implementation to db4o.

Here's an example of a logging reflector. Its only difference from standard reflector is that information about loaded classes is outputted to console:

```
class LoggerReflector implements Reflector{
    private   final Reflector readReflector;

    public LoggerReflector() {
        this(new JdkReflector(Thread.currentThread().getContextClassLoader()));
    }

    public LoggerReflector(Reflector readReflector) {
        this.readReflector = readReflector;
    }

    @Override
    public   void configuration(ReflectorConfiguration reflectorConfiguration) {
        readReflector.configuration(reflectorConfiguration);
    }

    @Override
    public ReflectArray array() {
        return readReflector.array();
    }

    @Override
    public ReflectClass forClass(Class aClass) {
        System.out.println("Reflector.forClass("+aClass+")");
        return readReflector.forClass(aClass);
    }

    @Override
    public ReflectClass forName(String className) {
        System.out.println("Reflector.forName("+className+")");
        return readReflector.forName(className);
    }

    @Override
    public ReflectClass forObject(Object o) {
        System.out.println("Reflector.forObject("+o+")");
        return readReflector.forObject(o);
    }

    @Override
    public   boolean isCollection(ReflectClass reflectClass) {
        return readReflector.isCollection(reflectClass);
    }

    @Override
    public   void setParent(Reflector reflector) {
        readReflector.setParent(reflector);
    }

    @Override
    public Object deepClone(Object o) {
        return   new LoggerReflector((Reflector) readReflector.deepClone(o));
    }
}
```

ReflectorExamples.java: Logging reflector

GenericReflector

db4o uses reflection internally for persisting and instantiating user objects. Reflection helps db4o to manage classes in a general way while saving. It also makes instantiation of objects using class name possible. However db4o reflection API can also work on generic objects when a class information is not available.

db4o uses a generic reflector as a decorator around specific reflector. The generic reflector is set when an object container is opened. All subsequent reflector calls are routed through this decorator.

The generic reflector keeps list of known classes in memory. When the generic reflector is called, it first checks its list of known classes. If the class cannot be found, the task is transferred to the delegate reflector. If the delegate fails as well, generic objects are created, which hold simulated "field values" in an array of objects.

Generic reflector makes possible the following use cases:

- Running a db4o server without deploying application classes.
- Easier access to stored objects where classes or fields are not available.
- Building interfaces to db4o from any programming language.

One of the live use cases is the ObjectManager, which uses the generic reflector to read C# objects from Java.

The generic reflector is automatically used when the class of a stored object is not found.

Runtime Monitoring

The db4o runtime statistics is a monitoring feature allowing to collect various important runtime data. This data can be crucial in resolving performance issues, analyzing usage patterns, predicting resource bottlenecks etc. Runtime Statistics can be collected both in the application testing stage and in an application deployed to production system. In the first case this data can help to estimate hardware requirements and analyze the stability of the system. In the latter case, the data can be used to fine-tune performance, find problems and fix bugs. Take a look how you install the statistics and monitor them.

Install And Monitor

On the Java platform all runtime statistics are published through the JMX interface. This means that you can use any JMX client to monitor the statistics. See "Install and Monitor" on page 121

Available Statistics

Currently following statistics are available.

- Query Monitoring: Monitor how queries behave and perform.
- Object Lifecycle Monitoring: Monitor how many objects are stored, deleted and activated.
- IO Monitoring: Monitor the IO-operations of db4o.
- Network Monitoring: Monitor the network operations of db4o.
- Reference System Monitoring: Monitor db4o's reference system.

Install and Monitor

On the Java platform all runtime statistics are published through the JMX interface. This means that you can use any JMX client to monitor the statistics.

Installing the Monitoring Support

The first thing we need to do is to add the monitoring support to the db4o configuration. The monitoring-support are in the optional-jars. So you need to add those for the monitoring support.

Monitoring adds a small overhead to the regular db4o operations. Therefore the monitoring support is distributed across different monitoring options, so that you can add only the options you need.

Currently following options are available:

- Query Monitoring: Monitor how queries behave and perform.
- Object Lifecycle Monitoring: Monitor how many objects are stored, deleted and activated.
- IO Monitoring: Monitor the IO-operations of db4o.
- Network Monitoring: Monitor the network operations of db4o.
- Reference System Monitoring: Monitor db4o's reference system.

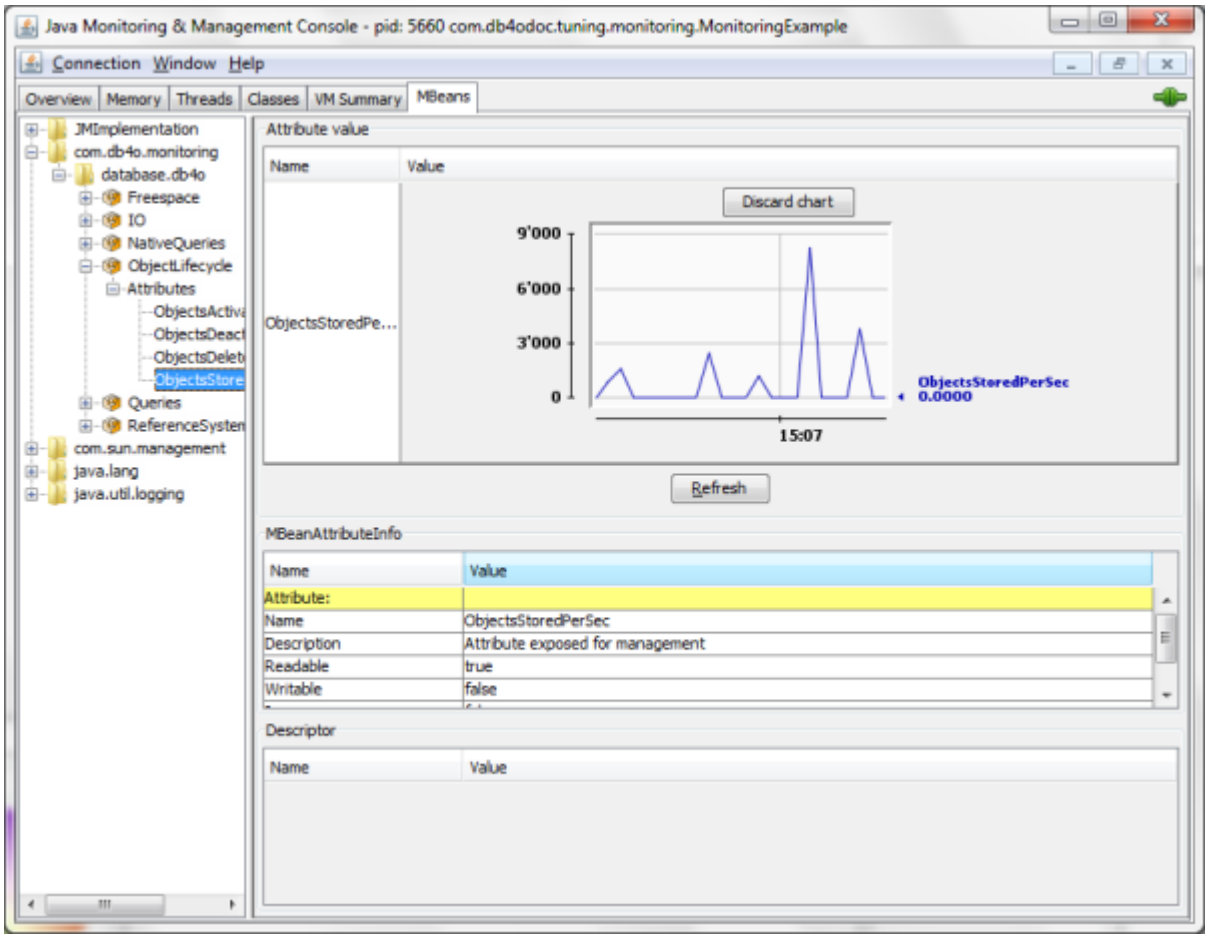
Monitor With JConsole

As said, the statistics are published through the JMX- interface (Java Management Extensions). Therefore you can use any JMX-client to monitor db4o. The JDK brings its own JMX-client with it: The JConsole-application is located in the bin-folder of the JDK-installation. Start the JConsole and connect your running db4o application. Then you will find the db4o related readings on the "MBeans" tab. In the treeview on the left, you will see "com.db4o.monitoring".

Below this tree node you will find all open ObjectContainers that have JMX monitoring enabled. For each of them the respective runtime statistics categories will be listed. If you select the "Attributes" node for any of them, the corresponding set of attribute values will be shown along with their most recent reading.

By double clicking on the number value a graph will be displayed.

Some Monitoring-Nodes also provide notifications. You can subscribe to this notifications. In the categories where the "Notifications" node is visible, you can select it and click "Subscribe" on the bottom right. For example we provide notifications about unoptimized native queries and about class index scans.



Monitoring Queries

You can monitor queries to find out more about the runtime behavior of your application.

Configure the Query Monitoring Support

First you need to add the monitoring support to the db4o configuration. There are two separate items. The QueryMonitoringSupport will monitor the very basic query operations. The NativeQueryMonitoringSupport adds additional statistics about native queries.

```
configuration.common().add(new QueryMonitoringSupport());
configuration.common().add(new NativeQueryMonitoringSupport());
```

QueryMonitoring.java: Add query monitoring

The Query Statistics

AverageQueryExecutionTime : Tells you how long on average a query takes to execute. Of course this heavily depends on the complexity of the queries. However if this number is high, you maybe should improve you query-performance. For example by adding additional indexes. See "Indexing" on page 63

ClassIndexScansPerSecond: Tells you the number of queries which required to scan through all objects. This means that a query couldn't use a field index and therefore required to go through all

objects. This is of course slow. You should try to keep this number low by adding the right indexes on fields. See "Indexing" on page 63

QueriesPerSecond: Tells you how many queries run per second.

Loaded From Class Index Notifications: The query statistics can notify you every time a query used the class-index and couldn't utilize a field index. You should try to avoid loading by class index by adding the right indexes on fields. See "Indexing" on page 63

The Native Query Statistics

NativeQueriesPerSecond: Tells you how many native queries per second run.

UnoptimizedNativeQueriesPerSecond: Tells you how many unoptimized native queries run per second. Such queries need to instantiate all objects which is a slow operation. If this number is high, you should try to simplify your queries.

Native Query Not Optimized Notifications: The native query statistics can notify you every time a query couldn't be optimized. You should try to avoid such queries. Try to simplify the native query or fall back to SODA-queries.

Monitor Object Lifecycle

You can monitor the object lifecycle statistics of db4o to find out more about the runtime behavior of your application.

Configure the Object Lifecycle Monitoring Support

In order to monitor the object lifecycle statistics, you need to add the monitoring support to the configuration.

```
configuration.common().add(new ObjectLifecycleMonitoringSupport());
```

ObjectLifecycleMonitoring.java: Monitor the object lifecycle statistics

The Object Lifecycle Statistics

ObjectsActivatedPerSec: Tells you how many objects are activated per second. Activation can consume a lot of time for complex object graphs. If there's a lot of time spend with activating objects, you may want to reduce the amount of activated objects. One of the best way to activate only the minimum set of objects is to use transparent activation.

ObjectsDeactivatedPerSec: Tells you how many objects are deactivate per second.

ObjectsDeletedPerSec: Tells you how many objects are deleted per second.

ObjectsStoredPerSec: Tells you how many objects are stored per second.

Monitor IO

You can monitor the IO-operations of db4o to find out more about the runtime behavior of your application.

Configure the IO Monitoring Support

In order to monitor the IO activities of db4o you need to configure the IO monitoring support.

```
configuration.common().add(new IOMonitoringSupport());
```

IOMonitoring.java: Add IO-Monitoring

The IO Statistics

BytesReadPerSecond: Tells you how many bytes are read per second from the disk.

BytesWrittenPerSecond: Tells you how many bytes are written to disk per second.

ReadsPerSecond: Tells how many read requests are issues per second.

SyncsPerSecond: Tells you how many sync-operations are done per second. The sync-operation is a part of the commit-process. I forces the runtime and operating system to flush all buffers to the disk to ensure that the data are safely stored. The sync operation is normally a costly operation, since it need to wait on the physical disk to finish all write operations

.**WritesPerSecond**: Tells you how may write requests are issued per second.

Monitor Freespace

You can monitor the freespace-manager to find out more about the runtime behavior of your application.

Configure the Freespace Monitoring Support

To monitor the freespace manager you need to add the monitoring support to the db4o configuration.

```
configuration.common().add(new FreespaceMonitoringSupport());
```

FreespaceMonitoring.java: Monitor the free-space system

The Freespace Statistics

AverageSlotSize: Tells you how big the average slot is. When you store larger objects, they need larger slots to fit it.

ReusedSlotsPerSecond: Tells you how many slots can be reused. When an object is deleted or modified, its old slot is released and can be reused. If you modify and delete a lot of objects, but the slots cannot be reused, then the database-file will fragment.

When this number is low but the free-space increases and increases then you have a fragmentation issue. Defragment your database. Use the btree-id-system.

SlotCount: Tells you how many slots are used by the database. As more and more objects are stored, the slot count will increase.

TotalFreespace: Tells you how much freespace there is in the database-file.A hight free-space number with a low reused slots number indicates database fragmentation. Defragment your database. Use the btree-id-system.

Monitor Network

You can monitor the network-operations of db4o in client-server mode.

Configure the Network Monitoring Support

First you need to add the monitoring support to the db4o configuration. The network monitoring support is in two separate configuration-items. The NetworkingMonitoringSupport can be used on the server and client to monitor the network statistics.

```
configuration.common().add(new NetworkingMonitoringSupport());
```

CSMonitoring.java: Add the network monitoring support

The ClientConnectionsMonitoringSupport can be added to the db4o server to monitor the connected clients.

```
configuration.addConfigurationItem(new ClientConnectionsMonitoringSupport());
```

CSMonitoring.java: Add the client connections monitoring support

The Network Statistics

BytesReceivedPerSecond: Tells you how many bytes this client or server has receives per second.

BytesSentPerSecond: Tells you how many bytes this client or server sends per second.

MessagesSentPerSecond: Tells you how many messages this client or server sends per second.

ConnectedClientCount: Tells you how many clients are connected to this server.

Monitor Reference System

You can monitor the reference-system to find out more about the runtime behavior of your application. The reference-system ensures that each object has only one in memory representation.

Configure the Reference System Monitoring Support

First you need to add the monitoring support to the db4o configuration.

```
configuration.common().add(new ReferenceSystemMonitoringSupport());
```

ReferenceSystemMonitoring.java: Add reference system monitoring

The Freespace Statistics

ObjectReferenceCount: Tells you how many references are currently hold in the reference-system. By default db4o uses weak references to objects. If this count is very high, you might hold unnecessary references to persisted objects in your application.

Diagnostics

The db4o engine provides user with a special mechanism showing runtime diagnostics information. Diagnostics can be switched on in the configuration before opening the database file:

The DiagnosticListener is a callback interface tracking diagnostic messages from different parts of the system.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().diagnostic().addListener(new DiagnosticToConsole());
```

CommonConfigurationExamples.java: Add a diagnostic listener

Built-in Listeners

There are two build in listeners, which print the output to the console

- DiagnosticToConsole:Prints diagnostic messages to the console.
- DiagnosticToTrace: Only on .NET, prints diagnostic messages to the debug output window.

Messages-Types

Every diagnostic message is represented by it's own type. You can filter the messages by checking for certain instances. Take a look how you can filter for certain messages.

- **MissingClass**: Notifies you that a class couldn't be found. You should add that class in order to avoid problems. If you've renamed the class, you should rename it in the database or add an alias.
- **DefragmentRecommendation**: Notifies you that you should consider to defragment the database.
- **LoadedFromClassIndex**: Notifies you that db4o couldn't use a field-index to perform a query. This means that the query runs extremely slow on larger data sets. Consider adding a field-index.
- **DescendIntoTranslator**: Means that you a query couldn't be optimized, because the query touches a class with a translator. Therefore the query runs slow. You should consider working without a translator or changing the query.
- **ClassHasNoFields**: You stored a class which has no fields. Even when the class has no fields it need to maintain its class index and needs to be stored. So it adds overhead for storing 'empty' objects. You should consider removing the empty class.
- **DeletionFailed**: db4o failed to delete a object from the database.
- **UpdateDepthGreaterOne**: You have configured a update depth greater one. A large update depth slows down updates significantly. Consider reducing the update-depth and use another strategies to update objects correctly, like transparent persistence.
- **NativeQueryNotOptimized**: A native query couldn't be optimized. An unoptimized query runs significantly slower than a optimized query. Consider to simplify you're query.
- **NativeQueryOptimizerNotLoaded**: Couldn't load the native query optimizer. Ensure that all required jar are added to your application.
- **NotTransparentActivationEnabled**: Notifies you when a class doesn't support transparent activation. Such object need to be fully activated and slow down the activation process.
- **ObjectFieldDoesNotExist**: A query uses a object-field which doesn't exist. Check you're queries to use only exising fields.

Diagnostic Messages Filter

The standard listeners can potentially produce quite a lot of messages. By writing your own DiagnosticListener you can filter that information.

On the stage of application tuning you can be interested in optimizing performance through indexing. Diagnostics can help you with giving information about queries that are running on un-indexed fields. By having this information you can decide which queries are frequent and heavy and should be indexed, and which have little performance impact and do not need an index. Field indexes dramatically improve query performance but they may considerably reduce storage and update performance.

In order to get rid of all unnecessary diagnostic information and concentrate on indexes let's create special diagnostic listener:

```
private  static  class DiagnosticFilter implements DiagnosticListener{
    private  final Set<Class> filterFor;
    private  final DiagnosticListener delegate;

    private DiagnosticFilter(DiagnosticListener delegate,Class<? extends Diagnostic>...filterFor) {
        this.delegate = delegate;
        this.filterFor = new HashSet<Class>(Arrays.asList(filterFor));
    }

    public  void onDiagnostic(Diagnostic diagnostic) {
        Class<?> type = diagnostic.getClass();
        if(filterFor.contains(type)){
            delegate.onDiagnostic(diagnostic);
        }
    }
}
```

DiagnosticsExamples.java: A simple message filter

After that we can use the filter-listener. It takes two arguments. The first one is a regular listener, the second is a list of all messages which are passed through.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().diagnostic()
        .addListener(new DiagnosticFilter(new DiagnosticToConsole(), LoadedFromClassIndex.class));
```

DiagnosticsExamples.java: Filter for unindexed fields

Configuration

db4o is configured by its configuration API. The configuration allows you to adjust db4o to your scenarios.

Configure db4o

In order to configure db4o, you need to create a new configuration-instance and set the desired settings on it. After that, you pass the configuration-instance to the object-container factory.

Note that you cannot share a configuration-instance. For each object-container you create, you need to pass in a new configuration-instance. It's recommended to create a method which will return a new configuration instance on request:

Embedded-Configuration

For an embedded container you can configure the common-, file- and id-system-configuration.

Server-Configuration

For an server you can configure the common-, file-, networking-, server- and id-system-configuration.

Client-Configuration

For an client you can configure the common-, networking-, client- and id-system-configuration.

Configuration Is Not Persistent

The db4o configuration is not persistent with a few exception. This means that you need to configure db4o each time you create a object-container instance.

Configuration in Client/Server-Mode

For using db4o in client/server mode it is recommended to use the same configuration on the server and on the client. To set this up nicely it makes sense to create one application class with one method that returns the required configuration and to deploy this class both to the server and to all clients.

Configuration-Settings Overview

The configuration-settings which are common across -client, embedded and db4o-server are summed up here: See "Common Configuration" on page 129

Common Configuration

The common-configuration applies to the embedded-, client- and the server-mode of db4o. All the common configuration is accessible via the common-getter on the configuration-object.

Overview

Here's a overview over all common configuration-settings which you can change:

	Same in C/S[1][2]	Can not change [3]
ActivationDepth: Change globally the activation-depth.		
Aliases: Configure aliases for class and package-names.		
AllowVersionUpdates: Allow/Disallow to update the database-format.	Yes	
AutomaticShutDown: Close the database when the application exits.		
BTreeNodeSize: Tune the size of the B-tree-node which are used for the indexes.		
Callbacks: Turn object-callbacks on an off.		
CallConstructors: Use or bypass the constructor for creating objects.		
DetectSchemaChanges: Disable/Enable schema changes detection.		
Diagnostic: Add diagnostic-listeners.		
ExceptionsOnNotStorable: Enable/Disable exceptions on not storable objects.		
InternStrings: Will call the intern-method on the retrieved strings.		
MarkTransient: Configure a Annotation for marking fields as transient.		
MessageLevel: Configure the logging-message level.		
NameProvider: Configure the toString() value of the object-container.		
MaxStackSize: Configure how many recursive operations are executed directly on the stack.		
ObjectClass: Configure class-specific settings.		
OptimizeNativeQueries: Enable runtime query optimization.		
OutStream: Configure the log message output stream		
Queries: Configure query behaviors		
ReflectWith: Configure a reflector.		
RegisterTypeHandler: Register a new TypeHandler.	Required	
StringEncoding: Configure the string-encoding.	Required	Yes
TestConstructors: Configure if db4o checks for valid con-		

[1] Client-Server
[2] This setting needs to be the same on the server and all clients.
[3] This setting has to be set the first time when the database is created. You cannot change is for an existing database.

structors.		
UpdateDepth: Change the update-depth.		
WeakReferenceCollectionInterval: Change the weak-reference cleanup interval. Default setting is 1000 milliseconds.		
WeakReferences: Enable/disable weak references.		

Additional Configuration Items

There are additional configuration items which add for additional features. You can add then on the common- configuration. For example to enable transparent persistence you add the TransparentPersistenceSupport configuration item. Take a look a the available configuration items. See "Common Configuration Items" on page 131

Common Configuration Items

Configuration items add special capability to the system. Here's a list of all configuration items available.

TransparentActivationSupport

Support for transparent activation. See "Transparent Activation" on page 47

TransparentPersistenceSupport

Support for transparent persistence. See "Transparent Persistence" on page 50

UniqueFieldValueConstraint

Set up unique Field constraints. See "Unique Constraints" on page 68

BigMathSupport

Add support for the BigDecimal and BigInteger classes. See "BigMath" on page 100

FreespaceMonitoringSupport

Enables you to monitor the free-space-manager. See "Runtime Monitoring" on page 121

IOMonitoringSupport

Enables you to monitor the IO-activity of db4o. See "Runtime Monitoring" on page 121

NativeQueryMonitoringSupport

Enables you to monitor the native queries. See "Runtime Monitoring" on page 121

NetworkingMonitoringSupport

Enables you to monitor the network activity of db4o. See "Runtime Monitoring" on page 121

ObjectLifecycleMonitoringSupport

Enables you to monitor the object lifecycles. See "Runtime Monitoring" on page 121

QueryMonitoringSupport

Enables you to monitor db4o queries. See "Runtime Monitoring" on page 121

ReferenceSystemMonitoringSupport

Enables you to monitor db4o's reference system. See "Runtime Monitoring" on page 121

Activation Depth

db4o uses the concept of activation to avoid loading to much data into memory. You can change the global activation depth with this setting.

Note: As soon as you use transparent activation/persistence this configuration option has no effect.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().activationDepth(2);
```

CommonConfigurationExamples.java: Change activation depth

A higher activation depth is usually more convenient to work with, because you don't face inactivated objects. However, a higher activation depth costs performance, because more data has to read from the database. Therefore a good balance need to be found. Take also a look a transparent activation, since it solves the activation issue completely.

Class Specific Configuration

You can also configure a class specific activation depth. See "Class Specific Configuration" on page 158

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).minimumActivationDepth(2);
```

ObjectConfigurationExamples.java: Set minimum activation depth

Update Depth

By default db4o only stores changes on the updated object but not the changes on referenced objects. With a higher update-depth db4o will traverse along the object graph to a certain depth and update all objects. See "Update Concept" on page 47
Note: As soon as you use transparent persistence this configuration option has no effect and isn't needed.

With the update-depth you configure how deep db4o updates the object-graph.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().updateDepth(2);
```

CommonConfigurationExamples.java: Increasing the update-depth

A higher update depth is usually more convenient, because you don't need to explicitly store each changed object. However the higher the update depth is the more time it takes to update the objects. Therefore it is a tradeoff. Note that you can also use transparent persistence, which takes of updating the right objects.

Class Specific Configuration

You can also configure a class specific update depth. See "Class Specific Configuration" on page 158

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).updateDepth(2);
```

ObjectConfigurationExamples.java: Set the update depth

Aliases

Aliases allow you to use different names for your persistent classes in the database and in your application. Before a class is saved, db4o checks if an alias exists. If this is the case, the alias-name is used

instead of the original name.

Adding Aliases

You can add aliases by adding instance of the Alias-interface to the configuration. You can add as many aliases to the configuration as you want. Following options are available.

- TypeAlias: Allows you to give an alias for a certain type.
- WildCardAlias: Allows you to give an alias for a namespace / multiple types. It allows you to use a wildcard for the name. (The *-character)
- Your own implementation, by implementing the Alias-interface.

Each alias has two arguments. The first argument it the name which the type has in the database. The second argument is the type which is currently used. Ensure that you alias only types which can be resolved.

Furthermore the order in which you add the aliases matters. Add first the most specific alias and the go to the more general alias. For example add first all TypeAlias and then the WildCardAlias.

```
// add an alias for a specific type
configuration.common().addAlias(
        new TypeAlias("com.db4odoc.configuration.alias.OldTypeInDatabase",
                "com.db4odoc.configuration.alias.NewType"));
// or add an alias for a whole namespace
configuration.common().addAlias(
        new WildcardAlias("com.db4odoc.configuration.alias.old.location.*",
                "com.db4odoc.configuration.alias.current.location.*"));
```

AliasExamples.java: Adding aliases

Allow Version Updates

The db4o database file format is a subject to change to allow progress for performance and additional features. db4o does not support downgrades back to previous versions of database files. In order to prevent accidental upgrades when using different db4o versions, db4o does not upgrade databases by default. Database upgrading can be turned on with the following configuration switch:

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().allowVersionUpdates(true);

// reopen and close the database to do the update
ObjectContainer container = Db4oEmbedded.openFile(configuration, DATABASE_FILE);
container.close();
```

CommonConfigurationExamples.java: Update the database-format

Please note that, once the database file version is updated, there is no way to get back to the older version. When a database file is opened successfully with the new db4o version, the upgrade of the file will take place automatically. You can simply upgrade database files by opening and closing a db4o database once.
Recommendations for upgrading:

- Backup your database file to be able to switch back.
- Defragmenting a database file with the new db4o version after upgrading can make the database more efficient.

Automatic Shutdown

With this setting you can disable the shutdown monitoring for db4o. By default, db4o will close the database when the JVM exits. However on some embedded devices this can lead to issues. So disable it when you experience problems when terminating the application.

Note that its recommended to close the object-container / -server when you close the application anyway, even when this setting is enabled.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().automaticShutDown(false);
```

CommonConfigurationExamples.java: Disable automatic shutdown

B-Tree Node Size

db4o uses B-tree indexes for increased query performance and reduced memory consumption. B-trees are used for field-index, class-index and for the id-system. B-trees are optimized for scenarios where a part of the data is on secondary storage such as a hard disk, since disk accesses is extremely expensive. B-trees minimize the number of disk accesses required. You find more information about B-trees on the internet.

You can tune the B-trees by configuring the size of a node. Larger node sizes require less disk access, since a node is read on one read-operation. However it consumes more memory to keep the larger nodes available. Also larger nodes take longer to write back to disk. You need to tune the B-tree node size according to your application requirements. When benchmarking the settings, use large data sets. The influence of B-trees is more significant for larger dabases with 100'000 and more objects.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().bTreeNodeSize(256);
```

CommonConfigurationExamples.java: Change B-tree node size

Disable Callbacks

db4o supports callback-methods on each class. In order to support this, db4o scans all persistent classes and looks for the callback-method signature. When a lot of different classes are stored, this may take some time, especially on embedded devices. Therefore you can disable the object callbacks, when you don't need them.

Note that this doesn't disable the global events. These can still be used, even when callbacks are disabled.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().callbacks(false);
```

CommonConfigurationExamples.java: Disable callbacks

Calling Constructors

By default db4o by passes the constructor to instantiate the objects. Therefore it uses the reflection capabilities of the current platform. On some embedded platform this is not possible because of security and platform constrains. On other platforms bypassing the constructor is significantly slower than calling it. Therefore you change the behavior so that db4o calls the constructor. Note that when you enable this setting, you classes need a constructor which can be called with default-arguments without throwing an exception.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().callConstructors(true);
```

CommonConfigurationExamples.java: Call constructors

Class Specific Configuration

You can enable this setting only for certain classes. For example when the constructor needs to initialize transient fields.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).callConstructor(true);
```

ObjectConfigurationExamples.java: Call constructor

Disable Schema Change Detection

db4o scans the class structure to find out the schema of the objects. This takes a little time. When a lot of classes are persistent this may take some time, especially on embedded devices.

Therefore you can disable this check. You can disable it only, when db4o already knows all stored classes. This means a object of each class has already been stored once. Furthermore there shouldn't be any further changes.

This setting is only useful for very special scenarios with no schema evolution at all. Otherwise this setting may cause strange and subtle errors!

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().detectSchemaChanges(false);
```

CommonConfigurationExamples.java: Disable schema evolution

Diagnostics

Enables you to add diagnostic listeners to db4o. Read more in the tuning chapters. See "Diagnostics" on page 126

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().diagnostic().addListener(new DiagnosticToConsole());
```

CommonConfigurationExamples.java: Add a diagnostic listener

Exceptions On Not Storable Objects

When db4o cannot store a object, it will throw an exception. This is the default behavior. When you disable this object, db4o will silently ignore objects which cannot be stored instead of throwing an exception.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().exceptionsOnNotStorable(false);
```

CommonConfigurationExamples.java: Disable exceptions on not storable objects

Intern Strings

You can configure db4o to call the intern method on all strings. See more on the intern method for your platform.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().internStrings(true);
```

CommonConfigurationExamples.java: intern strings

Benefits

When a lot of strings contain a the exact same content, calling intern on them can safe some memory.

