

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. Take a look at additional resources on [the community website](#). 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](#), [Ricoh](#) and [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 than 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 dis-

tributed 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.

These requirements can be fulfilled by using an Object Oriented Database Management System (**OODBMS**³). 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:

¹Relational Database Management System

²Relational Database Management System

³Object Oriented Database Management System

- 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**¹ 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 [Seagate](#) to 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 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.

¹Relational Database Management System

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 developer's 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
Educational systems	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 a memory database, providing better performance through 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 ² . 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

¹Relational Database Management System

²db4o Replication System

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 can potentially be used for rather large databases - up to 254 GB per database file. However, this would be highly unusual and likely due to having the kind of application that was storing relatively few, but very large objects in the database. As a general rule, if you expect your database to grow beyond 10 gigabytes, you should probably be looking at the [Versant object database](#).

Also db4o is currently not designed to scale for highly concurrent access. If you expect a high load of concurrent access to the database, you should also 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 also capture to complexity of the model and don't use a simplified flat data model.
- Use client-server version to enable many clients using data at the same time.

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

¹Relational Database Management System

²Native Query

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 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.

¹Object Oriented Database Management System

Basics Operations & Concepts

This topic gives you an overview of the most important and basic operations 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 8

Basic Operations

The basic operations are, not surprising, storing, updating, querying and deleting objects. See "The Basic Operations" on page 9

```
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 11

Basic Concepts

There are some basic concepts which are used in db4o. Understanding them helps 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 27

db4o manages objects by identity. This means db4o doesn't need additional id-field on your object. See "Identity Concept" on page 30

Other concepts are more unique to db4o. For example the activation concept, which controls which objects are loaded from the storage to memory. See "Activation Concept" on page 33. The same principals are also applied when updating (See "Update Concept" on page 54) or deletion objects (See "Delete Behavior" on page 73

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 300](#)

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

Additional online resources are available here: <http://developer.db4o.com/Resources>

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 318](#). 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 file-name 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 a object

Queries

Querying for objects is also easy. There are different query interfaces available with different benefits.
See "["Querying" on page 11](#)

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.

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](#)

Concept

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

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

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 there 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 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 what it does. When the operations are 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 318. 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 132

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 an 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 132

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="${basedir}/bin">
            <include name="**/*.class"/>
        </sources>
        <jars dir="${basedir}/lib">
            <include name="*.jar"/>
            <exclude name="db4o-*.jar"/>
            <exclude name="ant.jar"/>
            <exclude name="bloat-1.0.jar"/>
        </jars>
    </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="${basedir}/bin">
            <include name="**/*.class"/>
        </sources>
        <jars dir="${basedir}/lib">
            <include name="*.jar"/>
            <exclude name="db4o-*.jar"/>
            <exclude name="ant.jar"/>
            <exclude name="bloat-1.0.jar"/>
        </jars>
    </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. See "[Enhancement For Java](#)" on page 134

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 at the various examples of query by example examples. See "[Query By Example Examples](#)" on page 17

Query by Example has fundamental limitations, which limits its use cases. See "[Query By Example Limitations](#)" on page 19

Query By Example Examples

Query By Example Basics

For a query you pass in an example object to db4o. db4o will examine the object with reflection. Each field which doesn't have the default value will be used as a constraint. 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 20

Also SODA has special capabilities for certain types like collections etc. See "SODA Special Cases Examples" on page 22

Additionally SODA evaluations can help you implementing queries which go beyond the capabilities of pure SODA queries. See "SODA Evaluations" on page 24

At last, you need a way to sort the results of a query. See "SODA Sorting" on page 25

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 constraints 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-comparison is available for strings. For this you can specify if the comparison is case sensitive or not.

```

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 .startsWith constraints 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 constraint, 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 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("db40");

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 constraints 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.

```
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. One 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 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 constraints. If the constraints are [regular SODA-constraints](#), 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 28](#)
- **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 the db4o transactions are isolated from each other. [See "db4o Transactions" on page 27](#)

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 Transaction

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 123](#)

Note that in this mode, db4o uses the read committed isolation. [See "Isolation" on page 28](#)

```
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. If two or more transactions are executed at the same time, they must be executed in a way so that transaction A won't be impacted by the intermediate data of transaction B. Note that isolation does not dictate the order of the transactions. Another important thing to understand about isolation is serializability of transactions. If the effect on the database is the same when transactions are executed concurrently or when their execution is interleaved, these transactions are called serializable.

There are several degrees of isolation to be distinguished:

- Degree 0: A transaction does not overwrite data updated by another user or process ("dirty data") of other transactions.
- Degree 1: Degree 0 plus a transaction does not commit any writes until it completes all its writes (until the end of transaction).
- Degree 2: Degree 1 plus a transaction does not read dirty data from other transactions.
- Degree 3: Degree 2 plus other transactions do not dirty data read by a transaction before the transaction commits.

The more common classification is by isolation levels:

1. **Serializable**: In this case, transactions are executed serially so that there is no concurrent data access. Transactions can also be executed concurrently but only when the illusion of serial transactions is maintained (i.e. no concurrent access to data occurs). If the system uses locks, a lock should be obtained over the whole range of selected data. If the system does not use locks, no lock is acquired; however, if the system detects a concurrent transaction in progress, which would violate the serializability illusion, it must force that transaction to rollback, and the application will have to restart the transaction.
2. **Repeatable Read**: In this case, a lock is acquired over all the data retrieved from a database. Phantom reads can occur (i.e. new data from the other committed transactions included in the result)
3. **Read Committed**: In this case, read locks are acquired on the result set, but released immediately. Write locks are acquired and released only at the end of the transaction. Non-repeatable reads can occur, i.e. deletions or modifications from the other committed transactions will be visible by the current transaction. Phantom reads are also possible. This is the model db4o uses.
4. **Read Uncommitted**: With this isolation level dirty reads are allowed. Uncommitted modifications from the other transactions are visible. Both phantom and nonrepeatable reads can occur.

Read more about [isolation levels on Wikipedia](#).

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));
```

DangerousPractices.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();
```

DangerousPractices.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 free-space 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 be 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 stored 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 171](#)

Further Information

Maybe you're wondering why db4o manages objects by identity. Why not by equality? There are good reasons why this is the case. [See "Identity Vs Equality" on page 31](#)

In order to manage objects by identity db4o has a reference cache which contains all loaded objects. [See "The Reference Cache" on page 32](#)

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 runtime 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 updated pilot with the 'John'? What happens if there are hundreds of pilots which 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?

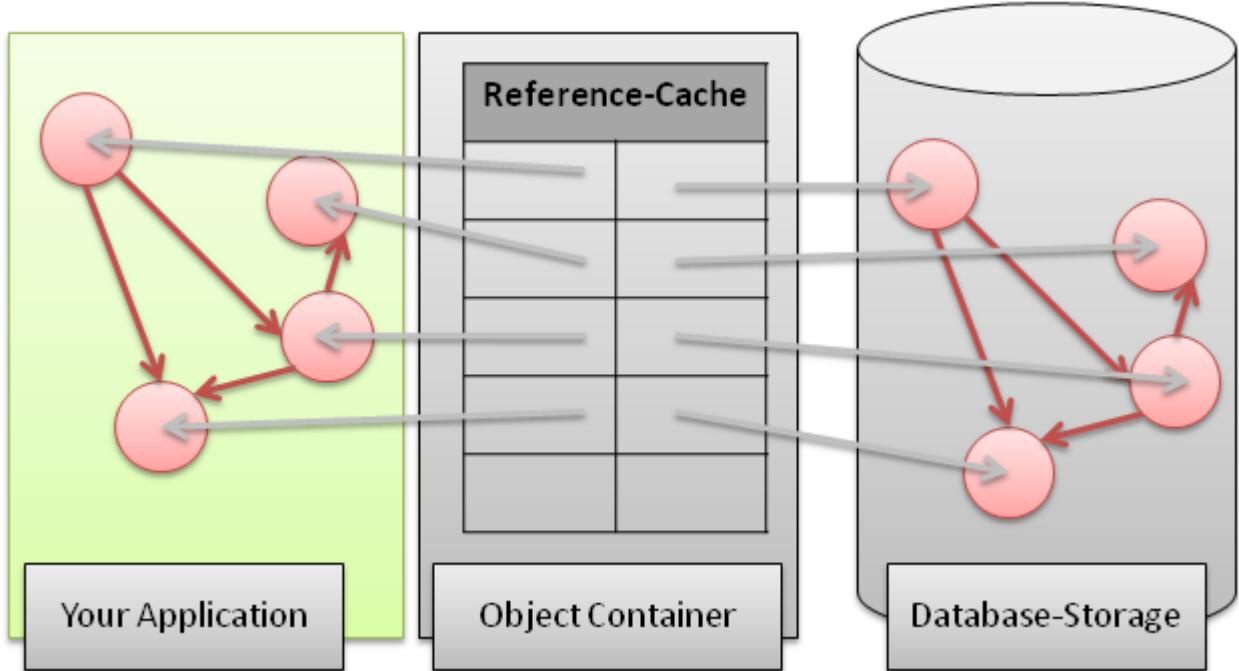
This question shows that the update-issue is not solvable when the database manages objects by equality. Objects without identity also make Transparent Persistence and Activation impossible, as there will be no way to decide which instance is the right one for update or activation.

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 90](#)

You even can disable the weak reference. [See "Disable Weak References" on page 90](#). 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 123](#)

Manually Remove A Object From The Reference Cache

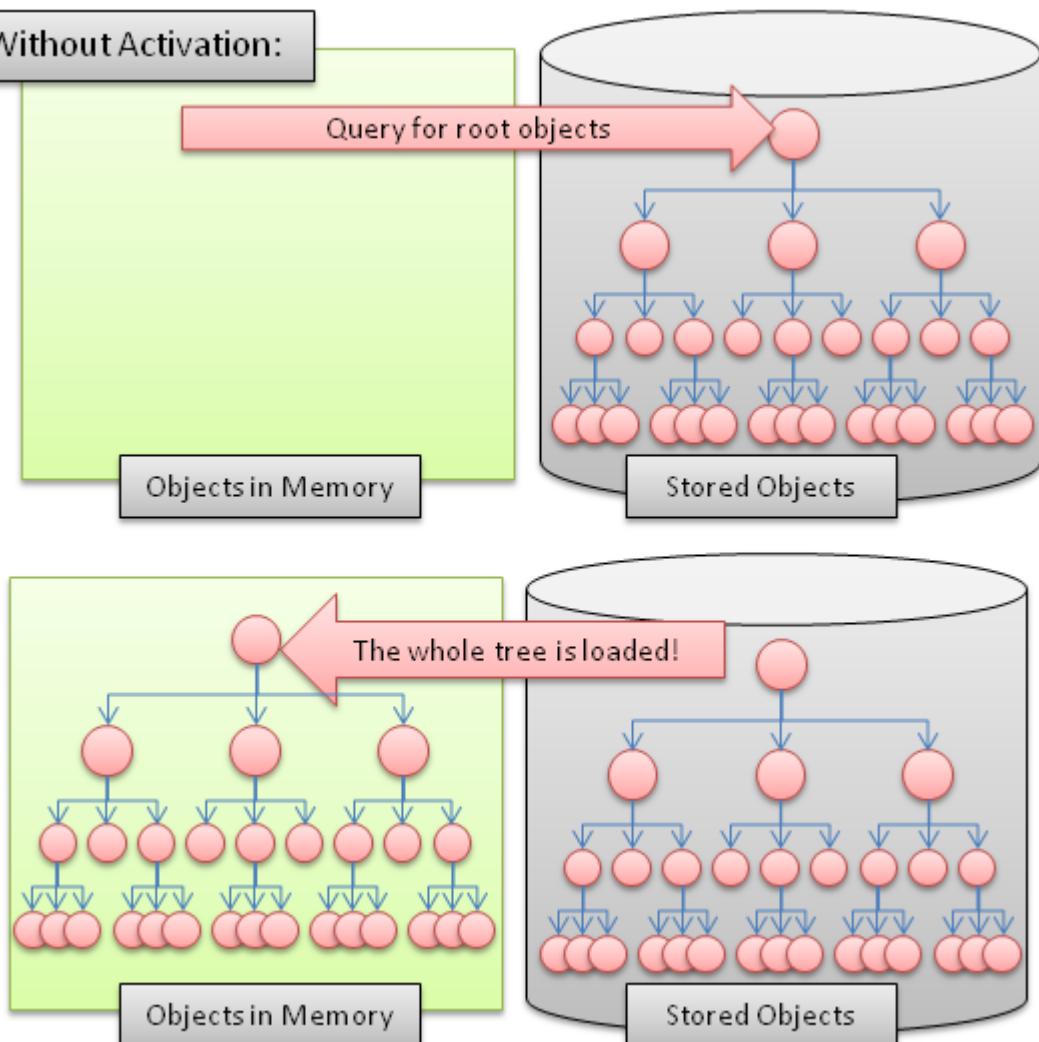
You can manually remove a object from the reference cache. This only required when you have disabled the [weak-references](#).

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

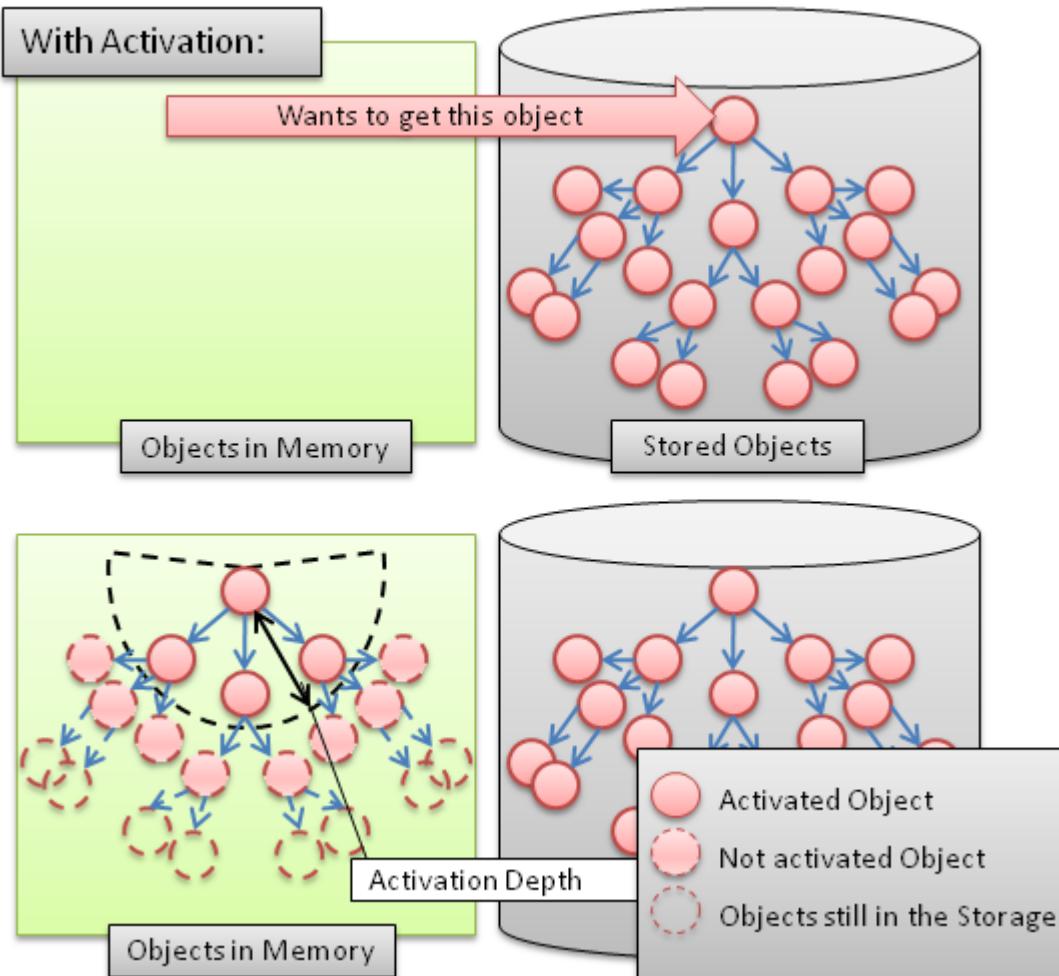
IdentityConcepts.java: With purge you can remove objects from the reference cache

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 retrieves objects, their fields 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 at lower levels, below activation depth are set to null or to default values. So db4o doesn't load the whole object graph from the database. Instead, db4o loads only parts of the object graph you're 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 34

If you want to automate the activation process: See "Transparent Activation" on page 36

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 we store a deep hierarchy of persons. Let's say we store a hierarchy of seven people. Then we query for it and traverse this object graph. When we 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 we traverse deep object graphs, we know that we 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 36](#)

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 TA enabled, objects are fetched on demand and only those that are used are being loaded.

First take a look at a simple example: [See "Transparent Activation Example" on page 36](#)

For more information how transparent activation works: [See "TA Implementation" on page 38](#)

You probably want to automate the process of implementing the required interfaces manually: [See "TA Enhanced Example" on page 39](#)

It's also possible to mix transparent activation aware object and other objects. [See "Object Types In TA" on page 49](#)

Collections can be also integrated into the transparent activation framework. [See "TA Aware Collections" on page 48](#)

Public fields can also be integrated with transparent activation to a certain limit: [See "TA For Public Fields" on page 51](#)

For diagnosis, take a look here: [See "TA Diagnostics" on page 51](#)

Also take a look at the pitfalls: [See "Transparent Activation Pitfalls" on page 51](#)

Transparent Activation Example

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 framework](#). Therefore it's marked as transient, to avoid that it's stored in the database.

¹Transparent Activation

```

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

Then implement the two methods of the Activatable-interface. 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 the important part. Every time a field of the class is accessed you need to call the activate-method with the purpose. For example in every getter/setter or other 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 within 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 "[TA Enhanced Example](#)" on page 39

The last step is to enable transparent persistence via configuration.

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

TransparentActivationExamples.java: Activate transparent activation

Now transparent persistence is enabled. Now you can navigate the object-graph as deep as you want. The transparent activation will load the objects from the database as you need them.

```
{
    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

TA Implementation

The basic idea for Transparent Activation:

- Classes can be modified to activate objects on demand by implementing the Activatable interface.
- To add the Activatable code to classes you can choose from one of the following three options:
 - Let db4o tools add the code to your persistent classes at compile time.
 - Use a special ClassLoader to add the code to persistent classes at load time.
 - Write the Activatable code by hand. [See "Transparent Activation Example" on page 36](#)
- To instruct db4o to operate in Transparent Activation mode, call: configuration.common().add(new TransparentActivationSupport());
- In Transparent Activation mode when objects are returned from a query:
 - objects that implement the Activatable interface will not be activated immediately
 - objects that do not implement the Activatable interface will be fully activated. Activatable objects along the graph of members break activation.
- Whenever a field is accessed on an Activatable object, the first thing that is done before returning the field value is checking it's activation state and activating the parent object if it is not activated. Similar as in querying, members that implement Activatable will not be activated themselves. Members that do not implement Activatable will be fully activated until Activatable objects are found.

With Transparent Activation the user does not have to worry about manual activation at all. Activatable objects will be activated on demand. Objects that do not implement Activatable will always be fully activated.

The basic sequence of actions to get this scheme to work is the following:

1. Whenever an object is instantiated from db4o, the database registers itself with this object. To enable this on the database level `TransparentActivationSupport` has to be registered with the db4o configuration. On the object level an object is made available for **TA**¹ by implementing the `Activatable/IActivatable` interface and providing the according `bind(activator)` method. The default implementation of the bind method stores the given `activator` reference for later use. Note, that only one activator can be associated with an object: trying to bind a different activator (from another object container) will result in an exception. More on this in [Migrating Between Databases](#).
2. All methods that are supposed to require activated object fields should call `activate(ActivationPurpose)/Activate(ActivationPurpose)` at the beginning of the message body. This method will check whether the object is already activated and if this is not the case, it will act depending on which activation reason was supplied.
3. The `ActivationPurpose` can be READ or WRITE. READ is used when an object field is requested for viewing by an application. In this case `Activate` method will request the container to activate the object to level 1 and set the activated flag accordingly (more on this case in the following chapters). WRITE activation purpose is used when an object is about to be changed; a simple example is setter methods. In this case the object is activated to depth 1 and registered for update. More on `ActivationPurpose.Write` in [Transparent Persistence](#).

This implementation requires quite many modifications to the objects. That is why db4o provides an automated TA implementation through bytecode instrumentation. With this approach all the work for TA is done behind the scenes.