Disadvantage

Calling intern on a string adds that string to a global pool. Therefore this string cannot be garbage collected. So when you load a lot of strings which you use only once, you can run into memory-problems.

Mark Transient

This allows you to configure additional Annotations to mark fields as transient. You can add multiple such Annotations by calling the method for each Annotation once.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().markTransient(TransientMarker.class.getName());
```

CommonConfigurationExamples.java: add an transient marker annotatin

Message Level

This allows you to enable debug-messages of db4o. Currently four message levels are supported:

- Level = 0: No messages, default configuration.
- Level> 0: Normal messages.
- Level> 1: State messages (new object, object update, delete);
- Level> 2: Activation messages (object activated, deactivated).

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().messageLevel(4);
```

CommonConfigurationExamples.java: Change the message-level

By default the output is send to the console. But you can redirect the output to any output stream. See "Changing the Output Stream" on page 137

Max Stack Size

This settings allows you to set how many recursive operations (like storing a new complex object graph) are executed on the stack. When you have a small stack size you can use a small value below 10. The default is 20.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().maxStackDepth(16);
```

CommonConfigurationExamples.java: Set the stack size

Name Provider

You can overwrite how the object-containers is named. This can be helpful for debugging multiple-container scenarios.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().nameProvider(new SimpleNameProvider("Database"));
```

CommonConfigurationExamples.java: set a name-provider

Disable Optimize Native Queries

Normally db4o tries to optimize native queries at runtime. See "Native Query Optimization" on page 20

However on some limited embedded platforms like Android this doesn't work. In such cases you can disable the native query optimizer and use instead the compile time optimizer.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().optimizeNativeQueries(false);
```

CommonConfigurationExamples.java: Disable the runtime native query optimizer

Register Type Handler

You can add special type-handlers for your datatypes. This allows you to plug in your own serialization-handling for that type. See "TypeHandlers" on page 108

Note that you need the type-handler on the client and server installed.

Changing the Output Stream

Normally the debug messages of db4o are printed to the default console. However you can configure db4o to send the information to any output stream:

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
// you can use any output stream for the message-output
configuration.common().outStream(System.err);
```

CommonConfigurationExamples.java: Change the output stream

Query Modes

What is the best way to process queries? How to get the optimum performance for your application needs?

Situations, when query result is bigger than the memory available or when query time is longer than the whole functional operation.

Luckily db4o takes most of the trouble for itself. There are three query modes allowing to fine tune the balance between speed, memory consumption and availability of the results:

- Immediate: The default mode. The result is determined immediately and only the objects are lazy loaded.
- Lazy: The query is evaluated while iterating to the result.
- Snapshot: Runs the index-processing and creates a snapshot of the result. Further processing is done while iterating over the result.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().queries().evaluationMode(QueryEvaluationMode.IMMEDIATE);
```

CommonConfigurationExamples.java: Change the query mode

Immediate Queries

This is the default query mode: the whole query result is evaluated upon query execution and an object IDs list is produced as a result.

Obviously evaluation takes some time and in a case of big result sets you will have to wait for a long time before the first result will be returned. This is especially unpleasant in a client-server setup, when query processing can block the server for seconds.

This mode makes the whole objects result set available at once. An result list is built based on the committed state in the database. As soon as a result is delivered it won't be changed neither by changes in current transaction neither by committed changes from another transactions.

Note that the result set contains only references to objects you were querying for, which means that if an object field has changed by the time of the actual object retrieval from the object set - you will get the new field value:

Advantages

- If the query is intended to iterate through the entire resulting ObjectSet, this mode will be slightly faster than the others.
- The query will process without intermediate side effects from changed objects (by the caller or by other transactions).

Disadvantages

- Query processing can block the server for a long time.
- In comparison to the other modes it will take longest until the first results are returned.
- The ObjectSet will require a considerate amount of memory to hold the IDs of all found objects.

Lazy Queries

With the Lazy Querying no criteria evaluated at all. Instead an iterator is created against the best index found. Further query processing, including all index processing, will happen when the application iterates through the result. This allows you to get the first query results almost immediately.

In addition to the very fast execution this method also ensures very small memory consumption. Because lazy queries do not need an intermediate representation as a set of IDs in memory. With this approach it does not have to cache a single object or ID. The memory consumption for a query is practically zero, no matter how large the result set is going to be.

There are some interesting effects appearing due to the fact that the objects are getting evaluated only on a request. It means that all the committed modifications from the other transactions and uncommitted modifications from the same transaction will be taken into account when delivering the result objects.

Advantages

- The call to `Query.execute()` will return very fast. First results can be made available to the application before the query is fully processed.
- A query will consume hardly any memory at all because no intermediate ID representation is ever created.

Disadvantages

- Lazy queries check candidates when iterating through the 'result'. In doing so the query processor takes changes into account that may have happened since the Query.execute() call: committed changes from other transactions, **and uncommitted changes from the calling transaction**. There is a wide range of possible side effects:
  - The underlying index may have changed.
  - Objects themselves may have changed in the meanwhile.
  - There even is a chance of creating an endless loop. If the caller iterates through the `ObjectSet` and changes each object in a way that it is placed at the end of the index, the same objects can be revisited over and over.

  In lazy mode it can make sense to work in same ways as with collections to avoid concurrent modification exceptions. For instance one could iterate through the `ObjectSet` first and store all the objects to a temporary collection representation before changing objects and storing them back to db4o.
- Some method calls against a lazy `ObjectSet` will require the query processor to create a snapshot or to evaluate the query fully. An example of such a call is `ObjectSet.size()`.

Lazy mode can be an excellent choice for single transaction read use, to keep memory consumption as low as possible.

Snapshot Queries

Snapshot mode allows you to get the advantages of the Lazy queries avoiding their side effects. When query is executed, the query processor chooses the best indexes, does all index processing and creates a snapshot of the index at this point in time. Non-indexed constraints are evaluated lazily when the application iterates through the `ObjectSet` result set of the query.

Snapshot queries ensure better performance than Immediate queries, but the performance will depend on the size of the result set.

As the snapshot of the results is kept in memory the result set is not affected by the changes from the caller or from another transaction.

Advantages

- Index processing will happen without possible side effects from changes made by the caller or by other transaction.
- Since index processing is fast, a server will not be blocked for a long time.

Disadvantages

- The entire candidate index will be loaded into memory. It will stay there until the query ObjectSet

is garbage collected. In a C/S[1] setup, the memory will be used on the server side

Client/Server applications with the risk of concurrent modifications should prefer Snapshot over Lazy mode to avoid side effects from other transactions.

Changing The Reflector

This setting allows you to change the reflector for db4o. The reflector is responsible to inspect the meta-data of objects and report them to db4o. See "db4o Reflection API" on page 118

This setting also allows you also to specify which class-loader is used to find classes. For that you pass the right class-loader to the JdkReflector constructor.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().reflectWith(
        new JdkReflector(Thread.currentThread().getContextClassLoader()));
```

CommonConfigurationExamples.java: Change the reflector

It's also possible to use very special class resolving strategy by implementing the JdkLoader-interface. For example when you want to look up classes in multiple class loaders.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();

JdkLoader classLookUp = new ClassLoaderLookup(
        Thread.currentThread().getContextClassLoader(),
        new URLClassLoader(new URL[]{new URL("file://./some/other/location")}));
configuration.common().reflectWith(new JdkReflector(classLookUp));

ObjectContainer container = Db4oEmbedded.openFile("database.db4o");
```

CommonConfigurationExamples.java: Complex class loader scenario

[1] Client-Server

```
public   class ClassLoaderLookup implements JdkLoader {
    private   final List<ClassLoader> classLoaders;


    ClassLoaderLookup(ClassLoader...classLoaders) {
        this.classLoaders = Arrays.asList(classLoaders);
    }

    ClassLoaderLookup(Collection<ClassLoader> classLoaders) {
        this.classLoaders = new ArrayList<ClassLoader>(classLoaders);
    }

    @Override
    public Class loadClass(String className) {
        for (ClassLoader loader : classLoaders) {
            Class<?> theClass = null;
            try {
                theClass = loader.loadClass(className);
                return theClass;
            } catch (ClassNotFoundException e) {
                // first check the other loaders
            }
        }
        throw   new RuntimeException(new ClassNotFoundException(className));
    }

    @Override
    public Object deepClone(Object o) {
        return   new ClassLoaderLookup(classLoaders);
    }
}
```

ClassLoaderLookup.java: Complex class loader scenario

String Encoding

When db4o stores and loads strings it has to convert them to and from byte arrays. By default db4o provides three types of string encoding, which do this job: Unicode, UTF-8 and ISO 8859-1, Unicode being the default one. In general Unicode can represent any character set. However, it also imposes a certain overhead, as character values are stored in 4 bytes (less generic encoding usually use 2 or even 1 byte per character). In order to save on the database file size, it is recommended to use UTF-8, which only required one byte per character.

Note that the string encoding need to be the same on the server and clients. Also you cannot change the string encoding for an existing database. If you want to change the encoding, you need to defragment the database.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().stringEncoding(StringEncodings.utf8());
```

CommonConfigurationExamples.java: Use the utf8 encoding

Of course you can also implement your own string encoding. You just need to implement the string encoding interface and pass it to this configuration.

Test For Constructors

This setting is only relevant for some embedded platforms which need a constructor (for example Android).

On startup, db4o checks that the classes have a callable constructor. This may take some time, especially on embedded platforms. When you run in production, you can disable this check, when you are sure that all classes can be instantiated.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().testConstructors(false);
```

CommonConfigurationExamples.java: Disable testing for callable constructors

Weak Reference Collection Interval

By default db4o uses weak references to keep track of loaded objects. These weak references need to be clean up from time to time. You can change this collection-interval.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().weakReferenceCollectionInterval(10*1000);
```

CommonConfigurationExamples.java: change weak reference collection interval

Disable Weak References

By default db4o uses weak references cache to all loaded objects. This ensures that the objects can be garbage collected. However it does impose a small overhead. You can disable weak reference if you like. Then db4o uses regular references. When disabled you need to remove objects explicit from the cache.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().weakReferences(false);
```

CommonConfigurationExamples.java: Disable weak references

File Configuration

The file-configuration contains all the configuration-settings which are related to file access. It can be set in the db4o-embedded-container or on the db4o-server. All the file configuration is accessible via the file-getter on the configuration-object.

Overview

Here's a overview over all file configuration-settings which you can change:

	Can not change [1]	Only useful for recovery [2]
AsynchronousSync: Enables asynchronous commits.		

[1] This setting has to be set the first time when the database is created. You cannot change is for an existing database.

[2] This setting is only useful when you try to recover a corrupted database file.

BlobPath: Specify where blobs are stored.		
BlockSize: Set the block-size of the database. Larger Blocks allow larger databases.	Yes	
DatabaseGrowthSize: Set the grow step size when the data-base-file is to small.		
DisableCommitRecovery: Disable the commit-recovery.		Yes
Freespace: Configure the free-space system.		
GenerateUUIDs: Configure to generate UUIDs for objects.		
~~**GenerateVersionNumbers**~~: Deprecated. Generate commit timestamps instead.		
LockDatabaseFile: Enable/disable the database file lock.		
ReadOnly: Set the database to read only mode.		
RecoveryMode: Set the database to a recovery mode.		Yes
ReserveStorageSpace: Reserve storage-space up front.		
Storage: Configure the storage system.		

Asynchronous File-Sync

One of the most costly operations during commit is flushing the buffers of the database file. In regular mode the commit call has to wait until this operation has completed.

When asynchronous sync is turned on, the sync operation will run in a dedicated thread, blocking all other file access until it has completed. This way the commit can returns immediately. This allows db4o and other processes to continue running side-by-side while the flush call is executed.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().asynchronousSync(true);
```

FileConfiguration.java: Allow asynchronous synchronisation of the file-flushes

Advantage

The commit call will return much faster. Because it doesn't have to wait until everything is written to disk.

Disadvantage

After the commit-call, you have no guaranties that everything is persistent. Maybe the commit is still in progress. On a failure, this means that you can lose a commit.

A setup with this option still guarantees ACID transaction processing: A database file is always either in the state before commit or in the state after commit. Corruption can not occur. You can just not rely on the transaction already having been applied when the commit() call returns.

Blob Path

With this setting you can specify in which directory the Blob-files are stored. The db4o-Blobs are stored outside the database file as regular files. With this settings you can set where this directory is. The default is the 'blob'-directory.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
try {
    configuration.file().blobPath("myBlobDirectory");
} catch (IOException e) {
    e.printStackTrace();
}
```

FileConfiguration.java: Configure the blob-path

Block Size

The block-size determines how how large a database can be. This value can be between 1 and 127. The default is 1. The resulting maximum database file size will be a multiple of 2GB. For example a a block-size of 8 allows a database-size up to 16 G. With the largest possible setting, 127, the database can grow up to 254 GB. A recommended setting for large database files is 8, since internal pointers have this length.

This configuration-setting has to be set the first time the database is created. You cannot change it for an existing database. If you want to change it, you need to deframent the database.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().blockSize(8);
```

FileConfiguration.java: Increase block size for larger databases

Advantages and Disadvantages

A larger value allows a larger database. However, since objects are aligned to the block size, a larger value will result in less efficient storage space usage. Furthermore a larger value may decrease the performance, because it causes more cache misses.

Database Growth Size

Configures how much the database-file grows, when there not enough space. Whenever no free space in the database is large enough to store a object the database file is enlarged. This setting configures by how much it should be extended, in bytes. This configuration setting is intended to reduce fragmentation. Higher values will produce bigger database files and less fragmentation. To extend the database file, a single byte array is created and written to the end of the file in one write operation. Be aware that a high setting will require allocating memory for this byte array.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().databaseGrowthSize(4096);
```

FileConfiguration.java: Configure the growth size

Disable Commit Recovery

This option only makes sense when you try to recover a corrupted database.

Turns commit recovery off. db4o uses a two-phase commit algorithm to ensure ACID properties. In a first step all intended changes are written to a free place in the database file, the "transaction commit record". In a second step the actual changes are performed. If the system breaks down during commit, the commit process is restarted when the database file is opened the next time.

On very rare occasions (possibilities: hardware failure or editing the database file with an external tool) the transaction commit record may be broken. In this case, this method can be used to try to open the

database file without commit recovery. The method should only be used in emergency situations after consulting db4o support.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().disableCommitRecovery();
```

FileConfiguration.java: Disable commit recovery

Freespace Configuration

When objects are updated or deleted, the space previously occupied in the database file is marked as "free". The freespace management system takes care of this space by maintaining which places in the file are free.

Discarding Threshold

By default the free space system keeps and maintains every bit of free space even the smallest ones. Very small blocks of storage are hard to reuse, because larger objects don't fit in. Therefore overtime more and more small blocks of free space are maintained. Maintaining these small free spaces can cost performance. Therefore you can configure the free-space system to discard small blocks. Then small blocks are not maintained as free space. On the downside these space is lost until the next defragmentation.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
// discard smaller than 256 bytes
configuration.file().freespace().discardSmallerThan(256);
```

FreeSpaceConfiguration.java: Discard settings

Memory Freespace System

When you use the memory free-space system the information of the free space locations is hold in memory. This is the fastest way to manage free space. However when the database is shut down abnormally, for example by a crash or power off, the free space information is lost. The space only can be reclaimed by defragmentation. This is the default-setting used by db4o.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().freespace().useRamSystem();
```

FreeSpaceConfiguration.java: Use the in memory system

BTree Freespace System

The B-tree free space system hold the information a B-tree which is stored on commits. Since the free space information is stored on disk, it is usually a slower then the memory free space system. However it doesn't lose the information on abnormal termination. Additionally the B-tree free space system uses less memory resources than the memory free space system.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().freespace().useBTreeSystem();
```

FreeSpaceConfiguration.java: Use BTree system

Freespace Filler

When you delete a object in db4o the storage it consumed isn't deleted. Instead only the storage space is marked as free and can be reused. Therefore it's possible to read also the content of deleted objects.

If you want to avoid that, you can specify a free-space filler. This filler is responsible to overwrite the free-space.

Note that this costs performance, since additional IO operations are performed.

```
class MyFreeSpaceFiller implements FreespaceFiller {
    @Override
    public  void fill(BlockAwareBinWindow block) throws IOException {
        byte[] emptyBytes = new  byte[block.length()];
        block.write(0,emptyBytes);
    }
}
```

MyFreeSpaceFiller.java: The freespace filler

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().freespace().freespaceFiller(new MyFreeSpaceFiller());
```

FreeSpaceConfiguration.java: Using a freespace filler

Generate UUIDs

db4o can generate UUIDs for each stored object. These UUIDs are mainly used for replication together with commit timestamps. Of course it can be used also for other purposes.

Enable UUIDs for all objects.

You can enable UUIDs for all objects. Set the global scope on the UUID setting.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().generateUUIDs(ConfigScope.GLOBALLY);
```

FileConfiguration.java: Enable db4o uuids globally

Enable UUIDs for certain classes

You can also enable uuids only for certain classes:

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().generateUUIDs(ConfigScope.INDIVIDUALLY);
configuration.common().objectClass(SpecialClass.class).generateUUIDs(true);
```

FileConfiguration.java: Enable db4o uuids for certain classes

Generate Commit Timestamps

db4o can store transaction timestamps. Those timestamps to compare and check if an objects has been changed. These commit timestamps are mainly used for replication together with UUIDs.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().generateCommitTimestamps(true);
```

FileConfiguration.java: Enable db4o commit timestamps

Lock Database File

You can disable the database lock. This is useful for two some scenarious. For example accessing a pure-readonly database at the same time. Or for increasing the performance on some small embedded devices.

However this risks the database integrity. **Never access the database file at the same time with multiple object containers!**

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().lockDatabaseFile(false);
```

FileConfiguration.java: Disable the database file lock

Read Only

You can set db4o to a read only mode. In this mode db4o won't do any changes to the database file. This mode is optimal for reading the database without doing some accidental changes to it.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().readOnly(true);
```

FileConfiguration.java: Set read only mode

Recovery Mode

This option only makes sense when you try to recover a corrupted database.

Turns recovery mode on and off. Recovery mode can be used to try to retrieve as much as possible out of an corrupted database. In recovery mode internal checks are more relaxed. Null or invalid objects may be returned instead of throwing exceptions. Use this method with care as a last resort to get data out of a corrupted database.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().recoveryMode(true);
```

FileConfiguration.java: Enable recovery mode to open a corrupted database

Reserve Storage Space

Reserves a number of bytes in database files. Without this setting storage space will be allocated continuously as necessary.

The allocation of a fixed number of bytes at one time makes it more likely that the database will be stored in one chunk on the mass storage. This will result in less read/write head movement on disk based storage. Note: Allocated space will be lost on abnormal termination of the database engine (hardware crash, VM crash). A Defragment run will recover the lost space. For the best possible performance, this method should be called before the Defragment run to configure the allocation of storage space to be slightly greater than the anticipated database file size.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().reserveStorageSpace(1024*1024);
```

FileConfiguration.java: Reserve storage space

Storage

db4o allows a user to configure the IO storage mechanism to be used. Currently db4o provides the following mechanisms:

- FileStorage: Raw file access.
- CachingStorage: A caching layer, used on top of other storages.

- MemoryStorage and PagingMemoryStorage: For in memory databases.
- NonFlushingStorage: A storage which disables the costly flush operation

It is also possible to create your own custom mechanism by implementing Storage interface. Possible use cases for a custom IO mechanism:

- Improved performance with a native library.
- Mirrored write to two files.
- Encryption of the database file.
- Read-on-write fail-safety control.

FileStorage

FileStorage is the base storage mechanism, providing the functionality of file access. The benefit of using FileStorage directly is in decreased memory consumption.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
Storage fileStorage = new FileStorage();
configuration.file().storage(fileStorage);
ObjectContainer container = Db4oEmbedded.openFile(configuration, "database.db4o");
```

IOConfigurationExamples.java: Using the pure file storage

Without cache, the file storage is significantly slower than with cache. Therefore this storage is normally used as underlying storage for other purposes. Typically it is used together with a CachingStorage on top of it:

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
Storage fileStorage = new FileStorage();
// A cache with 128 pages of 1024KB size, gives a 128KB cache
Storage cachingStorage = new CachingStorage(fileStorage,128,1024);
configuration.file().storage(cachingStorage);
ObjectContainer container = Db4oEmbedded.openFile(configuration, "database.db4o");
```

IOConfigurationExamples.java: Using a caching storage

Memory Storage

The MemoryStorage allows you to create and use a db4o database fully in RAM. This strategy eliminates long disk access times and makes db4o much faster.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
MemoryStorage memory = new MemoryStorage();
configuration.file().storage(memory);
ObjectContainer container = Db4oEmbedded.openFile(configuration, "database.db4o");
```

IOConfigurationExamples.java: Using memory-storage

MemoryStorage can be created without any additional parameters passed to the constructor. In this case default configuration values will be used.

PagingMemoryStorage

The regular MemoryStorage implementation keeps all the content in a single byte-array. However this brings some issues. When the database outgrows the array-size, a new, larger array is created and the

content is copied over. This can be quite slow. Also can cause this a out of memory exception, because during the copying these two large arrays are present. Also, on some runtimes large objects are treated different by the garbage-collector and are less often collected.

To avoid all this issues, the PagingMemoryStorage uses multiple, small arrays to keep the database in memory. When the database outgrows the storage, only such a smaller arrays needs to be allocated. The old content stays in the existing arrays. No coping is required.

However managing these arrays cost some small overhead. But for lots of cases, the PagingMemoryStorage is the better choice.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
PagingMemoryStorage memory = new PagingMemoryStorage();
configuration.file().storage(memory);
ObjectContainer container = Db4oEmbedded.openFile(configuration, "database.db4o");
```

IOConfigurationExamples.java: Using paging memory-storage

Growth Strategy for MemoryStorage

Growth strategy defines how the database storage (reserved disk or memory space) will grow when the current space is not enough anymore.

DoublingGrowthStrategy - default setting. When the size of the database is not enough, the reserved size will be doubled.

ConstantGrowthStrategy - a configured amount of bytes will be added to the existing size when necessary.

```
GrowthStrategy growStrategy = new ConstantGrowthStrategy(100);
MemoryStorage memory = new MemoryStorage(growStrategy);
configuration.file().storage(memory);
```

IOConfigurationExamples.java: Using memory-storage with constant grow strategy

MemoryBin

Each memory storage can contain a collection of memory bins, which are actually just names memory storages. You can reuse the MemoryBin created earlier for this MemoryStorage. MemoryBins are identified by their URI, i.e. when an object container is opened with:

Java:

```
Db4oEmbedded.openFile(embeddedConfiguration, "myEmbeddedDb.db4o");
```

A MemoryBin with URI = "myEmbeddedDb.db4o" will be used. If this memory bin does not exist in the storage when the container is opened, a new MemoryBin will be created and associated with this URI. When you pass the same memory storage to multiple object containers these containers can access to the same in memory file when they are using the same name.

More Reading:

- Storing MemoryBin Data

Storing MemoryBin Data

You can use db4o backup functionality to backup your in memory container. See "Backup" on page 68

```
container.ext().backup(new FileStorage(),"advanced-backup.db4o");
```

BackupExample.java: Store a backup with storage

CachingStorage

The CachingStorage is db4o's default storage. The default implementation uses the LRU/Q2 caching mechanism to reduce disk access times and to improve performance. The cache is characterized by the amount of pages that can be utilized and the page size. The multiplication of these two parameters gives the maximum cache size that can be used.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
Storage fileStorage = new FileStorage();
// A cache with 128 pages of 1024KB size, gives a 128KB cache
Storage cachingStorage = new CachingStorage(fileStorage,128,1024);
configuration.file().storage(cachingStorage);
ObjectContainer container = Db4oEmbedded.openFile(configuration, "database.db4o");
```

IOConfigurationExamples.java: Using a caching storage

Page Count and Page Size

The CachingStorage takes two parameters, the page count and the page size. Bigger page size means faster handling of information as there is no need to switch between pages for longer. On the other hand a bigger page size will mean higher memory consumption, as memory will be reserved for the whole page size, when the page is needed. Modify these values and run performance tests to find the best performance/memory consumption combination for your system. The default values are the following:

Page count = 64 Page size = 1024KB

This gives a total of 64 KB of cache memory.

Caching Type

By default db4o uses a LRU/Q2 caching strategy. You can more about the LRU/Q2 cache on the Internet or you can look for the concrete implementation in db4o source code: LRU2QCache, LRU2QXCache and LRUCache. The LRU2QCache is a simplified implementation of the 2 Queue algorithm described here:http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1..34.2641

It's possible to exchange the cache-type. Inherit from the CachingStorage class and overwrite the new cache method. There you return you're caching implementation. For example use a simple LRU-Cache.

```
public  class LRUCachingStorage extends CachingStorage {
    private  final  int pageCount;

    public LRUCachingStorage(Storage storage) {
        super(storage);
        this.pageCount = 128;
    }

    public LRUCachingStorage(Storage storage, int pageCount, int pageSize) {
        super(storage, pageCount, pageSize);
        this.pageCount = pageCount;
    }

    @Override
    protected Cache4<Long, Object> newCache() {
        return CacheFactory.newLRUCache(pageCount);
    }
}
```

LRUCachingStorage.java: Exchange the cache-implementation

NonFlushingStorage

NonFlushingStorage is a special IO Storage, which can be used to improve commit performance. Committing is a complex operation and requires flushing to the hard drive after each stage of commit. This is necessary as most operating system try to avoid the overhead of disk access by caching disk write data and only flushing the resulting changes to the disk. In the case of db4o commit it would mean that the physical write of some commit stages will be partially skipped and the data will be irreversibly lost.

However, physical access to the hard drive is a time-consuming operation and may considerably affect the performance. That is where NonFlushingStorage comes in: it allows the operating system to keep commit data in cache and do the physical writes in a most performant order. This may sound very nice, but in fact a system shutdown while the commit data is still in cache will lead to the database corruption.

The following example shows how to use the NonFlushingStorage. You can run it and see the performance improvement on commit stage.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
Storage fileStorage = new FileStorage();
// the non-flushing storage improves performance, but risks database corruption.
Storage cachingStorage = new NonFlushingStorage(fileStorage);
configuration.file().storage(cachingStorage);
ObjectContainer container = Db4oEmbedded.openFile(configuration, "database.db4o");
```

IOConfigurationExamples.java: Using the non-flushing storage

Please, remember, that NonFlushingStorage is potentially dangerous and any unexpected system shutdown may corrupt your database. Use with caution and avoid using in production environments.

Pluggable Storage

Pluggable nature of db4o Storage adapters allows you to implement any persistent storage behind the database engine. It can be memory, native IO, encrypted storage, mirrored storage etc.

The implementation is guided by 2 interfaces:

Java:

`Storage` and `Bin`

Storage/IStorage interface defines the presence of this particular storage implementation and Bin/IBin implements actual physical access to the information stored. To simplify the implementation db4o provides StorageDecorator and BinDecorator classes with the base functionality, that can be extended/overridden.

To sort out the details of the implementation let's look at an example.

Logging Storage

In this example we will implement a simple file base storage, which will log messages about each IO operation. In the implementation you can see that most of the functionality is derived from StorageDecorator and BinDecorator classes with additional logging added:

```
LoggingStorage.java
/**//* Copyright (C) 2004 - 2009  Versant Corporation http://www.versant.com */
package com.db4odoc.Storage;

import java.util.logging.*;

import com.db4o.ext.*;
import com.db4o.io.*;

public class LoggingStorage extends StorageDecorator  {

  public LoggingStorage() {
    // delegate to a normal file storage
    this(new FileStorage());
  }

  public LoggingStorage(Storage storage)  {
    // use submitted storage as a delegate
    super(storage);
  }

  /** *//**
   * opens a logging bin for the given URI.
   */
  @Override
  public Bin open(BinConfiguration config) throws Db4oIOException  {
    final Bin storage = super.open(config);
    if (config.readOnly())  {
      return new ReadOnlyBin(new LoggingBin(storage));
    }
    return new LoggingBin(storage);
  }

  /** *//**
   * LoggingBin implementation. Allows to log information
   * for each IO operation
   *
   */
  static class LoggingBin extends BinDecorator  {

    public LoggingBin(Bin bin)  {
      super(bin);
    }
```

```
    // delegate to the base class and log a message
    public void close() throws Db4oIOException  {
      super.close();
      Logger.getLogger(this.getClass().getName()).log(Level.INFO,
          "File closed");
    }

    // log a message, then delegate
    public int read(long pos, byte[] buffer, int length)
        throws Db4oIOException  {
      Logger.getLogger(this.getClass().getName()).log(
          Level.INFO,
          String.format("Reading %d bytes and %d position", length,
              pos));
      return super.read(pos, buffer, length);

    }

    // log a message, then delegate
    public void write(long pos, byte[] buffer, int length)
        throws Db4oIOException  {
      Logger.getLogger(this.getClass().getName()).log(
          Level.INFO,
          String.format("Writing %d bytes and %d position", length,
              pos));
      super.write(pos, buffer, length);
    }

    // log a message, then delegate
    public void sync() throws Db4oIOException  {
      Logger.getLogger(this.getClass().getName()).log(
          Level.INFO,"Syncing");
      super.sync();
    }

  }
}
```

Use the LoggingStorage implementation with the following code (you can find a working example if you download LoggingStorage class).

Java:

```
config.file().storage(new LoggingStorage());
```

Client Configuration

The client-configuration applies only to the db4o client. All the client specific configuration is directly on the client configuration interface.

You can also configure the common-and the network-settings for a client.

Overview

Here's a overview over all client-settings which you can change:

MessageSender: Gives access to the message-sender for sending messages to the server.

PrefetchDepth: Configures how depth the object-graph is prefetched from the server.
PrefetchIDCount:Configures how many object ids are prefetched for new objects from the server.
PrefetchObjectCount: Configures how many objects are prefetched from the server.
PrefetchSlotCacheSize: Configures the slot-cache size.
TimeoutClientSocket: Configures the connection timeout.

Message Sender

This gives you access to the message sender, which allows you to send messages to the server. See "Messaging" on page 171

Prefetch Depth

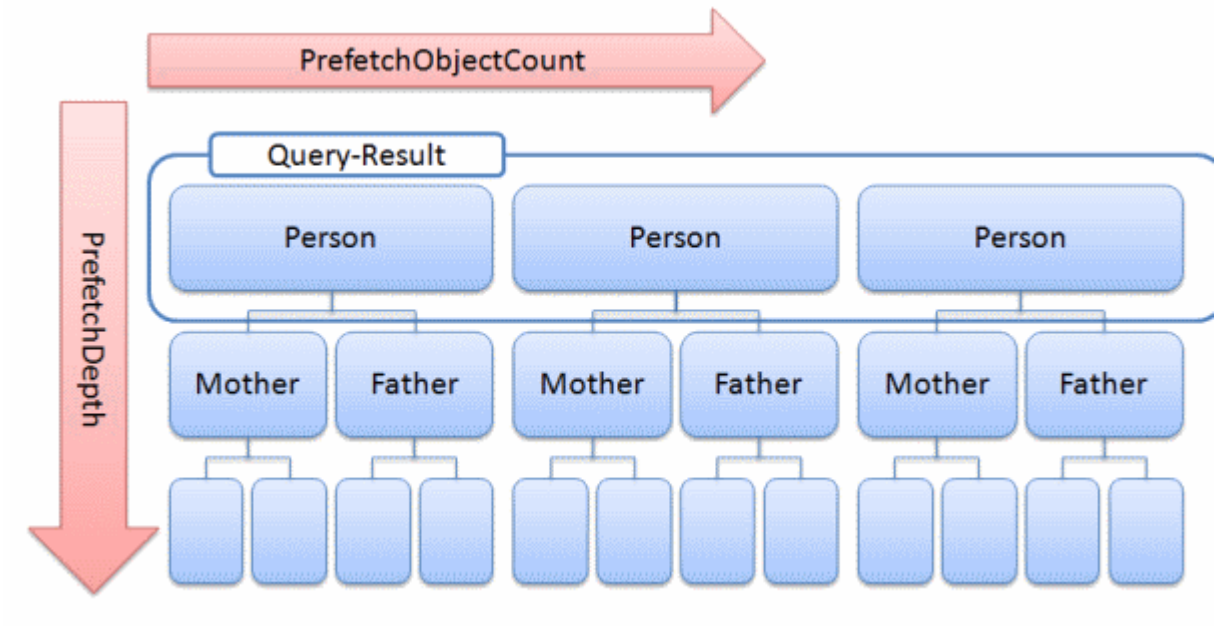
In a client server scenario, network latency is one of the biggest performance concerns. Instead of making lots of little requests, it's more to do bulk operations. A good way is to prefetch objects which are maybe required later.

The prefect depth influences to which depth the object-graph is loaded from the server when query for objects. Prefetched objects avoid additional roundtrip's to the server for getting the data. However more data needs to be sent to the clients.

```
ClientConfiguration configuration = Db4oClientServer.newClientConfiguration();
configuration.prefetchDepth(5);
```

ClientConfigurationExamples.java: Configure the prefetch depth

The prefetch depth and the prefetch object count are closely related to each other. The prefetch object count configures how many objects are prefetched from a query-result. The prefetch-depth configures how deep the object-graph is fetched.



Prefetch Object Count

In a client server scenario, network latency is one of the biggest performance concerns. Instead of

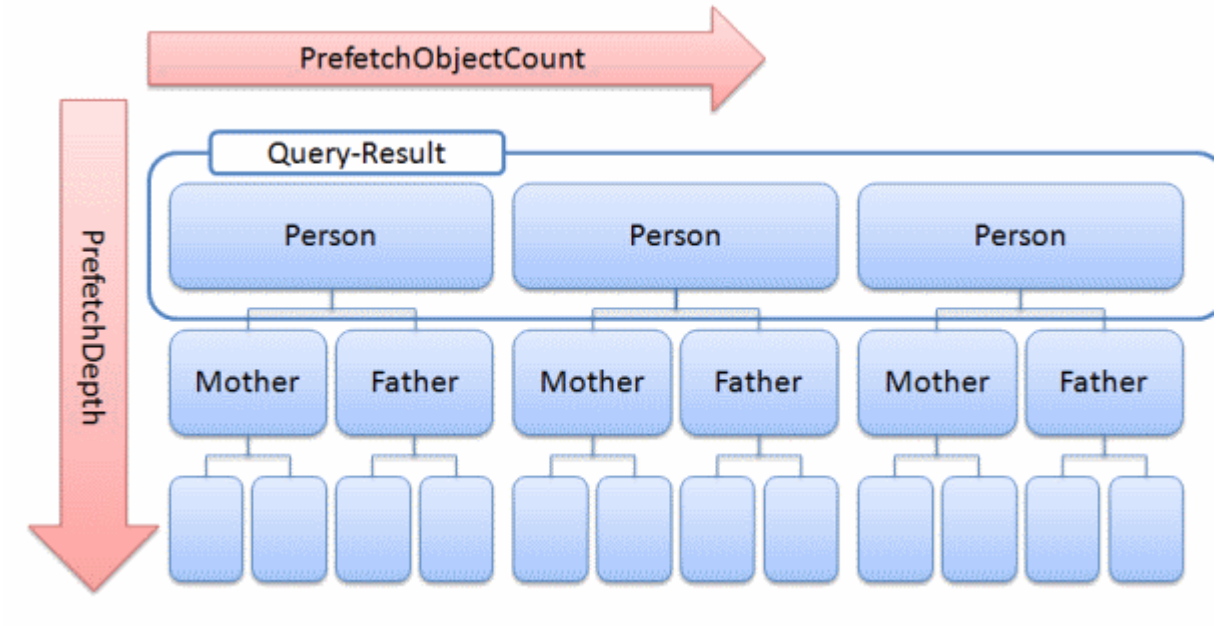
making lots of little requests, it's more to do bulk operations. A good way is to prefetch objects which are maybe required later.

The prefect count configures how many objects of a query-result are loaded from the server. Prefetched objects avoid additional roundtrip's to the server for getting the data. However more data needs to be sent to the clients.

```
ClientConfiguration configuration = Db4oClientServer.newClientConfiguration();
configuration.prefetchObjectCount(500);
```

ClientConfigurationExamples.java: Configure the prefetch object count

The prefetch depth and the prefetch object count are closely related to each other. The prefetch object count configures how many objects are prefetched from a query-result. The prefetch-depth configures how deep the object-graph is fetched.



Prefetch Slot Cache Size

Configures how big the cache for prefetched data is. A larger cache can improve performance, but consumes more memory.

```
ClientConfiguration configuration = Db4oClientServer.newClientConfiguration();
configuration.prefetchSlotCacheSize(1024);
```

ClientConfigurationExamples.java: Configure the slot cache

Prefetching IDs For New Objects

Sets the number of IDs to be pre-allocated new objects created on the client. This avoids requests to allocate new ids when new objects are stored on a client.

When a new object is created on a client, the client should contact the server to get the next available object ID. PrefetchIDCount allows to specify how many IDs should be pre-allocated on the server and prefetched by the client. This method helps to reduce client-server communication.

PrefetchIDCount can be tuned to approximately match the usual amount of objects created in one operation to improve the performance.

When the PrefetchIDCount is one, the client will have to connect to the server for each new objects created. On the other hand,, when the PrefetchIDCount is bigger than the amount of new objects the database will keep unnecessary preallocated ids.

The default PrefetchIDCount is 10.

Timeout Client Socket

Configure how long it takes, before an inactive connection is timed out.

```
ClientConfiguration configuration = Db4oClientServer.newClientConfiguration();
configuration.timeoutClientSocket(1*60*1000);
```

ClientConfigurationExamples.java: Configure the timeout

Networking Configuration

The networking-configuration apply to the client- and the server-mode of db4o. All the networking configuration is accessible via the networking-getter on the configuration-object.

Note that the networking configuration should be the same on the server and all clients.

Overview

Here's a overview over all networking configuration-settings which you can change:

BatchMessages: Enable/Disable batch messages.
MessageRecipient: Set the message receiver.
ClientServerFactory: Replace the client-server factory.
MaxBatchQueueSize: Set the maximum size of the batch queue.
SingleThreadedClient: Set the client to single threaded.
SocketFactory: Set the socket-factory.

Batch Mode

Batch mode allows to increase the performance by reducing client/server communication. It's activated by default.

```
ClientConfiguration configuration = Db4oClientServer.newClientConfiguration();
configuration.networking().batchMessages(true);
```

NetworkConfigurationExample.java: enable or disable batch mode

db4o communicates with the server by means of messaging. Without batch mode db4o sends a message for each operation and waits for the response. This might be quite inefficient when there are many small messages to be sent (like bulk inserts, updates, deletes). The network communication becomes a bottleneck. Batch messaging mode solves this problem by caching the messages and sending them only when required.

Advantages

- Reduced network load.
- Increased performance for bulk operations.

Disadvantages

- Increased memory consumption on both the client and the server. See "Max Batch Queue Size" on page 157

Client Server Factory

Allows you to change the behavior how a client or a server is created.

This setting should be set to the same value on the server and the clients.

```
ClientConfiguration configuration = Db4oClientServer.newClientConfiguration();
configuration.networking().clientServerFactory(new StandardClientServerFactory());
```

NetworkConfigurationExample.java: exchange the way a client or server is created

Message Recipient

Register a message recipient for the this object container / server. See "Messaging" on page 171

Max Batch Queue Size

To avoid unnecessary network traffic, the client and server can queue up operations, which don't need to be executed immediately. As soon as a message need to be send, all queued messages are send as well. You can configure the maximum of this message queue.

A larger size allows to queue up more messages and avoid unnecessary network roundtrip's. However queued up messages consume additional memory.

```
ClientConfiguration configuration = Db4oClientServer.newClientConfiguration();
configuration.networking().maxBatchQueueSize(1024);
```

NetworkConfigurationExample.java: change the maximum batch queue size

Single Threaded Client

You can configure the client to be single-threaded. When you enable this option, the client doesn't use background threads to handle the client-server communication.

```
ClientConfiguration configuration = Db4oClientServer.newClientConfiguration();
configuration.networking().singleThreadedClient(true);
```

NetworkConfigurationExample.java: single threaded client

Advantage

On some smaller embedded systems reducing the running threads improves the performance significantly.

Disadvantage

Since all operations run in a single thread, the operations may take longer. Additionally you cannot receive messages and cannot use commit-callbacks on a single-threaded client.

Pluggable Sockets

db4o allows to customize client-server communication by using pluggable socket implementations.

```
ClientConfiguration configuration = Db4oClientServer.newClientConfiguration();
configuration.networking().socketFactory(new StandardSocket4Factory());
```

NetworkConfigurationExample.java: Exchange the socket-factory

One use case for changing the socket-implementation is encryption. In fact, db4o's SSL-support uses this mechanism: See "Using SSL For Client-Server Communication" on page 172

Class Specific Configuration

Some settings are object-specific and are configured for the class. It's part of the common-configuration, which is available on the client, server and embedded-mode of db4o.

Its recommended that you use the same configuration for the client and the server.

Access the Class Configuration

The configuration for a specific class follows always the same pattern. First you specify for which type the configuration applies. You pass the type, the name as string or event an instance of the specific class to the configuration.

From the class-configuration, you also can go a level deeper to the field configuration.

Overview

Here's a overview over all common configuration-settings which you can change:

CallConstructor: Configure db4o to call constructors when instantiating objects.
CascadeOnDelete: When a object is deleted, delete also referenced objects.
CascadeOnUpdate: When a object is updated, update also referenced objects.
CascadeOnActivation: When a object is activated, activate also referenced objects.
Index: Don't index the objects of this type.
~~**EnableReplication**~~: Deprecated. Generate uuids and commit timestamps to enable replication.
GenerateUUIDs: Generate UUIDs, mainly used for replication.
~~**GenerateVersionNumbers**~~: Deprecated. Generate commit timestamps instead.
MaximumActivationDepth: Set a maximum activation-depth.
MinimumActivationDeph: Set a minimum activation-depth.
PersistStaticFields: Persist also the static fields of this type.
Rename: Rename this type. Used for refactorings.
Translate: Set a translator for this type.
StoreTransientFields: Store also transient fields.
UpdateDepth: Set the update-depth for this type.

Call Constructor

By default db4o bypassed the constructor when it loads the objects from the database. When you're class relies on the constructor to be called, for example to initialize transient state, you can enable it. You can do this also on the global configuration for all types. See "Calling Constructors" on page 135

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).callConstructor(true);
```

ObjectConfigurationExamples.java: Call constructor

Cascade on Delete

When turned on the deletion cascades to all referred objects.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).cascadeOnDelete(true);
```

ObjectConfigurationExamples.java: When deleted, delete also all references

Cascade on Update

When turned on, the update cascades to all referred objects.
Note: As soon as you use transparent persistence this configuration option has no effect and isn't needed.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).cascadeOnUpdate(true);
```

ObjectConfigurationExamples.java: When updated, update also all references

Cascade on Activate

db4o uses the concept of activation to avoid loading to much data into memory. When turned on the activation cascades for this types. This means that when an instance is activated, all referenced objects are also activated.
Note: As soon as you use transparent activation/persistence this configuration option has no effect.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).cascadeOnActivate(true);
```

ObjectConfigurationExamples.java: When activated, activate also all references

Disable Class Index

By default there's a index which indexes all instances of a certain type. When you never query this type and only access it by navigation, you can safely disable this index.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).indexed(false);
```

ObjectConfigurationExamples.java: Disable class index

Generate UUIDs

Generate UUIDs for instances of this type. UUIDs are mainly used for replication.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).generateUUIDs(true);
```

ObjectConfigurationExamples.java: Generate uuids for this type

Maximum Activation Depth

Sets the maximum activation depth for this type. Useful to avoid accidentally activating large objects.
Note: As soon as you use transparent activation/persistence this configuration option has no effect.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).maximumActivationDepth(5);
```

ObjectConfigurationExamples.java: Set maximum activation depth

Minimum Activation Depth

Sets the minimum activation depth for this type. Useful to load always load referenced objects.
Note: As soon as you use transparent activation/persistence this configuration option has no effect.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).minimumActivationDepth(2);
```

ObjectConfigurationExamples.java: Set minimum activation depth

Persist Static Fields

By default db4o doesn't store static fields. With this setting you force db4o to also store the value of the static fields. Note that only reference types are stored. Value-types like ints, longs etc are not persisted.

In general you shouldn't store static fields, because this can lead to unexpected behaviors. It's mostly useful for enumeration classes. See "Storing Static Fields" on page 104

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).persistStaticFieldValues();
```

ObjectConfigurationExamples.java: Persist also the static fields

Rename Class

You can rename a class. Useful when you reactor you're code. See "Refactoring and Schema Evolution" on page 110

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).rename("com.db4odoc.new.package.NewName");
```

ObjectConfigurationExamples.java: Rename this type

Set a Translator

You can specify a special translator for this type. A translator is a special way to add you're custom serialization for a type. See "Translators" on page 106

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).translate(new TSerializable());
```

ObjectConfigurationExamples.java: Use a translator for this type

Store Transient Fields

By default db4o doesn't store transient fields. With this setting you can force db4o to event store transient fields.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).storeTransientFields(true);
```

ObjectConfigurationExamples.java: Store also transient fields

Update Depth

By default db4o only stores changes on the updated object, but not the changes on referenced objects. With a higher update-depth db4o will traverse along the object graph to a certain depth and update all objects. See "Update Concept" on page 47. You can also specify this globally.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).updateDepth(2);
```

ObjectConfigurationExamples.java: Set the update depth

Field Specific Configuration

Some settings are field-specific. It's part of the object-configuration, which is available on the client, server and embedded-mode of db4o.

Its recommended that you use the same configuration for the client and the server.

Access the Field Configuration

The configuration for a field follows the same pattern. First you specify for which type this configuration applies. You pass the type or the name as string. Then you navigate to the specific field by passing the field name as string.

Adding a Field Index

Index dramatically speed up queries. You should index all fields which you run queries on. See "Indexing" on page 63

```
@Indexed
private String zipCode;
```

City.java: Index a field

As alternative you can externally configure a field index:

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).objectField("name").indexed(true);
```

ObjectFieldConfigurations.java: Index a certain field

As an alternative you also can use the appropriate Annotation on the field which you want to index.

Cascade On Activate

db4o uses the concept of activation to avoid loading to much data into memory. When this setting is turned on, the object referenced by this field is activated, when the object is activated.
Note: As soon as you use transparent activation/persistence this configuration option has no effect.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).objectField("father").cascadeOnActivate(true);
```

ObjectFieldConfigurations.java: When activated, activate also the object referenced by this field

Cascade On Update

When the object is updated, the object referenced by this field is also updated.
Note: As soon as you use transparent persistence this configuration option has no effect and isn't needed.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).objectField("father").cascadeOnUpdate(true);
```

ObjectFieldConfigurations.java: When updated, update also the object referenced by this field

Cascade On Delete

When the object is deleted, the object referenced by this field is also deleted

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).objectField("father").cascadeOnDelete(true);
```

ObjectFieldConfigurations.java: When deleted, delete also the object referenced by this field

Rename Field

Allows you to rename this field. . See "Refactoring and Schema Evolution" on page 110

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().objectClass(Person.class).objectField("name").rename("sirname");
```

ObjectFieldConfigurations.java: Rename this field

Id System

The id-system configuration applies to the embedded- and the server-mode of db4o. There are tree different id-systems-available. All the id system configuration is accessible via the id-system-getter on the configuration-object.

Note that you cannot change the id-system for an existing database. You need to defragment the database in order to change the id-system.

The id-system is responsible to mapping a object id to the physical location of an object. This mapping can have significant impact on the performance.

Stacked BTree Id-System

This setting uses a stack of two BTree's on top of an InMemoryIdSystem. This system is scalable for a large number of ids.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.idSystem().useStackedBTreeSystem();
```

IdSystemConfigurationExamples.java: Use stacked B-trees for storing the ids

Single BTree Id-System

This setting uses a single BTree on top of a in memory id system. This system works great for small databases. However it cannot scale for a large number of ids.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.idSystem().useSingleBTreeSystem();
```

IdSystemConfigurationExamples.java: Use a single B-tree for storing the ids.

In Memory Id-System

This id-system keeps all ids in memory. While accessing the ids if fast, all ids have to be written to disk on every commit. Therefore it can be used only for tiny databases.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.idSystem().useInMemorySystem();
```

IdSystemConfigurationExamples.java: Use a in-memory id system

Pointer Based Id-System

This id system uses pointers to handle ids. Each id represents a pointer into the database-file. This makes the id-mapping simple. However since it's a pointer, you cannot change the location. Therefore this system leads to more fragmentation and performance degradation as the database grows.

This id system is here to ensure backward-compatibility. It's not recommended to use for new databases.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.idSystem().useInMemorySystem();
```

IdSystemConfigurationExamples.java: Use a in-memory id system

Custom Id-System

It's possible to implement your own id system. You can pass an factory which creates your id-system implementation.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.idSystem().useCustomSystem(new CustomIdSystemFactory());
```

IdSystemConfigurationExamples.java: use a custom id system

Server Configuration

The server-configuration applies to the db4o-server. All the server specific configuration is directly on the client configuration interface.

You can also configure the common-, file-, id-system- and the network-settings for a server.

Server Socket Timeout

Configure how long it takes, before an inactive connection is timed out.

```
ServerConfiguration configuration = Db4oClientServer.newServerConfiguration();
configuration.timeoutServerSocket(10*60*1000);
```

ServerConfigurationExamples.java: configure the socket-timeout

Client-Server

db4o's is an embedded database which usually runs in process. Nevertheless db4o also supports a client server setup. There's no ready to use db4o server, instead you embedded the db4o server in a regular application. This gives you full how you want to run the db4o server.

In order to use the db4o client server mode you need to include the db4o-X.XX-cs-javaC.jar if your not using the db4o-X.XX-all-javaX.jar Db4objects.Db4o.CS.dll-assembly in your project. See "Dependency Overview" on page 252

Start The Server

You start the server by creating a object server instance with the client server factory. Pass the database file path and the port to the factory. After that you need to specify a user-name and password which clients can use to login to this server. You can add multiple users by calling the grant access method multiple times.

```
ObjectServer server = Db4oClientServer.openServer("database.db4o",8080);
try{
    server.grantAccess("user","password");

    // Let the server run.
    letServerRun();
} finally {
    server.close();
}
```

ServerExample.java: Start a db4o server

Connect To The Server

After the server is up an running you can connect to it. Pass the URL, port, user name and password to the client server factory. This will return a client object container. After that the container is ready to use.

```
final ObjectContainer container
        = Db4oClientServer.openClient("localhost", 8080, "user", "password");
try{
    // Your operations
}finally {
    container.close();
}
```

Db4oClientExample.java: Connect to the server

Reference Cache In Client-Server Mode

db4o uses an object reference cache for easy access to persistent objects during one transaction. In client/server mode each client has its own reference cache, which helps to achieve good performance. However it gets complicated, when different clients work on the same object, as this object's cached value is used on each side. It means, that even when the operations go serially, the object's value won't be updated serially unless it is refreshed before each update.

```
Person personOnClient1 = queryForPerson(client1);
Person personOnClient2 = queryForPerson(client2);
System.out.println(queryForPerson(client2).getName());

personOnClient1.setName("New Name");
client1.store(personOnClient1);
client1.commit();

// The other client still has the old data in the cache
System.out.println(queryForPerson(client2).getName());

client2.ext().refresh(personOnClient2,Integer.MAX_VALUE);

// After refreshing the date is visible
System.out.println(queryForPerson(client2).getName());
```

ReferenceCacheExamples.java: Reference cache in client server

There are multiple strategies to deal with this.

- If you client's are not updating the same object or it very unlikely that it happens, you don't need to take any action.
- You can refresh objects as they are updated on all clients. However this need a lot of communication between the server and client. See "Refreshing Objects" on page 165
- You can do short unit of work on the client, to minimize the chance of outdated objects.

Clean Cache For Work

Often you don't want to refresh object by object. Instead you want to work with a clean cache. You can do this by using a separate 'session' on the client. This container brings its own cache with it. So you can create a new container which a clean cache.

Note that open session on the client doesn't create a separate transaction. Instead it only creates a fresh cache. The transaction is always the same.

```
ObjectContainer container = client.ext().openSession();
try{
    // do work
}finally {
    container.close();
}
// Start with a fresh cache:
container = client.ext().openSession();
try{
    // do work
}finally {
    container.close();
}
```

ReferenceCacheExamples.java: Clean cache for each unit of work

Refreshing Objects

When working with multiple client object-containers each container has its own reference cache. When you query for objects, you get always the cached objects. This means, that objects probably have been updated in the mean-time but the client still sees the old state.

For some scenarios you might need to refresh to objects to bring it up to date. There are two strategies for this. You can explicit refresh a object at any time. Or you can use callbacks to refresh object on each commit. Both methods have their advantage.

	Explicit Refreshing	Using Callbacks
Advantage	• Selective refreshing possible, which reduces network traffic. • Can decide when a refresh is required. If no refresh is required, you can save the network traffic.	• The objects are always up to date. • There's no need to explicitly refresh-calls in your data-access layer.
Disadvantage	• Partial refreshed objects may lead to a inconsistent object-graph. • Probably a lot of refresh-calls spread over the code-base.	• A lot of network traffic is required to send the committing-events and object to the clients.

Explicitly Refreshing Objects

You can refresh objects with the refresh-method on the object-container. The pass the object to refresh and the refresh-depth to the method:

```
db.ext().refresh(objToRefresh,Integer.MAX_VALUE);
```

RefreshingObjects.java: refresh a object

Using Callbacks For Refreshing Objects

You can use the committed-event to refresh objects as soon as another client commits. Take a look at the example: See "Committed Event Example" on page 98

Embedded Client Server

db4o supports embedded containers or session container. It's a separate object-container with its own transaction and own reference cache. See "Session Containers" on page 66

Now if you're using the client-server mode for db4o, you also can create such sessions directly from the server. When you pass a 0 to the open server method, the server only supports embedded clients. With any other port you can connect with regular clients and also create embedded clients.

```
ObjectServer server = Db4oClientServer.openServer(DATABASE_FILE_NAME, 0);

// open the db4o-embedded client. For example at the beginning for a web-request
ObjectContainer container = server.openClient();
try {
    // do the operations on the session-container
    container.store(new Person("Joe"));
} finally {
    // close the container. For example when the request ends
    container.close();
}
```

Db4oSessions.java: Embedded client

OpenSession On A Client

You might noted that the open-session is available on any object-container. Normally this creates a session-container with its own transaction and reference cache. However on a db4o-client this is not true. There it only creates a container with new cache, but shares the transaction with the client. The only use case for this is to implement connection pooling.See "Client-Container Pooling" on page 169

Native Queries

Native Queries are optimized to SODA under the hood. Therefore they can be normally used in a client-server environment. However, there is a catch: as soon as the native query is not optimized there will be two things required on the server:

- The persistent classes definitions will be required because the objects will need to be instantiated to execute the query code.
- The native predicate classwill be required to run on the server to do the actual job.

To meet these requirements you can keep your persistent classes and **NQ**[1] predicates in separate libraries, which will make it easy to deploy it on both the client and the server side. Otherwise you may try to express the unoptimised query in SODA, this option will make it more performant, but the disadvantage is less robust and maintainable code.

Anonymous Classes In Client Server Mode

Anonymous classes are used to implement Native Query predicates, Comparators and Evaluations. In this case it is important to remember that in client/server mode they will be marshaled and sent together with the graph of referenced objects to the server. The catch here is that anonymous classes contain a reference to their parent class, meaning that the parent class will be marshalled as well!

This has two issues:

- All the stuff referenced by the parent class will be marshaled and sent too. This means additional traffic between the server and the client.
- The parent class maybe contains stuff which cannot be serialized.
- You can only use anonymous classes when the client and the server have all classes available refered by the anonymous class. This means in fact that the server should have the same classes available as the client.

Note that when a native query can be optimized, the anonymous class isn't used, but rather SODA. In such cases this issues don't apply.

Server Without Persistent Classes Deployed

In an ordinary Client/Server setup persistent classes are present on both client and server side. However this condition is not mandatory and a server side can work without persistent classes deployed utilizing db4o GenericReflector functionality.

How it works?

---

[1] Native Query

When classes are unknown GenericReflector creates generic objects, which hold simulated "field values" in an array of objects. This is done automatically by db4o engine and does not require any additional settings from your side: you can save, retrieve and modify objects just as usual. An example of this functionality is db4o's Object Manager.

Unfortunately in a server without persistent classes mode there are still some limitations:

- Evaluation queries won't work. This also implies that native queries which cannot be optimized don't work.
- Native queries will only work if they can be optimized
- Multidimensional arrays can not be stored.

Native Queries

Native queries are usually translated to SODA queries. When a native query can be translated to SODA, it will run fine on a server without persisted classes. However when the query cannot be optimized the server needs the native query instance and the data model which isn't available on a server without persistent classes. Therefore native queries only work when it can be optimized. You can fallback to pure SODA queries in such cases.

Events In Client Server-Mode

Events of course also work in client server mode. This topic only applies to the networked client/server mode. Embedded client/server-mode isn't affected. There the events work the same way as in the embedded-mode. See "Event Registry API" on page 92

Separate Event Registry For Each Client

Each client has its own event registry. When you register to a event on the client-event-registry, events will be fired for actions on the client. You won't receive events for actions on other clients.

Furthermore you cannot register for the delete-events, the delete events is only on the server available.

The Server Event Registry

The server has its own event registry. When you register to an event on the server registry, events will be fired for all action of the clients and the server itself. So you can monitor all operations on the server.

Note that in some events the server isn't involved. For example when a client activates some object, the server is no involved, an therefore no event is fired on the server.

Use central server object container to register for the events on the server.

```
ObjectServer server =
        Db4oClientServer.openServer(DATABASE_FILE_NAME, PORT_NUMBER);
EventRegistry eventsOnServer =
        EventRegistryFactory.forObjectContainer(server.ext().objectContainer());
```

EventRegistryExamples.java: register for events on the server

Committed Event On All Parties

The committed event is an exception. When a client or the server commits, every client and the server will fire the committed event. This way a client can inform itself that another client has made changes to the database.

However this involves communication overhead to send the message to all the clients. Therefore you should use this event only when absolutely necessary on the clients.

Events Are Fired Asynchronous

In client-server-mode the events are fired asynchronous. This means that you event-handler is invoked in a separate thread. You need to ensure that you lock any shared data-structure you access from the event-handlers.

Client-Container Pooling

When a client connects to the server, there's overhead to establish the connection. First the regular TCP-connection needs to be established. Then the server and client exchange meta data and finally the connection is ready. If you don't need a single or a few long during connections but rather short units of work on the client it is quite inefficient to open a client connection for each unit of work. In such scenarios you should consider to pool the client-containers.

Now simply pooling the raw client container might lead to issues. Each object-container has a reference-cache. When you pool the object-container and reuse it for some other work, this cache isn't cleared. This means that you might get dirty objects from the reference cache. You want to avoid this and have a clean cache when reusing the client container.

There's a way to archive that. On the client container the open session method creates a fresh object container with a clean reference cache which is sharing the transaction with the client. With this building block you can build a proper container pool.