Automatic and manual TA approaches are discussed in detail in the following examples.

- [TA Enhanced Example](#)
- [Detailed Example](#)

TA Enhanced Example

You can inject **TA**² awareness in your persistent classes without modifying their original code. In the current scenario this means:

- generate the `Activatable` interface declaration;
- add `bind(objectContainer)` method implementation;
- generate a field to keep a reference to the corresponding `Activator` instance;
- generate `activate()` call at the beginning of every method.

These tasks can be fulfilled in the classes bytecode by using [Enhancement Tools](#).

TA Enhancement In Java

TA can be enabled by bytecode injection of the above-mentioned code into the persistent classes when they are loaded or built. (Currently persistent classes have to be "tagged" by providing an appropriate

¹Transparent Activation

²Transparent Activation

`ClassFilter instance.)` In addition to this db4o also explicitly needs to be configured to use the Transparent Activation instrumentation of the persistent classes (`TransparentActivationSupport`).

Transparent Activation functionality requires the following jars:

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

The following topics explain how TA enhancement can be applied to built classes:

- [TA Enhancement At Loading Time](#)
- [TA Enhancement At Build Time](#)

TA Enhancement At Loading Time

TA¹ Instrumentation at loading time is the most convenient as the classes do not have to be modified, only a separate runner class should be created to enable special instrumenting classloader to deal with the classes.

The following configuration should be used (note that reflector set-up is not necessary for the loading time instrumentation):

```
TAInstrumentationExample.java: configureTA
private static Configuration configureTA() {
    Configuration configuration = Db4o.newConfiguration();
    configuration.add(new TransparentActivationSupport());
    // configure db4o to use instrumenting classloader
    // This is required for build time optimization!
    configuration.reflectWith(new JdkReflector(
        TAIInstrumentationExample.class.getClassLoader()));

    return configuration;
}
```

The `main` method should provide the testing code:

```
TAInstrumentationExample.java: main
public static void main(String[] args) {
    testActivation();
}

TAInstrumentationExample.java: storeSensorPanel
private static void storeSensorPanel() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database(configureTA());
    if (container != null) {
        try {
            // create a linked list with length 10
            SensorPanel list = new SensorPanel().createList(10);
            container.store(list);
        } finally {
            closeDatabase();
        }
    }
}
```

¹Transparent Activation

```

TAInstrumentationExample.java: testActivation
private static void testActivation() {
    storeSensorPanel();
    Configuration configuration = configureTA();
    activateDiagnostics(configuration);

    ObjectContainer container = database(configuration);
    if (container != null) {
        try {
            Query query = container.query();
            query.constrain(SensorPanel.class);
            query.descend("_sensor").constrain(new Integer(1));
            ObjectSet result = query.execute();
            listResult(result);
            if (result.size() > 0) {
                SensorPanel sensor = (SensorPanel) result.queryByExample(0);
                SensorPanel next = sensor._next;
                while (next != null) {
                    System.out.println(next);
                    next = next._next;
                }
            }
        } finally {
            closeDatabase();
        }
    }
}

```

A separate class should be used to run the instrumented example. This class creates a filter to point to the classes that should be instrumented, in this case ByNameClassFilter is used. You can see other filters in ClassFilter hierarchy. A special BloatClassEdit is created to instruct, which type of instrumentation will be used (InjectTransparentActivationEdit in our case). This configuration together with the path of the classes to be instrumented is passed to Db4oInstrumentationLauncher.

```

TAInstrumentationRunner.java
/** Copyright (C) 2007 Versant Inc. http://www.db4o.com */

package com.db4odoc.taexamples.instrumented;

import java.io.File;
import java.net.URL;

import com.db4o.instrumentation.classfilter.ByNameClassFilter;
import com.db4o.instrumentation.core.BloatClassEdit;
import com.db4o.instrumentation.core.ClassFilter;
import com.db4o.instrumentation.main.Db4oInstrumentationLauncher;
import com.db4o.ta.instrumentation.InjectTransparentActivationEdit;

public class TAInstrumentationRunner {

    public static void main(String[] args) throws Exception {
        // list the classes that need to be instrumented
        ClassFilter filter = new ByNameClassFilter(new String[]
        { SensorPanel.class.getName() });
        // inject TA awareness
        BloatClassEdit edits[] = new BloatClassEdit[]
        {new InjectTransparentActivationEdit(filter)};
        // get URL for the classloader
        URL[] urls = {

```

```

        new File("e:\\sb4o\\trunk\\reference\\bin").toURI().toURL() };
        Db4oInstrumentationLauncher
.launch(edits, urls, TAInstrumentationExample.class.getName(), new String[] {});

    }
    // end main

}

```

you can run the example by running Db4oInstrumentationLauncher, which will start TAInstrumentationExample in a correct configuration.

TA Enhancement At Build Time

In this topic we will look at even more convenient and performant way of enhancing classes to support **TA**¹: during application build time.

For simplistic example our build script should:

- Use classes, created by normal build script
- Create a new enhanced-bin folder for the enhanced classes
- Use TAAntClassEditFactory to create InjectTransparentActivationEdit (can be based on class filter)
- Call Db4oFileEnhancerAntTask#execute, which will call Db4oClassInstrumenter#enhance passing the previously created InjectTransparentActivationEdit to instrument classes with TA.

All these can be done with the following script:

```

Build.Xml
<?xml version="1.0"?>

<!--
  TA build time enhancement sample.
-->

<project name="taenhance" default="buildall">

<!--
  Set up the required class path for the enhancement task.
  In a production environment, this will be composed of jars, of course.
-->
<path id="db4o.enhance.path">
  <pathelement path="${basedir}" />
  <fileset dir="lib">
    <include name="**/*.jar"/>
  </fileset>
</path>

<!-- Define enhancement task. -->
<taskdef
  name="db4o-enhance"
  classname="com.db4o.instrumentation.ant.Db4oFileEnhancerAntTask"
  classpathref="db4o.enhance.path"
  loaderref="db4o.enhance.loader" />

<typedef
  name="transparent-activation"
  classname="com.db4o.ta.instrumentation.ant.TAAntClassEditFactory"

```

¹Transparent Activation

```

    classpathref="db4o.enhance.path"
    loaderref="db4o.enhance.loader" />

<target name="builddall">

    <!-- Create enhanced output directory-->
    <mkdir dir="${basedir}/enhanced-bin" />
    <delete dir="${basedir}/enhanced-bin" quiet = "true">
        <include name="**/*"/>
    </delete>

    <db4o-enhance classTargetDir="${basedir}/enhanced-bin"
    jarTargetDir="${basedir}/enhanced-lib">

        <classpath refid="db4o.enhance.path" />
            <!-- Use compiled classes as an input -->
        <sources dir="${basedir}/bin" />

            <!-- Call transparent activation enhancement -->
        <transparent-activation />

    </db4o-enhance>

```

```
</target>
```

```
</project>
```

In order to test this script:

- Create a new project, consisting of TAInstrumentationExample and SensorPanel classes from the [previous example](#)
- Add lib folder to the project root and copy the following jars from db4o distribution:
 - ant.jar
 - bloat-1.0.jar
 - db4o-X.XX-instrumentation.jar
 - db4o-X.XX-java5.jar
 - db4o-X.XX-taj.jar
 - db4o-X.XX-tools.jar (Note, that the described functionality is only valid for db4o releases after 7.0)
- Build the project with your IDE or any other build tools (it is assumed that the built class files go to the project's bin directory)
- Copy build.xml into the root project folder and execute it

Successfully executed build script will produce an instrumented copy of the project classes in enhanced-bin folder. You can check the results by running the following batch file from bin and enhanced-bin folders:

```
set CLASSPATH=.;${PROJECT_ROOT}\lib\db4o-X.XX-java5.jar
java com.db4odoc.taexamples.enhancer.TAInstrumentationExample
```

(In enhanced version the warning about classes that do not support TA should disappear).

Of course, the presented example is very simple and limited in functionality. In fact you can do a lot more things using the build script:

- o Add **NQ**¹ optimization in the same enhancer task
- o Use ClassFilter to select classes for enhancement
- o Use regex to select classes for enhancement
- o Use several source folders
- o Use jar as the source for enhancement

An example of the above features can be found in our [Project Spaces](#).

Transparent Persistence For Java Collections

One of the most valuable usages of **TA**²/**TP**³ is in collections:

- Using TA you don't have to worry about activating the next level of objects
- Using TP you can add new members to the collection or update existing ones without worrying about storing the changes to the object container.

If you remember the TA basics, in order to make an object activatable its class must implement Activatable interface. This can be done easily with the new classes, however in collections case you will probably want to use standard Java collection, which can't be modified. Db4o solves this problem by deriving special activatable classes from the standard Java collections and overriding all the public methods adding activate() call before calling the base method. At present db4o supports all Java5 standard collections:

ActivatableArrayList ActivatableHashMap ActivatableHashSet ActivatableHashtable ActivatableLinkedList ActivatableStack ActivatableTreeSet

In order to make use of these classes you will only need to add TransparentPersistenceSupport to your configuration before opening the object container:

```
TACollectionsExample.java: openObjectContainer
private static ObjectContainer openObjectContainer() {
    EmbeddedConfiguration embeddedConfiguration = Db4oEmbedded
        .newConfiguration();
    embeddedConfiguration.common().add(new TransparentPersistenceSupport());
    return Db4oEmbedded.openFile(embeddedConfiguration, DB4O_FILE_NAME);
}
```

And in the end enhance the project adding TP support (example will be shown below).

In your code you will simply use Java collections as usual, i.e.:

```
Pilot.java
package do4odoc.tacollections;

public class Pilot {

    private String name;

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

    @Override
    public String toString() {
```

¹Native Query

²Transparent Activation

³Transparent Persistence

```

        return name;
    }

}

Team.java
package do4odoc.tacollections;

import java.util.*;

public class Team {

    private String name;

    private List<Pilot> pilots;

    public Team(String name) {
        this.name = name;
    }

    public void addPilot(Pilot pilot) {
        if(pilots == null) {
            pilots = new ArrayList<Pilot>();
        }
        pilots.add(pilot);
    }

    @Override
    public String toString() {
        StringBuilder str = new StringBuilder();
        str.append(String.format("Team %s", name));
        if(pilots != null) {
            str.append(" with pilots: \n");
            for (Pilot p : pilots) {
                str.append(String.format("    %s \n", p));
            }
        }
        return str.toString();
    }

    public String getName() {
        return name;
    }
}

```

The following example code shows how Team class can be used:

```

TACollectionsExample.java: main
public static void main(String[] args) {
    new File(TACollectionsExample.DB40_FILE_NAME).delete();
    ObjectContainer objectContainer = openObjectContainer();
    Team team = new Team("Ferrari");
    // Store the team first time to make it persistent
    objectContainer.store(team);
    objectContainer.close();

    objectContainer = openObjectContainer();
    Team team1 = queryTeamByName(objectContainer, "Ferrari");
    team1.addPilot(new Pilot("Felipe Massa"));
    team1.addPilot(new Pilot("Kimi Raikkonen"));
    // No need to re-store team - it will be stored automatically

```

```

objectContainer.close();

objectContainer = openObjectContainer();
// check the results
ObjectSet<Team> objectSet = objectContainer.query(Team.class);
while (objectSet.hasNext()) {
    System.out.println(objectSet.next());
}
objectContainer.close();
}

```

Conversion to Activatable db4o collections and TA enhancement will be done by an ant script using Db4oEnhancerAntTask. This task can be integrated in compile and run script as follows:

```

build.xml
<project name="db4o.TP.sample" basedir="." default="run">

<property name="dir.src" value="${basedir}/src" />
<property name="dir.bin" value="${basedir}/bin" />

<path id="project.classpath">
    <fileset dir="lib">
        <include name="**/*.jar"/>
    </fileset>
</path>

<taskdef name="db4o-enhance"
    classname="com.db4o.enhance.Db4oEnhancerAntTask"
    classpathref="project.classpath" />

<target name="compile">
    <mkdir dir="${dir.bin}" />
    <javac srcdir="${dir.src}" destdir="${dir.bin}" failonerror="true">
        <classpath refid="project.classpath" />
    </javac>
</target>

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

<target name="run" depends="enhance">
    <java classname="do4odoc.tacollections.TACollectionsExample"
failonerror="true">
        <classpath>
            <pathelement path="${dir.bin}" />
            <path refid="project.classpath" />
        </classpath>
    </java>
</target>

</project>

```

If you will run this script you will see that the new team members were saved automatically when the object container was closed. To see the effect of the enhancer, please try to run TACollectionExample from IDE (without enhancement), in this case standard Java collections will be used and TA/TP will have no effect.

SensorPanel

```
SensorPanel.Java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */

package com.db4odoc.tpbuilder;

public class SensorPanel {

    private Object _sensor;

    private SensorPanel _next;

    public SensorPanel() {
        // default constructor for instantiation
    }
    // end SensorPanel

    public SensorPanel(int value) {
        _sensor = new Integer(value);
    }
    // end SensorPanel

    public SensorPanel getNext() {
        return _next;
    }
    // end getNext

    public Object getSensor() {
        return _sensor;
    }
    // end getSensor

    public void setSensor(Object sensor) {
        _sensor = sensor;
    }
    // end setSensor

    public SensorPanel createList(int length) {
        return createList(length, 1);
    }
    // end createList

    public SensorPanel createList(int length, int first) {
        int val = first;
        SensorPanel root = newElement(first);
        SensorPanel list = root;
        while (--length > 0) {
            list._next = newElement(++val);
            list = list._next;
        }
        return root;
    }
    // end createList
```

```

protected SensorPanel newElement(int value) {
    return new SensorPanel(value);
}
// end newElement

public String toString() {
    return "Sensor #" + getSensor();
}
// end toString
}

```

TA Aware Collections

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

db4o brings special, transparent activation 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.

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 TA Collections Directly" on page 48](#)

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 49](#)

Using TA Collections Directly

You can use the transparent activation aware collections directly in your code. They 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 useful to create a collection factory in your code. Then you have only one play in your code which decides which classes are used. So 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 "["TA Enhanced Example"](#) on page 39

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

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

Object Types In TA

When working in **TA**¹ enabled environment you must remember that db4o treats Activatable (TA Aware) and non Activatable (other) types differently.

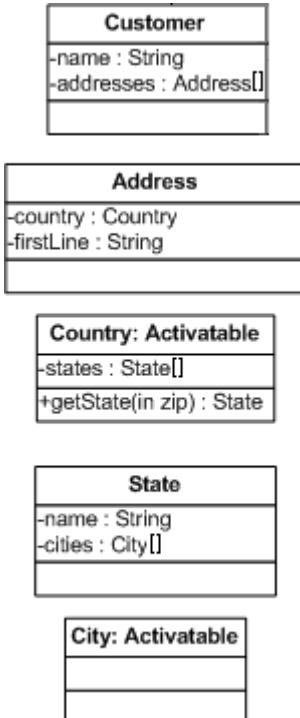
In general we can distinguish the following types:

- Value types with no identity (char, boolean, integer etc). These types are handled internally by db4o engine and behave the same in TA enabled and disabled modes.
- Activatable types, as it is clear from the name, implement Activatable interface and are responsible for their own activation.
- Non Activatable type - all the other types, including user types or third-party classes.

As it was mentioned [before](#) in TA enabled mode non-Activatable types are fully activated whereas Activatable types have 0 activation depth and are getting activated as requested.

Let's look at an example model below, which includes Activatable and non-Activatable classes:

¹Transparent Activation



Querying and traversing in TA enabled mode:

Java:

```
Customer c = container.queryByExample(Customer.class).next();
```

At this point the following paths should be already activated (Customer is not Activatable):

```
c.namec.addresses.addresses[N].firstLine
```

c.addresses[N].country - available but not activated (Activatable type).

Country.getState would cause the Country object to be activated

Java:

```
State state = c.getAddress(0).country().getState(someZipCode);
```

At this point the following paths become activated

c.addresses[0].country.states	.addresses[0]
.country.states[N].name	
c.addresses[0].country.states[N].city	.addresses[0]
.country.states[N].cities[N]	
- available but not activated (Activatable type)	

The following general rules apply:

1. Arrays of Arrays of non Activatable types: non Activatable behavior
2. Arrays of Arrays of Activatable types: non Activatable behavior except for leaves
3. JDK collections: non Activatable behavior

- Value types with references to non Activatable reference types and to Activatable reference types: the non Activatable path should be activated fully; Activatable path stops activation.

TA Diagnostics

You can use [Diagnostics](#) to get runtime information about classes with and without **TA**¹ support. Add a call to the following method in the `configureTA()` method and run the example from the previous topic:

```
TAEExample.java: activateDiagnostics
private static void activateDiagnostics(Configuration configuration) {
    // Add diagnostic listener that will show all the classes that are not
    // TA aware.
    configuration.diagnostic().addListener(new DiagnosticListener() {
        public void onDiagnostic(Diagnostic diagnostic) {
            if (!(diagnostic instanceof NotTransparentActivationEnabled)) {
                return;
            }
            System.out.println(diagnostic.toString());
        }
    });
}
```

The example should show you diagnostic messages about the classes without TA support. In this case it should be Image class (`Pilot._image`) and `BlobImpl`(used in `Image` class).

TA For Public Fields

Accessing public fields through **TA**² seems to be natural - if properties and getters are activated transparently, so should be public fields. However, if we look at [TA Implementation](#), we can see that we need to embed Activate call in the method accessing the field. It is easy in the case of getters and Properties as the method is the part of the persistent class. However, in the case of public fields, the access method can exist anywhere in the code. Effectively, TA enhancer has to browse through all the classes and find all references to public fields and instrument them as necessary. Well, this is still feasible.

However, there is a certain catch: if persistent classes are in a separate library/assembly from the main code, you have to make sure that BOTH persistent classes library/assembly and the accessing library/assembly are both instrumented. This is necessary to make sure that TA code is injected both in the persistent class and in the class accessing persistent class public field.

Transparent Activation Pitfalls

[Transparent Activation](#) is a powerful feature that can make development much faster, easier and error-proof. But as any power 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 can 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 to string method or the hash code 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.

¹Transparent Activation

²Transparent Activation

Migrating Between Databases

Problem

Transparent Activation is implemented through `Activatable/IActivatable` interface, which binds an object to the current object container. In a case when an object is stored to more than one object container, this logic won't work, as only one binding (activator) is allowed per object.

Solution

To allow correct behavior of the object between databases, the object should be unbinded before being stored to the next database. This can be done with the following code:

Java:

```
myObject.bind(null);
```

For more information see an [example](#).

Instrumentation Limitations

Problem

For Java instrumentation [instrumenting classloader](#) must know the classes to instrument, i.e. all application classes 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 you use 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](#).

Migrating Between Databases

Transparent activation and persistence functionality depends on an association between an object and an object container, which is created when an activator is bound to the object. Each object allows only one activator. Typically this limitation won't show up, however there is a valid use case for it:

- 1) suppose you need to copy one or more objects from one object container to another;
- 2) you will retrieve the object(s) from the first object container using any suitable query syntax;
- 3) optionally you can close the first object container;
- 4) you will now save the object to the second object container.