First you need to create client connections on demand which will be pooled.

```
ObjectContainer client = Db4oClientServer.openClient("localhost",PORT,
                USER_AND_PASSWORD, USER_AND_PASSWORD);
```

ConnectionPoolExamples.java: Open clients for the pool

On a request for a object container, get a client container from the pool. Rollback the transaction on it to ensure that it is in a clean state.Then open a session container on it and use this session. The session-container ensures that the reference-cache is empty. Make sure that each client container is always used only once at any time, which means that there's always only one session-container open per client-container. The session-containers share the transaction with the client and you don't want to share transactions across multiple object containers.

```
// Obtain a client container. Either take one from the pool or allocate a new one
ObjectContainer client = obtainClientContainer();
// Make sure that the transaction is in clean state
client.rollback();
// Then create a session on that client container and use it for the database operations.
// The client-container is now in use. Ensure that it isn't leased twice.
final ObjectContainer sessionContainer = client.ext().openSession();
```

ConnectionPool.java: Obtain a pooled container

Now when you're done with your operations, you can return the client to your pool of object containers. First close the session-container, which commits the changes and releases the resources. Then get the underlying client-container for that session and return it to the pool.

```
// First you need to get the underlying client for the session
final ObjectContainer client = clientForSession(session);
// then the client is ready for reuse
returnToThePool(client);
```

ConnectionPool.java: Returning to pool

Concurrency Control

As soon as you will start using db4o with multiple client connections you will recognize the necessity of implementing a concurrency control system. db4o itself works as an overly optimistic scheme, i.e. an object is locked for read and write, but no collision check is made. This approach makes db4o very flexible and gives you an opportunity to organize a concurrency control system, which will suit your needs the best. Your main tools to build your concurrency control system would be:

- Semaphores
- Callbacks

The articles in this topic set will show you how to implement locking and will give you an advice which strategy to use in different situations.

Semaphores

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

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

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

```
// Grab the lock. Specify the name and a timeout in milliseconds
boolean hasLock = container.ext().setSemaphore("LockName", 1000);
try {
    // you need to check the lock
    if (hasLock) {
        System.out.println("Could get lock");
    } else{
        System.out.println("Couldn't get lock");
    }
} finally {
    // release the lock
    container.ext().releaseSemaphore("LockName");
}
```

SemaphoreExamples.java: Grab a semaphore

Building Block

Semaphores are a low level building block. Usually they are not used directly. Instead you can build the abstractions you need with semaphores. For example you can build object-locking, critical sections etc with semaphores.

Messaging

In client/server it possible to send messages between a client and the server. Possible use cases could be:

- Shutting down and restarting the server.
- Triggering server backup.
- Using a customized login strategy to restrict the number of allowed client connections.
- Running special code on the server. For example batch updates.

Sending and Receiving Messages

First you need to decide on a class that you want to use as the message. Any object that is storable in db4o can be used as a message. Of course you use multiple classes for representing different messages. Let's create a dedicated class.

```
public    class HelloMessage {
    private    final String message;

    public HelloMessage(String message) {
        this.message = message;
    }

    @Override
    public String toString() {
        return message;
    }
}
```

HelloMessage.java: The message class

Now you need to register a handler to handle arriving messages. This can be done by configuring a message recipient on the server. Let's simply print out the arriving message. Additionally we answer to the client with another message.

```
ServerConfiguration configuration = Db4oClientServer.newServerConfiguration();
configuration.networking().messageRecipient(new MessageRecipient() {
    public    void processMessage(MessageContext messageContext, Object o) {
        System.out.println("The server received a '" + o + "' message");
        // you can respond to the client
        messageContext.sender().send(new HelloMessage("Hi Client!"));
    }
});
ObjectServer server = Db4oClientServer.openServer(configuration, DATABASE_FILE, PORT_NUMBER);
```

MessagingExample.java: configure a message receiver for the server

The same can be done on the client. Register a handler for the received messages.

```
ClientConfiguration configuration = Db4oClientServer.newClientConfiguration();
configuration.networking().messageRecipient(new MessageRecipient() {
    public    void processMessage(MessageContext messageContext, Object o) {
        System.out.println("The client received a '" + o + "' message");
    }
});
```

MessagingExample.java: configure a message receiver for a client

Now on the client we can get a message sender. The message sender allows you to send a message to the server. In this example we send a hello message.

```
MessageSender sender = configuration.messageSender();
ObjectContainer container = Db4oClientServer.openClient(configuration, "localhost", PORT_NUMBER, USER_AND_PASSWO


sender.send(new HelloMessage("Hi Server!"));
```

MessagingExample.java: Get the message sender and use it

Using SSL For Client-Server Communication

With the default settings db4o client-server communication is not encrypted and thus can potentially be a dangerous security hole. db4o supports SSL for client server communication. The implementation uses the pluggable socket to provide secure sockets.

The SSL-communication uses the standard Java Secure Socket Extensions, which are part of the normal JRE. You take a look a to full documentation here.

You simply need to add the SSLSupport on the server and the clients and you done. The default-implementation uses the default SSLContext for the client and the server.

```
ServerConfiguration configuration = Db4oClientServer.newServerConfiguration();
configuration.common().add(new SSLSupport());
```

SSLExample.java: Add SSL-support to the server

```
ClientConfiguration configuration = Db4oClientServer.newClientConfiguration();
configuration.common().add(new SSLSupport());
```

SSLExample.java: Add SSL-support to the client

Of course you also can build your own SSLContext with the Java API. After you've build the SSL-Context, you can pass it to the SSLSupport-constructor. Read in the Java documentation how to build a proper SSLContext:

```
// You can build your own SSLContext and use all features of Java's SSL-support.
// To keep this example small, we just use the default-context instead of building one.
SSLContext context = SSLContext.getDefault();


ServerConfiguration configuration = Db4oClientServer.newServerConfiguration();
configuration.common().add(new SSLSupport(context));
```

SSLExample.java: You can build up a regular SSL-Context and us it

Client-Server Timeouts

Every client/server application has to face a problem of network communications. Luckily modern protocols screen the end-application from all fixable problems. However there are still physical reasons that can't be fixed by a protocol: disconnections, power failures, crash of a system on the other end of communication channel etc. In these cases it is still the responsibility of the client-server application to exit the connection gracefully, releasing all resources and protecting data.

In order to achieve an efficient client/server communication and handling of connection problems the following requirements were defined for db4o:

- The connection should not be terminated when both client and server are still alive, even if either of the machines is running under heavy load.
- Whenever a client dies, peacefully or with a crash, the server should clean up all resources that were reserved for the client.
- Whenever a server goes offline, it should be possible for the client to detect that there is a problem.

- Since many clients may be connected at the same time, it makes sense to be careful with the resources the server reserves for each client.
- A client can be a very small machine, so it would be good if the client application can work with a single thread.

The current approach tries to keep things as simple as possible: any connection is closed immediately upon a timeout. In order to prevent closing connections when there is no communication between client and server due to reasons different from connection problems a separate timer thread was created to send messages to the server at a regular basis. The server must reply to the thread immediately, if this does not happen the communication channel gets closed.

This approach works effectively for both client and server side. However there's are small downside to this. When a server operation takes longer than the timeout, the connection will be closed. You can configure the timeouts for the client and the server.

An easy rule of thumb:

- If you experience clients disconnecting, raise the timeout value.
- If you have a system where clients crash frequently or where the network is very instable, lower the values, so resources for disconnected clients are freed faster.

Best Practices

This topic is a collection of best practices for db4o.

db4o doesn't have any support for limiting the result size or skipping objects in the result set. You can do this on top of db4o with little effort. See "Paging" on page 178

How should you scope the lifetime of an object container? See this guide-line: See "Lifetime Of An Object Container" on page 179

Deleting object is always a delicate process. See "Deleting Objects" on page 180

If you upgrade to new major db4o version you should do these steps to update the file format. See "Upgrade db4o Version" on page 179

Managing Relations

In db4o you manage relations by storing references to other objects. In db4o navigational access is usually a lot faster than queries. Therefore think about how you navigate to the right information. This topic gives an overview how to manage different relations.

1-N Relation: Navigating from 1 to N.

As example we use shopping cards which hold items. In most use cases you need to know which items are on a card. For that you navigate from the card (the 1-part) to the items (the N-part). In this case you simply can use a collection on the shopping card which references all items.

Keep in mind that a contains-query on collections is very slow. Finding out which items are on a certain shopping cards is very slow. If you want to know that, take a look at the suggestions below.

```
Set<Item> items = new HashSet<Item>();

public  void add(Item terrain) {
    items.add(terrain);
}

public  void remove(Item o) {
    items.remove(o);
}
```

ShoppingCard.java: Simple 1-n relation. Navigating from the card to the items

1-N Relation: Navigating from N to 1

Imagine that you store people and in which country a person is born. Here you usually navigate from the person (the N-part) to the country (the 1-part). Therefore you can have a field which refers to the country.

In the rare case where you want to know all people born in a certain country you can do a query. When the country reference is indexed, then that query is fast.

```
// Optionally we can index this field, when we want to find all people for a certain country
private Country bornIn;

public Country getBornIn() {
    return bornIn;
}
```

Person.java: Simple 1-n relation. Navigating from the person to the countries

Getting all people of a country is not that hard and fast when the 'bornIn' field is indexed.

```
final Country country = loadCountry(container,"USA");
final ObjectSet<Person> peopleBurnInTheUs = container.query(new Predicate<Person>() {
    @Override
    public  boolean match(Person p) {
        return p.getBornIn() == country;
    }
});
```

RelationManagementExamples.java: Query for people burn in a country

1-N Relation: Bidirectional

When you want to navigate a 1-N relationship bidirectional you can use the method above, because the query if fast enough. Just ensure that you index the field holding the reference.

Alternatively you also can add an additional collection which holds the items. However in that case you need to manually manage the consistency of the relation.

For example the relationship between dogs and their owners. The dog has a field with its owner, while the person has a collection of his dogs. The setters then manage the relationship and ensure that it is always consistent.

```
private Person owner;

// This setter ensures that the model is always consistent
public  void setOwner(Person owner) {
    if(null!=this.owner){
        Person oldOwner = this.owner;
        this.owner = null;
        oldOwner.removeOwnerShipOf(this);
    }
    if(null!=owner && !owner.ownedDogs().contains(this)) {
        owner.addOwnerShipOf(this);
    }
    this.owner = owner;
}

public Person getOwner() {
    return owner;
}
```

Dog.java: Bidirectional 1-N relations. The dog has a owner

```
private Set<Dog> owns = new HashSet<Dog>();

// The add and remove method ensure that the relations is always consistent
public  void addOwnerShipOf(Dog dog) {
    owns.add(dog);
    dog.setOwner(this);
}

public  void removeOwnerShipOf(Dog dog) {
    owns.remove(dog);
    dog.setOwner(null);
}

public Collection<Dog> ownedDogs() {
    return Collections.unmodifiableCollection(owns);
}
```

Person.java: Bidirectional 1-N relations. The person has dogs

N-N Relations: One Way Navigation

Like 1-N relations N-N relations also can be one directional. For example a person can have multiple citizenships in different countries. Let's suppose that you only want to know the citizenship of a person and not the citizens of a country. Then you navigate from people to countries. You can store that in a simple collection.

Keep in mind that a contains-query on a collection is very slow. Finding the people of a certain country will be very slow.

```
private Set<Country> citizenIn = new HashSet<Country>();

public  void removeCitizenship(Country o) {
    citizenIn.remove(o);
}

public  void addCitizenship(Country country) {
    citizenIn.add(country);
}
```

Person.java: One directional N-N relation

N-N Relations: Bidirectional Navigation

For managing bidirectional N-N relations you can use collections, but maintain a collection on both sides. For example a club has a member-list and each member has a list of clubs where he is a member.

Keep in mind that a contains-query on a collection is very slow. That's why you maintain two collections, so that you can navigate to the club-members or club-membership.

```
private Set<Person> members = new HashSet<Person>();

public  void addMember(Person person) {
    if (!members.contains(person)) {
        members.add(person);
        person.join(this);
    }
}

public  void removeMember(Person person) {
    if (members.contains(person)) {
        members.remove(person);
        person.leave(this);
    }
}
```

Club.java: Bidirectional N-N relation

```
private Set<Club> memberIn = new HashSet<Club>();

public  void join(Club club) {
    if (!memberIn.contains(club)) {
        memberIn.add(club);
        club.addMember(this);
    }
}

public  void leave(Club club) {
    if (memberIn.contains(club)) {
        memberIn.remove(club);
        club.removeMember(this);
    }
}

public Collection<Club> memberOf() {
    return Collections.unmodifiableCollection(memberIn);
}
```

Person.java: Bidirectional N-N relation

Paging

Currently db4o doesn't provide any paging mechanism at all. However all db4o query results are lazy loaded. db4o returns a result list which only contains the ids of the objects and will load the object as soon as you access it. This means you can page by only accessing the indexes of the range you're interested in.

Since the result sets of db4o implement the Java List interface it has the sub list method. With this method you easily get only a sub set of the result. Or you can build your own paging-method based on the sub list method on the lists.

```
public   static <T> List<T> paging(List<T> listToPage, int limit){
    return paging(listToPage,0,limit);
}

public   static <T> List<T> paging(List<T> listToPage, int start, int limit){
    if(start>listToPage.size()){
        throw   new IllegalArgumentException("You cannot start the paging outside the list." +
                " List-size: "+listToPage.size()+" start: "+start);
    }
    int end = calculateEnd(listToPage, start, limit);
    return listToPage.subList(start, end);

}

private   static <T> int calculateEnd(List<T> resultList, int start, int limit) {
    int end = start + limit;
    if(end>=resultList.size()){
        return resultList.size();
    }
    return end;
}
```

PagingUtility.java: Paging utility methods

And then of course you can use the utility methods on the result-sets of db4o.

```
final ObjectSet<StoredItems> queryResult = container.query(StoredItems.class);
List<StoredItems> pagedResult = PagingUtility.paging(queryResult, 2, 2);
```

TestPagingUtility.java: Use the paging utility

Upgrade db4o Version

How do you upgrade to a newer version safely? By default db4o can read an older version of a database and operate on it without any issues. However for major version updates you should consider doing this updating procedure. This ensures that the database uses the newest file format and performance optimizations.

The safest update procedure:

1. Defragment the database with the current version. This will automatically also create a backup.
2. Upgrade to the new db4o version.
3. After that immediately defragment the database with the current version. This will create another backup. After that the database file uses the most current database format.
4. Make sure that you can read the data from the database. If everything works you can delete the backups.

Lifetime Of An Object Container

The main interface to your database is the object container. You do all major operations with a object container instance. Now how long should you keep a object container open? Is it better to close the object container after each operation. Or keep the object container open? This topic is a small guideline on the lifetime of an object container.

Understanding Object Containers

The lifetime of a object container heavily depends on your application scenario. Therefore it's vital to understand what a object container represents. A object container consist of a transaction and a reference cache and is thread safe. This means that a object container is isolated from other containers. Furthermore an object container has a cache, so you don't want to throw a object container away if you don't have to.

One Object Container Per Unit Of Work

This means in fact that you should use one object container per unit of work. You can use a object container for succeeding units of work. But you should never use the same object container for concurrent or independent units of work. You can create new object containers at any time.

Desktop Application

What does this mean for a desktop application? Well in a simple desktop application you might can use only one object container. Since there only one user doing something at a time you can use the same object container for all operations.

For complexes desktop application multiple object container can be an option. For example a object container per tab, per wizard etc.

Be aware only because object container is thread safe it doesn't make your object model thread safe. If multiple threads are using the same object container they will use the same objects and modify the same object concurrently. You need to use an appropriate model for the concurrent access.

Web Application

In a web application you should use a object container per request. Each request is it unit of work. Maybe there are multiple unit of work successively in a request, but a request represents the top level unit of work.

Deleting Objects

Deleting object is always a delicate process. Deleting the wrong object can be catastrophic. Here are some best practices for deleting objects.

Delete Flag

When a end user deletes a object it's often better to use a deleted-flag instead of actually deleting the data. This has the advantage that you can undo the delete operation at any time. Also you don't break the model in cases where the user deleted the wrong object. However it has also some disadvantages. You need to honor the deleted-flag in your queries.

You can set the delete flag in a callback and use the regular db4o delete operation:

```
EventRegistry events = EventRegistryFactory.forObjectContainer(container);
events.deleting().addListener(new EventListener4<CancellableObjectEventArgs>() {
    public   void onEvent(Event4<CancellableObjectEventArgs> events,
                          CancellableObjectEventArgs eventArgument) {
        Object obj = eventArgument.object();
        // if the object has a deletion-flag:
        // set the flag instead of deleting the object
        if (obj instanceof Deletable) {
            ((Deletable) obj).delete();
            eventArgument.objectContainer().store(obj);
            eventArgument.cancel();
        }
    }
});
```

DeletionStrategies.java: Deletion-Flag

However you need to filter the deleted objects in every query.

Be Very Careful

db4o doesn't support any referential integrity. When you delete a object and there's still a reference to that object this reference is set to null. This means if you delete a object you may break the consistency of you're object model.

This means also that you need to implement any consistency check yourself on top of db4o. You can use db4o callbacks for doing so.

Use Cascade Deletion Wisely

You can configure db4o the cascade delete referenced objects. You can configure that for certain type or certain fields. As said there's no referential integrity checks for db4o, so you have to extreamly conscious where to use this feature. It makes sense to configure cascade deletion for composition roots, where you are sure that the children cannot be referenced from another location. In all other places it's a bad idea most of the time.

Platform Specific Issues

Db4o can be run in a variety of environments, which have Java virtual machine or .NET CLR. We use a common core code base, which allows automatic production of db4o builds for the following platforms:

- Java JDK 5 or newer
- .NET 3.5 or newer
- .NET 3.5 CompactFramework
- Mono (you need to compile db4o yourself)
- Android
- Silverlight

Db4o has a small database footprint and requires minimum processing resources thus being an excellent choice for embedded use in smartphones, photocopiers, car electronics, and packaged software (including real-time monitoring systems). It also shows good performance and reliability in web and desktop applications.

You can use db4o on desktop with:

- Windows (Java, .Net)
- Linux (Java, Mono).

On mobile and embedded devices with:

- Android
- Windows CE or Windows Mobile (.NET Compact Framework)

db4o provides the same API for all platforms, however each platform has its own features, which should be taken into consideration in software development process. These features will be discussed in the following chapters.

More Reading:

- db4o on Java Platforms
- Security Requirements On Java Platform
- Servlets
- Database For OSGi
- Android

Disconnected Objects

db4o manages objects by object-identity. db4o ensures that each stored object in the database is always represented by the same object in memory. When you load an object, change it and then store it, db4o recognizes the object by its identity and will update it.

This model works wonderful as long as the object-identity is preserved. However there are a lot of scenarios, where the object-identity is lost. As soon as serialize objects, the object-identity is lost. This is typical for web-scenarios or web-services, where a object needs to be identified across requests. For such scenarios objects need additional ids to identify object across requests- and object-container-boundaries.

There are several possibilities for such additional ids. You can use db4o internal ids, db4o uuids, or additional ids-fields on objects. Each has its advantages and disadvantages, so take a looks at this comparison: See "Comparison Of Different IDs" on page 183

Only identifying the object across object container boundaries is often not enough. You actually need to update a disconnected object. This is done by coping the new values to the existing object. See "Merging Changes" on page 187

Comparison Of Different IDs

There are a lot of possibilities for additional ids to identify objects across the object container boundary.

Internal db4o ids

db4o has internal ids to identify each object in the database. You can access these ids and use them yourself. See "Internal IDs". Take a look at the example. See "Example Internal Id" on page 184

Advantages	Disadvantages
• Internal ids are fast. • No additional field on the class required. • No additional configuration required.	• The id may change when defragmentating the database.

db4o UUIDs

db4o supports special UUIDs. You can enable them by configuration. See "Unique Universal IDs". Take a look at the example. See "Example db4o UUID" on page 184

Advantages	Disadvantages
• A UUID is a worldwide unique id. • No additional field on the class required.	• db4o UUIDs are large. • db4o UUID is db4o-specific type.

UUID-Fields on Classes

You can add UUID-fields to your classes. In the constructor of the object you assign a new UUID to the object. Then you can find the object by a regular query. Add UUID support before using UUIDs . Don't forget to index the id-field. Take a look at the example. See "Example UUID" on page 185

Advantages	Disadvantages
• A UUID is a worldwide unique id . • UUID are easy to generate and portable.	• You need an id-field on your objects. • UUIDs are quite large. • Additional index required.

ID-Field On Classes With a ID-Generator

You can add a id-field to your classes and then use an ID-Generator to assign new ids to stored objects. Don't forget to index the id-field. Take a look at the example. See "Example ID-Generator" on page 185

Advantages	Disadvantages

• A simple id on objects. • Familiar model from the **RDBMS**[1]-world.	• You need an id-field on your objects. • You need to implement an ID-Generator. Which isn't trivial. • Additional index required.

Example Internal Id

This example demonstrates how you can use internal object ids to identify objects across objects containers. Take a look advantages and disadvantages of internal ids: See "Comparison Of Different IDs" on page 183

For using the internal ids no additional configuration is required. You can get this id for any object.

You can get the internal id from the extended object container.

```
long interalId = container.ext().getID(obj);
```

Db4oInternalIdExample.java: get the db4o internal ids

Getting a object by its id is also easy. First you get the object from the container. Unlike queries this won't return a activated object. So you have to do it explicitly.

```
long internalId =idForObject;
Object objectForID = container.ext().getByID(internalId);
// getting by id doesn't activate the object
// so you need to do it manually
container.ext().activate(objectForID);
```

Db4oInternalIdExample.java: get an object by db4o internal id

Example db4o UUID

This example demonstrates how you can use db4o-UUIDs to identify objects across objects containers. Take a look advantages and disadvantages of db4o-UUIDs: See "Comparison Of Different IDs" on page 183

First you need to enable db4o-UUIDs in order to use it.

```
configuration.file().generateUUIDs(ConfigScope.GLOBALLY);
```

Db4oUuidExample.java: db4o-uuids need to be activated

With UUIDs turned on, db4o will create an UUID for each stored object. So you can get the UUID of the object from the object-container.

```
Db4oUUID uuid = container.ext().getObjectInfo(obj).getUUID();
```

Db4oUuidExample.java: get the db4o-uuid

Getting a object by its UUID is also easy. First you get the object from the container. Unlike queries this won't activate your object. So you have to do it explicitly.

---

[1] Relational Database Management System

```
Object objectForId = container.ext().getByUUID(idForObject);
// getting by uuid doesn't activate the object
// so you need to do it manually
container.ext().activate(objectForId);
```

Db4oUuidExample.java: get an object by a db4o-uuid

Example UUID

This example demonstrates how you can use UUIDs to identify objects across objects containers. Take a look advantages and disadvantages of UUIDs: See "Comparison Of Different IDs" on page 183

Don't forget to add UUID support.

This example assumes that all object have a common super class, IDHolder, which holds the UUID in a field.

```
private    final UUID uuid = UUID.randomUUID();

public UUID getObjectId(){
    return uuid;
}
```

IDHolder.java: generate the id

It's important to index the id-field, otherwise looking up for object by id will be slow.

```
configuration.common().add(new UuidSupport());
configuration.common().objectClass(IDHolder.class).objectField("uuid").indexed(true);
```

UuidOnObject.java: index the uuid-field

The id is hold by the object itself, so you can get it directly.

```
IDHolder uuidHolder = (IDHolder)obj;
UUID uuid = uuidHolder.getObjectId();
```

UuidOnObject.java: get the uuid

You can get the object you can by a regular query.

```
Query query = container.query();
query.constrain(IDHolder.class);
query.descend("uuid").constrain(idForObject);
IDHolder object= (IDHolder) query.execute().get(0);
```

UuidOnObject.java: get an object its UUID

Example ID-Generator

This example demonstrates how you can use an ID-generator to identify objects across objects containers. Take a look advantages and disadvantages of ID-generators: See "Comparison Of Different IDs" on page 183

This example assumes that all object have a common super class, IDHolder, which holds the id.

```
private    int id;
public    int getId() {
    return id;
}

public    void setId(int id) {
    this.id = id;
}
```

IDHolder.java: id holder

Don't forget to index the id-field. Otherwise finding objects by id will be slow.

```
configuration.common().objectClass(IDHolder.class).objectField("id").indexed(true);
```

AutoIncrementExample.java: index the id-field

The hard part is to write an efficient ID-Generator. For this example a very simple auto increment generator is used. Use the creating-callback-event in order to add the ids to the object. When committing, store the state of the id-generator.

```
final AutoIncrement increment = new AutoIncrement(container);
EventRegistry eventRegistry = EventRegistryFactory.forObjectContainer(container);
eventRegistry.creating().addListener(new EventListener4<CancellableObjectEventArgs>() {
    public    void onEvent(Event4<CancellableObjectEventArgs> event4,
                           CancellableObjectEventArgs objectArgs) {
        if(objectArgs.object() instanceof IDHolder){
            IDHolder idHolder = (IDHolder) objectArgs.object();
            idHolder.setId(increment.getNextID(idHolder.getClass()));
        }
    }
});
eventRegistry.committing().addListener(new EventListener4<CommitEventArgs>() {
    public    void onEvent(Event4<CommitEventArgs> commitEventArgsEvent4,
                           CommitEventArgs commitEventArgs) {
        increment.storeState();
    }
});
```

AutoIncrementExample.java: use events to assign the ids

The id is hold by the object itself, so you can get it directly.

```
IDHolder idHolder = (IDHolder)obj;
int id = idHolder.getId();
```

AutoIncrementExample.java: get the id

You can get the object you can by a regular query.

```
Object object = container.query(new Predicate<IDHolder>() {
    @Override
    public    boolean match(IDHolder o) {
        return o.getId() == id;
    }
}).get(0);
```

AutoIncrementExample.java: get an object by its id

Merging Changes

Merging the changes is the most challenging part when using disconnected objects. Imagine that we have a disconnected object, which contains the changes. We have to store the changes somehow. You cannot simply store the disconnected object, because db4o wouldn't recognize it and store a new object instead of updating the old one. See "Wrong Approach" on page 187

Instead you need to load the existing object and then copy the state from the disconnected object to the loaded object. Basically you traverse the object-graph and copy all changes over. See "Example Merge Changes" on page 188

db4o has no built-in merge support. However there are external libraries which can help you to merge changes, like Dozer.

Wrong Approach

The wrong approach is to try to store disconnected objects. db4o manages object by their object-identity and doesn't recognize objects which have been serialized or loaded by another object container instance. This example shows, that instead of updating the object, db4o will store a new instance of the object.

```
{
    ObjectContainer container = openDatabase();
    Pilot joe = queryByName(container,"Joe");
    container.close();

    // The update on another object container
    ObjectContainer otherContainer = openDatabase();
    joe.setName("Joe New");
    otherContainer.store(joe);
    otherContainer.close();
}
{
    // instead of updating the existing pilot,
    // a new instance was stored.
    ObjectContainer container = openDatabase();
    ObjectSet<Pilot> pilots = container.query(Pilot.class);
    System.out.println("Amount of pilots: "+pilots.size());
    for (Pilot pilot : pilots) {
        System.out.println(pilot);
    }
    container.close();
}
```

ObjectIdentityExamples.java: Update doesn't works when using the different object containers

So in order to update an object, you need to load and store it in the same object-container. If you cannot do this, you need to merge to object-changes. See "Example Merge Changes" on page 188

```
{
    ObjectContainer container = openDatabase();
    Pilot joe = queryByName(container,"Joe");
    joe.setName("Joe New");
    container.store(joe);
    container.close();
}
{
    ObjectContainer container = openDatabase();
    ObjectSet<Pilot> pilots = container.query(Pilot.class);
    System.out.println("Amount of pilots: "+pilots.size());
    for (Pilot pilot : pilots) {
        System.out.println(pilot);
    }
    container.close();
}
```

ObjectIdentityExamples.java: Update works when using the same object container

Example Merge Changes

This example shows how changes are merged from the disconnected object to the object to update. To do this, traverse the object-graph and copy all value types over. All reference types are first checked if they are an existing object. If it is,the primitives are copied over, otherwise it's a stored as a new object.

```
ObjectContainer container = openDatabase();

// first get the object from the database
Car carInDb = getCarById(container,disconnectedCar.getObjectId());

// copy the value-objects (int, long, double, string etc)
carInDb.setName(disconnectedCar.getName());

// traverse into the references
Pilot pilotInDB = carInDb.getPilot();
Pilot disconnectedPilot = disconnectedCar.getPilot();

// check if the object is still the same
if(pilotInDB.getObjectId().equals(disconnectedPilot.getObjectId())){
    // if it is, copy the value-objects
    pilotInDB.setName(disconnectedPilot.getName());
    pilotInDB.setPoints(disconnectedPilot.getPoints());
} else{
    // otherwise replace the object
    carInDb.setPilot(disconnectedPilot);
}

// finally store the changes
container.store(pilotInDB);
container.store(carInDb);
```

MergeExample.java: merging

You can use reflection to automated this process. You can also use existing libraries like Dozer which help you to do this.

Web Environment

db4o runs perfectly well in a web environment. It can be used to build your web-application.

In most web-application multiple concurrent requests are processes. Normally you want to isolate each request from another. You can use db4o transactions to archive this isolation. See "Isolation in Web-Applications" on page 189

In most web-applications a object is only alive during a request. So you have to identify objects across requests. Therefore you need to add an additional id to your object. There are different possibilities for this. See "Disconnected Objects" on page 182

When you run in a web-environment, you often have stricter security limitations. Take a look at the security requirements. See "Security Requirements" on page 214

Take a look how you create a object-container for each request. See "Servlets" on page 190

Take a look at a small example Spring MVC application.See "Spring MVC Example" on page 191

Isolation in Web-Applications

In most web-application multiple concurrent request are processes. You want to isolate the request from each other. db4o supports transactions, which are perfect for this kind of isolation. Each unit of work gets its own transaction, for example each request. You can create a new session object container for this purpose. Such a session-container brings its own transaction and reference-system. This ensures that the session container is isolated from other operations on the database.

```
ObjectContainer rootContainer = Db4oEmbedded.openFile(DATABASE_FILE_NAME);

// open the db4o-session. For example at the beginning for a web-request
ObjectContainer session = rootContainer.ext().openSession();
try {
    // do the operations on the session-container
    session.store(new Person("Joe"));
} finally {
    // close the container. For example when the request ends
    session.close();
}
```

Db4oSessions.java: Session object container

Or you can use embedded clients when your embedded clients.

```
ObjectServer server = Db4oClientServer.openServer(DATABASE_FILE_NAME, 0);

// open the db4o-embedded client. For example at the beginning for a web-request
ObjectContainer container = server.openClient();
try {
    // do the operations on the session-container
    container.store(new Person("Joe"));
} finally {
    // close the container. For example when the request ends
    container.close();
}
```

Db4oSessions.java: Embedded client

Servlets

Running db4o as the persistence layer of a Java web application is easy. There is no installation

procedure - db4o is just another library in your application. There are only two issues that make web applications distinct from standalone programs from db4o's point of view. One is the more complex classloader environment - db4o needs to know itself and the classes to be persisted.

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

On the Servlet API layer, you could bind db4o server handling to the Servlet.

You can implement the ServletContextListener-interface, open the database when the web application starts and close when it ends.

```
@Override
public   void contextInitialized(ServletContextEvent event) {
    ServletContext context = event.getServletContext();
    String filePath = context.getRealPath("WEB-INF/"
            + context.getInitParameter(KEY_DB4O_FILE_NAME));
    EmbeddedObjectContainer rootContainer = Db4oEmbedded.openFile(filePath);
    context.setAttribute(KEY_DB4O_SERVER, rootContainer);
    context.log("db4o startup on " + filePath);
}


@Override
public   void contextDestroyed(ServletContextEvent event) {
    ServletContext context = event.getServletContext();
    ObjectContainer rootContainer = (ObjectContainer) context.getAttribute(KEY_DB4O_SERVER);
    context.removeAttribute(KEY_DB4O_SERVER);
    close(rootContainer);
    context.log("db4o shutdown");
}
```

Db4oServletListener.java: db4o-instance for the web-application

Additionally you can implement the ServletRequestListener-interface and open a db4o-session on each request.

```
@Override
public   void requestInitialized(ServletRequestEvent requestEvent) {
    EmbeddedObjectContainer rootContainer = (EmbeddedObjectContainer) requestEvent
            .getServletContext().getAttribute(Db4oServletListener.KEY_DB4O_SERVER);

    ObjectContainer session = rootContainer.openSession();
    requestEvent.getServletRequest().setAttribute(KEY_DB4O_SESSION, session);
}


@Override
public   void requestDestroyed(ServletRequestEvent requestEvent) {
    ObjectContainer session = (ObjectContainer) requestEvent
            .getServletRequest().getAttribute(KEY_DB4O_SESSION);

    close(session);
}
```

Db4oServletListener.java: a db4o-session for each request

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

```
<context-param>
    <param-name>database-file-name</param-name>
    <param-value>database.db4o</param-value>
</context-param>
<listener>
    <listener-class>com.db4odoc.servlet.Db4oServletListener</listener-class>
</listener>
```

web.xml: register the listener for the web application

Now db4o should be available to your application classes. Each request has it own object-container. You can get the instance via its key, like this:

```
ObjectContainer container =
        (ObjectContainer)req.getAttribute(Db4oServletListener.KEY_DB4O_SESSION);
```

ServletExample.java: Get the session container

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

Spring MVC Example

This example is a tiny CRUD application which shows how to use db4o in a web-application. This example uses the Spring MVC framework. Of course, db4o works with any webframework.

Managing Object Containers

It uses the code from the servlet-example to have a object container for each request. On each new request a object container is opened. Then all operations are done on the container. When the request ends, the container is closed.

You also can use the features of your web framework or your dependency injection framework to archive the same goal.

Object Identification

This example uses a GUID for each object to identify it across requests. Persisted objects which inherit from the IDHolder class which contains the id-field. Take a look at alternatives for ids. See "Comparison Of Different IDs" on page 183

Using db4o

You can use db4o as expected. In this example we use a the db4o-container of the request:

```
@RequestMapping(value = "list.html", method = RequestMethod.GET)
public ModelAndView get() {
    ObjectSet pilots = db4o.objectContainer().query(Pilot.class);
    return    new ModelAndView("list", "pilots", new ArrayList<Pilot>(pilots));
}
```

HomeController.java: List all pilots on the index-page

Using IDs

Now the ids can be used in the views and controllers to identify objects. For example in a list-view you use the ids for the edit- and delete-links:

```
<c:forEach items="${pilots}" var="pilot">
    <tr>
        <td>
            <a href="edit${pilot.id}.html"/>Edit</a>
            <a href="delete${pilot.id}.html"/>Delete</a>
        </td>
        <td>
                ${pilot.name}
        </td>
        <td>
                ${pilot.points}
        </td>
    </tr>
</c:forEach>
```

list.jsp: In the view use the ids to identify the objects

Another location where the ids are used is in the controllers. For example when you need to store changes. First we get a object which contains all changes. Then we copy all changes to the existing object in the database and finally store it. See "Merging Changes" on page 187

```
@RequestMapping(value = "/edit{id}.html", method = RequestMethod.POST)
public ModelAndView editPilot(@PathVariable final String id, Pilot pilotFromForm) {
    Pilot pilotFromDatabase = db4o.objectContainer().query(new Predicate<Pilot>() {
        @Override
        public    boolean match(Pilot p) {
            return p.getId().equals(id);
        }
    }).get(0);
    pilotFromDatabase.setName(pilotFromForm.getName());
    pilotFromDatabase.setPoints(pilotFromForm.getPoints());
    db4o.objectContainer().store(pilotFromDatabase);
    return    new ModelAndView(new RedirectView("list.html"));
}
```

HomeController.java: Update the object

Reporting

You want to generate reports with from a db4o database? What capabilities does db4o have to support reporting? How can you implement reporting? This topic tries to answer these questions.

Understand db4o's Limitations

db4o has serious limitations when it comes to reporting. It doesn't support any aggregation operations like average, min, max or sum. Nor does it support any projection (SELECT statement in SQL) or grouping operations. You only can query for objects and not do any statistics or selection on them.

Reporting with db4o

Despite the missing features for reporting it's possible to do simple reports with db4o as long as your reporting framework can work with simple Java Beans instead of a relational database. Take a look at this small example with Jasper Reports.See "Reporting Example With MS Report Viewer" See "db4o with Jasper Reports Example" on page 193

Separate Application Data from Reporting Data

In some applications it can make sense to separate application data from the reporting data, for e example in a web application. The visits log data goes into a relational database which then is used for reporting, while the content of the site is stored in db4o. In this case you the can use the full power of the relational database for the reporting of the visitor data.

Export to a Relation Database

Another possibility would be to export the data to a relational database for reporting purposes. In that process you also can denormalize and restructure the data for the reporting task. That way you also can use a regular relational database for the reporting tasks.

db4o with Jasper Reports Example

This example illustrates how you can use Jasper Reports together with db4o to create a report. Take a look at the official Jasper Reports website for the documentation and tools around Jasper Reports.

Designing the Report

The report layout is written as an XML file or alternatively you can use an editor like the iReport-editor.

For adding data you can declare fields, which then are used for each row of the report. Just use the names of your properties which you want to report. For navigating along the data separate the properties with dots:

```
<field name="sirname" class="java.lang.String"/>
<field name="firstname" class="java.lang.String"/>
<field name="address.city" class="java.lang.String"/>
```

report.jrxml: declare fields

After that you can access these fields in the report:

```
<textField>
       <reportElement x="0" y="0" width="100" height="21"/>
       <textFieldExpression class="java.lang.String"><![CDATA[$F{sirname}]]></textFieldExpression>
</textField>
<textField>
       <reportElement x="100" y="0" width="100" height="21"/>
       <textFieldExpression class="java.lang.String"><![CDATA[$F{firstname}]]></textFieldExpression>
</textField>
<textField>
       <reportElement x="200" y="0" width="100" height="21"/>
       <textFieldExpression class="java.lang.String"><![CDATA[$F{address.city}]]></textFieldExpression>
</textField>
```

report.jrxml: Placing the fields on the report

Filling the Report

After that you can fill the report with data from the database. For simpler reports we can use the JRBeanCollectionDataSource, which reads the values from regular Java bean objects. For more complex reports you might need to use another data source which you can fill manually like the JRMapCollectionDataSource. For all the available datasources read the Jasper Reports documentation.

```
final ObjectSet<Person> queryResult = container.query(new Predicate<Person>() {
    @Override
    public   boolean match(Person p) {
        return p.getSirname().contains("a");
    }
});
final JRBeanCollectionDataSource dataSource = new JRBeanCollectionDataSource(queryResult);
final JasperPrint jasperPrint = JasperFillManager.fillReport(report, new HashMap(), dataSource);
JasperExportManager.exportReportToPdfFile(jasperPrint, "the-report.pdf");
```

JasperReportsExample.java: Run a query an export the result as report

Maven

Currently db4o only provides snapshots of the current beta and production versions.

Repository

All db4o artifacts are in this repository:

```
<repositories>
   <repository>
        <id>db4o</id>
        <name>Db4o</name>
        <url>https://source.db4o.com/maven/</url>
   </repository>
</repositories>
```

Artifacts

The db4o Maven artifacts represent the different functionality of db4o. All different artifacts use the groudID 'com.db4o'.

You can either include the whole db4o distribution like this:

```
<dependency>
    <groupId>com.db4o</groupId>
    <artifactId>db4o-full-java5</artifactId>
    <version>8.0-SNAPSHOT</version>
</dependency>
```

Or include the different part of db4o as you need them. Available are 'db4o-core-java5', 'db4o-cs-java5', 'db4o-cs-optional-java5', 'db4o-nqopt-java5', 'db4o-taj-java5' and 'db4o-tools-java5'. Take a look at this overview for more details.

```
<!--For example: Include native
 queries optimization and optional featueres-->
<dependency>
   <groupId>com.db4o</groupId>
   <artifactId>db4o-nqopt-java5</artifactId>
    <version>8.0-SNAPSHOT</version>
</dependency>
<dependency>
    <groupId>com.db4o</groupId>
    <artifactId>db4o-optional-java5</artifactId>
    <version>8.0-SNAPSHOT</version>
</dependency>
```

Validation

By default db4o doesn't support any validation and integrity checks except unique field values. However you can use .NET data annotations to validate objects. The .NET data annotations provide attributes for validating objects. However you can use other libraries like Hibernate-Validator to validate objects. Download the library from the official Hibernate site and include into your project. Objects can be validated when you store them in the database by using db4o events.

Add the Annotation to your classes which you want to validate:

Now we can write a validation method and register it to the db4o events:

```
private  static  class ValidationListener implements EventListener4<CancellableObjectEventArgs> {
    private  final Validator validator = Validation.buildDefaultValidatorFactory()
            .getValidator();

    @Override
    public  void onEvent(Event4<CancellableObjectEventArgs> events,
                         CancellableObjectEventArgs eventInfo) {
        Set<ConstraintViolation<Object>> violations = validator.validate(eventInfo.object());
        if (!violations.isEmpty()) {
            throw  new ValidationException(buildMessage(violations));
        }
    }

    private String buildMessage(Set<ConstraintViolation<Object>> violations) {
        final StringBuilder builder = new StringBuilder("Violations of validation-rules:\n");
        for (ConstraintViolation<Object> violation : violations) {
            builder.append(violation.getPropertyPath()).append(" ")
                    .append(violation.getMessage()).append("\n");
        }
        return builder.toString();
    }
}
```

DataValidation.java: Validation support

```
EventRegistry events = EventRegistryFactory.forObjectContainer(container);
events.creating().addListener(new ValidationListener());
events.updating().addListener(new ValidationListener());
```

DataValidation.java: Register validation for the create and update event

After that you can store and update objects. In case a object violates its validation rules an exception is thrown. That exception will contain information about the violations.

```
Pilot pilot = new Pilot("Joe");
container.store(pilot);
```

DataValidation.java: Storing a valid pilot

```
Pilot otherPilot = new Pilot("");
try {
    container.store(otherPilot);
} catch (EventException e) {
    ValidationException cause = (ValidationException) e.getCause();
    System.out.println(cause.getMessage());
}
```

DataValidation.java: Storing a invalid pilot throws exception

Android

db4o runs seamlessly on Android, enabling native storage and retrieval of objects of any complexity. db4o is a powerful alternative to the built in persistence-capabilities of the Android-platform. See "Comparison With SQLite" on page 203

It's easy to start.Setting up db4o only takes a few steps. See "Getting Started" on page 197.

Due to differences between Android and other Java platforms, there are some additional pitfalls. See "Pitfalls and Tested Functionality" on page 197

Android uses a special virtual machine which prevents db4o from optimizing native queries at runtime. The solution is optimize the queries as build-time. See "Native Queries" on page 199

Getting Started

It takes only a few steps to get started with db4o and Android. This description assumes that your using Eclipse to create your Android application.

Setup db4o for a Android project

1. First download the Java version of db4o at the download area
2. Unpack the distribution. Then copy the **db4o-xxx-java5.jar** the db4o-distribution to your Android project-folder.
3. Add the db4o-jar to the class path. In Eclipse you do it this way. Refresh you project. Right click on the **db4o-xxx-java5.jar** . Choose 'Build Path' -> 'Add to Build Path'
4. Your done! You can now use db4o in your Android application and it will be deployed automatically when running the Android emulator.

Using db4o in a Android application

You can use db4o on Android as normally. However you when you create a db4o-database you should use a file in the application-context. The start class of your application itself is usually the context.

Also add the AndroidSupport to your configuration, which tunes some configuration settings to work better with Android.

```
public   class Db4oOnAndroidExample  extends Activity  {
    /** Called when the activity is first created. */
    @Override
    public   void onCreate(Bundle savedInstanceState) {
               String filePath = this.getFilesDir() + "/android.db4o";
        final EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
        config.common().add(new AndroidSupport());
        ObjectContainer db = Db4oEmbedded.openFile(config,filePath);
               // do your stuff
               db.close();

    }
}
```

Db4oOnAndroidExample.java: open db4o on Android

Pitfalls and Tested Functionality

Android uses a special virtual machine, the Dalvik VM and uses its own set of the standard Java libraries. This creates some pitfalls and limitations.

Native Queries Limitation

On most platforms db4o can optimize native queries at runtime. For this it analyses the byte-code of the query. Since Android uses a different bytecode for the Dalvik VM, this optimization doesn't work. However you can do the same optimization at built-time. See "Native Queries" on page 199

BigMath-Limitation

The BitMath is slightly different implemented on Android. Therefore it doesn't work out of the box with db4o. But when you add the Big-Math support, it will work just fine. Add the BigMath-support to the configuration. See "BigMath" on page 100

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().add(new BigMathSupport());
```

ConfigurationItemsExamples.java: Add support for BigDecimal and BigInteger

Monitoring Not Supported

Android doesn't provide the Java monitoring and instrumenting facilities. Therefore db4o's support for monitoring doesn't work as well.

Tested Functionality

Currently the automated tests for db4o run against the Android version 2.1. db4o should run also on older Androids versions.

Since the Dalvik VM doesn't use regular Java bytecode and class-files and therefore uses different a different class-loader strategy, we don't test features of db4o which utilize this features. However this is only relevant for advanced db4o features which need those functionality.

Native Queries

Native Queries also work on Android. However there's a limitation. Android uses a special Java Virtual machine, which prevents optimizing Native Queries at runtime. The solution is to optimize the queries as build-time. See "Enhancement Tools" on page 83

Example Build Time Enhancement for Android

This example is based on the build time enhancements. This example only shows the important steps for optimizing the native queries for Android.

First define the enhancing task. It's important to also include the Android-platform libraries, otherwise some classes cannot be found.

```
<path     id="project.classpath">
    <pathelement     path="${basedir}/bin/classes"/>
       <!-- The sdk.dir points to the Android-platform libraries -->
    <fileset     dir="${sdk.dir}/platforms/${target}">
        <include     name="android.jar"/>
    </fileset>
    <fileset     dir="libs">
        <include     name="**/*.jar"/>
    </fileset>
</path>
<taskdef     name="db4o-enhance"
         classname="com.db4o.enhance.Db4oEnhancerAntTask"
         classpathref="project.classpath"/>
```

android-nq-optimisation.xml: Define the task for the enhancement

Then define the target which enhances the classes. This example only activates the Native Query optimization. However you can also activate **TA**[1] or **TP**[2] support if you like. See "Transparent Persistence" on page 50

```
<target     name="enhance">
    <db4o-enhance     classtargetdir="${basedir}/bin/classes"
                jartargetdir="${basedir}/lib"
                nq="true"     ta="false"
                collections="false">
        <classpath     refid="project.classpath"/>
        <sources     dir="${basedir}/bin/classes">
                <include     name="**/*.class"/>
        </sources>
    </db4o-enhance>
</target>
```

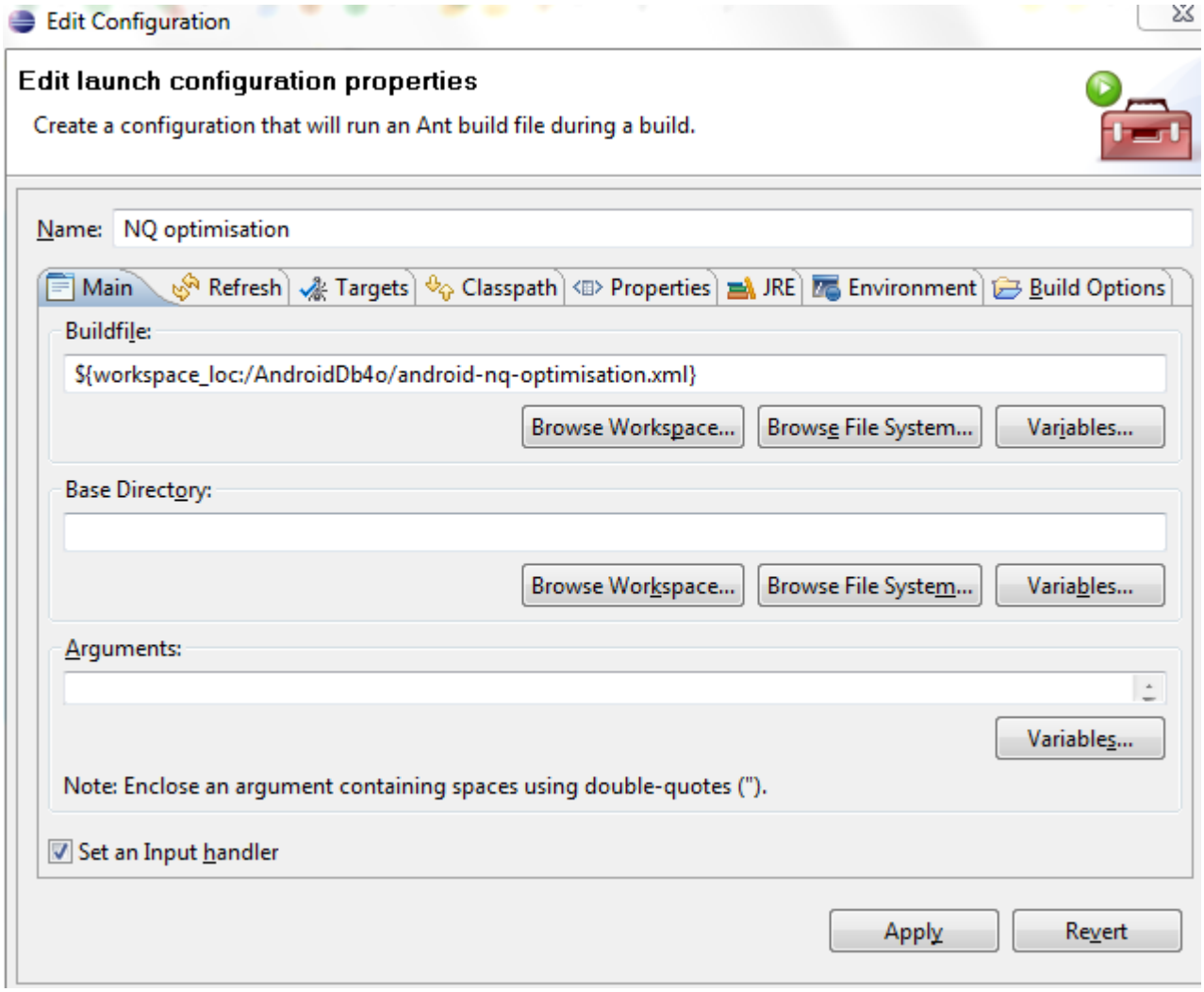
android-nq-optimisation.xml: Define a target which runs the task

The next step is to integrate this Ant-task in Eclipse. Right click on the project and then select 'Properties'. There switch to the 'Builders'-tab. Add a new 'Ant Builder'.

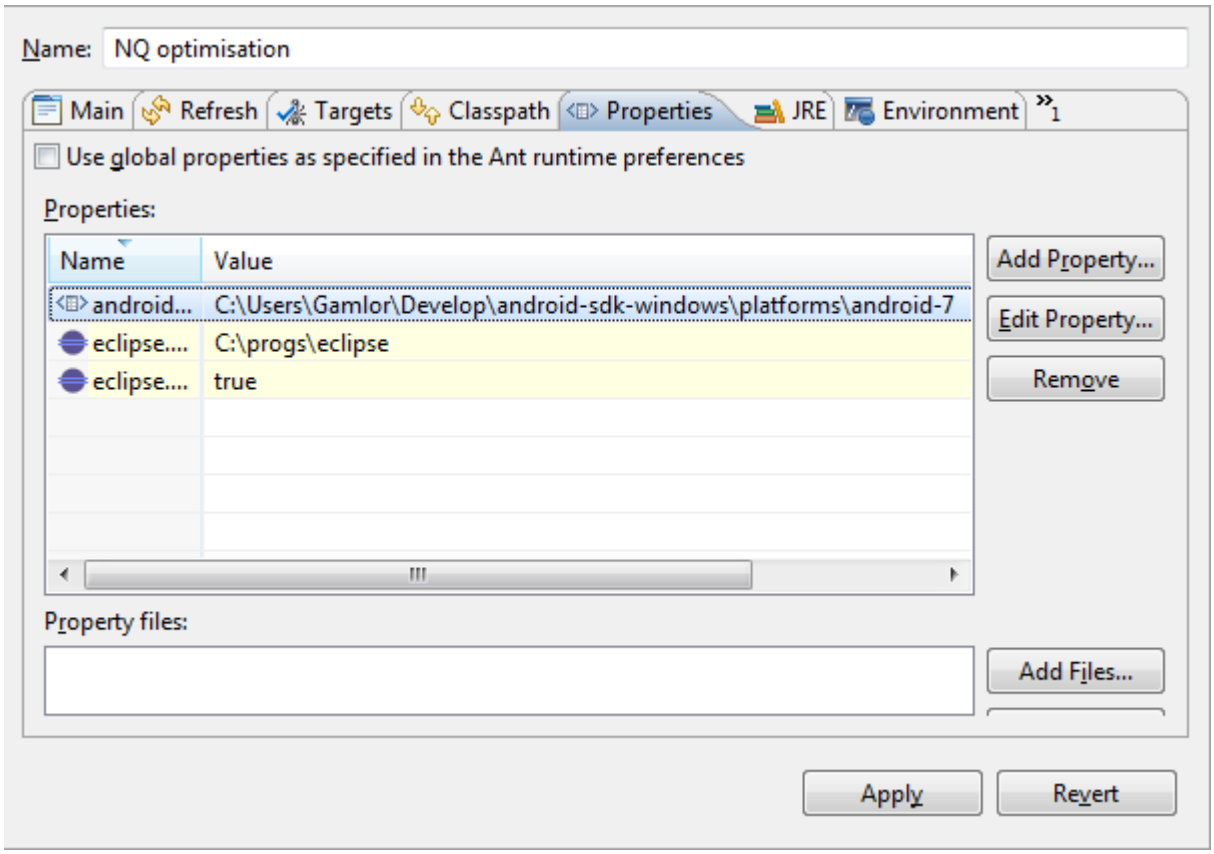
On the 'Main'-tab select the enhancement-script:

---

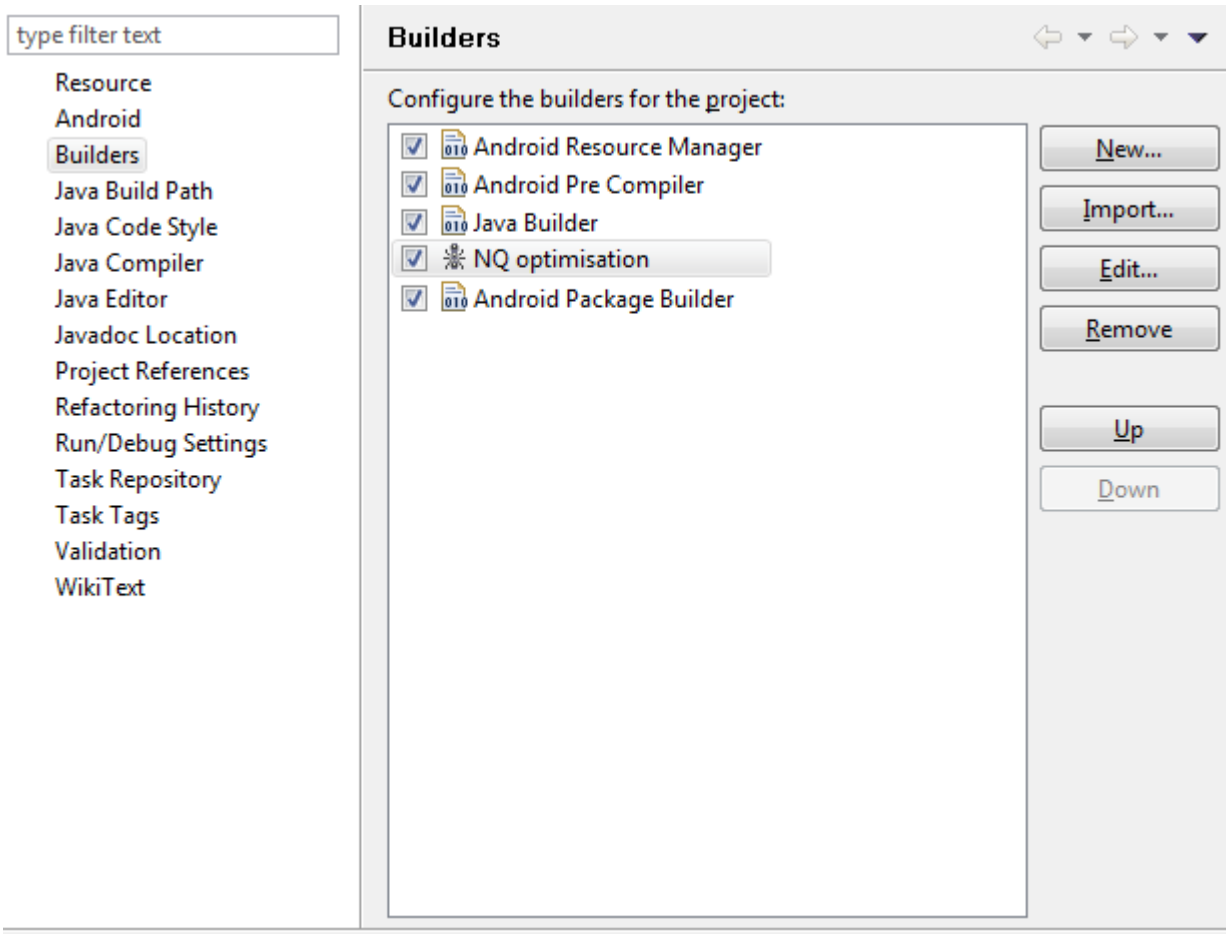
[1] Transparent Activation
[2] Transparent Persistence



Then go to the 'Properties'-tab. There add a new property 'android.platform' which points to the right Android-platform and version-path. You can find the platforms in you Android-SDK. The platforms are in the folder 'platforms' in the Android SDK. For example when your using the API-level 7 the platform is in the folder 'AndroidSDK/platforms/android-7'



Apply all settings. The last step is to place the new created builder between the 'Java Builder' and the 'Android Package Builder':



Integrating in the Regular Build

You simply can include the enhancer step into the regular Android build. For that include the enhancer Ant script in the build.xml and let it run during the compilation phase:

```
<import     file="${sdk.dir}/tools/ant/build.xml"     />

<!-- Add the db4o step below the predefined android targets -->
<import     file="android-db4o-optimisation.xml"     />


<target     name="-post-compile"     depends="enhance">
</target>
```

build.xml: Run the enhancer step during the build

Comparison With SQLite

Android is a new complete, open and free mobile platform. Android offers developers a Java based software developer kit with lots of helpful APIs, including geolocational services. Of course, there is a database support as well: Android has a built-in support for SQLite database. The basic API is similar to standard JDBC API, with some additional helpful methods

It may look better than SQL, but if you look closer it still has the same issues. Queries are specified as strings, so we still stay with a problem of run-time checking instead of compile-time. Furthermore you still need to map between your object-model and the relational SQLite database.

Luckily even for this very early Android release we already have an alternative - db4o. db4o runs on Android out of the box and produces very competitive results as well. The following examples compare db4o and SQLite usage for basic operations.

General Info

Both db4o and SQLite are embedded databases, i.e. they run within an application process, removing the overhead associated with a client-server configuration, although db4o can also be used in client-server mode. Both db4o and SQLite offer zero-configuration run modes, which allows to get the database up and running immediately.

Access Control

SQLite relies solely on the file system for its database permissions and has no concept of user accounts. SQLite has database-level locks and does not support client/server mode.

db4o can use encryption or client/server mode for user access control. Client/server can also be used in embedded mode, i.e on the same device.

Referential Integrity

Traditionally referential integrity in relational databases is implemented with the help of foreign keys. However SQLite does not support this feature. In db4o referential integrity is imposed by the object model, i.e. you can't reference an object that does not exist.

Transactions

Both db4o and SQLite support ACID transactions.

In db4o all the work is transactional: transaction is implicitly started when the database is open and closed either by explicit commit() call or by close() call through implicit commit. Data is protected from system crash during all application lifecycle. If a crash occurs during commit itself, the commit will be restarted when the system is up again if the system had enough time to write the list of changes, otherwise the transaction will be rolled back to the last safe state.

In SQLite autocommit feature is used by default: transaction is started when a SQL command other than SELECT is executed and commit is executed as soon as pending operation is finished. Explicit BEGIN and END TRANSACTION(COMMIT) or ROLLBACK can be used alternatively to specify user-defined

transaction limits. Database crash always results in pending transaction rollback. Nested transactions are not supported.

Database Size

Though embeddable db4o and SQLite support big database files:

- db4o up to 256 GB
- SQLite up to 2TB

all the data is stored in a single database file. db4o also supports clustered databases.

Opening A Database

Opening the database is very similar.

SQLite

Opening a SQLite database is very easy. However is necessary to generate the schema for the database.