If both object containers were using transparent activation or persistence - the 4-th step will throw an exception. Let's look at the case in more detail. Typical activatable class contains an `activator` field. When transparent activation functionality is used for the first time an object container activator will be bound to the object:

```
SensorPanelTA.java: bind
/**/*Bind the class to the specified object container, create the activator*/
public void bind(Activator activator) {
    if (_activator == activator) {
        return;
    }
    if (activator != null && _activator != null) {
```

```

        throw new IllegalStateException();
    }
    _activator = activator;
}

```

If `bind` method will be re-called with the same object container, activator parameter will always be the same. However, if another object container tries to bind the object (in our case with the `store` call) activator parameter will be different, which will cause an exception. (Exception will be thrown even if the first object container is already closed, as activator object still exists in the memory.) This behaviour is illustrated with the following example ([SensorPanelTA](#) class from Transparent Activation chapter is used):

```

TAExample.java: testSwitchDatabases
private static void testSwitchDatabases() {
    storeSensorPanel();

    ObjectContainer firstDb = Db4o.openFile(configureTA(), FIRST_DB_NAME);
    ObjectContainer secondDb = Db4o.openFile(configureTA(), SECOND_DB_NAME);
    try {
        ObjectSet result = firstDb.queryByExample(new SensorPanelTA(1));
        if (result.size() > 0) {
            SensorPanelTA sensor = (SensorPanelTA) result.queryByExample(0);
            firstDb.close();
            // Migrating an object from the first database
            // into a second database
            secondDb.store(sensor);
        }
    } finally {
        firstDb.close();
        secondDb.close();
    }
}

```

The solution to this problem is simple: activator should be unbound from the object:

Java:

```
sensor.bind(null);
```

Note, that the object will quit being activatable for the first object container. The following example shows the described behaviour:

```

TAExample.java: testSwitchDatabasesFixed
private static void testSwitchDatabasesFixed() {
    storeSensorPanel();

    ObjectContainer firstDb = Db4o.openFile(configureTA(), FIRST_DB_NAME);
    ObjectContainer secondDb = Db4o.openFile(configureTA(), SECOND_DB_NAME);
    try {
        ObjectSet result = firstDb.queryByExample(new SensorPanelTA(1));
        if (result.size() > 0) {
            SensorPanelTA sensor = (SensorPanelTA) result.queryByExample(0);
            // Unbind the object from the first database
            sensor.bind(null);
            // Migrating the object into the second database
            secondDb.store(sensor);

            System.out.println("Retrieving previous query results from "
                + FIRST_DB_NAME + ":");

    }
}

```

```

SensorPanelTA next = sensor.getNext();
while (next != null) {
    System.out.println(next);
    next = next.getNext();
}

System.out.println("Retrieving previous query results from "
    + FIRST_DB_NAME + " with manual activation:");
firstDb.activate(sensor, Integer.MAX_VALUE);
next = sensor.getNext();
while (next != null) {
    System.out.println(next);
    next = next.getNext();
}

System.out.println("Retrieving sensorPanel from " + SECOND_DB_NAME + ":");
result = secondDb.queryByExample(new SensorPanelTA(1));
next = sensor.getNext();
while (next != null) {
    System.out.println(next);
    next = next.getNext();
}
}
} finally {
    firstDb.close();
    secondDb.close();
}
}
}

```

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?

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 55](#)

Since collection are regular object in db4o the update depth also applies to collections. [See "Updating Collections" on page 56](#)

If you want to automate the update process of object: [See "Transparent Persistence" on page 56](#)

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. Then 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 stored 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 is 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 56

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.

Transparent persistence takes care of activating the objects and storing changes. Take a look a simple example. [See "Transparent Persistence Example" on page 57](#)

In order to support transparent persistence the persisted object need to implement certain interfaces. To manually implementing this interface is time consuming. Therefore it can be automated with enhancement tools. [See "TP Enhancement" on page 60](#)

When using Transparent Persistence collections might need special attention. [See "Transparent Persistence Collections" on page 73](#)

For a deeper understanding of the mechanism: [See "Transparent Persistence Implementation" on page 59](#)

Transparent Persistence Example

In order to support **TP**¹, the persistent objects 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 framework](#). Therefore it's marked as transient, to avoid that it's 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

Then implement the two methods of the Activatable-interface. 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.

¹Transparent Persistence

```

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 the important part. Every time a field of the class is accessed you need to call the activate-method with the purpose. For example in every getter/setter or other 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 within 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 "[TP Enhancement](#)" on page 60

The last step is to enable transparent persistence via configuration.

```

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

```

TransparentActivationExamples.java: Activate transparent persistence

Now transparent persistence is enabled. You have to store the object only once initially. After that, changes are automatically stored on the commit call.

```

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

    // With transparent persistence 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());

    // Updating a object doesn't require no store call.
    // Just change the objects and the call commit.
    beginOfDynasty.setName("New Name");
    container.commit();
    container.close();
}
{
    ObjectContainer container = openDatabaseWithTP();
    Person person = queryByName(container, "New Name");
    // The changes are stored, due to transparent persistence
    System.out.println(person.getName());
    container.close();
}

```

TransparentActivationExamples.java: Transparent persistence in action

Transparent Persistence Implementation

The basic logic of Transparent Persistence (**TP¹**) is the following:

- Classes available for Transparent Persistence should implement the Activatable -interface, which allows to bind an object in the reference cache to the current object container.
- Persistent objects should be initially explicitly stored to the database:
Java: `objectContainer.store(myObject)`

myObject can be an object of any complexity including a linked list or a collection (currently you must use db4o-specific implementation for transparent collections: `ArrayList4`). For complex objects all field objects will be registered with the database with this call as well.
- Stored objects are bound to the Transparent Persistent framework when they are instantiated in the reference cache. This happens after the initial `store()` or when an object is retrieved from the database through one of the querying mechanisms.
- Whenever a `commit()` call is issued by the user, Transparent Persistent framework scans for modified persistent objects and implicitly calls `store()` on them before committing the transaction. Implicit commit with the mentioned above changes also occurs when the database is closed.

¹Transparent Persistence

Note that Transparent Persistence is based on Transparent Activation, so it is strongly recommended to study the [Transparent Activation](#) documentation first.

In order to make use of Transparent Persistence you will need:

1. Enable Transparent Activation (required for binding object instances to the TP framework) on the database level:

```
Java: configuration.common().add(new TransparentPersistenceSupport());
```

2. Implement Activatable interface for the persistent classes, either manually or through using [enhancement tools](#).

3. Call activate method at the beginning of all class methods that modify class fields:

```
Java: activate(ActivationPurpose.WRITE)
```

Note that `TransparentPersistenceSupport` configuration implicitly adds `TransparentActivationSupport`. The fact is, that before modification each field object should be loaded into the reference cache and that is the job of **TA**¹. So TA should be utilized in any case before TP. You can also note that the way TA and TP links into objects is absolutely identical: TP also uses the same `activate` call, but in this case its purpose is WRITE.

TP Enhancement

Implementing the Activatable-interface manually is time consuming, repetitive and error-prone. Therefore it's desirable to automate this process. There are different ways to automatically enhance classes to support transparent persistence.

- One possibility is to use a build-script which enhances the classes. For this a special Ant-task enhances the classes to support **TP**². See "Build Time Enhancement Example" on page 133
- It's also possible to enhance classes at runtime. See "Load Time Enhancement" on page 135

TP Enhancement On Java

TP³ Enhancement on Java platform can be done by customizing the ant build script to include instrumentation for persistent classes.

For a simple example we will use [SensorPanel](#) class, which represents a simple linked list. In our example application we will first store several objects of this class, then retrieve and modify them. Transparent Persistence mechanism should take care of modified objects and persist them to the database when the transaction is committed or the database is closed. As `SensorPanel` does not implement Activatable interface, we will need to use db4o enhancement tools to implement this interface after the class is built.

Let's look at our example code.

First, we need to configure Transparent Persistence:

```
TPExample.java: configureTP
private static Configuration configureTP() {
    Configuration configuration = Db4o.newConfiguration();
    // add TP support
    configuration.add(new TransparentPersistenceSupport());
    return configuration;
}
```

¹Transparent Activation

²Transparent Persistence

³Transparent Persistence

Now we will store a linked list of 10 elements:

```
TPExample.java: storeSensorPanel
private static void storeSensorPanel() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database(Db4o.newConfiguration());
    if (container != null) {
        try {
            // create a linked list with length 10
            SensorPanel list = new SensorPanel().createList(10);
            container.store(list);
        } finally {
            closeDatabase();
        }
    }
}
```

And the following procedure will test the effect of TP:

```
TPExample.java: testTransparentPersistence
private static void testTransparentPersistence() {
    storeSensorPanel();
    Configuration configuration = configureTP();

    ObjectContainer container = database(configuration);
    if (container != null) {
        try {
            ObjectSet result = container.queryByExample(new SensorPanel(1));
            listResult(result);
            SensorPanel sensor = null;
            if (result.size() > 0) {
                System.out.println("Before modification: ");
                sensor = (SensorPanel) result.queryByExample(0);
                // the object is a linked list, so each call to next()
                // will need to activate a new object
                SensorPanel next = sensor.getNext();
                while (next != null) {
                    System.out.println(next);
                    // modify the next sensor
                    next.setSensor(new Integer(10 + (Integer)next.getSensor()));
                    next = next.getNext();
                }
                // Explicit commit stores and commits the changes at any time
                container.commit();
            }
        } finally {
            // If there are unsaved changes to activatable objects, they
            // will be implicitly saved and committed when the database
            // is closed
            closeDatabase();
        }
    }
    // reopen the database and check the modifications
    container = database(configuration);
    if (container != null) {
        try {
            ObjectSet result = container.queryByExample(new SensorPanel(1));
            listResult(result);
            SensorPanel sensor = null;
            if (result.size() > 0) {
                System.out.println("After modification: ");
            }
        }
```

```
        sensor = (SensorPanel) result.queryByExample(0);
        SensorPanel next = sensor.getNext();
        while (next != null)  {
            System.out.println(next);
            next = next.getNext();
        }
    }
} finally  {
    closeDatabase();
}
}
```

Of course, if you will run the code above as is, you will see that all the changes were lost. In order to fix it we will need to build the application with a special build script:

```

Build.Xml
<?xml version="1.0"?>

<!--
 TP build time enhancement sample.
-->

<project name="tpexamples" default="buildall">

<!--
 Set up the required class path for the enhancement task.
-->
<path id="db4o.enhance.path">
  <pathelement path="${basedir}" />
  <fileset dir="lib">
    <include name="**/*.jar" />
  </fileset>
</path>

<!-- Define enhancement task. -->
<taskdef name="db4o-enhance"
classname="com.db4o.instrumentation.ant.Db4oFileEnhancerAntTask"
classpathref="db4o.enhance.path" loaderref="db4o.enhance.loader" />

<typedef name="transparent-persistence"
classname="com.db4o.ta.instrumentation.ant.TAAntClassEditFactory"
classpathref="db4o.enhance.path" loaderref="db4o.enhance.loader" />

<target name="buildall" depends="compile">

<!-- Create enhanced output directory-->
<mkdir dir="${basedir}/enhanced-bin" />
<delete dir="${basedir}/enhanced-bin" quiet="true">
  <include name="**/*" />
</delete>

<db4o-enhance classtargetdir="${basedir}/enhanced-bin">

  <classpath refid="db4o.enhance.path" />
  <!-- Use compiled classes as an input -->
  <sources dir="${basedir}/bin" />

  <!-- Call transparent persistence enhancement -->

```

```

        <transparent-persistence />

    </db4o-enhance>

</target>

<!-- Simple compilation. Note that db4o version
should be adjusted to correspond to the version
you are using
--&gt;
&lt;target name="compile"&gt;
    &lt;javac fork="true" destdir="${basedir}/bin"&gt;
        &lt;classpath&gt;
            &lt;pathelement location="${basedir}/lib/db4o-X.XX-java5.jar" /&gt;
        &lt;/classpath&gt;
        &lt;src path="${basedir}/src" /&gt;
        &lt;include name="**/*.java" /&gt;
    &lt;/javac&gt;
&lt;/target&gt;

&lt;/project&gt;
</pre>

```

The build script relies on several jars:

- ant.jar - [Ant](#) library
- bloat-1.0.jar - bloat bytecode instrumentation library
- db4o-X.XX-instrumentation.jar - db4o instrumentation library on top of bloat
- db4o-X.XX-java5.jar - db4o jar
- db4o-X.XX-taj.jar - db4o transparent activation support
- db4o-X.XX-tools.jar - db4o tools

All these jars should be added to /lib folder in the project directory.

After running the build script above you will get /bin and /enhanced-bin folders produced in your project folder. /bin folder contains compiled application classes, whereas /enhanced-bin contains compiled and enhanced classes. For testing the result of the enhancement you can use the following batch file (to be run from /enhanced-bin folder):

```

set CLASSPATH=.;${PROJECT_ROOT}\lib\db4o-X.XX-java5.jar
java com.db4odoc.tpbuilder.TPEexample

```

SensorPanel

```

SensorPanel.Java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */

package com.db4odoc.tpbuilder;

public class SensorPanel {

    private Object _sensor;

    private SensorPanel _next;

    public SensorPanel() {
        // default constructor for instantiation
    }
    // end SensorPanel

    public SensorPanel(int value) {

```

```

        _sensor = new Integer(value);
    }
    // end SensorPanel

    public SensorPanel getNext() {
        return _next;
    }
    // end getNext

    public Object getSensor() {
        return _sensor;
    }
    // end getSensor

    public void setSensor(Object sensor) {
        _sensor = sensor;
    }
    // end setSensor

    public SensorPanel createList(int length) {
        return createList(length, 1);
    }
    // end createList

    public SensorPanel createList(int length, int first) {
        int val = first;
        SensorPanel root = newElement(first);
        SensorPanel list = root;
        while (--length > 0) {
            list._next = newElement(++val);
            list = list._next;
        }
        return root;
    }
    // end createList

    protected SensorPanel newElement(int value) {
        return new SensorPanel(value);
    }
    // end newElement

    public String toString() {
        return "Sensor #" + getSensor();
    }
    // end toString
}

```

Transparent Persistence Pitfalls

Transparent Persistence is closely coupled with Transparent Activation, therefore the same [pitfalls](#) apply here. However there are some additional catches:

- [Object Clone](#)
- [Rollback Strategies](#)

Rollback Strategies

[Transparent Persistence](#) makes development faster and more convenient. However without a clear understanding of what is happening under the hood you can easily get into trouble. Transparent Persistence is triggered by commit call and with rollback the changes are simply discarded. But is it really all that trivial?

More Reading:

- [Rollback And Cache](#)
- [Automatic Deactivation](#)
- [Car](#)
- [Id](#)
- [Pilot](#)

Rollback And Cache

Suppose we have [Car](#), [Pilot](#) and [Id](#) classes stored in the database. Car class is activatable, others are not. We will modify the car and rollback the transaction:

```
TPRollback.java: modifyAndRollback
private static void modifyAndRollback() {
    ObjectContainer container = database(configureTP());
    if (container != null) {
        try {
            // create a car
            Car car = (Car) container.queryByExample(new Car(null, null))
                .queryByExample(0);
            System.out.println("Initial car: " + car + "("
                + container.ext().getID(car) + ")");
            car.setModel("Ferrari");
            car.setPilot(new Pilot("Michael Schumacher", 123));
            container.rollback();
            System.out.println("Car after rollback: " + car + "("
                + container.ext().getID(car) + ")");
        } finally {
            closeDatabase();
        }
    }
}
```

If the transaction was going on normally (commit), we would have had the car modified in the database as it is supported by Transparent Persistence. However, as the transaction was rolled back - no modifications should be done to the database. The result that is printed to the screen is taken from the reference cache, so it will show modified objects. That is confusing and should be fixed:

```
TPRollback.java: modifyRollbackAndCheck
private static void modifyRollbackAndCheck() {
    ObjectContainer container = database(configureTP());
    if (container != null) {
        try {
            // create a car
            Car car = (Car) container.queryByExample(new Car(null, null))
                .queryByExample(0);
            Pilot pilot = car.getPilot();
            System.out.println("Initial car: " + car + "("
                + container.ext().getID(car) + ")");
            System.out.println("Initial pilot: " + pilot + "("
                + container.ext().getID(pilot) + ")");
            car.setModel("Ferrari");
            car.changePilot("Michael Schumacher", 123);
            container.rollback();
            container.deactivate(car, Integer.MAX_VALUE);
            System.out.println("Car after rollback: " + car + "("
```

```
        + container.ext().getID(car) + ")");
    System.out.println("Pilot after rollback: " + pilot + "("
        + container.ext().getID(pilot) + ")");
} finally {
    closeDatabase();
}
}
```

Here we've added a deactivate call for the car object. This call is used to clear the reference cache and its action is reversed to activate.

We've used max int to deactivate car fields to the maximum possible depth. Thus we can be sure that all the car fields will be re-read from the database again (no outdated values from the reference cache), but the trade-off is that all child objects will be deactivated and read from the database too. You can see it on Pilot object. This behaviour is preserved for both activatable and non-activatable objects.

Automatic Deactivation

The use of depth parameter in deactivate call from the [previous example](#) directly affects performance: the less is the depth the less objects will need to be re-read from the database and the better the performance will be. Ideally we only want to deactivate the objects that were changed in the rolled-back transaction. This can be done by providing a special class for db4o configuration. This class should implement `RollbackStrategy`/`IRollbackStrategy` interface and is configured as part of Transparent Persistence support:

```
TPRollback.java: rollbackDeactivateStrategy
private static class RollbackDeactivateStrategy implements RollbackStrategy {
    public void rollback(ObjectContainer container, Object obj) {
        container.ext().deactivate(obj);
    }
}

TPRollback.java: configureTPForRollback
private static Configuration configureTPForRollback() {
    Configuration configuration = Db4o.newConfiguration();
    // add TP1 support and rollback strategy
    configuration.add(new TransparentPersistenceSupport(
        new RollbackDeactivateStrategy()));
    return configuration;
}
```

`RollbackDeactivateStrategy#rollback` method will be automatically called **once** per each **modified** object after the rollback. Thus you do not have to worry about deactivate depth anymore - all necessary deactivation will happen transparently preserving the best performance possible.

```
TPRollback.java: modifyWithRollbackStrategy
private static void modifyWithRollbackStrategy()  {
    ObjectContainer container = database(configureTPForRollback());
    if (container != null)  {
        try  {
            // create a car
            Car car = (Car) container.queryByExample(new Car(null, null))
                .queryByExample(0);
            Pilot pilot = car.getPilot();
```

1 Transparent Persistence

```
        System.out.println("Initial car: " + car + "("
            + container.ext().getID(car) + ")");
        System.out.println("Initial pilot: " + pilot + "("
            + container.ext().getID(pilot) + ")");
        car.setModel("Ferrari");
        car.changePilot("Michael Schumacher", 123);
        container.rollback();
        System.out.println("Car after rollback: " + car + "("
            + container.ext().getID(car) + ")");
        System.out.println("Pilot after rollback: " + pilot + "("
            + container.ext().getID(pilot) + ")");
    } finally {
        closeDatabase();
    }
}
```

Note, that RollbackDeactivateStrategy **only works for activatable** objects. To see the different you can comment out Activatable implementation in Id class (id value will be preserved in the cache).

Car

```
Car.java
/* Copyright (C) 2004 - 2008 Versant Inc. http://www.db4o.com */
package com.db4odoc.tp.rollback;

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

public class Car implements Activatable, Cloneable {
    private String model;
    private Pilot pilot;
    transient Activator _activator;

    public Car(String model, Pilot pilot) {
        this.model = model;
        this.pilot = pilot;
    }
    // end Car

    // Bind the class to an object container
    public void bind(Activator activator) {
        if (_activator == activator) {
            return;
        }
        if (activator != null && _activator != null) {
            throw new IllegalStateException();
        }
        _activator = activator;
    }
    // end bind

    // activate the object fields
    public void activate(ActivationPurpose purpose) {
        if (_activator == null)
            return;
        _activator.activate(purpose);
    }
}
```

```

// end activate

public String getModel() {
    activate(ActivationPurpose.READ);
    return model;
}
// end getModel

public void setModel(String model) {
    activate(ActivationPurpose.WRITE);
    this.model = model;
}
// end setModel

public Pilot getPilot() {
    activate(ActivationPurpose.READ);
    return pilot;
}
// end getPilot

public void setPilot(Pilot pilot) {
    activate(ActivationPurpose.WRITE);
    this.pilot = pilot;
}
// end setPilot

public String toString() {
    activate(ActivationPurpose.READ);
    return model + "[" + pilot + "]";
}
// end toString

public void changePilot(String name, int id) {
    pilot.setName(name);
    pilot.setId(id);
}

}

```

Id

```

Id.java
/**/* Copyright (C) 2008 Versant Inc. http://www.db4o.com */

package com.db4odoc.tp.rollback;

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

public class Id implements Activatable {
    int number = 0;

    transient Activator _activator;

    public Id(int number) {
        this.number = number;
    }
}
```

```

public void bind(Activator activator) {
    if (_activator == activator) {
        return;
    }
    if (activator != null && _activator != null) {
        throw new IllegalStateException();
    }
    _activator = activator;
}

public void activate(ActivationPurpose purpose) {
    if (_activator == null)
        return;
    _activator.activate(purpose);
}

public String toString() {
    activate(ActivationPurpose.READ);
    return String.valueOf(number);
}

public void change(int i) {
    activate(ActivationPurpose.WRITE);
    this.number = i;
}
}

```

Pilot

```

Pilot.java
/** Copyright (C) 2008 Versant Inc. http://www.db4o.com */

package com.db4odoc.tp.rollback;

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

public class Pilot implements Activatable {

    private String name;
    private Id id;

    transient Activator _activator;
    // Bind the class to an object container
    public void bind(Activator activator) {
        if (_activator == activator) {
            return;
        }
        if (activator != null && _activator != null) {
            throw new IllegalStateException();
        }
        _activator = activator;
    }