```
SQLiteDatabase db = _context.openOrCreateDatabase(DATABASE_NAME,
               Context.MODE_PRIVATE, null);
```

SqlExample.java: opening SQLite database

```
db.execSQL("CREATE TABLE IF NOT EXISTS " + DB_TABLE_PILOT + " ("
               + "id INTEGER PRIMARY KEY AUTOINCREMENT, "
               + "name TEXT NOT NULL, points INTEGER NOT NULL);");
// Foreign key constraint is parsed but not enforced
// Here it is used for documentation purposes
db.execSQL("CREATE TABLE IF NOT EXISTS " + DB_TABLE_CAR + " ("
               + "id INTEGER PRIMARY KEY AUTOINCREMENT, "
               + "model TEXT NOT NULL, pilot INTEGER NOT NULL,"
               + "FOREIGN KEY (pilot)"
               + "REFERENCES pilot(id) ON DELETE CASCADE);");
db.execSQL("CREATE INDEX IF NOT EXISTS CAR_PILOT ON " + DB_TABLE_CAR
               + " (pilot);");
```

SqlExample.java: SQLite create the schema

db4o

Opening a db4o database is easy. First you need to create a file which is in the context of the application. Then you can open the database. To be faster, we configure additional indexes.

```
String filePath = context.getFilesDir() + "/android.db4o";
ObjectContainer db = Db4oEmbedded.openFile(configure(), filePath);
```

Db4oExample.java: open a db4o database

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
      configuration.common().add(new AndroidSupport());
configuration.common().objectClass(Car.class).objectField("pilot").indexed(true);
configuration.common().objectClass(Pilot.class).objectField("points").indexed(true);
```

Db4oExample.java: configure db4o

Conclusion

db4o code is a bit more compact, but the main advantage of db4o is in the fact that all APIs are pure java, they are compile-time checked and can be transferred into IDE templates (database opening should be a template as it most probably be the same for all your db4o applications including tests).

Storing Data

Storing a car-object in is very-different in SQLite and db4o.

SQLite

```
ContentValues initialValues = new ContentValues();

initialValues.put("id", number);
initialValues.put("name", "Tester");
initialValues.put("points", number);
db.insert(DB_TABLE_PILOT, null, initialValues);

initialValues = new ContentValues();

initialValues.put("model", "BMW");
initialValues.put("pilot", number);
db.insert(DB_TABLE_CAR, null, initialValues);
```

SqlExample.java: store a car in SQLite

db4o

```
Car car = new Car("BMW");
car.setPilot(new Pilot("Tester", points));
container.store(car);
```

Db4oExample.java: store a car in db4o

Conclusion

You can see that db4o handles adding objects to the database in a much more elegant way - #store(-object) method is enough. In SQLite case it is much more difficult as you mush store different objects into different tables. Some of the additional work that SQLite developer will have to do is not visible in this example, i.e:

- the developer will have to ensure that the sequence of insert commands starts from children objects and goes up to the parent (this can be a really difficult task for relational models including lots of foreign key dependencies);
- in most cases the data for insertion will come from business objects, which will mean that the object model will have to be transferred to relational model.

Retrieving Data

In order to test the retrieval abilities of both databases we will try to select a car with a pilot having15 points:

SQLite

```
SQLiteDatabase db = database();
Cursor cursor = db.rawQuery(
               "SELECT c.model, p.name, p.points, r.id, r.year" + " FROM "
                               + DB_TABLE_CAR + " c, " + DB_TABLE_PILOT + " p "
                               + "WHERE c.pilot = p.id AND p.points = ?;",
               new String[] { "15" });
cursor.moveToFirst();

Pilot pilot = new Pilot();
pilot.setName(cursor.getString(1));
pilot.setPoints(cursor.getInt(2));

Car car = new Car();
car.setModel(cursor.getString(0));
car.setPilot(pilot);
```

SqlExample.java: select a car from SQLite

db4o

```
ObjectContainer db = database();
ObjectSet<Car> cars = db.query(new Predicate<Car>() {
       @Override
       public  boolean match(Car car) {
               return car.getPilot().getPoints() == 15;
       }
});

Car car = cars.get(0);
```

Db4oExample.java: select a car from db4o

Conclusion

The db4o native queries are typesafe. This is a huge benefit, since the compiler can detect errors and the IDE help you with the refactoring. In the example above you can see that SQLite needs a lot of additional code to transfer the retrieved data into application's objects, whereas db4o does not need this code at all, as the result is already a collection of objects.

Changing Data

For this test we will select and update a car with a new pilot, where existing pilot has 15 points:

SQLite

```
SQLiteDatabase db = database();
db.execSQL("INSERT INTO REG_RECORDS (id,year) VALUES ('A1', DATETIME('NOW'))");

ContentValues updateValues = new ContentValues();
updateValues.put("reg_record", "A1");
int count = db.update(DB_TABLE_CAR, updateValues,
               "pilot IN (SELECT id FROM " + DB_TABLE_PILOT
                               + " WHERE points = 15)", null);
if (count == 0) {
       logToConsole(0, "Car not found, refill the database to continue.",
                       false);
} else {
       Cursor c = db.rawQuery("SELECT c.model, r.id, r.year from car c, "
                       + "REG_RECORDS r, pilot p where c.reg_record = r.id "
                       + "AND c.pilot = p.id AND p.points = 15;", null);
       if (c.getCount() == 0) {
               logToConsole(0,
                               "Car not found, refill the database to continue.",
                               false);
               return;
       }
       c.moveToFirst();
       String date = c.getString(2);

       Date dt = parseDate(date);
       RegistrationRecord record = new RegistrationRecord(c.getString(1),dt);

       Car car = new Car();
       car.setModel(c.getString(0));
       car.setRegistration(record);
       logToConsole(startTime, "Updated Car (" + car + "): ", true);
}
```

SqlExample.java: update a car with SQLite

db4o

```
ObjectContainer container = database();
if (container != null){
	ObjectSet<Car>  result = container.query(new Predicate<Car>(){
		@Override
		public  boolean match(Car car) {
			return car.getPilot().getPoints()==15;
		}

	});
	if (!result.hasNext()){
		logToConsole(0, "Car not found, refill the database to continue.", false);
	} else {
		Car car = result.next();
		logToConsole(startTime, "Selected Car (" + car + "): ", false);
		startTime = System.currentTimeMillis();
		car.setRegistration(new RegistrationRecord("A1", new Date()));
		logToConsole(startTime, "Updated Car (" + car + "): ", true);
	}
}
```

Db4oExample.java: update a car with db4o

Conclusion

In this example db4o and SQLite actually behave quite differently. For SQLite in order to update a pilot in an existing car in the database the following actions are needed:

1. A new pilot should be created and saved to the database.
2. New pilot's primary key (101) should be retrieved (not shown in this example, but is necessary for a real database application).
3. An update statement should be issued to replace pilot field in the car table.

For db4o database the sequence will be the following:

1. Retrieve the car from the database
2. Update the car with a new pilot object

As you can see the only benefit of SQLite API is that the car can be selected and updated in one statement. But in the same time there are serious disadvantages:

- A new pilot record should be created absolutely separately (in a real database will also include ORM)
- The pilot's ID needs to be retrieved separately (we must sure that it is a correct id)

In db4o we avoid these disadvantages as creating new pilot and updating the car value are actually combined in one atomic operation.

Deleting Data

The following methods will delete a car with a pilot having 5 points from each database:

SQLite

```
SQLiteDatabase db = database();
db.delete(DB_TABLE_CAR,
               "pilot in (select id from pilot where points = ?)",
               new String[]{"5"});
```

SqlExample.java: delete a car with SQLite

db4o

```
ObjectContainer db = database();
ObjectSet<Car> cars = db.query(new Predicate<Car>() {
       public  boolean match(Car car) {
               return car.getPilot().getPoints()==5;
       }
});
for(Car car : cars){
       db.delete(car);
}
```

Db4oExample.java: delete a car with db4o

Conclusion

In this example db4o code looks much longer. But should we consider it a disadvantage? My opinion is - No. Of course, SQLite seems to handle the whole operation in just one statement: db.delete(). But if you look attentively you will see that basically this statement just transfers all the difficult job to SQL: SQL statement should select a pilot with a given condition, then find a car. Using SQL can look shorter but it has a great disadvantage - it uses strings. So what will happen if the statement is wrong? You will never notice it till somebody in the running application will cause this statement to execute. Even then you might not see the reason immediately. The same applies to the schema changes - you may not even notice that you are using wrong tables and fields.

db4o helps to avoid all the above mentioned problems: query syntax is completely compile-checked and schema evolution will be spotted immediately by the compiler, so that you would not need to rely on code search and replace tools.

Backup

SQLite

SQLite does not support a special API to make a backup. However, as you remember SQLite database is stored in a single database file, so the backup can be simply a matter of copying the database file. Unfortunately, this can't be done if the database is in use. In this case you can use Android Debug Bridge (adb) tool to access sqlite3 command-line application, which has .dump command for backing up database contents while the database is in use:

```
E:\>adb shell
# sqlite3 /data/data/com.db4odoc.android.compare/databases/android.db
sqlite3 /data/data/com.db4odoc.android.compare/databases/android.db
SQLite version 3.5.0
Enter ".help" for instructions
sqlite> .dump > android200711.dmp
.dump > android200711.dmp
```

```
BEGIN TRANSACTION;
COMMIT;
sqlite>.exit
.exit
# ^D
```

Ctrl+D command is used to close adb session.

db4o

On db4o a ExtObjectContainer#backup call is used to backup a database in use. See "Backup" on page 68

Closing A Database

The following methods will close SQLite and db4o database accordingly:

SQLite

```
_db.close();
```

SqlExample.java: close SQLite

db4o

```
container.close();
```

Db4oExample.java: close db4o

Schema Evolution

When a new application development is considered it is important to think about its evolution. What happens if your initial model does not suffice and you need changes or additions? Let's look how db4o and SQLite applications can handle it.

To keep the example simple, let's add a registration record to our car:

```
private RegistrationRecord registration;

public RegistrationRecord getRegistration() {
	return registration;
}

public  void setRegistration(RegistrationRecord registration) {
	this.registration = registration;
}
```

Car.java: Add a new field to the car

Ok, the application is changed to take care for the new class. What about our databases?

Schema Evolution in db4o

db4o supports such schema change on the fly: we can select values and update the new field too:

```
ObjectContainer container = database();
if (container != null){
	ObjectSet<Car>  result = container.query(new Predicate<Car>(){
		@Override
		public  boolean match(Car car) {
			return car.getPilot().getPoints()==15;
		}

	});
	if (!result.hasNext()){
		logToConsole(0, "Car not found, refill the database to continue.", false);
	} else {
		Car car = result.next();
		logToConsole(startTime, "Selected Car (" + car + "): ", false);
		startTime = System.currentTimeMillis();
		car.setRegistration(new RegistrationRecord("A1", new Date()));
		logToConsole(startTime, "Updated Car (" + car + "): ", true);
	}
}
```

Db4oExample.java: update a car with db4o

Schema Evolution in SQLite

For SQLite database model should be synchronized with the object model:

```
db.execSQL("CREATE TABLE IF NOT EXISTS REG_RECORDS ("
		+ "id TEXT PRIMARY KEY, year DATE);");
db.execSQL("CREATE INDEX  IF NOT EXISTS  IDX_REG_RECORDS ON REG_RECORDS (id);");
db.execSQL("ALTER TABLE " + DB_TABLE_CAR + " ADD reg_record TEXT;");
```

SqlExample.java: upgrade schema in SQLite

Now we can try to retrieve and update records:

```
SQLiteDatabase db = database();
db.execSQL("INSERT INTO REG_RECORDS (id,year) VALUES ('A1', DATETIME('NOW'))");

ContentValues updateValues = new ContentValues();
updateValues.put("reg_record", "A1");
int count = db.update(DB_TABLE_CAR, updateValues,
               "pilot IN (SELECT id FROM " + DB_TABLE_PILOT
                              + " WHERE points = 15)", null);
if (count == 0) {
       logToConsole(0, "Car not found, refill the database to continue.",
                      false);
} else {
       Cursor c = db.rawQuery("SELECT c.model, r.id, r.year from car c, "
                      + "REG_RECORDS r, pilot p where c.reg_record = r.id "
                      + "AND c.pilot = p.id AND p.points = 15;", null);
       if (c.getCount() == 0) {
               logToConsole(0,
                              "Car not found, refill the database to continue.",
                              false);
               return;
       }
       c.moveToFirst();
       String date = c.getString(2);

       Date dt = parseDate(date);
       RegistrationRecord record = new RegistrationRecord(c.getString(1),dt);

       Car car = new Car();
       car.setModel(c.getString(0));
       car.setRegistration(record);
       logToConsole(startTime, "Updated Car (" + car + "): ", true);
}
```

SqlExample.java: update a car with SQLite

Conclusion

You can see that schema evolution is much easier with db4o. But the main difficulty that is not visible from the example is that schema evolution with SQLite database can potentially introduce a lot of bugs that will be difficult to spot. For more information see Refactoring and Schema Evolution.

Proguard

The Android SDK includes Proguard to shrink the size of the application by removing unused stuff, shorting names and other optimisations.

db4o relies heavily on reflection, which can lead to major issues when code is processed with Proguard. db4o has not been tested under such a environment. When you're using Proguard you need to test yourself if your application is still working.

You should to exclude all persisted classes from the shrinking process, because db4o manages objects by their name. Therefore db4o will have issue when those name changes by the shrinking process. Read the Proguard documentation for the right configuration. For example when your persisted classes in a certain package you can exclude them like this:

```
########################
-keep class com.db4odoc.android.**
-keepnames class com.db4odoc.android.**
-keepclassmembers class com.db4odoc.android.** {
    !static !transient <fields>;
    !private <fields>;
    !private <methods>;
}
```

proguard.cfg: Keep persisted classes intact

To be on the save side you should keep all db4o classes in the original state. Especially since db4o stores some internal data structures which shouldn't be renamed.

```
########################
## Monitoring requires JMX, which is not available on Android
-dontwarn com.db4o.monitoring.*
-dontwarn com.db4o.cs.monitoring.*
-dontwarn com.db4o.internal.monitoring.*

## Ant is usually not used in a running app
-dontwarn com.db4o.instrumentation.ant.*
-dontwarn com.db4o.ta.instrumentation.ant.*

## Keep internal classes.
-keep class com.db4o.** { *; }
```

proguard.cfg: keep db4o intact

db4o on Java Platforms

All Java

- root package is com.db4o

JDK 1.5 or newer

- Generics support introduced in JDK1.5 makes db4o Native Query syntax much simpler: `List <Pilot> pilots = db.query(new Predicate <Pilot> () { public boolean match(Pilot pilot) { return pilot.getPoints() == 100;}});`
- following JDK5 annotations db4o introduces its own annotations.
- you can use built-in enums
- db4o for JDK5 also has replication support.

Android

Read about how to use db4o on Android here.

Security Requirements

Java Security Manager can be used to specify Java application security permissions. It is usually provided by web-browsers and web-servers for applet and servlet execution, however any Java application can make use of a security manager. For example, to use the default security manager you will only need to pass `-Djava.security.manager` option to JVM command line. Custom security managers can be created and utilized as well (please refer to Java documentation for more information).

If you are going to use db4o in a Tomcat servlet container you will need to grant some additional permissions in {CATALINA_HOME}/conf/catalina.policy file:

```
// The permissions granted to the context
WEB-INF/classes directory
grant codeBase "file:${catalina.home}/webapps/{your_db4o_application}/WEB-INF/classes/-"
{
       permission java.util.PropertyPermission "user.home", "read";
       permission java.util.PropertyPermission "java.fullversion", "read";
       permission java.io.FilePermission "path_to_db4o_database_folder", "read";
       permission java.io.FilePermission "path_to_db4o_database_file", "read, write";
};
// The permissions granted to the context WEB-INF/lib directory, containing db4o jar
grant codeBase "file:${catalina.home}/webapps/{your_db4o_application}/WEB-INF/lib/-"
{
       permission java.io.FilePermission "path_to_db4o_database_file", "read, write";
};
```

An example catalina.policy file can be downloaded here.

In order to avoid db4o DatabaseFileLocked exception you will also need to add some configuration before opening the object container:

```
Configuration config = Db4o.newConfiguration();
config.lockDatabaseFile(false);
ObjectContainer container = Db4o.openFile(config, dbfile.getPath());
```

Having done that, you can package and deploy your application. To enable the security configuration start Tomcat with the following command:

{CATALINA_HOME}/bin/catalina start -security

Xml Import-Export In .NET

One of the most widely used platform independent formats of data exchange today is xml.

db4o does not provide any specific API to be used for XML import/export, but with the variety of XML serialization libraries available.

All that you need to export your database/query results is:

1. Retrieve objects from the database.
2. Serialize them in XML format using your favorite serialization API/library
3. Save XML stream (to a disc location, into memory, into another database).

Import process is just the reverse:

1. Read the XML stream.
2. Create an objects from XML.
3. Save objects to db4o.

Let's go through a simple example. We will use XStream library (http://xstream.codehaus.org/) for object serialization, but any other tool capable of serializing objects into XML will do as well.

```
ObjectContainer container = Db4oEmbedded.openFile("database.db4o");
try {
    Pilot[] pilots = container.query(Pilot.class).toArray(new Pilot[0]);
    XStream xstream = new XStream();
    OutputStream stream = new FileOutputStream("pilots.xml");
    try {
        xstream.toXML(pilots, stream);
    } finally {
        stream.close();
    }
} finally {
    container.close();
}
```

XMLSerialisationExamples.java: Serialize to XML

After the method executes all car objects from the database will be stored in the export file as an array. Note that child objects (Pilot) are stored as well without any additional settings. You can check the created XML file to see how it looks like.

Now we can clean the database and try to recreate it from the XML file:

```
ObjectContainer container = Db4oEmbedded.openFile("database.db4o");
try {
    XStream xstream = new XStream();
    InputStream stream = new FileInputStream("pilots.xml");
    try {
        Pilot[] pilots = (Pilot[]) xstream.fromXML(stream);
        for (Pilot pilot : pilots) {
            container.store(pilot);
        }
    } finally {
        stream.close();
    }
} finally {
    container.close();
}
```

XMLSerialisationExamples.java: Read objects from XML

Obviously there is much more to XML serialization: renaming fields, storing collections, selective persistence etc. You should be able to find detailed description together with the serialization library, which you will use.

OSGi

db4o can run in an **OSGi**[1] environment. db4o jars have a OSGi manifest so that you can use them in your project.

Using db4o with OSGi

In principal you can use db4o in OSGi without any special configuration, as long as the stored objects and db4o are loaded with the same OSGi classloader.

---

[1] OSGi framework, a module system and service platform

However as soon as you're using multiple bundles, and therefore multiple classloaders, you need to configure db4o. You can specify the class-loader explicitly for db4o, or even use multiple class-loaders. This way you can lookup in multiple class-loaders for your class. First write an implementation of the JdkLoader interface:

```
class OSGiLoader implements JdkLoader{
    private   final Bundle[] bundlesToLookIn;

    OSGiLoader(Bundle... bundlesToLookIn) {
        this.bundlesToLookIn = bundlesToLookIn;
    }

    public Class loadClass(String s) {
        for (Bundle bundle : bundlesToLookIn) {
            try {
                return bundle.loadClass(s);
            } catch (ClassNotFoundException e) {
                // just retry on other bundles
            }
        }
        try {
            return getClass().getClassLoader().loadClass(s);
        } catch (ClassNotFoundException e) {
            return   null;
        }
    }

    public Object deepClone(Object o) {
        return   new OSGiLoader(bundlesToLookIn);
    }
}
```

UsingDb4oDirectly.java: Load classes from multiple bundles

After that you can use your implementation to load classes for db4o:

```
// Specify the bundles from which the classes are used to store objects
Bundle[] bundles = new Bundle[]{bundleContext.getBundle()};
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common()
        .reflectWith(new JdkReflector(new OSGiLoader(bundles)));
```

UsingDb4oDirectly.java: Db4o with a special OSGi loader

Note also that some db4o jars have a dependency on Ant and need an Ant bundle in your OSGi application to use all features.

Using the db4o OSGi package

An alternative is to use the db4o OSGi package, which is in the db4o--X.XX-osgi.jar and includes all dependencies. See "Dependency Overview" on page 252. This provides a OSGi service which creates a db4o instance for you. It configures db4o in such a way that classes are loaded from the requesting OSGi service.

```
ServiceReference reference = bundleContext.getServiceReference("com.db4o.osgi.Db4oService");
Db4oService db4oService = (Db4oService) bundleContext.getService(reference);
ObjectContainer container = db4oService.openFile("database.db4o");
```

UsingDb4oViaService.java: Get db4o osgi service

Usage Pitfalls

The db4o team tries hard to make the db4o as easy to use as possible. However there are still some pitfalls.

When objects are only partially loaded and you run into null pointer exceptions, then you ran into not activated objects. This is a common issue. See "The Activation Pitfall" on page 219

When updates are not stored then you have issues with the update-depth.See "Update Depth Pitfall" on page 220

db4o tries hard to store every object without any mapping. However in practice the mapping is not perfect. For UUID you should add a configuration to support them propertly. See "UUID Support" on page 99. For big math types you need some additional configuration See "Storing BigDecimal And BigInteger" on page 221. Also be careful with collections and avoid SQL-Types. See "Special Type Handling" on page 221

In more complex Java environments you may run into class loading issues. See "Classloader And Generic Classes" on page 222

There are a few dangerous practices which you really should avoid. See "Dangerous Practices" on page 223

db4o has database size limit, which is by default 2 GByte. See how you can increase it. See "Working With Large Amounts Of Data" on page 223

The Activation Pitfall

In order to work effectively with db4o you should understand the concept of Activation. Activation controls the amount of objects loaded into the memory. There are two main pitfalls that you should be aware about.

Accessing Not Activated Objects

One common pitfall is to access unactivated objects. This usually results in null pointer exceptions or invalid values. This happens when you navigate beyond the activated object-graph area. For example, we have a complex relationships and follow them:

```
final Person jodie = queryForJodie(container);
Person julia = jodie.mother().mother().mother().mother().mother();
// This will print null
// Because julia is not activated
// and therefore all fields are not set
System.out.println(julia.getName());
// This will throw a NullPointerException.
// Because julia is not activated
// and therefore all fields are not set
String joannaName = julia.mother().getName();
```

ActivationDepthPitfall.java: Run into not activated objects

This will result in a exception. Because by default db4o only activates objects up the a depth of 5. This means that when you load an object, that object and all objects which are reachable via 4 references are activated.

There are multiple solutions to this issue.

- Activate the object explicitly as you dive deeper into the object graph.
- Increase the global activation-depth.
- Increase the activation-depth for certain types.
- Use wisely the cascading activation.
- The most elegant solution is transparent activation. With transparent activation db4o takes care of activating object as you access them.

To High Activation Depth Or Two Many Cascade Activation

Having a high activation-depth makes working with db4o much easier. However activation can take a long time with deeper object graphs and become a serious performance bottleneck. The same applies when using cascade activation on almost all types. To reduce the time spend on activating objects, you need to be more selective about what to activate and what not.

- Activate the object explicitly as you dive deeper into the object graph.
- The most elegant solution is transparent activation. With transparent activation db4o takes care of activating object as you access them.

Update Depth Pitfall

db4o update behavior is regulated by Update Depth. Understanding update depth will help you to improve performance and avoid unnecessary memory usage. A common pitfall is that the update-depth is to small and that the objects are not updated. In such cases you either need to explicitly store the related objects individually or increase the update-depth.

For example in this code we add a new friend and store the object. Since a collection is also handled like a regular object, it is affected by the update-depth. In this example we only store the person-object, but not the friend-collection-object. Therefore with the default-update depth of one the update isn't store. In order to update this properly, you either need to set the update depth to two or store the friend-list explicitly.

```
ObjectContainer container = Db4oEmbedded.openFile(DATABASE_FILE);
try {
    Person jodie = queryForJodie(container);
    jodie.add(new Person("Jamie"));
    // Remember that a collection is also a regular object
    // so with the default-update depth of one, only the changes
    // on the person-object are stored, but not the changes on
    // the friend-list.
    container.store(jodie);
} finally {
    container.close();
}
container = Db4oEmbedded.openFile(DATABASE_FILE);
try {
    Person jodie = queryForJodie(container);
    for (Person person : jodie.getFriends()) {
        // the added friend is gone, because the update-depth is to low
        System.out.println("Friend="+person.getName());
    }
} finally {
    container.close();
}
```

UpdateDepthPitfall.java: Update doesn't work on collection

So for this example setting the update-depth to two will fix the issue. For lots of operation a update-depth of two is pretty reasonable. This allows you to update collections without storing them explicitly.

```
EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
config.common().updateDepth(2);
ObjectContainer container = Db4oEmbedded.openFile(config,DATABASE_FILE);
```

UpdateDepthPitfall.java: A higher update depth fixes the issue

When the update depth is set to a big value on objects with a deep reference hierarchy it will cause each update on the top-level object to trigger updates on the lower-level objects, which can impose a huge performance penalty.

Try transparent persistence, which removes the issue of the update-depth completly.

Storing BigDecimal And BigInteger

Problem

BigDecimal or BigInteger objects can't be stored by db4o with the default configuration because those object contains some transient state. You need to add big decimal support to db4o to store a BigDecimal or a BigInteger. See "BigMath" on page 100

Special Type Handling

For some often used types db4o implements a special type-handling. This can inflict some issues.

SQL-Date cannot be stored properly

The java.sql.Date extends the standard java.util.Date-type. The date-type is treated like other value-types such as strings, integers etc. Unfortunatly this isn't true for the java.sql.Date-type. This also means that the SQL-date type cannot be store correctly. When you retrieve a SQL-Date type it will contain the wrong date.

The solution is to use only the normal JDK-Date. Theirs no reason to use the java.sql.Date-type.

Be Careful with own Collection-Types

db4o uses special type-handlers for collections to improve efficiency. This also means that db4o makes some assumptions about the collection. It assumes that all collection-methods are implemented and work properly. This isn't true for special collections like the JDK unemployable collections or maybe for you own special collection.

For example this simple collection which doesn't implement all methods:

```
class MyFixedSizeCollection<E> extends AbstractCollection<E>{
    private E[] items;

    public MyFixedSizeCollection(E[] items) {
        this.items = items;
    }

    @Override
    public Iterator<E> iterator() {
        return Arrays.asList(items).iterator();
    }

    @Override
    public   int size() {
        return items.length;
    }
}
```

MyFixedSizeCollection.java: This collection doesn't implement all collection methods

This collection will create exceptions when you try to load it. Because db4o uses the clear and add-methods on the collection.

```
try {
    ObjectSet<CollectionHolder> holders = container.query(CollectionHolder.class);
    MyFixedSizeCollection<String> collection = holders.get(0).getNames();
} catch (Exception e) {
    // this will fail! The db4o collection-storage
    // assumes that collections support all operations of the collection interface.
    // db4o uses the regular collection-methods to restore the instance.
    e.printStackTrace();
}
```

TypeHandlingEdgeCases.java: db4o fails to load partially implemented collections

Classloader And Generic Classes

db4o uses class information available from the classloader to store and recreate class objects. When a class definition is not available from the classloader db4o resolves to Generic Objects, which represent the class information stored in object arrays. With this approach db4o is ready to function both with and without class definitions available. However, the problem can appear when your application and db4o use different classloaders, because in this case db4o won't match objects in the database to their definitions in the runtime. In order to avoid this:

1. Make sure that your db4o lib is not in JRE or JDK lib folder. Libraries in these folders get a special

classloader, which is unaware of your application classes. Instead put db4o library into any other suitable for you location and make it available to your application through CLASSPATH or using IDE provided methods.

2. If your application design does not guarantee that application classes and db4o will be loaded by the same classloader, specify the right classloader with setting the reflector explicitly.

The above-mentioned cases should be distinguished from a case when Java application uses a db4o database created from a .NET application. In this particular case .NET class definitions should be replaced by Java class definitions with the help of Aliases.

Dangerous Practices

Db4o databases are well protected against corruption. However some specific configurations can make your database file vulnerable.

- Disabling the file-lock. When two db4o instance write to the same database file at the same time, the database will be corrupted. Therefore you should avoid disabling this setting. You can safely disable the file-lock, when you're using the database in a read-only mode.
- Using the non-flushing storage. This storage will disable the flush-operation to the disk. While this improves the performance, it endangers consistent commits in a crash.
- db4o cannot deal with some class-hierarchy-changes. You cannot add a class between two existing classes in the class-hierarchy. Or remove a class from the top of the class-hierarchy. See "Refactoring Class Hierarchy" on page 113
- You cannot change a field from a array to a simple field of the same type and back.This only applies when you change it from a type to the same array-type. So for example from a string to an array of strings. A change from string to an array of integer is fine. See "Field Refactoring Limitation" on page 112

Working With Large Amounts Of Data

The following paragraphs highlight some information important for using db4o with large data.

Size of Database Files

In the default setting, the maximum database file size is 2GB. You can increase this value by configuring the internal db4o block size. The maximum possible size is 254GB.

You cannot change this setting for an existing database. In order to change it for an existing database, you need to defragment the database.

Troubleshooting

Here are a few tips on troubleshooting issues with db4o.