    // activate the object fields
}

```

```

public void activate(ActivationPurpose purpose) {
    if (_activator == null)
        return;
    _activator.activate(purpose);
}

public Pilot(String name, int id) {
    this.name = name;
    this.id = new Id(id);
}

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

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

public String toString() {
    activate(ActivationPurpose.READ);
    return getName() + "[" + id + "]";
}

public void setId(int i) {
    activate(ActivationPurpose.WRITE);
    this.id.change(i);
}
}

```

Car

```

Car.java
/* Copyright (C) 2004 - 2008 Versant Inc. http://www.db4o.com */
package com.db4odoc.tp.rollback;

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

public class Car implements Activatable, Cloneable {
    private String model;
    private Pilot pilot;
    transient Activator _activator;

    public Car(String model, Pilot pilot) {
        this.model = model;
        this.pilot = pilot;
    }
    // end Car

    // Bind the class to an object container
    public void bind(Activator activator) {
        if (_activator == activator) {
            return;
        }
    }
}

```

```

        }
        if (activator != null && _activator != null)  {
            throw new IllegalStateException();
        }
        _activator = activator;
    }
    // end bind

    // activate the object fields
    public void activate(ActivationPurpose purpose)  {
        if (_activator == null)
            return;
        _activator.activate(purpose);
    }
    // end activate

    public String getModel()  {
        activate(ActivationPurpose.READ);
        return model;
    }
    // end getModel

    public void setModel(String model)  {
        activate(ActivationPurpose.WRITE);
        this.model = model;
    }
    // end setModel

    public Pilot getPilot()  {
        activate(ActivationPurpose.READ);
        return pilot;
    }
    // end getPilot

    public void setPilot(Pilot pilot)  {
        activate(ActivationPurpose.WRITE);
        this.pilot = pilot;
    }
    // end setPilot

    public String toString()  {
        activate(ActivationPurpose.READ);
        return model + "[" + pilot + "]";
    }
    // end toString

    public void changePilot(String name, int id)  {
        pilot.setName(name);
        pilot.setId(id);
    }
}

```

Object Clone

Platform implementations of #clone is not compatible with **TP¹**.

¹Transparent Persistence

Both java and .NET object implementations provide `#clone` method for default objects, which is enabled by implementing `Cloneable/ICloneable` interface. This implementation is a shallow clone, i.e. only the top-level object fields are duplicated, all the referenced(children) objects are only copied as references to the same object in the parent clone. But how it affects Transparent Persistence?

If you remember [Transparent Persistence Implementation](#) you must know that a special `Activator` field is used to bind an object to the object container. Consequently, the default clone will copy this `Activatable` field to the object's duplicate, which will produce ambiguity as the object container won't know which object should be activated for write.

Let's look how it will affect db4o in practice. We will use a usual `Car` class and make it cloneable. Use the following code to store a car object and it's clone:

```
TPCloneExample.java: storeCar
private static void storeCar() throws CloneNotSupportedException {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database(Db4o.newConfiguration());
    if (container != null) {
        try {
            // create a car
            Car car = new Car("BMW", new Pilot("Rubens Barrichello"));
            container.store(car);
            // clone
            Car car1 = (Car)car.clone();
            container.store(car1);
        } finally {
            closeDatabase();
        }
    }
}
```

So it works for the first store, but what if we will clone an object retrieved from the database?

```
TPCloneExample.java: testClone
private static void testClone() throws CloneNotSupportedException {
    storeCar();
    Configuration configuration = configureTP();

    ObjectContainer container = database(configuration);
    if (container != null) {
        try {
            ObjectSet result = container.queryByExample(new Car(null, null));
            listResult(result);
            Car car = null;
            Car car1 = null;
            if (result.size() > 0)
            {
                car = (Car)result.queryByExample(0);
                System.out.println("Retrieved car: " + car);
                car1 = (Car)car.clone();
                System.out.println("Storing cloned car: " + car1);
                container.store(car1);
            }
        } finally {
            closeDatabase();
        }
    }
}
```

The code above throws an exception when the cloned object is being bound to the object container. Luckily we can easily fix it by overriding `#clone` method and setting activator to null:

```
Car.java: Clone
public Object clone() throws CloneNotSupportedException {
    Car test = (Car)super.clone();
    test._activator = null;
    return test;
}
```

Transparent Persistence Collections

In the [previous example](#) we reviewed how Transparent Persistence should be used with simple types. For your domain-classes this is quite easy to archive. But what about the Java-collections? Wouldn't it be nice when the collections also work together the transparent persistence framework? So that when you add a object to a persistent collection, the object is also stored when committing?

For this purpose db4o brings special, transparent persistence aware collections with it. These collections store the added objects when on commit.

These are the same collections as for transparent activation and you can use them exactly the same way. [See "TA Aware Collections" on page 48](#)

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.getDriver());
```

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 74](#)

Also collections are treated like regular objects and need to be deleted explicitly. See "Collections And Arrays" on page 74

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 than 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 110

```
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.

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 75

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 76. 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

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 78](#)

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	Can not
--	---------	---------

	C/S ¹ ²	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 <code>toString()</code> value of the object-container.		
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 constructors.		
UpdateDepth : Change the update-depth.		
WeakReferenceCollectionInterval : Change the weak-reference cleanup interval. Default setting is 1000 milliseconds.		
WeakReferences : Enable/disable weak references.		

¹Client-Server

²This setting needs to be the same on the server and all clients.

³This setting has to be set the first time when the database is created. You cannot change it for an existing database.

Additional Configuration Items

There are additional configuration items which add for additional features. You can add them on the common-configuration. For example to enable transparent persistence you add the TransparentPersistenceSupport configuration item. Take a look at the available configuration items. [See "Common Configuration Items" on page 80](#)

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 36](#)

TransparentPersistenceSupport

Support for transparent persistence. [See "Transparent Persistence" on page 56](#)

UniqueFieldValueConstraint

Set up unique Field constraints. [See "Unique Constraints" on page 125](#)

BigMathSupport

Add support for the BigDecimal and BigInteger classes. [See "BigMath" on page 143](#)

FreespaceMonitoringSupport

Enables you to monitor the free-space-manager. [See "Runtime Monitoring" on page 282](#)

IOMonitoringSupport

Enables you to monitor the IO-activity of db4o. [See "Runtime Monitoring" on page 282](#)

NativeQueryMonitoringSupport

Enables you to monitor the native queries. [See "Runtime Monitoring" on page 282](#)

NetworkingMonitoringSupport

Enables you to monitor the network activity of db4o. [See "Runtime Monitoring" on page 282](#)

ObjectLifecycleMonitoringSupport

Enables you to monitor the object lifecycles. [See "Runtime Monitoring" on page 282](#)

QueryMonitoringSupport

Enables you to monitor db4o queries. [See "Runtime Monitoring" on page 282](#)

ReferenceSystemMonitoringSupport

Enables you to monitor db4o's reference system. [See "Runtime Monitoring" on page 282](#)

Activation Depth

db4o uses the concept of [activation](#) to avoid loading too much data into memory. You can change the global activation depth with this setting.

```
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 107](#)

```
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 54](#)

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 107](#)

```
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 is 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 then 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 expen-

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 databases 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, your 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 289](#)

```
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 save 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 annotation

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 86

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 14

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 154

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 considerable 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 164

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

¹Client-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 ₂	Only useful for recovery
<u>AsynchronousSync</u> : Enables asynchronous commits.		
<u>BlobPath</u> : Specify where blobs are stored.		
<u>BlockSize</u> : Set the block-size of the database. Larger Blocks	Yes	

¹This setting has to be set the first time when the database is created. You cannot change is for an existing database.

²This setting is only useful when you try to recover a corrupted database file.

allow larger databases.		
DatabaseGrowthSize : Set the grow step size when the database-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 <u>commit timestamps</u> 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 guarantees that everything is persistent. Maybe the commit is still in progress. On a failure, this means that you can loose 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 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 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 [defragment 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 loose 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 scenerios. 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 2 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 125

```
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 newCache 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

Cache Configuration Example

The following example helps to see the effect of cache by modifying the cache size. Note, that this example is not a good illustration of the LRU cache algorithm, and only shows the effect of simple caching of the most recent data.

For the test we will use the following 2 classes: Item and ItemStore, holding a collection of items:

```
CachingStorageExample.java: Item
private static class Item {
    int id;
    String name;

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

    public String toString() {
        return String.format("%s %d", name, id);
    }
}
```

```
CachingStorageExample.java: ItemStore
```

```
private static class ItemStore {
    ArrayList<Item> items;

    public ItemStore() {
        this.items = new ArrayList<Item>();
    }

    public void addItem(Item item) {
        this.items.add(item);
    }
}
```

The following methods will be used to fill in and query the database:

```
CachingStorageExample.java: createDatabase
private static void createDatabase(EmbeddedConfiguration config)  {
    new File(DB4O_FILE_NAME).delete();

    long startTime = System.currentTimeMillis();
    ObjectContainer container = Db4oEmbedded.openFile(config, DB4O_FILE_NAME);
    try  {
        ItemStore itemStore = new ItemStore();
        for (int i = 0; i < OBJECT_COUNT; i++)  {
            itemStore.addItem(new Item(i, "title" + i));
        }
        container.store(itemStore);
        Item item  = (Item) container.queryByExample(
            new Item(1, "title1")).queryByExample(0);
        System.out.println(item);
    } finally  {
        container.close();
    }
    System.out.println(String.format("Time to create a database: %d ms",
        System.currentTimeMillis() - startTime));
}
```

```
CachingStorageExample.java: queryDatabase
```

```
private static void queryDatabase(EmbeddedConfiguration config)  {

    ObjectContainer container = Db4oEmbedded
        .openFile(config, DB4O_FILE_NAME);
    try  {
        ArrayList<Item> temp = new ArrayList<Item>();
        long startTime = System.currentTimeMillis();
```

```

Query q = container.query();
q.constrain(Item.class);
q.descend("id").constrain(1).greater();
ObjectSet <Item> result = q.execute();
for (Item i: result) {
    temp.add(i);
}
System.out.println(String.format("Time to get an objects first time: %d ms",
    System.currentTimeMillis() - startTime));
//
temp = new ArrayList<Item>();
startTime = System.currentTimeMillis();
for (Item i: result) {
    temp.add(i);
}
System.out.println(String.format("Time to get an objects second time: %d ms",
    System.currentTimeMillis() - startTime));
} finally {
    container.close();
}
}

```

In general you should see the benefits of using cache with the majority of db4o operations. However, the effect is most obvious when a large amount of database information should be accessed fast (query) and this information is not loaded in the applications hash memory. We are trying to reproduce this situation in the second method querying database. If the amount of items in the ItemStore collection is 10000, the database size will be around 900 KB. With the default cache size, i.e. page count multiplied by page size (64 * 1024), this will mean that the whole collection won't fit into the cache (It can still fit into your hash memory, so you may want to decrease the available hash memory or increase the size of the collection to see the effect.).

We will use the following configuration methods to compare the default cache allocation and the custom one:

```

CachingStorageExample.java: getConfig
private static EmbeddedConfiguration getConfig() {
    EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
    config.file().storage(new CachingStorage(new FileStorage(), PAGE_COUNT, PAGE_SIZE));
    return config;
}

CachingStorageExample.java: getDefaultConfig
private static EmbeddedConfiguration getDefaultConfig() {
    EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
    return config;
}

```

Using custom page count = 1024 will make the cache size enough to fit the whole collection, and you should see some performance improvement browsing the retrieved collection second time (Unless it all fits into your hash memory in any case). Please, also try decreasing page size and count to see the opposite effect.

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 119](#)

Prefetch Depth

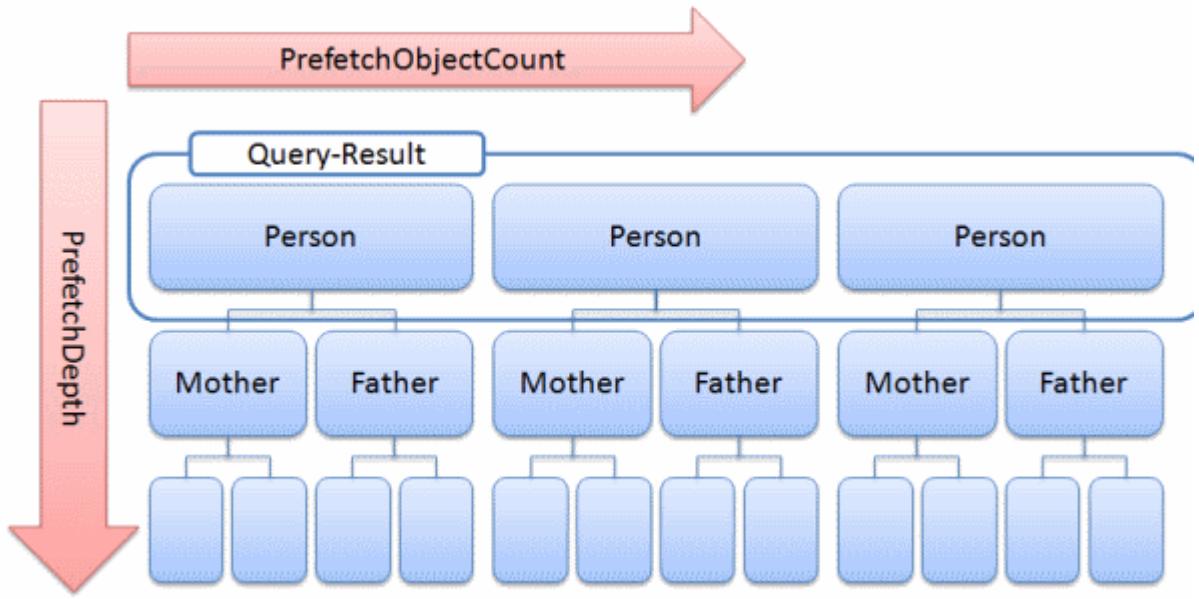
In a client server scenario, network latency is one of the biggest performance concerns. Instead of making lots of little requests, it's more to do bulk operations. A good way is to prefetch objects which are maybe required later.

The prefetch 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

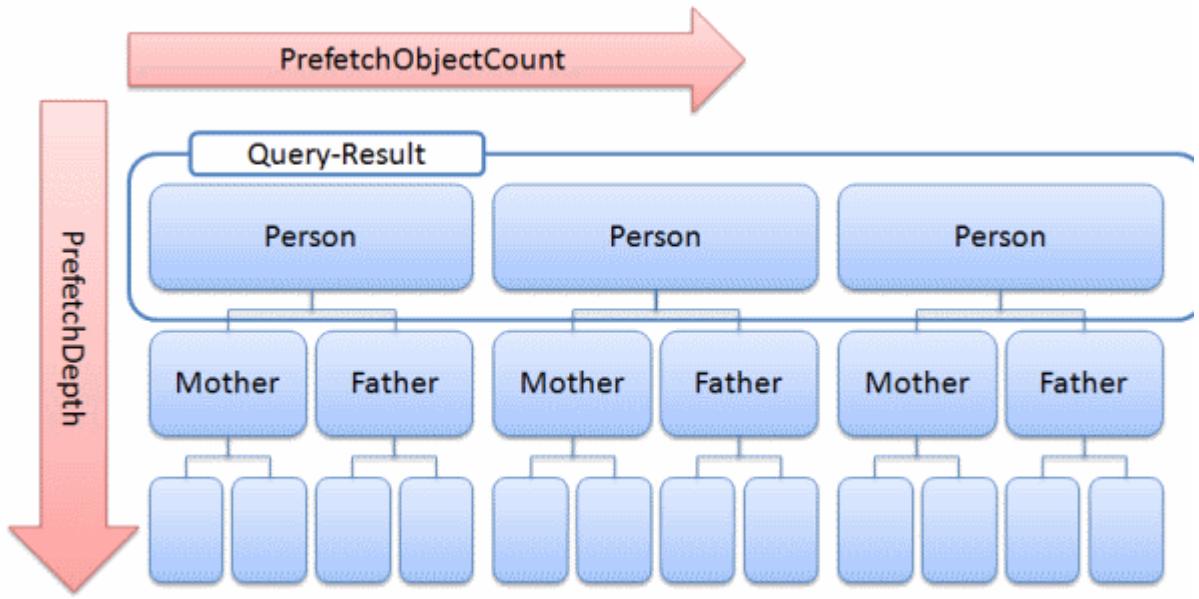
In a client server scenario, network latency is one of the biggest performance concerns. Instead of making lots of little requests, it's more to do bulk operations. A good way is to prefetch objects which are maybe required later.

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 pre-fetched 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 107

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 119

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 120](#)

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 even 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.

MinimumActivationDepth: 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 83](#)

```
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.

```
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 too 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.

```
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.

```
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.

```
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 150

```
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 156](#)

```
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 152](#)

```
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 54](#). 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 75](#)

```
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 too much data into memory. When this setting is turned on, the object referenced by this field is activated, when the object is activated.

```
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.

```
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 156

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

ObjectFieldConfigurations.java: Rename this field

Id System

The id-system configuration applies to the embedded- and the server-mode of db4o. There are three 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 is 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.jarDb4objects.Db4o.CS.dll-assembly in your project. [See "Dependency Overview" on page 318](#)

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 114](#)
- 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	<ul style="list-style-type: none"> 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. 	<ul style="list-style-type: none"> The objects are always up to date. There's no need to explicitly refresh-calls in your data-access layer.
Disadvantage	<ul style="list-style-type: none"> Partial refreshed objects may lead to an inconsistent object-graph. Probably a lot of refresh-calls spread over the code-base. 	<ul style="list-style-type: none"> 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. You 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 142](#)

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 123](#)

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 118](#)

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 class will 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 referred 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?

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.

¹Native Query

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 137](#)

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 your 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_PASSWORD);

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.

Unfortunately all the requirements are difficult to achieve for a cross-platform application, as Java and .NET sockets behave differently.

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.

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 123](#)

db4o support unique constrains on fields. [See "Unique Constraints" on page 125](#)

You can also do backups of your database at runtime. [See "Backup" on page 125](#)

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 132](#)

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 128](#)

Callback allow you to perform additional logic for different database operations. [See "Callbacks" on page 136](#)

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 156](#)

You can access the internal ids or generate UUIDs for objects. [See "IDs and UUIDs" on page 126](#)

It's also possible to access the meta information of all stored types in the database. [See "db4o Meta-Information" on page 127](#). 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 143](#) 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 164](#)

Unfortunately failure happen. db4o communicates failures with exceptions. [See "Exception-Handling" on page 161](#)

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 consider 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 with each other.

They don't share the objects loaded and stored with each other. That means you need to load and store the object with the same session container. When you try to load a object from one session-container and store it with another, you will 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();

// now the changes are visible 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 won't 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.

```
configuration.common().objectClass(UniqueId.class).objectField("id").indexed(true);
configuration.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

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.

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

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 page 30

If you really need to have ids for your object, take a look at this comparison. See "Comparison Of Different IDs" on page 172

Internal IDs

Each object, except value objects like ints, floats or strings, do have an internal id. This id is unique within the 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 internalId = 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 UUID that 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);
```

MetalInfoExample.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 type 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();
```

MetalInfoExample.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);
}
```

MetalInfoExample.java: Access via meta data

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.

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 129

Take a look at the defragmentation configuration options. [See "Defragmentation Configuration" on page 129](#)

You can register a defragmentation listener which notifies you for certain issues: [See "Tracking Defragmentation Errors" on page 132](#)

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 creates a defragmented version of this database at the original position. If the backup file already exists this an IOException is thrown 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 defragmentation process. [See "Defragmentation Configuration" on page 129](#)

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 130](#)

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 130](#)

When you defragment large database you should configure a commit-frequency to speed up the defragmentation process. [See "Commit Frequency" on page 130](#)

If you have refactored your classes you might want to remove old meta data. This is possible with the class filters. [See "Class Filter" on page 131](#)

By default the backup file isn't deleted after a successful defragmentation. You can change that. [See "Delete The Backup" on page 131](#)

You can force a database update before defragmenting the database-file. [See "Upgrade Database File" on page 131](#)

You can disable the read-only mode for backup file. [See "Disable Read Only Mode" on page 131](#)

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 132](#)

You can change the id-mapping implementation for the defragmentation-process. [See "Configure IDMapping" on page 132](#)

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 your 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 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.

- It's possible to integrate the enhancement step into the build process by using a Ant-task. See "[Build Time Enhancement Example](#)" on page 133
- It's possible to enhance the classes a runtime. See "[Load Time Enhancement](#)" on page 135
- More details about the enhancement process: See "[Enhancement For Java](#)" on page 134

Build Time Enhancement Example

It possible to integrate the db4o enhancements into the build process, using a special Ant-Task.

Preparation

The enhance-functionality is implemented in the db4o-X.XX-instrumentation and db4o-X.XX-tools-jar and its dependencies. This files are in the db4o-distribution. Take a look a the detailed documentation for detailed infos. See "[Enhancement For Java](#)" on page 134 Or you can take the db4o-X.XX-all-jar which includes all db4o-functionality and all dependencies. For this example the db4o-X.XX-all-jar is in the 'lib'-folder of the project.

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](#).