General Diagnostics

To get more diagnostics information you can register diagnostics listeners.

- See "Diagnostics" on page 126
- See "Debug Messaging System"
- See "Runtime Monitoring" on page 121

Investigate a Corrupt Database

Take a look at this small guide which gives tips on how to deal with a corrupt database. See "Deal With a Corrupt Database" on page 224

Performance and Scalability Issues

If you experience issues with the performance of db4o take a look at this topic: See "Performance and Scalability Issues" on page 225

Ask for Help and Advice on the db4o Forums

If you hit a problem you can ask for help in the db4o forums. You can find the forums here: http://-community.versant.com/Forums.aspx

Deal With a Corrupt Database

In general you should do regular backups of your database. db4o tries to do its best to never corrupt the database. However due to bugs or external reasons corruption can occur. Check also your configuration to not use any risky settings.

If you have a corrupted database you may think about reporting it: See "Report Bugs" on page 237

The fist step is to run the consistency check on your database. If the database is really corrupt you can try to restore at least some your data.

Consistency Check

To run consistency checks use the com.db4o.consistency.ConsistencyChecker class, like this:

```
java -cp db4oX-X.XX-all-java5.jar com.db4o.consistency.ConsistencyChecker databaseFile.db4o
```

This consistency check doesn't check the content of objects. It only checks if the overall structure of the database file is still intact. Also it doesn't offer any repair functionality. It only tells you if the file is corrupted or not.

When the tool reports inconsistencies you've defiantly have a corrupted database. Then you should fall back to the latest backup or try to recover parts of the database.

In case the tool doesn't report any issues you maybe still have a corrupted object in your database. In that case you usually can read all intact objects and copy them to a new database.

Try to Restore Intact Objects

When you cannot read some particular object then you can try to read and copy the intact objects over to a new database. You can get the id of all objects for a certain type. With these ids you then can load each single object. When loading an object fails then that object is lost. However you still can store the intact objects to another storage.

```
final    long[] idsOfPersons = container.ext().storedClass(Person.class).getIDs();
for (long id : idsOfPersons) {
    try{
        final Person person = (Person)container.ext().getByID(id);
        container.ext().activate(person,1);
        // store the person to another database
        System.out.println("This object is ok "+person);
    } catch (Exception e){
        System.out.println("We couldn't read the object with the id "+id +" anymore." +
                " It is lost");
        e.printStackTrace();
    }
}
```

TryToReadSingleObjects.java: Try to read the intact objects

Performance and Scalability Issues

You are experiencing speed or scalability problems with db4o? In this section gives an overview of the common issues.

Understand db4o's Limits

db4o is built for embedded use cases with small databases. It isn't optimized for database larger than a few gigabytes. See "Increasing the Maximum Database File Size" on page 227

db4o is explicitly single-threaded. Concurrent accesses will be synchronized against a global database lock.That means db4o cannot deal with a highly concurrent access, since it blocks on all operations. When you expect a high load and concurrent access, then you should consider a larger database like the Versant object database.

Loading Performance: Monitor Your Activation

One of the most common performance issues is that too many objects are activated. You should try to keep activation to a minimum. You can monitor activation of your objects either by installing a event handler or by using the monitoring capabilities.

Check your configuration for the global activation depth and cascading activations. These settings are often responsible for activating too many objects. In general consider transparent activation to avoid activating unnecessary objects.

Query Performance

If you have trouble with query performance take a look here.

Degrading Performance over Time

If your performance degrades over time you might need to defragement your database. See "Defragment" on page 79

Database Size Issues.

By default db4o databases can only grow up to two Gigabytes. For increasing that size take a look here. Also take a look what contributes to the database file size.

Tune Storage Adapters

A big performance point is how db4o stores the data to disk. By default db4o uses a relatively small file cache. A bigger cache might improve the performance. In general choosing the right storage can increase performance significantly.

In Client-Server Mode: Tune Prefetching Options

In client server mode a major performance bottleneck are network roundtrip's. You can reduce these by prefetching objects. Try to tune the prefetch options on the client settings.

Tune B-Tree Node Size

Another parameter which can influences performance quite bit is the size of the B-tree nodes. You can tune this parameter in the common-configuration.

Query Performance

Native Queries

Check the Required Dependencies

When you are using native queries you should ensure that the query is optimized. First check that the required jars are referenced.

For native queries you need to add these jars: db4o-X.XX-nqopt-javax.jar, db4o-X.XX-instrumentation-javax.jar, bloat-1.0.jar. See "Dependency Overview" on page 252

Check that the Query is Optimized

Native queries try to translate the original query to a SODA query. If this optimization fails the query runs an order of magnitude slower, because it cannot use indexes and instantiates all objects.

For the native queries see: See "Native Query Optimization" on page 20

Avoid Calling Methods in Queries

The optimizer only can optimize simple access patterns. When you call a complex method within a query the optimization almost certainly fails and the query runs slowly. For example when you call your equals method on your objects. Only calling the equals method of built in types will be recognized by the optimizer. However the optimizer almost certainly cannot find out what your equals method does.

Overview of Fast/Slow Queries

Also take a look at this overview to see which kind of queries run fast and which ones slowly See "Native Queries Performance Characteristics" on page 16.

Check That Fields Are Indexed

Ensure that fields which you are using in queries are indexed. Without an index db4o has to scan through all objects in order to find the right object.

Note also that currently you cannot index arrays and collections. That means that queries which do checks on array or collection members will be slow.

Also db4o currently cannot use the index for advanced operations on strings, like end-with, contains or start-with comparisons.

SODA-Queries

Take a look at this overview to see which kind of queries run fast and which ones slowly. See "SODA Performance Characteristics" on page 33

Increasing the Maximum Database File Size

To increase the database size you need to increase the block size. This configuration setting has only an effect when you create a new database or when you defragment an existing database. See "Block Size" on page 144

When you want to know what contributes to the database file size see here.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.file().blockSize(8);
```

FileConfiguration.java: Increase block size for larger databases

Advantage

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

Effect

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

Because of possible padding for objects that are not exact multiples in length of the block size, database files will naturally tend to be bigger when a higher value is chosen. It may also leads to more file cache misses and therefore can decrease performance.

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

What Contributes to the Database Size

If you are concerned about the size of your database file, it is important to understand what contributes to it and what the strategies to keep it down are.

Object Overhead

When you create a new db4o database file it contains only the header. As soon as you start storing information the file will grow. The size overhead per object depends on the object type.

In general an object consists of value types, i.e. integers, arrays and references to other objects. If you decide to use UUIDs and version number for your objects, you will get an additional overhead.

Your Objects Data

Of course your objects consume the space which is needed to store the data. For strings this space heavily depends on the encoding. If you want to keep the database size down consider using the UTF8 encoding.

Indexes

Indexes consume additional space. Therefore you should only index fields which you actually use in queries. Furthermore you might want to remove a class-index when you never query for that type.

Block Size

Block Size is a configurable value, which defines the way information is stored in db4o database. Using bigger block sizes can result in unnecessary growth of the database. For more information see Increasing The Maximum Database File Size

Freespace

Freespace appears in db4o database after unneeded objects have been deleted. The amount of the freespace can be controlled from the configuration. Another option to get rid of the freespace is Defragment. It is a good practice to run Defragment regularly to maintain the minimum database file size.

Lost Space on Crash / Process Killed

When you are using the default freespace manager the free space is lost when the database crashes or the process is killed. This is especially true when you kill your process with the debugger while developing. In regular operations you should always try to close the database regularly. When you defragment the database the lost freespace is reclaimed.

Commit Strategies

Objects stored or updated within one db4o transaction are written to a temporary transaction area in the database file and are only durable after the transaction is committed.

Commit is a costly operation as it includes disk writes and flushes caches. Too many commits can decrease your application's performance. On the other hand long transaction increases the risk of losing your data in case of a system or a hardware failure.

Best Strategies

- You should call commit() at the end of every logical operation, at a point where you want to make sure that all the changes are permanently stored to the database.
- When you are doing a bulk insert of many (say >100 000) objects, it is a good idea to commit after every few thousand inserts, depending on the complexity of your objects. If you don't do that, there is not only a risk of losing the objects in a case of a failure, but also a good chance of running out of memory. The exact amount of inserts that can be done safely and effectively within one transaction should be tested for the concrete system and will depend on available system resources, size and complexity of objects.
- Don't forget to close db4o object container before the application exits to make sure that all the changes will be saved to disk during implicit commit.

db4o Replication System

The db4o Replication System is a separate project. You can download it and its documentation on the official db4o website.

Object Manager Enterprise

Object Manager Enterprise (**OME**[1]) is an object browser for db4o databases. OME installation can be found in /ome folder of the distribution. The zip file in this folder contains the Eclipse plugin version of OME. To install the plugin, you need to have a version of Eclipse >= 3.3 installed. Unzip the file to a folder of your choice. Then open Eclipse, select 'Help' -> 'Software Updates...' -> 'Available Software' from the menu. Choose 'Add Site...' -> 'Local...' and select the unzipped folder. Follow the Eclipse Update Manager instructions for the OME feature from here on. The actual menu structure may vary over Eclipse versions. (The above applies to Eclipse 3.4 Ganymede.) When in doubt, please refer to the Eclipse documentation on Software Updates. Alternatively, you can install the plugin manually by simply copying the contents of the 'plugins' and 'features' folders from the unzipped folder to the corresponding subfolders in the root folder of your Eclipse installation.

More Reading:

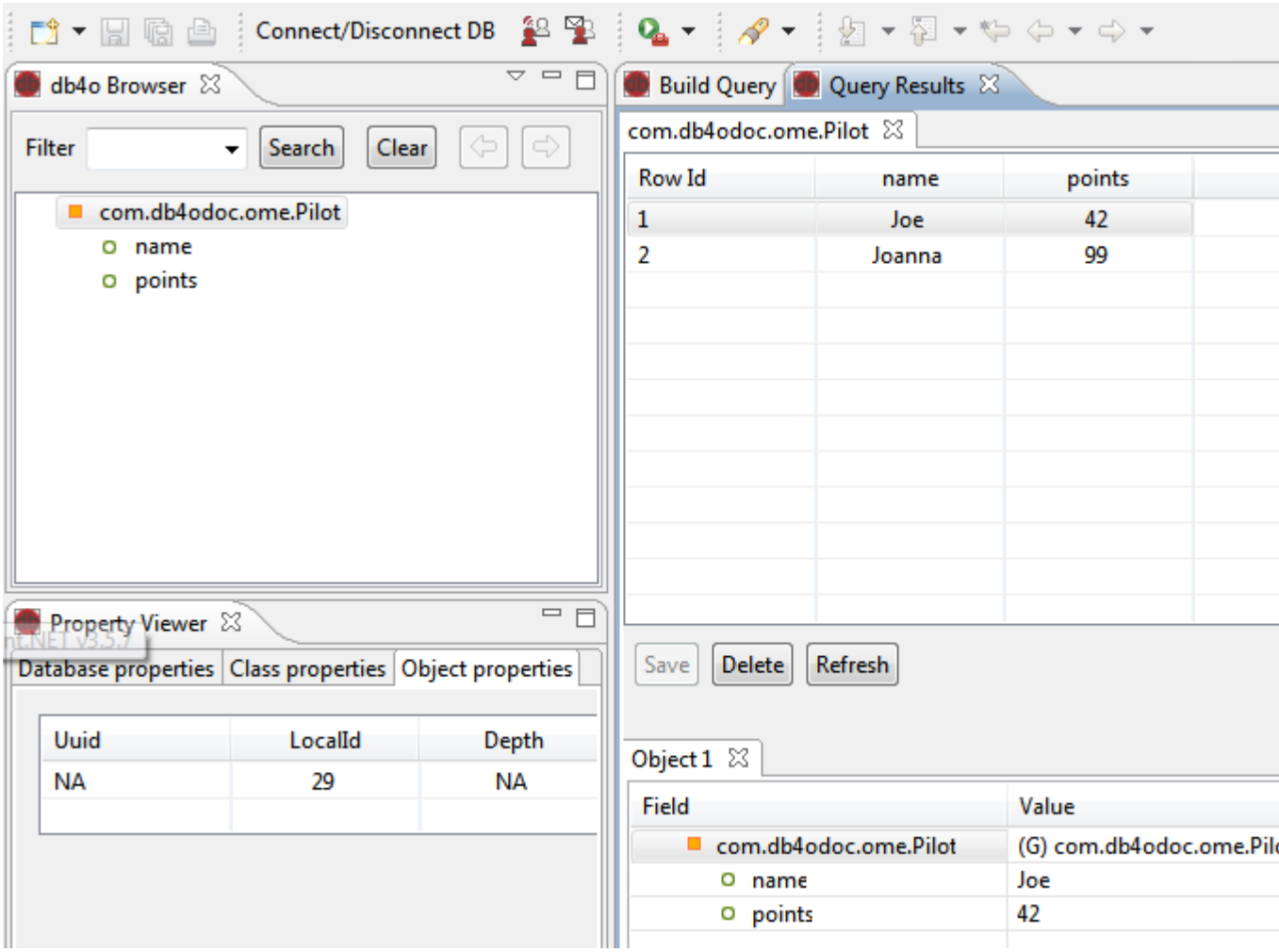
- OME Interface
- Browsing A Database
- Querying

OME Interface

Once the Object Manager (OM) is installed you can see it in Eclipse by selecting Window->Open Perspective->Other and choosing "**OME**[2]". Typically, OME window should look similar to this:

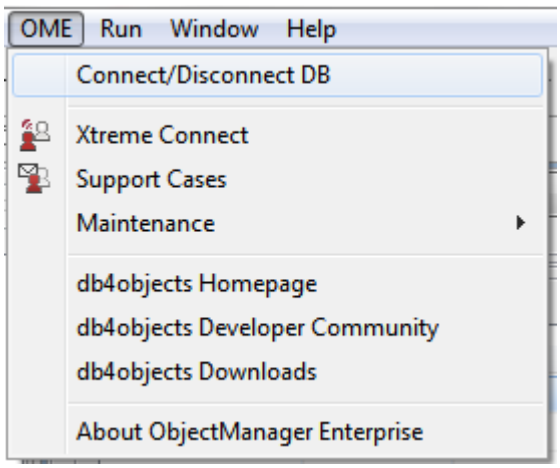
---

[1] Object Manager, a tool to view and edit a db4o database
[2] Object Manager, a tool to view and edit a db4o database



In the OME perspective you can see:

- OME menu:



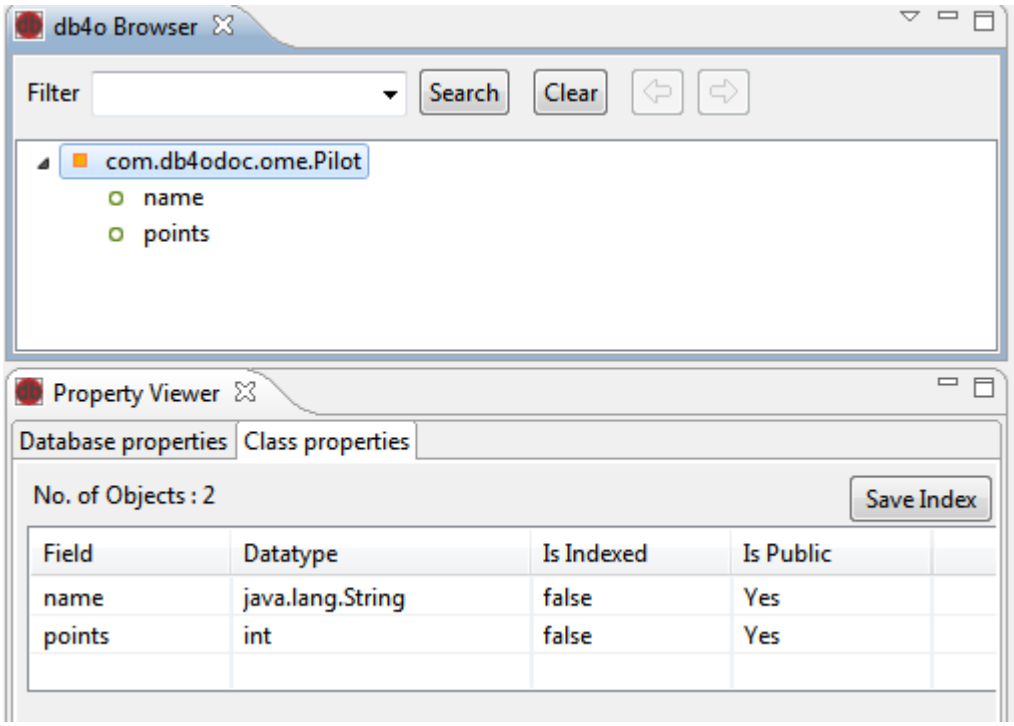
- OME toolbar buttons to access the frequently used functionality fast
- Db4o Browser: window displaying the contents of the open db4o database

- Property Viewer: window displaying the properties of the open database or the properties of the selected database class
- Build Query: windows allowing to build a query using drag&drop functionality
- Query Results: window to browse the results of the query execution

Browsing A Database

Suppose you have a simple database file containing Pilot and Car objects. Please select **OME**[1] ->Connect/Disconnect DB (or use a shortcut button from the toolbar menu) and browse to your database file.

Once you've connected you will see a screen similar to this:



The db4o Browser window in the picture above shows that there is 1 class in the database (Pilot), which contains 2 fields: name and points. In the Property Viewer you can see more information about the class fields. You can also change "Is indexed" field and add the index to the database by pressing "Save Index" button.

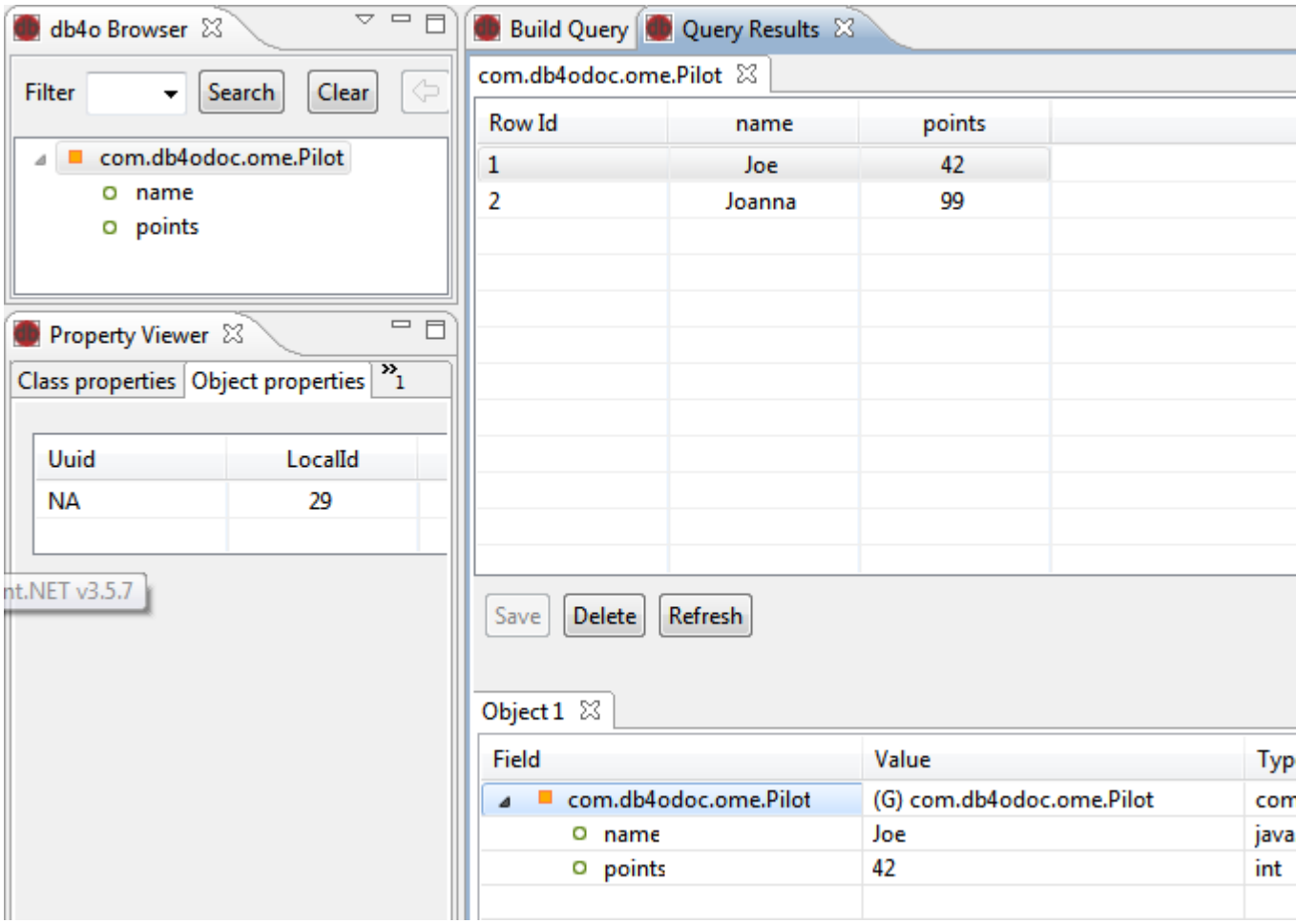
The filter panel on the top of the view allows easier navigation through the database with lots of different classes. You can use wildcard searches and benefit from the search history to make the selection faster.

Querying

It is easy to retrieve all of the Pilot instances from the database: just right-click the Pilot class in db4o Browser and select "View All Objects". The list of the Pilot objects will be shown in the Query Result view:

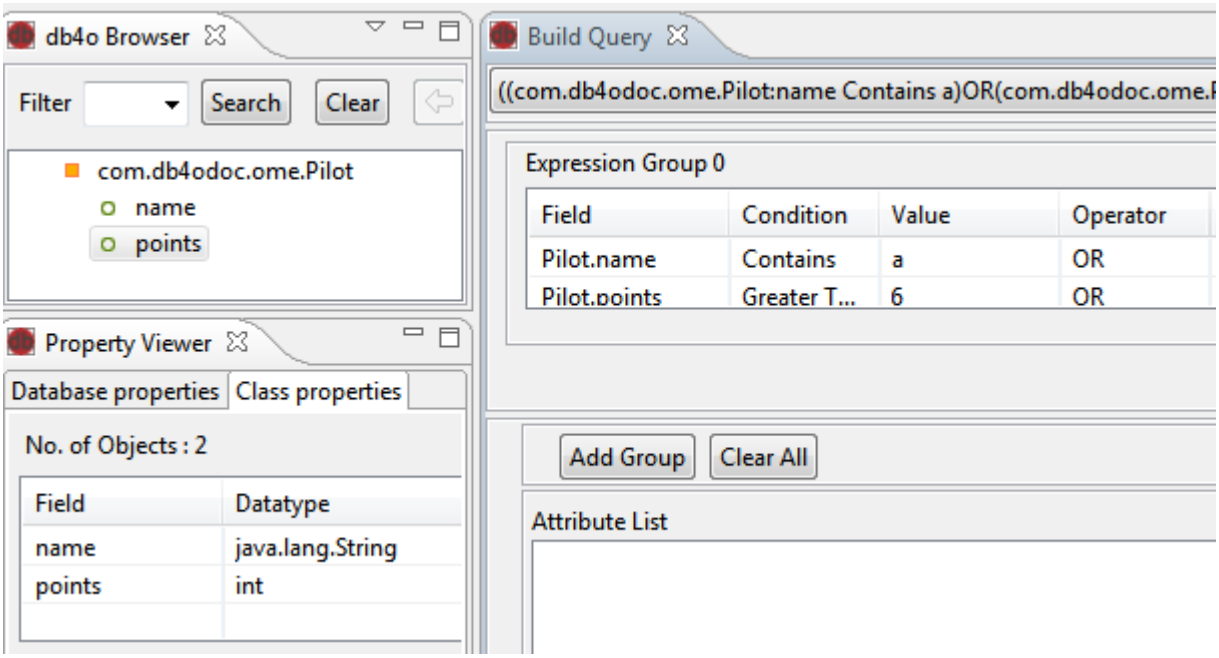
---

[1] Object Manager, a tool to view and edit a db4o database



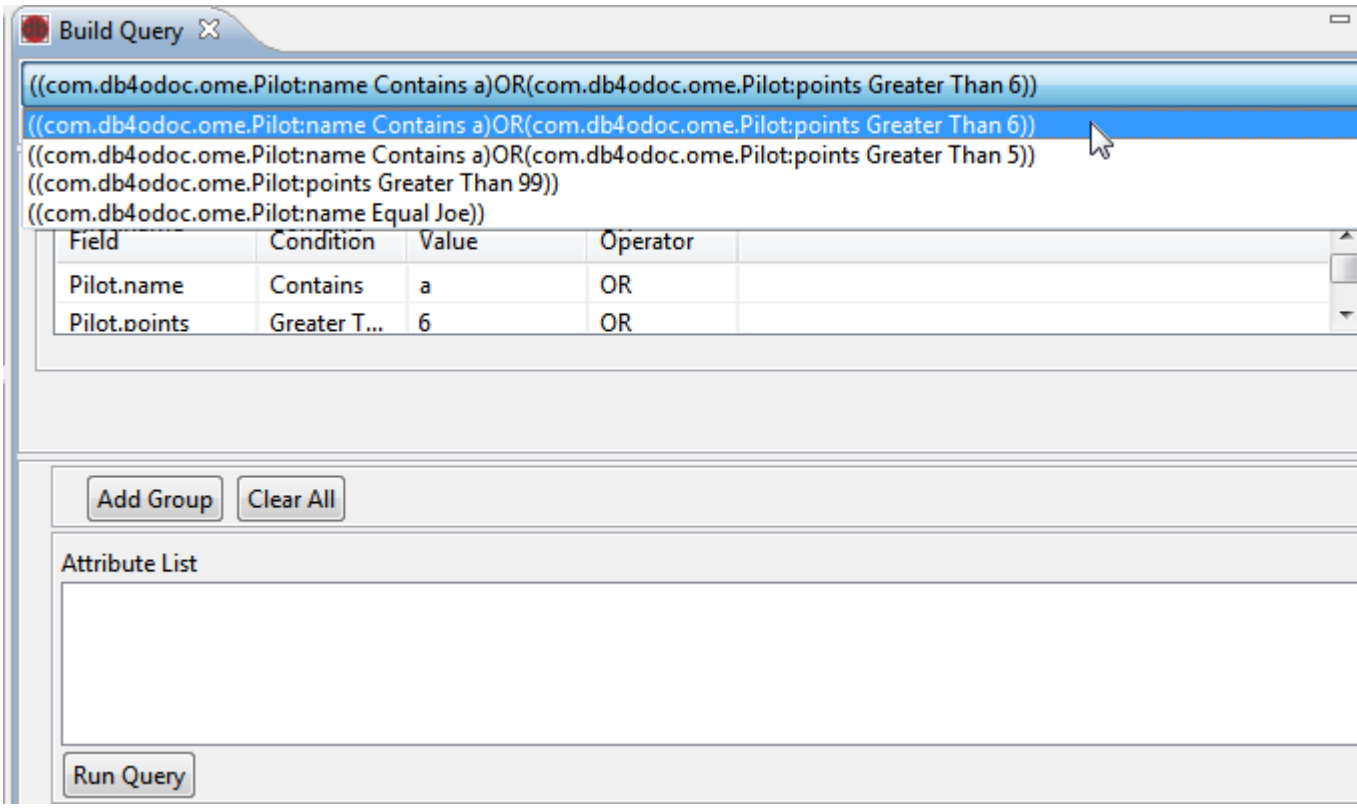
You can see object details in the detailed view below. Try to change any values and use Save button to persist the changes to the database. You can also use Delete button to delete objects from the database. For the objects containing field objects you will be prompted to use cascade on delete.

More complex queries can be done by using Build Query view:



Drag "points" field from the db4oBrowser view into the Build Query view, set condition "Greater Than", put a value "99" and run the query. You can return to the Built Query tab and modify the query later on again. For example: add "AND" operator, drag "name" field and set the value to "Michael Schumacher". Re-run the query.

When the new query is created, the previous query is stored and can be selected from the history drop-down:



More sophisticated queries can be build by joining grouped constraints using "Add Group" button.

When you are done working with the database in **OME**[1], you can close the connection by using OME->Connect/Disconnect DB menu command or by using the equivalent button on the toolbar.

Custom Configuration

When the database cannot be opened with OMJ you should try to pass your configuration to OMJ. This ensures that OMJ opens the database the same way as your application does.

Prepare Configuration Jar File

First you need to prepare a jar-file which contains code to configure OMJ. You do this by implementing the EmbeddedConfigurationItem-interface or implementing ClientConfigurationItem-interface.

In the implementation you can apply all the configuration settings to the object container which you use in your application.

```
public  class ConfigureDBForOmj implements EmbeddedConfigurationItem{
    public  void prepare(EmbeddedConfiguration embeddedConfiguration) {
        // Your configuration goes here.
        // For example:
        embeddedConfiguration.common().add(new UuidSupport());
    }

    public  void apply(EmbeddedObjectContainer embeddedObjectContainer) {
    }
}
```

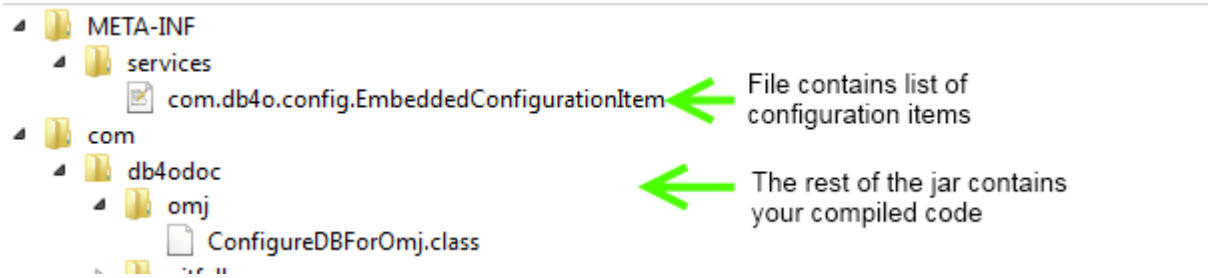
ConfigureDBForOmj.java: A configuration item for Java

Then create the file 'com.db4o.config.EmbeddedConfigurationItem' in the 'META-INF/services' folder of your jar. Add the fully qualified name of your EmbeddedConfigurationItem/ClientConfigurationItem-implementation. You can list as many implementations as you want.

```
com.db4odoc.omj.ConfigureDBForOmj
```

com.db4o.config.EmbeddedConfigurationItem: List all configuration items for OMJ

Finally you need to package the compiled code and the 'META-INF/service' folder into a jar-file. The jar-layout should be like this:



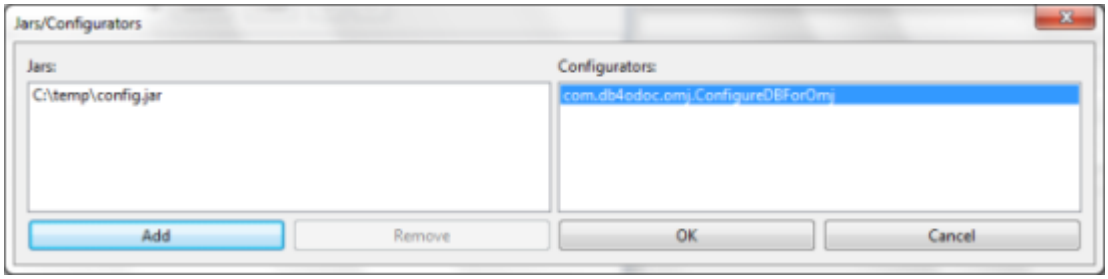
Using the Configuration in OMJ

After that you can choose the 'Custom config...' option on the open-dialog. There add the jar-file you've build previously. On the right all configuration-items are listed. Select the appropriate configuration

---

[1] Object Manager, a tool to view and edit a db4o database

items for your database.



Problems?

Are you experiencing problems with the object manager? You can take a look at the error log in the Eclipse workspace. The log is located here: "workspace/.metadata/.log". Maybe you need to first clean up the log and then reproduce the issue.

When you are using advanced configuration features like type handlers, special string encoding etc. you should add your configuration to the object manager.

Otherwise ask for help in the db4o forums. When you see errors in the log please include them in your post.

Community

Participate in the db4o community and help to improve db4o. The main community website for db4o is http://community.versant.com/. There you can find the db4o forums, blog posts additional resources and the newest db4o releases.

db4o Forums

The best place to ask questions about db4o, discuss and make suggestions are the db4o forums. You can find the db4o forums here:http://community.versant.com/Forums.aspx

Additional Resources on the Web

To stay informed about new features, community activity and other news subscribe to the different db4o blogs: http://community.versant.com/Blogs.aspx

Or follow us on Twitter: http://twitter.com/db4objects

Report Bugs

Every software contains bugs and so does db4o. If you find bugs please report them to us. In case you're not sure if you really found a bug, you maybe want to discuss your issue first on the db4o forum. As soon as you're sure that you found a bug it best to report in our bug-tracking system. The login is the same as in the db4o forums. Read more on tips and tricks for reporting bugs here.

When possible include a small test program which reproduces the issue with the bug report. A test case avoids confusion and effort to reproduce the bug.

Working with the Source Code

See "Working With Source Code" on page 238

Report Bugs

Found a bug in db4o? Here are a few tips and tricks for reporting bugs.

Where Do I Report Bugs?

The best way to report a bug is to report it on our issue tracking system. The login for the bug tracker is the same as in the db4o community page and db4o forums. If you don't have a login yet you can register here.

In cases you are not sure if the issue is an bug or another problem you maybe want to discuss it in the db4o forums.

What Should the Bug Report Contain?

The bug report should be as specific as possible, so that the room for interpretation is very small:

- Steps to reproduce the bug. In order to fix the bug a step by step instruction on how to reproduce the issue is essential.
- When possible include a the test-code which reproduces the issue.
- Include the db4o version in your bug report. The complete version of db4o is included in the jar-name.

- Include information about your platform. Which Java Version and what kind of device (like Android). Which .NET version and what kind of device (for example a embedded system)
- In case of exceptions, please add also the stack trace.

Working With Source Code

db4o is an open-source project. The source is available for reviewing, modifying for own needs or contributing your modifications. You can use the source code from the downloaded distribution package or you may use our SVN repository to get the latest modifications.

SVN repository can be found at: https://source.db4o.com/db4o/trunk/

More Reading:

- Unit Tests for db4o
- Sharpen: Converting Java to C#
- Using The Sources From db4o Distribution
- Using The Repository
- Building Full Distribution
- Building Java Version
- Patch Submission

Sharpen

Sharpen[1] is an tool that allows you to convert your Java db4o project into C#. The difference between sharpen and other java-to-C# converters is the native support for db4o, .NET naming conventions and customization options.

More Reading:

- How To Setup Sharpen
- Doing a Sharpen Conversion
- Sharpen Command-Line Arguments
- Sharpen Annotations

How to Setup Sharpen

You can obtain sharpen source from db4o svn repository at:

https://source.db4o.com/db4o/trunk/sharpen

For the ease of use check-out sharpen projects:

- sharpen.builder
- sharpen.core
- sharpen.ui
- sharpen.ui.tests

---

[1] Sharpen is a tool to translate Java source code to C# source code

Additionally **Sharpen**[1] requires a valid Eclipse installation to run. Install Eclipse on your machine or reuse an existing installation. Check out the four projects into the same workspace as your java project which you want to convert. This is not required is easier to use and maintain.

The first step is to build sharpen. For that we can use Apache Ant:

```
<target  name="build-sharpen">
    <property  name="sharpen.core.dir"  location="${dir.workspace}/sharpen.core"/>
    <reset-dir  dir="${dir.dist.classes.sharp}"/>

    <javac  fork="true"
           debug="true"
           target="1.5"
           source="1.5"
           destdir="${dir.dist.classes.sharp}"
           srcdir="${sharpen.core.dir}/src"
           encoding="UTF-8">
        <classpath>
            <fileset  dir="${eclipse.home}/plugins">
                <include  name="org.eclipse.osgi_*.jar"/>
                <include  name="org.eclipse.core.resources_*.jar"/>
                <include  name="org.eclipse.core.runtime_*.jar"/>
                <include  name="org.eclipse.jdt.core_*.jar"/>
                <include  name="org.eclipse.jdt.launching_*.jar"/>
                <include  name="org.eclipse.equinox.*.jar"/>
                <include  name="org.eclipse.core.jobs_*.jar"/>
            </fileset>
        </classpath>
    </javac>
    <jar  destfile="${dir.dist.classes.sharp}/sharpen.core_1.0.0.jar"  basedir="${dir.dist.classes.sharp}">
        <fileset  dir="${sharpen.core.dir}">
            <include  name="plugin.xml"/>
        </fileset>
    </jar>
</target>
```

sharpen-install.xml: Build Sharpen

To run Sharpen you should install it to an Eclipse instance:

```
<target  name="install-sharpen-plugin"  depends="build-sharpen">
    <copyfile  src="${dir.dist.classes.sharp}/sharpen.core_1.0.0.jar"  dest="${plugins.home}/sharpen.core_1.0.0.
</target>
```

sharpen-install.xml: Install Sharpen to Eclipse

Put the paths for the build in a property file, so that you can easily change them. Here's a example of the property file. You have to configure the JDK-path, the Eclipse path and the path to the Sharpen source.

---

[1] Sharpen is a tool to translate Java source code to C# source code

```
#The workspace where the sharpen projects are
dir.workspace=C:/Users/Gamlor/Develop/db4o/sharpenProject
# Java executable
jdk.home=${env.JAVA_HOME}
jdk.home.java=${jdk.home}/bin/java.exe
# Eclipse home directory
eclipse.home=C:/progs/eclipse
# Sandcastle can be used to convert javadoc to .NET xml comments
# dir.lib.sandcastle=e:/sandcastle/
# sharpen compile directory
dir.dist.classes.sharp=${dir.workspace}/dist/
# Eclipse plugins home
plugins.home=${eclipse.home}/plugins
```

sharpen.properties: The configuration for building sharpen

After that Sharpen is set up to run.

Doing a Sharpen Conversion

Ensure that you've installed sharpen to an existing eclipse installation as explained here.

Use Ant scripts to run **Sharpen** [1] and translate your Java code to C#. The best way for this is to define an Ant macro which you then can reuse. This task takes two arguments. The first argument is the path to a valid Eclipse workspace which contains the project to translate. The second parameter is the project in the workspace which you want to translate.

[1] Sharpen is a tool to translate Java source code to C# source code

```
<macrodef  name="sharpen">
    <attribute  name="workspace"/>
    <attribute  name="resource"/>

    <element  name="args"  optional="yes"/>

    <sequential>
        <java  taskname="sharpen"
              fork="true"
              classname="org.eclipse.core.launcher.Main"
              failonerror="true"  timeout="1800000">

            <classpath>
                <fileset  dir="${eclipse.home}/plugins">
                    <include  name="org.eclipse.equinox.launcher_*.jar"/>
                </fileset>
            </classpath>

            <arg  value="-clean"/>
            <arg  value="-data"/>
            <arg  file="@{workspace}"/>
            <arg  value="-application"/>
            <arg  value="sharpen.core.application"/>
            <arg  value="@{resource}"/>
            <args/>
        </java>
    </sequential>
</macrodef>
```

sharpen-install.xml: The sharpen task

Now you can use this task to sharpen your project. First ensure that your project is in a valid Eclipse workspace. Then you specify the workspace and the sources of the project:

```
<target  name="sharpen">
    <sharpen
        workspace="C:\temp\sharpenExamples\"
        resource="example/src">
        <args>
            <arg  value="@sharpen-config"/>
        </args>
        </sharpen>
</target>
```

sharpen-example.xml: Sharpen a example project

Additionally you can pass the sharpen configuration as a file-name. When you add a '@' in front of the file-name sharpen will read that file and use all configuration flags of that. For example:

```
-pascalCase+
-nativeTypeSystem
-nativeInterfaces
```

You can find a list of all Sharpen configuration flags here and a list of all Sharpen annotations here.

Sharpen Command-Line Arguments

Sharpen[1] command-line arguments can be defined in an options file

```
<sharpen workspace="${target.dir}" resource="sharpened_examples/src">
   <args>
      <!-- Sharpen options are defined in a separate file -->
      <arg value="@sharpen-all-options" />
    </args>
 </sharpen>
```

Here sharpen-all-options file contains all command-line options needed to convert current project. For an example of command-line options file see the previous topic.

Command-line arguments can also be specified directly in an ant script:

```
<sharpen workspace="${dir.sharpen}" resource="db4oj/core/src">
    <args>
        <arg value="-xmldoc"/>
        <arg file="config/sharpen/ApiOverlay.xml" />
        <arg value="@sharpen-all-options" />
     </args>
 </sharpen>
```

The following table shows available command-line options, their meaning and example usage:

Argument	Usage
-pascalCase	Convert Java identifiers to Pascal case
-pascalCase+	Convert Java indentifiers and package names (namespaces) to Pascal case
-cp	Adds a new entry to classpath: <arg value="-cp" /> <arg path="lib/db4o-7.2.37.10417-java5.jar" />
-srcFolder	Adds a new source folder for sharpening
-nativeTypeSystem	Map java classes to .NET classes with a similar functionality. For example: java.lang.Class - System.Type
-nativeInterfaces	Adds an "I" in front of the interface name
-organizeUsings	Adds "using" for the types used
-fullyQualify	Converts to a fully-qualified name: -fullyQualify File
-namespaceMapping	Maps a java package name to a .NET namespace. For example: -namespaceMapping com.db4o Db4objects.Db4o
-methodMapping	Maps a java method name to a .NET method (can be method in another class). For example: - methodMapping java.util.Date.getTime Sharpen.Runtime.ToJavaMilliseconds
-typeMapping	Maps a java class to .NET type: -typeMapping com.db4o.Db4o Db4objects.Db4o.Db4oFactory

[1] Sharpen is a tool to translate Java source code to C# source code

-propertyMapping	Maps a java method to .NET property: -propertyMapping com.db4odoc.structured.Car.getPilot Pilot
-runtimeTypeName	Name of the runtime class. The runtime class provides implementation for methods that don't have a direct mapping or that are simpler to map at the language level than at the sharpen level. For instance: String.substring, String.valueOf, Exception.printStackTrace, etc. For a complete list of all the method that can be mapped to the runtime class see Configuration#runtimeMethod call hierarchy.
-header	Header comment to be added to all converted files. -header config/copyright_comment.txt
-xmldoc	Specifies an xml- overlay file, which overrides javadoc documentation for specific classes: -xmldoc config/sharpen/ApiOverlay.xml
-eventMapping	Converts the methods to an event.
-eventAddMapping	Marks the method as an event subscription method. Invocations to the method in the form <target>.method(<argument>) will be replaced by the c# event subscription idiom: <target> += <argument>
-conditionalCompilation	Add a condition when to translate the Java code - - conditionalCompilation com.db4o.db4ounit.common.cs !SILVERLIGHT
-configurationClass	Change the configuration class. The default is 'sharpen.core.DefaultConfiguration'

Sharpen Annotations

Sharpen[1] annotations decorate java source code and are used to notify sharpener about how the code should be processed and converted. Annotations can be used to specify how a code element should be converted (for example class to enum), to skip conversion of some code elements, to rename classes, to change visibility etc.

The following table shows existing annotations, their meaning and examples.

Annotation	Meaning
@sharpen.enum	Mark java class to be processed as a .NET enum
@sharpen.rename	Specifies a different name for the converted type, takes a single name argument. For example: @sharpen.rename Db4oFactory

@sharpen.private	Specifies that the element must be declared private in the converted file, though it can be not private in the java source:
	`/*`
	`* @sharpen.private`
	`*/`

[1] Sharpen is a tool to translate Java source code to C# source code

	public List4 _first;

@sharpen.internal	Specifies that the element must be declared internal in the converted file:
	`/**`
	` * @sharpen.internal`
	`*/`
	public abstract int size();

@sharpen.protected	Specifies that the element must be declared protected in the converted file:
	`/**`
	` * @sharpen.protected`
	`*/`
	public abstract int size();

@sharpen.new	Adds the C#-'new' modifier to the translated code.

@sharpen.event	Links an event to its arguments. For example:
	Java:
	`/**`
	`* @sharpen.event com.db4o.events.QueryEventArgs`
	`*/`
	`public Event4 queryStarted();`
	is converted to:
	`public delegate void QueryEventHandler(`
	`  object sender,`
	`  Db4objects.Db4o.Events.QueryEventArgs args);`
	`.......`
	`event Db4objects.Db4o.Events.QueryEventHandler QueryStarted;`

@sharpen.event.add	Marks the method as an event subscription method. Invocations to the method in the form <target>.method(<argument>) will be replaced by the c# event subscription idiom: <target> += <argument>
@sharpen.event.onAdd	Valid for event declaration only (SHARPEN_EVENT). Configures the method to be invoked whenever a new event handler is subscribed to the event.
@sharpen.if	Add #if <expression>#endif declaration: @sharpen.if <expression>

@sharpen.property	Convert a java method as a property:
	`/**`
	` * @sharpen.property`
	`*/`
	public abstract int size();

@sharpen.indexer	Marks an element as an indexer property
@sharpen.ignore	Skip the element while converting
@sharpen.ignore.extends	Ignore the extends clause in Java class definition
@sharpen.ignore.implements	Ignore the implements clause in Java class definition
@sharpen.extends	Adds an extends clause to the converted class definition. For example:

	Java:
	`/**`
	`* @sharpen.extends System.Collections.IList`
	`*/`
	`public interface ObjectSet {...`
	converts to
	`public interface IObjectSet : System.Collections.IList`

@sharpen.partial	Marks the converted class as partial
@sharpen.remove	Marks a method invocation that should be removed

@sharpen.remove.first	Removes the first line of the method/constructor when converting to C#:
	`/**`
	`* @sharpen.remove.first`
	`*/`
	`public void doSomething(){`
	`  System.out.println("Java");`
	`  NextMethod();`
	`}`
	converts to:
	`public void DoSomething(){`
	`  NextMethod();`
	`}`

@sharpen.struct	Marks class to be converted as c# struct

@sharpen.unwrap	When a method is marked with this annotation all method calls are removed. This is useful for removing conversion methods when their aren't required in C#.
	`/*`
	`* @sharpen.unwrap`
	`*/`
	`public Iterable toIterable(Object[] array){`
	`  return Arrays.asList(array);`
	`}`
	`public void doSomething(Object[] objs){`
	`  Iterable iterable = toIterable(objs);`
	`  // do something with the iterable`
	`}`
	Is converted to:
	`public IEnumerable ToIterable(object[] array){`
	`  return Arrays.AsList(array);`
	`}`
	`public void doSomething(object[] objs){`
	`  Iterable iterable = objs;`
	`  // do something with the iterable`
	`}`

@sharpen.attribute	Adds an attribute to the converted code:
	`/*`
	`* @sharpen.attribute TheAttribute`
	`*/`
	`public void doSomething(){}`
	Will be converted to:

	`[TheAttribute]` `public void DoSomething(){}`
@sharpen.macro	Add a replace-pattern macro to your code.

Db4o Testing Framework

db4ounit is a minimal xUnit (jUnit, NUnit) style testing framework. The db4ounit framework was created to fulfill the following requirements:

- The core tests should be run against JDK1.1
- It should be possible to automatically convert test cases from Java to .NET.

db4ounit design deviates from vanilla xUnit in some respect, but if you know xUnit, db4ounit should look very familiar.

db4ounit itself is completely agnostic of db4o, but there is the db4ounit.extensions module which provides a base class for db4o specific test cases with different fixtures, etc.

Db4ounit and db4ounit.extensions are supplied as a source code for both java and .NET. Java version also comes with a compiled library: db4o-X.XX-db4ounit.jar, which allows you to run your tests from a separate package.

If you've found a bug and want to supply a test case to help db4o to fix the issue quickly, the best option would be to supply your code in the java db4ounit format. This format allows very easy integration of a new test case into db4o test suite: only copy/paste is required to put your test class code into the framework using Eclipse.

More Reading:

- Creating A Sample Test
- Db4ounit Methods

Creating A Sample Test

Let's create the a extremely simple test case. The first step is to setup db4ounit. The best way is to use db4ounit directly from the source. db4ounit is in the db4o source code folder of the distribution.

Open the db4o source-code of the db4o test-projects in Eclipse (or another IDE). The db4ounit projects are called 'db4ounit' and 'db4ounit.extensions'. Then create a new project which references the db4ounit projects.

After that we're ready to write our first db4ounit test. Create a new class which inherits from AbstractDb4oTestCase. Then add a test-method. Any method which starts with the prefix 'test' is a test method. Then add a main method which starts the test.

```
public  class ExampleTestCase extends AbstractDb4oTestCase{

    public  static  void main(String[] args) {
        new ExampleTestCase().runEmbedded();
    }

    public  void testStoresElement(){
        db().store(new TestItem());
        ObjectSet<TestItem> result = db().query(TestItem.class);
        Assert.areEqual(1, result.size());
    }


    static  class TestItem{

    }
}
```

ExampleTestCase.java: Basic test case

Db4ounit Methods

Let's look through the basic API , which will help you to build your own test. This document is not a complete API reference and its intention is to give you a general idea of the methods usage and availability.

AbstractDb4oTestCase

AbstractDb4oTestCase is a base class for creating test cases. It will setup a db4o instance for you which you can use in your tests. Additionally it provides different utility methods for configuring, querying and modifying the database.

Additionally it contains methods to run the test. You can create new instance of a class which extends AbstractDb4oTestCase and run the test in different environments. For example the method 'runSolo' will run the test with an embedded local container.

TestCase

This interface provides the basic for a unit test in the db4ounit framework. When you implement this interface the class is a valid unit test. All methods starting with 'test' will be executed as test.

For most tests it is more continent to use the AbstractDb4oTestCase.

TestLifeCycle

This interface provides a test case with additional setup and tear down methods. Those will be called before and after each test method

ConsoleTestRunner

A test-runner which runs the tests as console application.

Usually it's more convenient to extend the AbstractDb4oTestCase class and use the provided run methods instead of the console test runner.

Db4ounit.Assert

This class provides a variety of methods for asserting certain conditions

Using The Sources From db4o Distribution

The sources for db4o are in the 'src' subdirectory of the db4o distribution. You simply can open these projects in Eclipse (or another IDE). Compile and use the db4o projects like any other Eclipse project.

Following projects are shipped with db4o:

- bloat: The bytecode manipulation library used by db4o.
- db4o.cs: The db4o client server mode.
- db4o.cs.optional: Optional additions for the db4o client server mode.
- db4o.instrumentation: The base for different instrumentations provided by db4o.
- db4o_osgi: The db4o **OSGi**[1] service
- db4o_osgi_test: The test cases for the db4o OSGi service.
- db4oj: The db4o core.
- db4oj.optional: Optional features for db4o.
- db4oj.tests: The db4o test-suite.
- db4onqopt: The db4o native query optimization.
- db4otaj: Transparent activation/persistence enhancers.
- db4otools: Tooling for db4o like Ant-tasks
- db4ounit: The db4o unit test framework.
- db4ounit.extensions: Utilities and extensions for the db4o unit test framework.

Using The Repository

If you enjoy being on the "cutting edge" and want to follow up with the development process, you can use our SVN repository to get the most up-to-date db4o source code.

Access to the public projects on our Subversion server is available under the following public URL. No login is required.

https://source.db4o.com/db4o/trunk/

The following projects are currently available.

Projects may be under constant development. Source code is not guaranteed to be stable.

Most top-level modules in svn directly map to Eclipse projects, i.e. the root folder contains the Eclipse project metadata.

- bloat: The byte code manipulation library db4o uses.
- db4o.cs: The client server implementation for db4o.
- db4o.cs.optional: Optional client server features for db4o.
- db4o.instrumentation: The instrumentation basic functionality for db4o.
- db4o.net: The .NET version of db4o.

---

[1] OSGi framework, a module system and service platform

- db4obuild: The build scripts for db4o.
- db4onqopt: The db4o native query optimizer.
- db4otaj: The db4o transparent activation/persistence enhancer.
- db4otools: The db4o tools like Ant scripts.
- db4ounit: The db4o test project.
- db4ounit.extensions: Additional unit test functionality.
- decaf: The Java 1.5 to Java 1.4 converter. You need to checkout all sub-projects on the same level as other db4o projects.
- drs: The db4o replication system.
- sharpen: The Java to C# converter. You need to checkout all sub-projects on the same level as other db4o projects.

Building db4o

Building full distribution will allow you to get the same db4o packages as you can get from db4o download center. However, the flexibility of the build project also allows you to get only parts of it, like only java distro, only documentation, only tests etc.

The following documentation explains how to build a full distribution using Eclipse version 3.4 Ganymede. It is assumed that you have ant and one of Eclipse SVN clients (Subclipse or Subversive) installed.

Projects Required

In order to build db4o you will need to check out the following projects.

- bloat: The byte code manipulation library db4o uses.
- db4o.cs: The client server implementation for db4o.
- db4o.cs.optional: Optional client server features for db4o.
- db4o.instrumentation: The instrumentation basic functionality for db4o.
- db4o.net: The .NET version of db4o.
- db4obuild: The build scripts for db4o.
- db4onqopt: The db4o native query optimizer.
- db4otaj: The db4o transparent activation/persistence enhancer.
- db4otools: The db4o tools like Ant scripts.
- db4ounit: The db4o test project.
- db4ounit.extensions: Additional unit test functionality.
- decaf: The Java 1.5 to Java 1.4 converter. You need to checkout all sub-projects on the same level as other db4o projects.
- drs: The db4o replication system.
- sharpen: The Java to C# converter. You need to checkout all sub-projects on the same level as other db4o projects.
- doctor: A tool for building the tutorial.
- tutorial: The db4o tutorial

machine.properties

You will need to create machine.properties file in db4obuild folder. The contents of the file can be copied from build.xml (see the comments at the beginning of the file). Modify the paths where applicable to set the build variables for your environment.

Read the instructions at top of the build.xml to find out what options are available. Here's an example:

```
file.compiler.jdk1.3=%JAVA_HOME%/bin/javac.exe
file.compiler.jdk1.3.args.optional=-source 1.3
file.jvm.jdk1.5=%JAVA_HOME%/bin/java.exe
dir.workspace=C:/Users/Gamlor/Develop/db4o/db4o-src/
eclipse.home=C:/progs/eclipse
msbuild.executable="C:/Windows/Microsoft.NET/Framework/v4.0.30319/MSBuild.exe"
```

Build Preparation

First you will need to run some preparation scripts. This is done only once per workspace and should not be repeated in the future.

Run build-db4obuild.xml, this will compile some of the tools used in the build process.

You will need to generate a key to sign the tutorial applet. Use the following commands:

keytool -genkey -alias db4objects -keyalg rsa

keytool -export -alias db4objects -file [path]/db4obuild/config/db4objects.crt

Use "kistoa" (without quotes) as your keypass and storepass.

Replace [path] with the path to db4obuild project on your system and make sure that db4objects.crt file is created in db4obuild/config folder.

If you've already generated db4objects key pair before, you will need to delete it before re-generating:

keytool -delete -alias db4objects

You will need to add ant-contrib.jar to your eclipse ant. You can download ant-contib.jar at:

http://sourceforge.net/projects/ant-contrib

- Add ant-contib jar to ant folder in eclipse/plugins.
- After this is done go to Window->Preferences menu in Eclipse.
- Select Ant->Runtime in the list.
- Then select "Ant Home Entries".
- Press "Add External Jar" and select ant-contib.jar location in the plugins folder.

Running The Build

Now everything is ready to run db4o build. Right-click build.xml file and select "Run As/Ant Build". You will need to run "buildall" target to generate java and .NET distribution.

Patch Submission

This topic explains how to prepare your patch.

Before writing a patch, please, familiarize yourself with our Coding Style conventions.

You can create a patch using "Create Patch" SVN command.

If you are using Subversive plugin you can use the following steps:

- select the project in Package Explorer;

- right-click and select Team/Create Patch;
- select "Save In File System" and choose the file name;
- click "Next" and "Finish";
- check unversioned resources that should be included in the patch;
- click "OK"

Once the patch is created you should register the functionality provided with our Jira tracking system by creating new issue, submitting the description, patch and test case if applicable.

Dependency Overview

The different functionality of db4o is implemented in multiple jars. You need to include only the jar which are required for you're application. Here's an overview:

db4o-X.XX-core-javaX.jar

This jar contains the core functionality of db4o. It doesn't have any dependencies to other jars.

db4o-X.XX-cs-javaX.jar

This jar contains the client server mode of db4o. It only depends on the db4o-core jar.

db4o-X.XX-nqopt-javax.jar

This jar contains the native query optimizations for db4o. Make sure that you include this jar when you use native queries. It depends on the db4o-core, db4o-instrumentation and the bloat jar for the runtime optimization.

When you use the compile-time optimization instead of the runtime optimization, it also requires the Ant library. However when you do compile-time optimization, you don't need this jar and its dependencies at runtime.

Note that the native queries work also without this jar in the classpath. However the queries cannot be optimized without this jar and will run a lot slower.

db4o-X.XX-optional-javaX.jar

This jar contains some additional features and functionality for db4o, like monitoring capabilities, Big-Math-support and .NET support. It depends on the db4o-core jar.

db4o-X.XX-cs.optional-javaX.jar

This jar contains additional features for the client-server-mode, like SSL and monitoring support. It depends on the db4o-cs, db4o-optional and the db4o-core jar.

db4o-X.XX-taj-javaX.jar

This jar contains the utilities to enhance your classes with transparent activation. It is normally used at compile time, but its possible to use it also at runtime. It depends on the db4o-core, db4o-instrumentation and the bloat-jar. Since the enhancement is usually done with Ant, it also requires the Ant-library.

db4o-X.XX-tools-javaX.jar

This jar contains the high level functionality for enhancing your classes. It requires the db4o-core, db4o-instrumentation, db4o-taj, db4o-nqopt and the bloat-jar. Additionally it needs the Ant-library.

db4o-X.XX-osgi.jar

This jar contains a db4o **OSGI**[1] service. It depends on the db4o-core and the db4o-cs jars.

db4o-X.XX-all-javaX.jar

This jar contains all db4o functionality with all dependencies, except the Ant-library and the db4o-osgi library. If you just want to use db4o and aren't limited by space constrains use this jar only.

---

[1] OSGi framework, a module system and service platform

License

Licensing the db4o Engine

Versant Inc. offers three difference license options for the db4o object database engine db4o:

General Public License (GPL) Version 3

db4o is free under the GPL, where it can 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 receive a copy of the GPL in the file db4o.license.txt together with the db4o distribution.

If you have questions whether the GPL is the right license for you, please read:

- db4objects and the GPL - frequently asked questions FAQ
- the free whitepaper db4objects and the Dual Licensing Model
- Versant's GPL interpretation policy for further clarification

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 commercial information on db4o website for licensing terms and pricing.

db4o Opensource Compatibility License (dOCL)

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

3rd Party Licenses

When you download the db4o distribution, you receive the following 3rd party libraries:

In java versions

- Apache Ant(Apache Software License)

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

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

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

- BLOAT(GNU LGPL)

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

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

These products are not part of db4o's licensed core offer and for development purposes. You receive and license those products directly from their respective owners.

Contacts

Join The db4o Community

Join the db4o community for help, tips and tricks. Ask for help in the db4o forums at any time. If you want to stay informed, subscribe to db4o blogs.

Address and Contact Information

Versant Corporation255 Shoreline Drive, Suite 450
Redwood City, CA 94065
USA

Phone:

+1 (650) 232-2436

Fax:

+1 (650) 232-2401

Sales:

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

Careers

career@db4o.com

Partnering
partner@db4o.com