```
<path id="project.classpath">
    <pathelement path="${basedir}/bin"/>
    <fileset dir="lib">
        <include name="**/*.jar"/>
    </fileset>
</path>
<taskdef name="db4o-enhance"
    classname="com.db4o.enhance.Db4oEnhancerAntTask"
    classpathref="project.classpath"/>
```

simple-enhance-example.xml: Define a task for the enhancement

And the execute the task after the compilation.

¹Native Query

```

<target name="enhance">
    <db4o-enhance classtargetdir="${basedir}/bin"
        jartargetdir="${basedir}/lib"
        nq="true" ta="true"
        collections="true">
        <classpath refid="project.classpath"/>
        <sources dir="${basedir}/bin">
            <include name="**/*.class"/>
        </sources>
        <jars dir="${basedir}/lib">
            <include name="*.jar"/>
            <exclude name="db4o-*.jar"/>
            <exclude name="ant.jar"/>
            <exclude name="bloat-1.0.jar"/>
        </jars>
    </db4o-enhance>
</target>

```

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

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. [See "Enhancement For Java" on page 134](#)

Enhancement For Java

The db4o enhancement framework relies on the following jars:

bloat-1.0.jar	Third-party bytecode instrumentation library
db4o-X.XX-instrumentation	Instrumentation library on top of bloat
db4o-X.XX-tools	Enhancement and other utilities

In addition:

- For **TA**¹/**TP**² instrumentation enhancement db4o-X.XX-taj.jar should be used (contains TA / TP instrumentation classes).
- For **NQ**³ optimization db4o-X.XX-nqopt.jar is used (provides instrumentation functionality for NQ).

The basic steps required to enhance classes are:

1. Create a ClassFilter instance to select the classes for enhancement. ClassFilter is an interface in db4oinstrumentation project and is implemented by several classes, like AcceptAllClassesFilter, ByNameClassFilter and others.

¹Transparent Activation

²Transparent Persistence

³Native Query

2. Create BloatClassEdit array of classes capable of editing class bytecode. BloatClassEdit is an interface in db4o instrumentation project. Among its implementations are TranslateNQToSODAEedit (implements NQ optimization) and InjectTransparentActivationEdit (injects TA/TP awareness). Filter can be used in some of the edit classes (InjectTransparentActivationEdit).
3. For load-time instrumentation the edit classes created above are passed to Db4oInstrumentationLauncher together with the application entry point class. Db4oInstrumentationLauncher is a public class in db4o instrumentation project, which creates a special instrumenting classloader and uses it to load the application's main class.
4. For build time instrumentation Db4oFileEnhancerAntTask is used to create an enhancer task in Ant, which must call the class edit classes inside. Db4oFileEnhancerAntTask is a class extending Ant task in db4o instrumentation project. It loads and instruments the classes using class edits supplied as parameters to the enhancer task and copies the resulted classes to the output directory. It can also work on Jars instead of classes.

The examples below shows how enhancer works at load and build time:

- [Simplest Enhancement Script](#)
- [TA Enhancement at Loading Time](#)
- [NQ Enhancement at Build Time](#)

Load Time Enhancement

It's possible to enhance your classes at runtime. This mechanism uses the Java class loading mechanism. Instead of using the regular class-loader, the application needs to use a db4o-classloader which enhances the classes when loaded.

To enable load time enhancement, you need to create a special launcher for your application. This launcher sets up the class-loader and then starts the application.

The db4o instrumenting classloader has following configuration options available:

- A *ClassFilter* specifies which classes should be instrumented. In the example, we are using a filter that will only accept classes whose fully qualified name starts with a given prefix. The instrumentation API already comes with a variety of other filter implementations, and it's easy to create custom filters.
- A sequence of *ClassEdits*. A ClassEdit is a single instrumentation step. In the example, we are applying two steps: First, we preoptimize all Native Query Predicates, then we instrument for Transparent Activation. Note that the order of steps is significant: Switching the order would leave the generated **NQ**¹ optimization code unaware of **TA**². The db4otools package provides a convenience launcher with a hardwired sequence for combined NQ/TA instrumentation.
- The *classpath* for the instrumented classes, represented by a sequence of URLs. This must contain all classes "reachable" from the classes to be instrumented - the easiest way probably is to provide the full application class path here. The classes to be instrumented need not be listed here, they are implicitly added to this classpath, anyway.

A start of your application might look like this:

```
EnhancerStarter.java: main
public static void main(String[] args) throws Exception {
```

¹Native Query

²Transparent Activation

```

// use a class-filter to only enhance certain classes
ClassFilter filter = new ByNameClassFilter("yourpackages.datamodel.", true);
// specify which enhancements you need. Typically the native-query
// optimisation and transparent activation support
BloatClassEdit[] edits = { new TranslateNQToSODAEEdit() , new InjectTransparentActivationEdit(filter) };
// specify the where the jars & classes of your project are
URL[] urls = { new File("/work/worksaces/db4o/tatest/bin").toURI().toURL() };
// the launch the application
Db4oInstrumentationLauncher.launch(edits, urls, EnhancerMain.class.getName(), args);
}

```

Try this code now - if everything is correct you will see that the objects are getting activated as they are requested. NQ info also should say that the queries are preoptimized.

Note that for load time instrumentation to work, the application code has to make sure db4o operates on the appropriate classloader for the persistent model classes.

This means that you need to set explicitly the class-loader for the reflection. Set the thread-context class loader in the db4o configuration.

EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration(); configuration.common().reflectWith(new JdkReflector(Thread.currentThread().getContextClassLoader()));
ClassloadingStrategies.java: Use the context classloader

Don't forget that you still need to [enable transparent activation / persistence](#) in order to use it.

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 136](#)

Additionally you can register event listener to a object-container which will be called on certain operations.
[See "Event Registry API" on page 137](#)

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 139](#)

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<CancelableObjectEventArgs>() {
    public void onEvent(Event4<CancelableObjectEventArgs> events,
                       CancelableObjectEventArgs 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 its own event-registry, and therefore only sees its own events. Except the committed-event. See "Events In Client Server-Mode" on page 117

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.
- 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**². 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.

¹This event is asynchronously distributed across all clients

²Relational Database Management System

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.

```
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 committing 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<CancellableEventArgs>() {
    @Override
    public void onEvent(Event4<CancellableEventArgs> events,
                       CancellableEventArgs 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.
- Database replication 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.moveToNext();){
            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 and easy to use. One 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 object. db4o has different storing strategies for different types.

For most types the regular db4o type handling is sufficient.

Since collections are so important, db4o treats collections specially to improve efficiency. [See "Collections" on page 146](#)

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 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

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

Internally the regular Java-collections are stored as arrays. When you store a collection, db4o will store all the collection-items into a array. And when you retrieve a collection, db4o will create a new instance of the collection and add back the items. Map-instances are stored as an array of key-values.

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 "TA Aware Collections" on page 48

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 146

You might wonder what is better to use, collection or arrays. Most of times it doesn't matter. See "Collections Or Arrays" on page 146

Collections Or Arrays

If you are planning an application with db4o, you may be asking yourself, what is better to use: collections or arrays? In the current implementation it is not really a difficult choice, as collections internally are stored as arrays, which is explained in [Collections](#) chapter. 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 "TA Aware Collections" on page 48

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: Create 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

¹Transparent Activation

²Transparent Persistence

Final Fields

This topic will give you an overview of some specifics concerning final fields usage in persistent objects.

More Reading:

- [Final Fields Specifics](#)
- [Possible Solutions](#)

Final Fields Specifics

Db4o uses reflection to store and retrieve objects from the database file. In the case of final fields db4o needs a successful call to `java.lang.Field#setAccessible` to allow write access to those fields. Unfortunately different Java versions produce different results in this case. To be more specific:

- In (Sun) JDK 1.1.2 `java.lang.Field#setAccessible` call will be successful for the fields with the final modifier.
- This behavior was changed for JDK1.3-1.4 as the API documentation for `java.lang.Field#set()` made a quite clear distinction between 'Java language access control' (visibility modifiers, affected by `setAccessible()`) and final fields (not affected by `setAccessible()`). For more information refer to [java bug 4250960](#).
- The behavior of `java.lang.Field#setAccessible` method was changed again for JDK5 and JDK6. The access to final fields was made manageable by `setAccessible()` call to accommodate for the extended semantics of the final modifier for the revised Java memory model. The API documentation of `java.lang.Field#set()` was changed accordingly. See [java bug 5044412](#).

You can use the following example code to check final fields behavior with different java versions:

```
TestFinal.java
/**/* Copyright (C) 2007 Versant Inc. http://www.db4o.com */
package com.db4odoc.finalfields;
import java.io.File;

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

public class TestFinal
{
    private static final String DB4O_FILE_NAME = "reference.db4o";
    // non-final fields
    public int _i;
    public String _s;
    // final fields storing the same values as above
    public final int _final_i;
    public final String _final_s;

    public static void main(String[] args)
    {
        new File(DB4O_FILE_NAME).delete();
        ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
        try {
            TestFinal test = new TestFinal(1,"test");
            container.store(test);
            System.out.println("Added: " + test);
        } finally {
            // Close does implicit commit and refreshes the reference cache
            container.close();
        }
    }
}
```

```

        container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        ObjectSet result = container.queryByExample(null);
        listResult(result);
    } finally {
        container.close();
    }
}
// end main

public TestFinal(int i, String s)
{
    // initialize final and non-final fields with the same values
    _i      = i;
    _s      = s;
    _final_i = i;
    _final_s = s;
}
// end TestFinal

public String toString()
{
    return "Int - " + _i + "; String - " + _s + "; FINAL Int - "
+ _final_i + "; FINAL String - " + _final_s;
}
// end toString

private static void listResult(ObjectSet result)
{
    while(result.hasNext()) {
        System.out.println(result.next());
    }
}
// end listResult
}

```

If you are using Eclipse it is easy to switch between java versions - you can switch to the versions lower than the one installed on your computer without having to install them all. For example if you are using JDK6 you can easily test your project on JDK1.1 - 1.4 and JDK5. Just go to the project properties, select "Java Build Path" on the left panel and "Libraries" tab on the right panel. Remove the system library currently used. Select "Add library->JRE System Library"; on the next screen check the "Execution Environment" radio button and select the desired environment from the list.

Don't forget to use the appropriate db4o version for the selected java environment version. See [db4o on Java Platforms](#) for more information.

Possible Solutions

Of course, if you only use JDK5 or 6 there are no worries about the final fields at all. But if you do not want to stick to the definite java version and need to have the flexibility of switching to different java versions you currently have 2 solutions:

- avoid using the final modifier in the persistent objects;
- use translator.

An example of the final fields translator can look like this:

```

FinalFieldTranslator.java
/**/* Copyright (C) 2007 Versant Inc. http://www.db4o.com */
package com.db4odoc.finalfields;

```

```

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

// Translator allowing to store final fields on any Java version
public class FinalFieldTranslator implements ObjectConstructor {

    public Object onStore(ObjectContainer container,
        Object applicationObject) {
        System.out.println("onStore for " + applicationObject);
        TestFinal notStorable = (TestFinal) applicationObject;
        // final fields values are stored to an array of objects
        return new Object[] { new Integer(notStorable._final_i),
            notStorable._final_s };
    }

    public Object onInstantiate(ObjectContainer container,
        Object storedObject) {
        System.out.println("onInstantiate for " + storedObject);
        Object[] raw = (Object[]) storedObject;
        // final fields values are restored from the array of objects
        int i = ((Integer) raw[0]).intValue();
        String s = (String) raw[1];
        return new TestFinal(i, s);
    }

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

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

```

The following call should be issued before opening the object container to connect the translator to the TestFinal class:

```
configuration.common().objectClass(TestFinal.class).translate(new FinalFieldTranslator());
```

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.

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 109](#)

<pre>EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration(); configuration.common().objectClass(Person.class).persistStaticFieldValues();</pre>
--

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 of 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.

```

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 storables 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 storables type

```
NonStorableType instance = container.query(NonStorableType.class).get(0);
```

TranslatorExample.java: Load the non storables 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 storables or need special treatment.

- TTransient: Doesn't store the object at all. Usable when you don't want to store instances of certain type. It 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 to 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

One of the most important and convenient things that db4o provides is the ability to store any object just as it is: no interfaces to be implemented, no custom fields, no attributes/annotations - nothing, just a plain object. However, it is not as simple as it may seem - objects are getting more and more complex and sometimes the generic solution is not good enough for specific objects.

This problem was recognized by db4o team long ago, and various solutions were provided to customize the way an object is stored: [Translators](#), transient [fields in Java](#) and .NET etc. However all these means were rather fixing the symptoms but not the disease itself. And the fact is that there is no single generic way to store just any available or future object in the best possible way. But luckily we don't even need it - all we need is a way to write a specific persistence solution for any custom object, and now db4o provides this way through a pluggable TypeHandler4 interface. You can register any number of type handlers to the configuration. Additionally you need to register a predicate which decides which classes are handled by the type handler.

```
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](#).

Use cases and other benefits of the pluggable typehandler interface are reviewed [here](#).

Pluggable Typehandler Benefits

As the name suggests Pluggable Typehandler allows anybody to write custom typehandlers, and thus control the way the class objects are stored to the database and retrieved in a query. Why would you do this? There can be various reasons:

- You know a more performant way to convert objects to byte array or to compare them.
- You need to store only part of the object's information and want to skip unneeded fields to keep the database smaller. You can also do the same using [Transient](#) marker, but this is only possible for classes with available code. Using custom typehandler you can configure partial storage for any third-party class.

- You need to keep information that will allow you to restore fields that cannot be stored as is, for example: references to environmental variables (like time zone), proxy objects or variables of temporary state (like current memory usage). Previously, this job was done by [Translators](#), but certainly custom Typehandler gives you more control and unifies the approach.
- You need to perform a complex refactoring on-the-fly (use typehandler versioning)
- You want to cipher each object before putting it into the database

Other not so common and more difficult in realization behaviours that can be realized with the new Typehandler:

- Customary indexes
- Versioning of typehandlers (can be used for refactoring and db4o version upgrades)

Of course, writing typehandlers is not totally simple, but once you understand how to do that - you will also gain a much deeper understanding of db4o itself. You can start with a [simple example](#) provided in this documentation and continue by looking into existing db4o typehandler implementations: StringHandler, VariableLengthTypeHandler, IndexableTypeHandler etc.

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. Note 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 157

For renaming classes and fields you can use the renaming API. See "Renaming API" on page 157.

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 157 And there are some small limitations when refactoring the field-type. See "Field Refactoring Limitation" on page 158

Unfortunatly db4o doesn't support changing the inheritance hierarchy. See "Refactoring Class Hierarchy" on page 159

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("surname");
```

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-class 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-meta data 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 conversion 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 convert 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 157](#)

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.

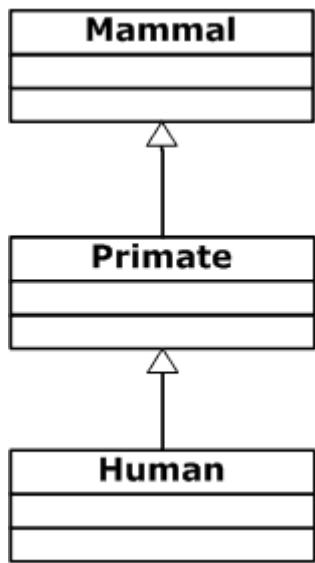
Refactoring Class Hierarchy

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 a class from the inheritance-hierarchy.

Currently the only possible solution for this refactoring is this.

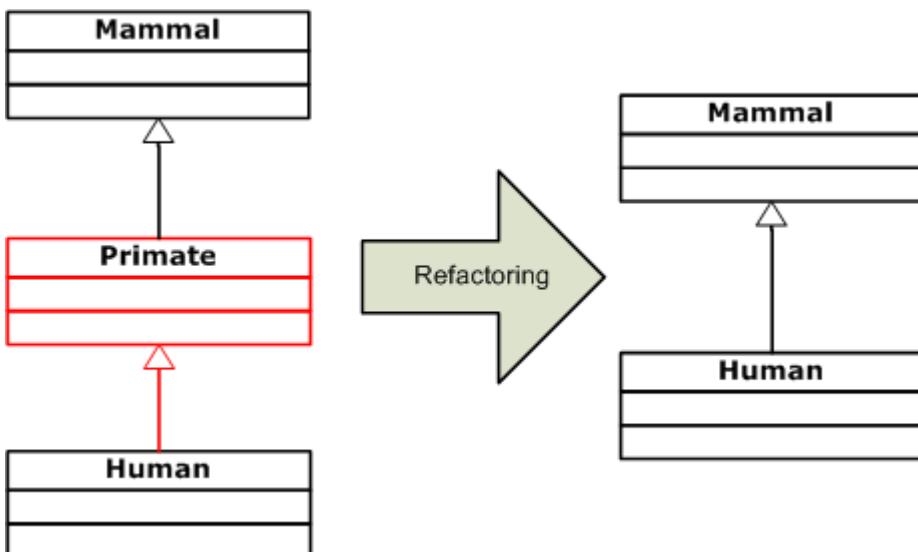
1. Create the new hierarchy with different names, preferably in a new package
2. Copy all values from the old classes to the new classes.
3. Redirect all references from existing objects to the new classes.

Take a look at the example to how to add a class into the hierarchy. See "Inserting Class Into A Hierarchy" on page 161.

Or how you can remove a class from the inheritance hierarchy. See "Removing Class From A Hierarchy" on page 160

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());

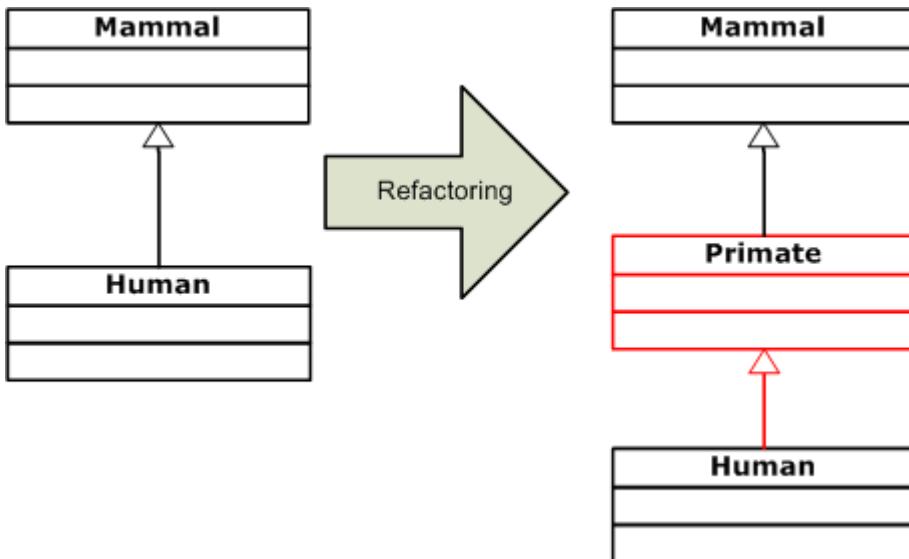
    container.store(newHuman);
    container.delete(oldHuman);
}
```

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 which 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 is to 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 at the list of the most common db4o related exceptions. See "Exception Types" on page 162 Handling db4o exceptions is nothing special and complies with regular exception handling. See "How To Work With db4o Exceptions" on page 163

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 RuntimeException 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 280

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.

IncompatibleFormatException - 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 your own reflector implementation. See "Creating your own reflector" on page 165

db4o has also a generic reflector which can deal with stored objects without using the original class. See "GenericReflector" on page 167

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.

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 168](#)

How should you scope the lifetime of an object container? See this guide-line: [See "Lifetime Of An Object Container" on page 169](#)

Deleting object is always a delicate process. [See "Deleting Objects" on page 170](#)

If you upgrade to new major db4o version you should do these steps to update the file format. [See "Upgrade db4o Version" on page 169](#)

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 - .NET 4.0 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](#)
- [Xml Import-Export In Java](#)
- [Classloader issues](#)
- [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 172](#)

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 175](#)

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 173](#)

Advantages	Disadvantages
<ul style="list-style-type: none"> Internal ids are fast. No additional field on the class required. No additional configuration required. 	<ul style="list-style-type: none"> 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 173](#)

Advantages	Disadvantages
<ul style="list-style-type: none"> A UUID is a worldwide unique id. No additional field on the class required. 	<ul style="list-style-type: none"> 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 173](#)

Advantages	Disadvantages
<ul style="list-style-type: none"> A UUID is a worldwide unique id . UUID are easy to generate and portable. 	<ul style="list-style-type: none"> 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 174](#)

Advantages	Disadvantages
<ul style="list-style-type: none"> A simple id on objects. Familiar model from the RDBMS¹-world. 	<ul style="list-style-type: none"> You need an id-field on your objects. You need to implement an ID-Generator. Which isn't trivial. Additional index required.

¹Relational Database Management System

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 172](#)

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 internalId = 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 172](#)

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.

```
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 172](#)

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 172](#)

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 175](#)

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 176](#)

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 176](#)

```

{
    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're an existing object. If it is, the primitives are copied over, otherwise it's 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 automate 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 177](#)

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 171](#)

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 196](#)

Take a look how you create a object-container for each request. [See "Servlets" on page 178](#)

Take a look at a small example Spring MVC application. [See "Spring MVC Example" on page 180](#)

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. Please refer to the [classloader](#) chapter for more information.

The other issue is configuring, starting and shutting down the db4o correctly. This can be done at the Servelt 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_DB40_FILE_NAME));
    EmbeddedObjectContainer rootContainer = Db4oEmbedded.openFile(filePath);
    context.setAttribute(KEY_DB40_SERVER, rootContainer);
    context.log("db4o startup on " + filePath);
}

@Override
public void contextDestroyed(ServletContextEvent event) {
    ServletContext context = event.getServletContext();
    ObjectContainer rootContainer = (ObjectContainer) context.getAttribute(KEY_DB40_SERVER);
    context.removeAttribute(KEY_DB40_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_DB40_SERVER);

    ObjectContainer session = rootContainer.openSession();
    requestEvent.getServletRequest().setAttribute(KEY_DB40_SESSION, session);
}

@Override
public void requestDestroyed(ServletRequestEvent requestEvent) {
    ObjectContainer session = (ObjectContainer) requestEvent
        .getServletRequest().getAttribute(KEY_DB40_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 its 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 recommendations, 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 172](#)

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 175

```
@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

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 186

Take a look at the [PasswordSafe](#) and [MapMe](#) sample applications for Android. The examples show how easy it is to use db4o on Android.

It's easy to start. Setting up db4o only takes a few steps. See "Getting Started" on page 181.

Due to differences between Android and other Java platforms, there are some additional pitfalls. See "Pitfalls and Tested Functionality" on page 182

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 183

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. For example:

```

public class Db4oOnAndroidExample extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        String filePath = this.getFilesDir() + "/android.db4o";
        ObjectContainer db = Db4oEmbedded.openFile(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 183](#)

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 143](#)

```

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 132](#)

Example Build Time Enhancement for Android

This example is based on the [build time enhancement example](#). 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="${basedir}/bin"/>
        <!-- The android.platform points the Android-platform libraries -->
    <fileset dir="${android.platform}"/>
        <include name="android.jar"/>
    </fileset>
    <fileset dir="lib">
        <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 56

```
<target name="enhance">
    <db4o-enhance classtargetdir="${basedir}/bin"
        jartargetdir="${basedir}/lib"
        nq="true" ta="false"
        collections="false">
        <classpath refid="project.classpath"/>
        <sources dir="${basedir}/bin">
            <include name="**/*.class"/>
        </sources>
        <jars dir="${basedir}/lib">
            <include name="*.jar"/>
            <exclude name="db4o-*.jar"/>
            <exclude name="ant.jar"/>
            <exclude name="bloat-1.0.jar"/>
        </jars>
    </db4o-enhance>
</target>
```

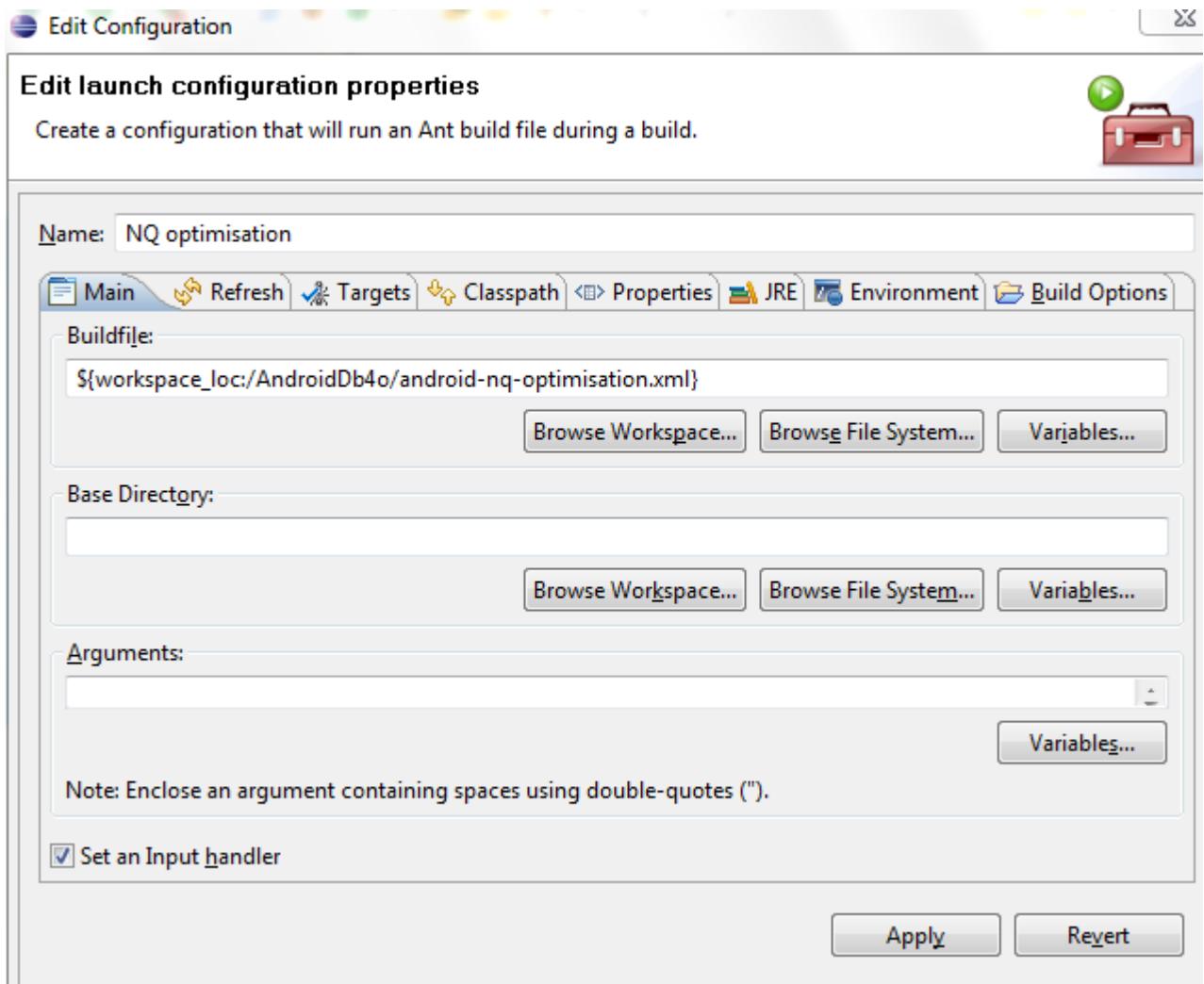
android-nq-optimisation.xml: Define a target which runs the task

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'.

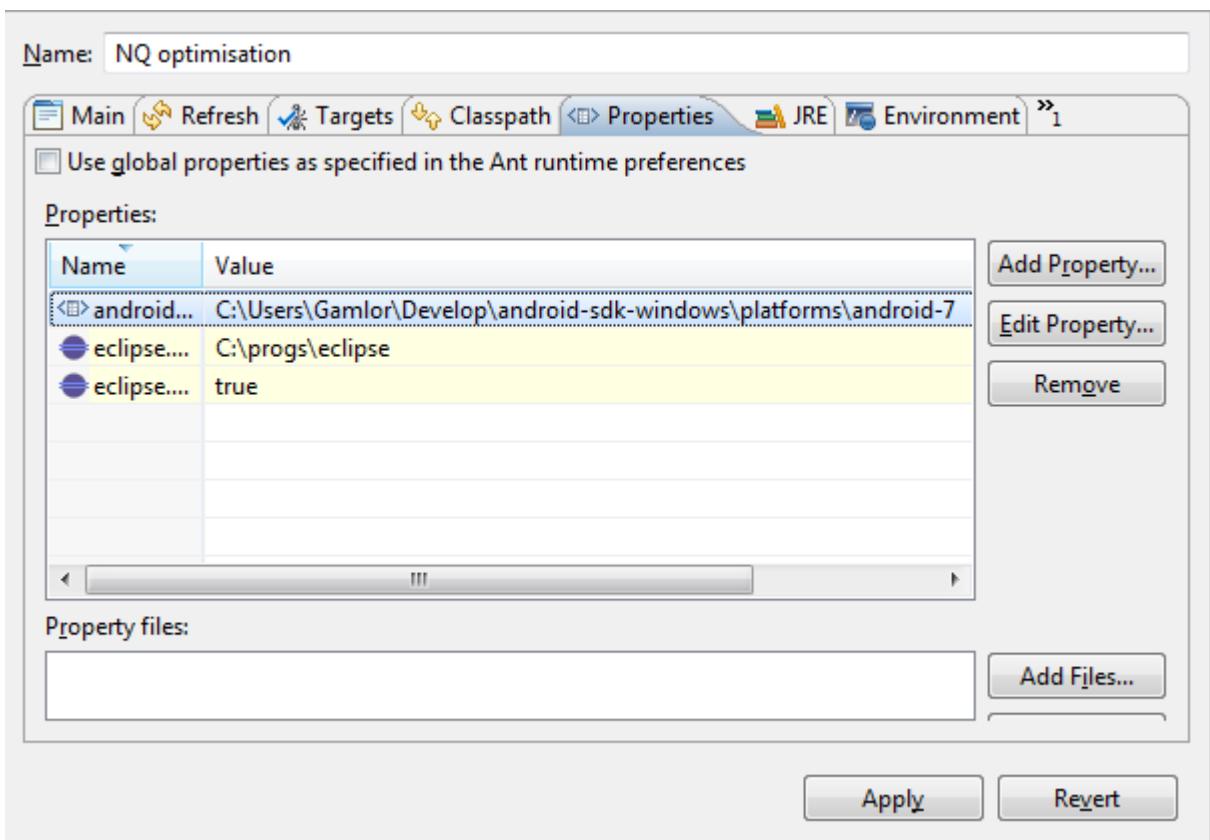
On the 'Main'-tab select the enhancement-script:

¹Transparent Activation

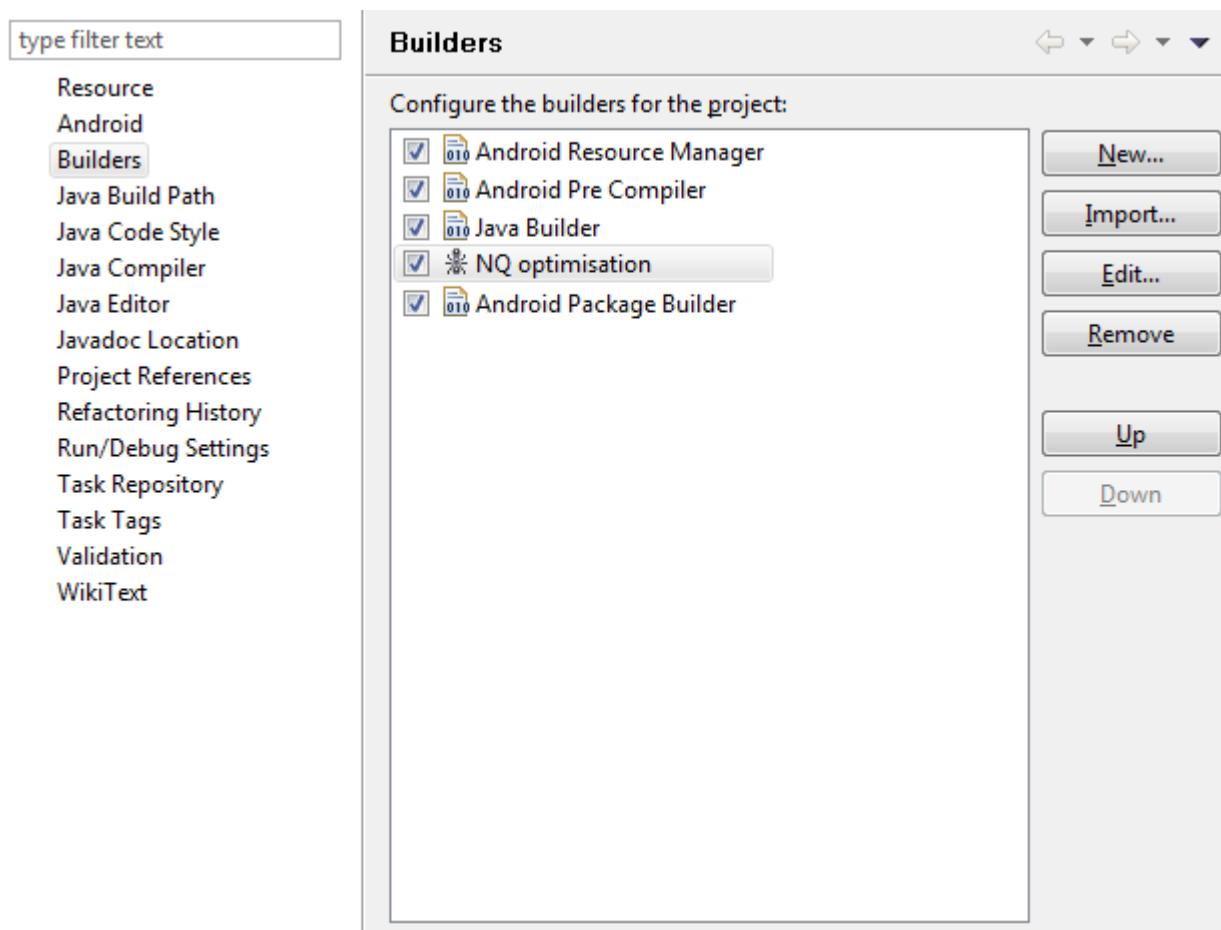
²Transparent Persistence



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 your Android-SDK. The platforms are in the folder 'platforms' in the Android SDK. For example when you're 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':



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 geolocation 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](#)
- [Opening A Database](#)
- [Storing Data](#)
- [Retrieving Data](#)
- [Changing Data](#)

- [Deleting Data](#)
- [Backup](#)
- [Closing A Database](#)
- [Schema Evolution](#)

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().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 must 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 having 15 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 [125](#)

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](#).

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 Java

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 tools available for Java and .NET (freeware and licensed) this is not really necessary. All that you need to export your database/query results is:

1. Retrieve objects from the database.
2. Serialize them in XML format (using language, or external tools, or your own serializing software).
3. Save XML stream (to a disc location, into memory, into another database).

Import process is just the reverse:

1. Read 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.

First, let's prepare a database:

```
SerializeExample.java: setObjects
private static void setObjects() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        Car car = new Car("BMW", new Pilot("Rubens Barrichello"));
        container.store(car);
        car = new Car("Ferrari", new Pilot("Michael Schumacher"));
        container.store(car);
    } finally {
        container.close();
    }
}
```

We will save the database to XML file "formula1.xml":

```
SerializeExample.java: exportToXml
private static void exportToXml() {
    XStream xstream = new XStream(new DomDriver());
    try {
        FileWriter xmlFile = new FileWriter(XMLXML_FILE_NAME);
        ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
        try {
            ObjectSet result = container.query(Car.class);
            Car[] cars = new Car[result.size()];
            for (int i = 0; i < result.size(); i++) {
                Car car = (Car) result.next();
                cars[i] = car;
            }
            String xml = xstream.toXML(cars);
            xmlFile.write("<?xml version=\"1.0\"?>\n" + xml);
            xmlFile.close();
        } finally {
            container.close();
        }
    } catch (Exception ex) {
        System.out.println(ex.getMessage());
    }
}
```

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:

```
SerializeExample.java: importFromXml
private static void importFromXml() {
    new File(DB4O_FILE_NAME).delete();
    XStream xstream = new XStream(new DomDriver());
    try {
        FileReader xmlReader = new FileReader(XMLXML_FILE_NAME);
        Car[] cars = (Car[]) xstream.fromXML(xmlReader);
        ObjectContainer container;
        for (int i = 0; i < cars.length; i++) {
            container = Db4o.openFile(DB4O_FILE_NAME);
            try {
```

```

        Car car = (Car) cars[i];
        container.store(car);
    } finally {
        container.close();
    }
}
container = Db4o.openFile(DB4O_FILE_NAME);
try {
    ObjectSet result = container.query(Pilot.class);
    listResult(result);
    result = container.query(Car.class);
    listResult(result);
} finally {
    container.close();
}
xmlReader.close();
} catch (Exception ex) {
    System.out.println(ex.getMessage());
}
}
}

```

Easy, isn't it? Obviously there is much more about 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.

Classloader Issues

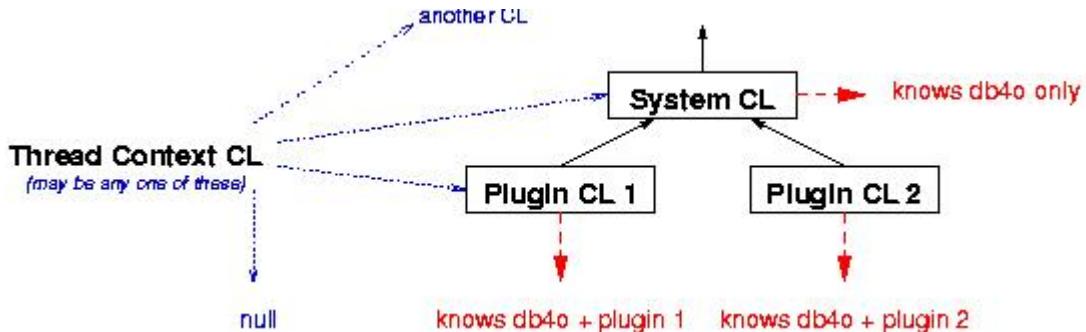
Db4o needs to know its own classes, of course, and it needs to know the class definitions of the objects it stores. (In Client/Server mode, both the server and the clients need access to the class definitions.) While this usually is a non-issue with self-contained standalone applications, it can become tricky to ensure this condition when working with plugin frameworks, where one might want to deploy db4o as a shared library for multiple plugins, for example.

More Reading:

- [Classloader basics](#)
- [Configuration](#)
- [Special Cases](#)

Classloader basics

Classloaders are organized in a tree structure, where classloaders deeper down the tree (usually) delegate requests to their parent classloaders and thereby 'share' their parent's knowledge.



An in-depth explanation of the classloaders functionality is beyond the scope of this documentation. Starting points might be found here:

<http://www.javaworld.com/javaworld/javaqa/2003-06/01-qa-0606-load.htm>

<http://java.sun.com/developer/technicalArticles/Networking/classloaders/>

<http://java.sun.com/products/jndi/tutorial/beyond/misc/classloader.htm>

Configuration

Db4o can be configured to use a user-defined classloader.

```
Java: configuration.common().reflectWith(new JdkReflector(classloader))
```

This line will configure db4o to use the provided classloader. Note that, as with most db4o configuration options, this configuration will have to occur before the respective database has been opened.

The usual ways of getting a classloader reference are:

- Using the classloader the class containing the currently executed code was loaded from. (*this.getTClass().getClassLoader()*)
- Using the classloader db4o was loaded from. (*Db4o.class.getClassLoader()*)
- Using the classloader your domain classes were loaded from. (*SomeDomainClass.class.getClassLoader()*)
- Using the context classloader that may have been arbitrarily set by the execution environment. (*Thread.currentThread().getContextClassLoader()*).

To choose the right classloader to use, you have to be aware of the classloader hierarchy of your specific execution environment. As a rule of thumb, one should configure db4o to use a classloader as deep/specialized in the tree as possible. In the above example this would be the classloader of the plugin db4o is supposed to work with.

Special Cases

In your average standalone program you'll probably never have to face these problems, but there are standard framework classics that'll force you to think about these issues.

Servlet container

In a typical servlet container, there will be one or more classloader responsible for internal container classes and shared libraries, and one dedicated classloader per deployed web application. If you deploy db4o within your web application, there should be no problem at all. When used as a shared library db4o has to be configured to use the dedicated web application classloader. This can be done by assigning the classloader of a class that's present in the web application only, or by using the context classloader, since all servlet container implementations we are aware of will set it accordingly.

You will find more detailed information on classloader handling in Tomcat, the reference servlet container implementation, here:

<http://jakarta.apache.org/tomcat/tomcat-4.1-doc/class-loader-howto.htm>

Eclipse

Eclipse uses the system classloader to retrieve its core classes. There is one dedicated classloader per plugin, and the classloader delegation tree will resemble the plugin dependency tree. The context classloader will usually be the system classloader that knows nothing about db4o and your business classes. So the best candidate is the classloader for one of your domain classes within the plugin.

Running Without Classes

db4o can also cope with missing class definitions. This is a by-product of the work on our [object manager](#) application. Another implementation is a [server without persistent classes deployed](#).

Database For OSGi

db4o osgi project (since db4o-6.3) provides a service, which allows to use db4o in **OSGi**¹ environment. Its usage is the usage of an OSGI service, which is well documented in the Internet. Short and essential guide to OSGI service usage can be found [here](#), but you are surely free to use any suitable for you technique to access db4o osgi service.

The main purpose of db4o osgi service is to configure an **OSGi**² bundle aware reflector for the database instance, so that classes that are owned by the client bundle are accessible to the db4o engine. To emulate this behavior when using db4o directly through the exported packages of the db4o osgi plugin, db4o can be configured like this:

```
Configuration config = Db4o.newConfiguration();
config.reflectWith(new JdkReflector(SomeData.class.getClassLoader()));
// ...
ObjectContainer database = Db4o.openFile(config,fileName);
```

Access through the service is recommended over the direct usage, though, as the service may implement further OSGi specific features in the future.

Db4o-Osgi Usage

db4o-osgi service can be accessed like any other **OSGi**³ service:

```
ServiceReference serviceRef = _context.getServiceReference(Db4oService.class.getName());
Db4oService db4oService = (Db4oService)_context.getService(serviceRef);
```

db4o-osgi uses Bundle-ActivationPolicy:lazy header to define the lazy bundle loading policy (only utilized in some environments, like Eclipse).

Db4oService instance can be used as Db4o class in usual environment:

```
Configuration config = db4oService.newConfiguration();
ObjectContainer db = db4oService.openFile(config, filename);
```

Also available are methods for opening db4o server and client. For more information see the API documentation.

Once the service instance is obtained, you can continue to work with db4o API as usual.

Installation in Eclipse

1. Put the db4o osgi jar file on eclipse/plugins folder
2. Start/restart Eclipse
3. While on Eclipse switch from the "Package Explorer" tab to the "Plug-ins" tab
4. Find the db4o bundle (should be listed there), right click on it and select "Import As" -> "Source Project"

Eclipse Plug-In With Db4o Service

The following example was created to show a practical usage of db4o osgi service. Though targeting a wide auditory, it can be especially helpful to people new to **OSGi**⁴ and plug-in development.

¹OSGi framework, a module system and service platform

²OSGi framework, a module system and service platform

³OSGi framework, a module system and service platform

⁴OSGi framework, a module system and service platform

In this example we will create a simple Eclipse UI plug-in, which will store notes between Eclipse sessions using db4o as storage.

To be able to follow the explanation you will need:

- JDK 1.5
- db4o for Java ([download](#))
- Eclipse IDE ([download](#)).

More Reading:

- [Creating A Plugin](#)
- [Code Overview](#)
- [Connecting To Db4o](#)
- [Testing MemoPlugin](#)

Creating A Plugin

First of all you will need to install db4o osgi into the Eclipse environment. You can do that by copying db4o osgi.jar into ECLIPSE_HOME\plugins folder, restart Eclipse, then switch to Plug-Ins view, select db4o-osgi plug-in, right-click and select "Import As->Source Project".

If you do not want to do that, you can open Eclipse and create a usual java project from the db4o osgi sources.

Now you are ready to create a new plug-in project.

- Open Eclipse workspace if not yet opened.
- Select File/New from the menu and select "Plug-in Project" as the project type.
- Select MemoPlugin as the project name, leave the default values for the other settings and press "Next"
- Leave all the default values and press "Next"
- In the "Templates" screen select "Hello, World" template. This template creates a menu in the Eclipse environment, which we will use for our example. Click "Finish"

You might be asked to switch to "Plug-in development" perspective, which you can surely do.

You should see a MemoPlugin window opened in the environment. This window represents important plug-in properties and it can be opened by double-clicking plugin.xml file in Package Explorer.

You can use the tab-scroll at the bottom to navigate to different pages. Please, open the "Overview" page of the plugin.xml window and review the information presented there. Note, that this page can be used to start testing and debugging (see Testing paragraph).

Our plug-in will depend on db4o osgi bundle; therefore we must define this dependency somewhere. Select "Dependencies" hyperlink in the "Plug-in Content" paragraph. (You can gain the same effect by selecting "Dependencies" tab page.) In the "Required Plug-ins" list click "Add" and select "db4o-osgi".

Please, note that you should not specify Java Build path as in a normal Java project, otherwise the environment will find duplicates in your project dependencies.

Code Overview

Now the plug-in environment is configured and we can look at the code itself.

At this point in time the project contains the following classes:

- memoplugin.Activator
- memoplugin.actions.SampleAction

Activator class is called to start and stop the plug-in. It is responsible for managing its lifecycle. We will use it to initialize and clean up db4o resources.

SampleAction is a class that performs the action specified in the action set in the manifest file. It can be used to specify the behavior on the action. We will use it to call a custom dialog for memo viewing and editing.

From the said above we can see that we will need 2 more classes:

- Db4oProvider: will be used to keep db4o connection, provide it to the users on request, and close it on dispose.
- DataDialog: will provide a simple UI for viewing and editing the data, it will use Db4oProvider to access and store the data.

These 2 classes are very basic and are not specific to **OSGI**¹ environment. Please, review their code below:

```
Db4oProvider.java
package memoplugin;

import com.db4o.ObjectContainer;
import com.db4o.osgi.Db4oService;
/**/*
 * This class is used to store db4o osgi service instance and
 * provide db4o services on request.
 */
public class Db4oProvider {

    private static ObjectContainer _db;
    private static String FILENAME = "sample.db4o";

    public static void Initialize(Db4oService db4oService) {
        _db = db4oService.openFile(FILENAME);
    }

    public static ObjectContainer database() {
        return _db;
    }

    public static void UnInitialize() {
        _db.close();
    }
}
```

```
DataDialog.java
package memoplugin.ui;

import java.util.ArrayList;

import org.eclipse.jface.dialogs.Dialog;
import org.eclipse.jface.dialogs.IDialogConstants;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
```

¹OSGi framework, a module system and service platform

```
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.List;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;

import memopugin.Db4oProvider;

import com.db4o.ObjectSet;

public class DataDialog extends Dialog {
    private static int ID_ADD = 100;
    private static int ID_DELETE = 101;
    private Shell _shell;
    /**
     * The title of the dialog.
     */
    private String title;

    /**
     * The message to display, or <code>null</code> if none.
     */
    private String message;

    /**
     * The input value; the empty string by default.
     */
    private String value = "";//$NON-NLS-1$

    /**
     * Add button widget.
     */
    private Button addButton;

    /**
     * Delete button widget.
     */
    private Button deleteButton;

    /**
     * Input text widget.
     */
    private Text text;

    /**
     * List widget.
     */
    private List list;

    public DataDialog(Shell parentShell, String dialogTitle,
                     String dialogMessage, String initialValue) {
        super(parentShell);
        this.title = dialogTitle;
        message = dialogMessage;
        if (initialValue == null) {
            value = "";//$NON-NLS-1$
        } else {
            value = initialValue;
        }
    }
}
```

```

}

/**
 *
 *
 * @see org.eclipse.jface.window.Window#configureShell(org.eclipse.swt.widgets.Shell)
 */
protected void configureShell(Shell shell) {
    super.configureShell(shell);
    _shell = shell;
    if (title != null) {
        shell.setText(title);
    }
}

/**
 * Clears the database before adding new data
 */
private void clearDb() {
    ObjectSet result = Db4oProvider.database().queryByExample(ArrayList.class);
    while (result.hasNext()) {
        Db4oProvider.database().delete(result.next());
    }
}

/**
 *
 * Makes sure that all the data is saved to the
 * database before closing the dialog
 */
protected void handleShellCloseEvent() {
    clearDb();
    ArrayList data = new ArrayList();
    for (int i=0; i < list.getItemCount(); i++) {
        data.add(list.getItem(i));
    }
    Db4oProvider.database().store(data);
    Db4oProvider.database().commit();
    Db4oProvider.database().ext().purge(ArrayList.class);
    super.handleShellCloseEvent();
}

/**
 * Button events handler
 */
protected void buttonPressed(int buttonId) {
    if (buttonId == ID_ADD) {
        value = text.getText();
        list.add(value);
    } else if (buttonId == ID_DELETE) {
        int selectedId = list.getSelectionIndex();
        if (selectedId == -1) {
            new MessageDialog(_shell, "Error",
                null, "No item selected", MessageDialog.ERROR,
                new String[] {"Ok"}, 0).open();
        } else {
            list.remove(selectedId);
        }
        value = null;
    } else {

```

```

        super.buttonPressed(buttonId);
    }
}

/**
 *
 *
 * @see org.eclipse.jface.dialogs.Dialog#createButtonsForButtonBar(org.eclipse.swt.widgets.Composite)
 */
protected void createButtonsForButtonBar(Composite parent) {
    // create Add and Delete buttons by default
    addButton = createButton(parent, ID_ADD,
        "Add", true);
    createButton(parent, ID_DELETE,
        "Delete", false);
    //do this here because setting the text will set enablement on the ok
    // button
    text.setFocus();
    if (value != null) {
        text.setText(value);
        text.selectAll();
    }
}

/**
 * Creates the visual dialog representation
 */
protected Control createDialogArea(Composite parent) {
    // create composite
    Composite composite = (Composite) super.createDialogArea(parent);
    // create message
    if (message != null) {
        Label label = new Label(composite, SWT.WRAP);
        label.setText(message);
        GridData gridData = new GridData(GridData.GRAB_HORIZONTAL
            | GridData.GRAB_VERTICAL | GridData.HORIZONTAL_ALIGN_FILL
            | GridData.VERTICAL_ALIGN_CENTER);
        gridData.widthHint = convertHorizontalDLUsToPixels(IDialogConstants.MINIMUM_MESSAGE_AREA_WIDTH);
        label.setLayoutData(gridData);
        label.setFont(parent.getFont());
    }
    text = new Text(composite, SWT.SINGLE | SWT.BORDER);
    text.setLayoutData(new GridData(GridData.GRAB_HORIZONTAL
        | GridData.HORIZONTAL_ALIGN_FILL));

    list = new List(composite, SWT.SINGLE|SWT.H_SCROLL|SWT.V_SCROLL);
    GridData gridData = new GridData(SWT.FILL,SWT.FILL, true, true);
    gridData.heightHint = 50;
    list.setLayoutData(gridData);
    ObjectSet result = Db4oProvider.database().query(ArrayList.class);
    if (result.size() != 0) {
        ArrayList data = (ArrayList)result.next();
        String[] items = new String[data.size()];
        for (int i=0; i < data.size(); i++) {
            items[i] = (String)data.queryByExample(i);
        }
        list.setItems(items);
    }

    applyDialogFont(composite);
}

```

```

        return composite;
    }

    /**
     * Returns the string typed into this input dialog.
     *
     * @return the input string
     */
    public String getValue() {
        return value;
    }
}

```

In order to call the above-mentioned DataDialog we will need to modify the generated `run` method in `SampleAction` class:

```

SampleAction.java: run
/**
 * The action has been activated. The argument of the
 * method represents the 'real' action sitting
 * in the workbench UI.
 * @see IWorkbenchWindowActionDelegate#run
 */
public void run(IAction action) {
    /**
     * Call DataDialog to view and edit memo notes
     */
    DataDialog d = new DataDialog(window.getShell(), "db4o-osgi",
"Enter an item to add to the list:",null);
    d.open();
}

```

Connecting To Db4o

The only thing left - is a connection to the db4o_osgi plug-in. It can be established upon the plug-in start and terminated upon the plug-in stop:

```

Activator.java: start
/**
 * (non-Javadoc)
 * @see org.eclipse.ui.plugin.AbstractUIPlugin#start(BundleContext)
 * Obtains a db4o_osgi service reference and registers it with Db4oProvider
 */
public void start(BundleContext context) throws Exception {
    super.start(context);
    ServiceReference serviceRef = context.
getServiceReference(Db4oService.class.getName());
    Db4oService db4oService = (Db4oService)context.getService(serviceRef);
    Db4oProvider.Initialize(db4oService);
}

Activator.java: stop
/**
 * (non-Javadoc)
 * @see org.eclipse.ui.plugin.AbstractUIPlugin#stop(BundleContext)
 * Unregisters the db4o_osgi service from Db4oProvider
 */
public void stop(BundleContext context) throws Exception {
    Db4oProvider.UnInitialize();
    plugin = null;
}

```

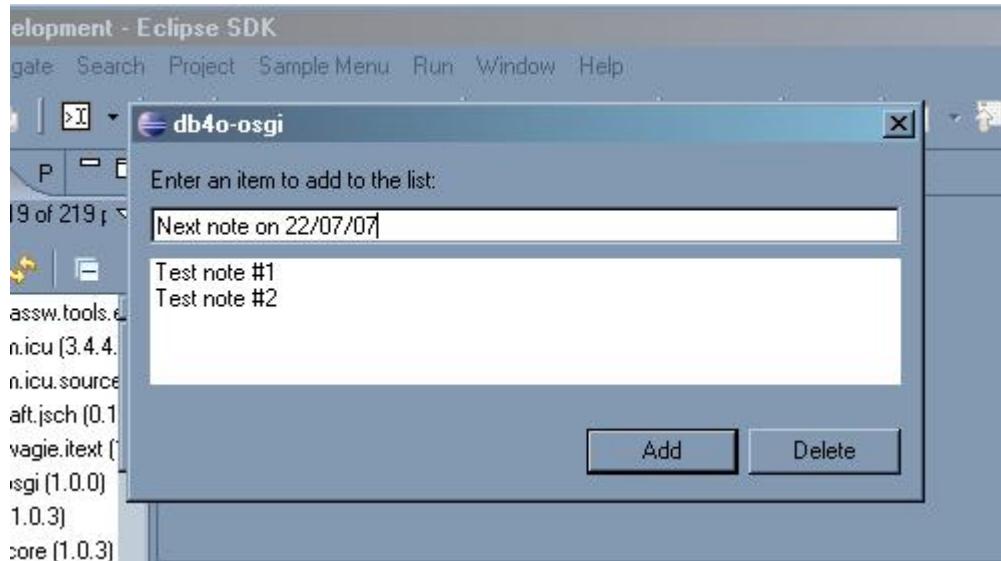
```
    super.stop(context);
}
```

Testing MemoPlugin

Eclipse environment makes it very easy to test plug-in projects: you do not need to exit your workspace or manually activate plug-ins.

In order to test the new MemoPlugin, open plugin.xml by double-clicking the file in Package Explorer, select Overview tab and click "Launch an Eclipse application" link.

If everything worked out right, you should see "Sample Menu" with a "Sample Action" submenu. Click "Sample Action", you should see a window like this:



Try to add and delete several items. You can close the window and open it again to check that the changes are saved. You can also test the same after restarting Eclipse to see that the changes are not lost between sessions.

Note, that the database file is created in the ECLIPSE_HOME, so you must have write and create writes there.

Tuning

This topic set explains different configuration, debugging and diagnostics issues. This information will help you to fine-tune your db4o usage and chase away bugs and performance pitfalls.

More Reading:

- [Main Operations Performance](#)
- [IO Benchmark Tools](#)
- [Selective Persistence](#)
- [Indexing](#)
- [Performance Hints](#)
- [Debugging db4o](#)
- [Diagnostics](#)
- [Native Query Optimization](#)
- [Utility Methods](#)

Main Operations Performance

One of the most important factors in database usage is performance. In the same time it is something difficult to measure and predict as there are too many factors affecting it. These factors can be dependent or independent of database implementation. Independent factors, such as operating memory, processor speed etc are general for all applications and in many cases are given as initial conditions, which do not allow frequent or tuning at all (for example, embedded mobile devices). On the other hand, dependent factors can usually be changed programmatically and provide valuable effect.

The following articles will give you some average numbers of db4o performance, providing the testing code that can be easily modified to accommodate your object model and environment and pointing out the most influencing performance factors.

More Reading:

- [Insert Performance](#)
- [Delete Performance](#)
- [Update Performance](#)
- [Query Performance](#)

Insert Performance

The following chapters provide some performance testing examples, revealing the most influential performance factors. Together with the examples there are some approximate time measurement values that were achieved on a Toshiba Sattelite Pro A120 notebook with 1Gb RAM 120GB ATA drive running on Vista. Please, note that these values are not guaranteed and can vary considerably depending on a hardware and software used.

In most of the tests the following simple object was used:

```
InsertPerformanceBenchmark.java: Item
public static class Item {

    public String _name;
    public Item _child;

    public Item() {
```

```

    }

    public Item(String name, Item child) {
        _name = name;
        _child = child;
    }
}

```

In the tests Item objects were created with 3 levels of embedded Item objects. The amount of objects was varied for different tests.

Please, be cautious to compare results of different tests presented as different configurations are used in each test.

More Reading:

- [Hardware Resources](#)
- [Local And Remote Modes](#)
- [Commit Frequency](#)
- [Object Structure](#)
- [Indexes](#)
- [Inherited Objects](#)
- [Configuration Options](#)

Hardware Resources

Initial object storing requires little calculation, but can be resource consuming on disk access. Therefore the main hardware resource that will affect db4o insert performance is the hard drive. The faster is the hard drive the better performance you will get.

An alternative to a hard drive database storage can be a database file stored in RAM. This can be done by placing the database file in a designated RAM-drive or by using db4o memory storage:

```

EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
MemoryStorage memory = new MemoryStorage();
configuration.file().storage(memory);
ObjectContainer container = Db4oEmbedded.openFile(configuration, "database.db4o");

```

IOConfigurationExamples.java: Using memory-storage

Local And Remote Modes

Of course local and client/server modes cannot give the same performance and it is difficult to say what will be the impact of inserting the objects over the network, as the network conditions can vary.

You can use the following test to compare the performance on your network:

```

InsertPerformanceBenchmark.java: runClientServerTest
private void runClientServerTest() {

    configureClientServer();

    init();
    clean();
    System.out.println("Storing " + _count + " objects of depth "
+ _depth + " locally:");
    open();
    store();
    close();
}

```

```

        initForClientServer();
        clean();
        System.out.println("Storing " + _count + " objects of depth "
+ _depth + " remotely:");
        open();
        store();
        close();

    }

InsertPerformanceBenchmark.java: configureClientServer
private void configureClientServer() {
    Configuration config = Db4o.configure();
    config.lockDatabaseFile(false);
    config.weakReferences(false);
    config.flushFileBuffers(false);
    config.clientServer().singleThreadedClient(true);
}

InsertPerformanceBenchmark.java: init
private void init() {
    _count = 1000;
    _depth = 3;
    _isClientServer = false;

}

InsertPerformanceBenchmark.java: initForClientServer
private void initForClientServer() {
    _count = 1000;
    _depth = 3;
    _isClientServer = true;
    _host = "localhost";
}

InsertPerformanceBenchmark.java: store
private void store() {
    startTimer();
    for (int i = 0; i < _count ;i++) {
        Item item = new Item("load", null);
        for (int j = 1; j < _depth; j++) {
            item = new Item("load", item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store " + totalObjects() + " objects");
}

```

With a good and reliable network you can use the same methods to improve the insert performance as in a local mode. However, if your network connection is not always perfect you will need to use commits more often to ensure that the objects do not get lost. See the [next chapter](#) for recommendations on commit performance.

Commit Frequency

Commit is an expensive operation as it needs to physically access hard drive several times and write changes. However, only commit can ensure that the objects are actually stored in the database and won't be lost.

The following test compares different commit frequencies (one commit for all objects or several commits after a specified amount of objects). The test runs against a hard drive:

```
InsertPerformanceBenchmark.java: runCommitTest
private void runCommitTest() {

    configureForCommitTest();
    initForCommitTest();

    clean();
    System.out.println("Storing objects as a bulk:");
    open();
    store();
    close();

    clean();
    System.out.println("Storing objects with commit after each "
+ _commitInterval + " objects:");
    open();
    storeWithCommit();
    close();
}

InsertPerformanceBenchmark.java: configureForCommitTest
private void configureForCommitTest() {
    Configuration config = Db4o.configure();
    config.lockDatabaseFile(false);
    config.weakReferences(false);
    // flushFileBuffers should be set to true to ensure that
    // the commit information is physically written
    // and in the correct order
    config.flushFileBuffers(true);
}

InsertPerformanceBenchmark.java: initForCommitTest
private void initForCommitTest() {
    _count = 100000;
    _commitInterval = 10000;
    _depth = 3;
    _isClientServer = false;

}

InsertPerformanceBenchmark.java: store
private void store() {
    startTimer();
    for (int i = 0; i < _count ;i++)  {
        Item item = new Item("load", null);
        for (int j = 1; j < _depth; j++)  {
            item = new Item("load", item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store "+ totalObjects() + " objects");
}

InsertPerformanceBenchmark.java: storeWithCommit
private void storeWithCommit() {
    startTimer();
    int k = 0;
    while (k < _count) {
```

```

        for (int i = 0; i < _commitInterval ;i++)  {
            Item item = new Item("load", null);
            k++;
            for (int j = 1; j < _depth; j++)  {
                item = new Item("load", item);
            }
            objectContainer.store(item);
        }
        objectContainer.commit();
    }
    objectContainer.commit();
    stopTimer("Store "+ totalObjects() + " objects");
}

```

Note, that you can get an OutOfMemory exception when running the part of the test with a single commit. To fix this use -Xmx500m setting for your Java machine.

The following results were achieved for the [testing configuration](#):

Java:

Storing objects as a bulk:

Store 300000 objects: 11974ms

Storing objects with commit after each 10000 objects:

Store 300000 objects: 14692ms

Object Structure

Object Structure naturally has a major influence on insert performance: inserting one object, which is a linked list of 1000 members, is much slower than inserting an object with a couple of primitive fields.

The following test compares storing time of similar objects with one different field:

```

InsertPerformanceBenchmark.java: runDifferentObjectsTest
private void runDifferentObjectsTest() {

    configure();
    init();
    System.out.println("Storing " + _count + " objects with " + _depth
+ " levels of embedded objects:");

    clean();
    System.out.println(" - primitive object with int field");
    open();
    storeSimplest();
    close();

    open();
    System.out.println(" - object with String field");
    store();
    close();

    clean();
    open();
    System.out.println(" - object with StringBuffer field");
    storeWithStringBuffer();
    close();

    clean();
    open();

```

```

        System.out.println(" - object with int array field");
        storeWithArray();
        close();

        clean();
        open();
        System.out.println(" - object with ArrayList field");
        storeWithArrayList();
        close();

    }

InsertPerformanceBenchmark.java: configure
private void configure() {
    Configuration config = Db4o.configure();
    config.lockDatabaseFile(false);
    config.weakReferences(false);
    config.io(new MemoryIoAdapter());
    config.flushFileBuffers(false);
}

InsertPerformanceBenchmark.java: init
private void init() {
    _count = 10000;
    _depth = 3;
    _isClientServer = false;

}

InsertPerformanceBenchmark.java: storeSimplest
private void storeSimplest() {
    startTimer();
    for (int i = 0; i < _count ;i++) {
        SimplestItem item = new SimplestItem(i, null);
        for (int j = 1; j < _depth; j++) {
            item = new SimplestItem(i, item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store "+ totalObjects() + " objects");
}

InsertPerformanceBenchmark.java: store
private void store() {
    startTimer();
    for (int i = 0; i < _count ;i++) {
        Item item = new Item("load", null);
        for (int j = 1; j < _depth; j++) {
            item = new Item("load", item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store "+ totalObjects() + " objects");
}

InsertPerformanceBenchmark.java: storeWithStringBuffer
private void storeWithStringBuffer() {
    startTimer();
    for (int i = 0; i < _count ;i++) {
        ItemWithStringBuffer item = new ItemWithStringBuffer(

```

```

new StringBuffer("load"), null);
        for (int j = 1; j < _depth; j++) {
            item = new ItemWithStringBuffer(new StringBuffer("load"), item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store "+ totalObjects() + " objects");
}

InsertPerformanceBenchmark.java: storeWithArray
private void storeWithArray() {
    startTimer();
    int[] array = new int[] {1,2,3,4};
    for (int i = 0; i < _count ;i++) {
        int[] id = new int[] {1,2,3,4};
        ItemWithArray item = new ItemWithArray(id, null);
        for (int j = 1; j < _depth; j++) {
            int[] id1 = new int[] {1,2,3,4};
            item = new ItemWithArray(id1, item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store "+ totalObjects() + " objects");
}

InsertPerformanceBenchmark.java: storeWithArrayList
private void storeWithArrayList() {
    startTimer();
    ArrayList idList = new ArrayList();
    idList.add(1);
    idList.add(2);
    idList.add(3);
    idList.add(4);
    for (int i = 0; i < _count ;i++) {
        ArrayList ids = new ArrayList();
        ids.addAll(idList);
        ItemWithArrayList item = new ItemWithArrayList(ids, null);
        for (int j = 1; j < _depth; j++) {
            ArrayList ids1 = new ArrayList();
            ids1.addAll(idList);
            item = new ItemWithArrayList(ids1, item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store "+ totalObjects() + " objects");
}

InsertPerformanceBenchmark.java: SimplestItem
public static class SimplestItem {

    public int _id;
    public SimplestItem _child;

    public SimplestItem() {
    }

    public SimplestItem(int id, SimplestItem child) {
        _id = id;
    }
}

```

```
        _child = child;
    }
}

InsertPerformanceBenchmark.java: ItemWithArray
public static class ItemWithArray {

    public int[] _id;
    public ItemWithArray _child;

    public ItemWithArray() {
    }

    public ItemWithArray(int[] id, ItemWithArray child) {
        _id = id;
        _child = child;
    }
}
```

```
InsertPerformanceBenchmark.java: ItemWithArrayList
public static class ItemWithArrayList {

    public ArrayList _ids;
    public ItemWithArrayList _child;

    public ItemWithArrayList() {
    }

    public ItemWithArrayList(ArrayList ids, ItemWithArrayList child) {
        _ids = ids;
        _child = child;
    }
}
```

```
InsertPerformanceBenchmark.java: ItemWithStringBuffer
public static class ItemWithStringBuffer {

    public StringBuffer _name;
    public ItemWithStringBuffer _child;

    public ItemWithStringBuffer() {
    }

    public ItemWithStringBuffer(StringBuffer name,
ItemWithStringBuffer child) {
        _name = name;
        _child = child;
    }
}
```

The following results were achieved for the [testing configuration](#):

Java:

Storing 10000 objects with 3 levels of embedded objects:

- primitive object with int field

- Store 30000 objects: 820ms

- object with String field

- Store 30000 objects: 803ms

- object with StringBuffer field

Store 30000 objects: 2182ms

- object with int array field

Store 30000 objects: 810ms

- object with ArrayList field

Store 30000 objects: 2178ms

Indexes

One more feature that inevitably decreases the insert performance: indexes. When a new object with indexed field is inserted an index should be created and written to the database, which consumes additional resources. Luckily indexes do not only reduce the performance, actually they will improve the performance to a much more valuable degree during querying.

An example below provides a simple comparison of storing objects with and without indexes:

```
InsertPerformanceBenchmark.java: runIndexTest
private void runIndexTest() {

    init();
    System.out.println("Storing " + _count + " objects with "
+ _depth + " levels of embedded objects:");

    clean();
    configure();
    System.out.println(" - no index");
    open();
    store();
    close();

    configureIndex();
    System.out.println(" - index on String field");
    open();
    store();
    close();
}

InsertPerformanceBenchmark.java: configure
private void configure() {
    Configuration config = Db4o.configure();
    config.lockDatabaseFile(false);
    config.weakReferences(false);
    config.io(new MemoryIoAdapter());
    config.flushFileBuffers(false);
}

InsertPerformanceBenchmark.java: configureIndex
private void configureIndex() {
    Configuration config = Db4o.configure();
    config.lockDatabaseFile(false);
    config.weakReferences(false);
    config.io(new MemoryIoAdapter());
    config.flushFileBuffers(false);
    config.objectClass(Item.class).objectField("_name").indexed(true);
}

InsertPerformanceBenchmark.java: init
private void init() {
    _count = 10000;
    _depth = 3;
```

```

        _isClientServer = false;
    }

InsertPerformanceBenchmark.java: store
private void store() {
    startTimer();
    for (int i = 0; i < _count ;i++) {
        Item item = new Item("load", null);
        for (int j = 1; j < _depth; j++) {
            item = new Item("load", item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store "+ totalObjects() + " objects");
}

```

The following results were achieved for the [testing configuration](#):

Java:

Storing 10000 objects with 3 levels of embedded objects:

- no index

Store 30000 objects: 877ms

- index on String field

Store 30000 objects: 1076ms

Inherited Objects

Inherited objects are stored slower than simple objects. That is happening, because parent class indexes are created and stored to the database as well.

The following example shows the influence of a simple inheritance on the insert performance:

```

InsertPerformanceBenchmark.java: runInheritanceTest
private void runInheritanceTest() {

    configure();
    init();
    clean();
    System.out.println("Storing " + _count + " objects of depth " + _depth);
    open();
    store();
    close();

    clean();
    System.out.println("Storing " + _count + " inherited objects of depth "
+ _depth);
    open();
    storeInherited();
    close();

}

InsertPerformanceBenchmark.java: configure
private void configure() {
    Configuration config = Db4o.configure();
    config.lockDatabaseFile(false);
    config.weakReferences(false);
}

```

```

        config.io(new MemoryIoAdapter());
        config.flushFileBuffers(false);
    }

InsertPerformanceBenchmark.java: init
private void init() {
    _count = 10000;
    _depth = 3;
    _isClientServer = false;

}

InsertPerformanceBenchmark.java: store
private void store() {
    startTimer();
    for (int i = 0; i < _count ;i++) {
        Item item = new Item("load", null);
        for (int j = 1; j < _depth; j++) {
            item = new Item("load", item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store "+ totalObjects() + " objects");
}

InsertPerformanceBenchmark.java: storeInherited
private void storeInherited() {
    startTimer();
    for (int i = 0; i < _count ;i++) {
        ItemDerived item = new ItemDerived("load", null);
        for (int j = 1; j < _depth; j++) {
            item = new ItemDerived("load", item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store "+ totalObjects() + " objects");
}

InsertPerformanceBenchmark.java: ItemDerived
public static class ItemDerived extends Item {

    public ItemDerived(String name, ItemDerived child) {
        super(name, child);
    }
}

```

The following results were achieved for the [testing configuration](#):

Java:

Storing 10000 objects of depth 3

Store 30000 objects: 883ms

Storing 10000 inherited objects of depth 3

Store 30000 objects: 938ms

Configuration Options

Configuration options can also affect the insert performance. Some of them we've already came across in the previous topics:

MemoryStorage- improves the insert performance, by replacing disk access with memory access.

lockDatabaseFile- reduces the resources consumption by removing database lock thread. Should only be used for JVM versions < 1.4

weakReferences- switching weak references off during insert operation releases extra resources and removed the cleanup thread.

NonFlushingStorage- switching off flushFileBuffers can improve commit performance as the commit information will be cached by the operating system. However this setting is potentially dangerous and can lead to database corruption.

Delete Performance

Delete operation only consists of marking an object as deleted in the database file and usually is very fast. For the clarity in this case we do not include time necessary to locate the object in the database. The considerations for the best delete performance would be the same as the considerations to make db4o database generally faster and would include:

- fast storage location (hard drive or RAM)
- enough operational memory

Let's consider some of the application setups and their influence on delete performance. In order to distinguish delete time from query time, all the deletions will be done by object id.

More Reading:

- [Commit Frequency](#)
- [Complexity Of Objects](#)
- [Storage Characteristics](#)
- [Client-Server](#)
- [Cascade On Delete](#)

Commit Frequency

Commit time has a major influence on all db4o operations. Commit is generally a slow operation as it requires physical disk access. The following example shows how the frequency of commits affects the delete performance:

```
DeletePerformanceBenchmark.java: runCommitTest
private void runCommitTest() {
    System.out.println("Delete test with different commit frequency");

    configureForCommitTest();
    initForCommitTest();

    clean();
    System.out.println("Test delete all:");
    open();
    store();
    deleteAll();
    close();

    clean();
    System.out.println("Test delete all with commit after each "
+ _commitInterval + " objects:");
    open();
    store();
```

```
        deleteAllWithCommit();
        close();
    }
```

The following initial setup is used:

```
DeletePerformanceBenchmark.java: initForCommitTest
private void initForCommitTest() {
    _count = 10000;
    _commitInterval = 1000;
    _depth = 3;
    _isClientServer = false;
}
```

Delete procedures are as follows:

```
DeletePerformanceBenchmark.java: deleteAll
private void deleteAll() {
    ObjectSet result = objectContainer.queryByExample(null);
    ArrayList<Long> ids = new ArrayList<Long>();
    for (Object obj: result) {
        ids.add(objectContainer.ext().getID(obj));
    }
    startTimer();
    for (long id: ids) {
        objectContainer.delete(objectContainer.ext().getByID(id));
    }
    objectContainer.commit();
    stopTimer("Deleted all objects");
}
```

```
DeletePerformanceBenchmark.java: deleteAllWithCommit
private void deleteAllWithCommit() {
    ObjectSet result = objectContainer.queryByExample(null);
    ArrayList<Long> ids = new ArrayList<Long>();
    for (Object obj: result) {
        ids.add(objectContainer.ext().getID(obj));
    }
    startTimer();
    int i = 0;
    for (long id: ids) {
        objectContainer.delete(objectContainer.ext().getByID(id));
        if (i++ > _commitInterval) {
            objectContainer.commit();
            i = 0;
        }
    }
    objectContainer.commit();
    stopTimer("Deleted all objects");
}
```

Item class used in this example has depth = 3:

```
DeletePerformanceBenchmark.java: Item
public static class Item {

    public String _name;
    public Item _child;
```

```

public Item() {
}

public Item(String name, Item child) {
    _name = name;
    _child = child;
}
}

```

The results on the test computer are:

Delete test with different commit frequency

Test delete all:

Store 30000 objects: 2432ms

Deleted all objects: 3731ms

Test delete all with commit after each 1000 objects:

Store 30000 objects: 2103ms

Deleted all objects: 8615ms

You can see that the time required for deletion increased with the amount of commits. To improve the performance commit frequency should be kept relatively low, in the same time it must ensure the integrity of the logical transactions.

Complexity Of Objects

Unlike other database operations deletion is mostly unaffected by the complexity of objects. The delete operation basically marks an object as deleted and is the same for any type of object. The main time required is to locate the object record in the database. This is demonstrated by the following example:

```

DeletePerformanceBenchmark.java: runDifferentObjectsTest
private void runDifferentObjectsTest() {
    System.out.println("Delete test with different objects");

    configure();
    init();
    System.out.println("Deleting 1 of " + _count + " objects with "
+ _depth + " levels of embedded objects:");

    clean();
    System.out.println(" - primitive object with int field");
    open();
    storeSimplest();
    deleteAny(10);
    close();

    clean();
    open();
    System.out.println(" - object with String field");
    store();
    deleteAny(10);
    close();

    clean();
    open();
    System.out.println(" - object with StringBuffer field");
}

```

```

        storeWithStringBuffer();
        deleteAny(10);
        close();

        clean();
        open();
        System.out.println(" - object with int array field");
        storeWithArray();
        deleteAny(10);
        close();

        clean();
        open();
        System.out.println(" - object with ArrayList field");
        storeWithArrayList();
        deleteAny(10);
        close();

    }
}

```

The following classes are used for the test:

```

DeletePerformanceBenchmark.java: SimplestItem
public static class SimplestItem {

    public int _id;
    public SimplestItem _child;

    public SimplestItem() {
    }

    public SimplestItem(int id, SimplestItem child) {
        _id = id;
        _child = child;
    }
}

```

```

DeletePerformanceBenchmark.java: Item
public static class Item {

    public String _name;
    public Item _child;

    public Item() {
    }

    public Item(String name, Item child) {
        _name = name;
        _child = child;
    }
}

```

```

DeletePerformanceBenchmark.java: ItemWithStringBuffer
public static class ItemWithStringBuffer {

    public StringBuffer _name;
    public ItemWithStringBuffer _child;

    public ItemWithStringBuffer() {
    }
}

```

```

        public ItemWithStringBuffer(StringBuffer name,
ItemWithStringBuffer child) {
            _name = name;
            _child = child;
        }
    }

DeletePerformanceBenchmark.java: ItemWithArray
public static class ItemWithArray {

    public int[] _id;
    public ItemWithArray _child;

    public ItemWithArray() {
    }

    public ItemWithArray(int[] id, ItemWithArray child) {
        _id = id;
        _child = child;
    }
}

```

```

DeletePerformanceBenchmark.java: ItemWithArrayList
public static class ItemWithArrayList {

    public ArrayList _ids;
    public ItemWithArrayList _child;

    public ItemWithArrayList() {
    }

    public ItemWithArrayList(ArrayList ids, ItemWithArrayList child) {
        _ids = ids;
        _child = child;
    }
}

```

The deletion procedure deletes several items from the whole amount of objects in the database:

```

DeletePerformanceBenchmark.java: deleteAny
private void deleteAny(int i) {
    long time = 0;
    int counter = 0;
    while (counter++ < i) {
        long id = selectAny();
        time += deleteById(id);
    }
    System.out.println("Deleted " + i + " objects in: " + time + " ms.");
}

```

Approximate results on the test computer are presented below:

Delete test with different objects

Deleting 1 of 10000 objects with 3 levels of embedded objects:

- *primitive object with int field*
Store 30000 objects: 1720ms
Deleted 10 objects in: 0 ms.

- *object with String field*
Store 30000 objects: 1690ms
Deleted 10 objects in: 1 ms.
- *object with StringBuffer field*
Store 30000 objects: 4424ms
Deleted 10 objects in: 1 ms.
- *object with int array field*
Store 30000 objects: 2071ms
Deleted 10 objects in: 0 ms.
- *object with ArrayList field*
Store 30000 objects: 4264ms
Deleted 10 objects in: 1 ms.

You can see that the complexity and structure of an object play little or no role in the performance.

Storage Characteristics

As any other db4o operation delete is dependent on the performance of a storage location. This can be easily compared with a database stored on a hard drive and in RAM:

```
DeletePerformanceBenchmark.java: runRamDiskTest
private void runRamDiskTest() {
    System.out.println("Delete test: RAM disk");

    configureRamDrive();
    initForRamDriveTest();
    clean();
    open();
    store();
    System.out.println("Deleting 1 object of depth " + _depth
+ " on a RAM drive:");
    deleteAll();
    close();
}

DeletePerformanceBenchmark.java: initForRamDiskTest
1

DeletePerformanceBenchmark.java: runHardDriveTest
private void runHardDriveTest() {
    System.out.println("Delete test: hard drive");

    initForHardDriveTest();
    clean();
    open();
    store();
    System.out.println("Deleting 1 object of depth " + _depth
+ " on a hard drive:");
    deleteAll();
    close();
}

DeletePerformanceBenchmark.java: initForHardDriveTest
private void initForHardDriveTest() {
    _count = 3000;
    _depth = 3;
    _filePath = "performance.db4o";
    _isClientServer = false;
}
```

```

DeletePerformanceBenchmark.java: deleteAll
private void deleteAll() {
    ObjectSet result = objectContainer.queryByExample(null);
    ArrayList<Long> ids = new ArrayList<Long>();
    for (Object obj: result) {
        ids.add(objectContainer.ext().getID(obj));
    }
    startTimer();
    for (long id: ids) {
        objectContainer.delete(objectContainer.ext().getByID(id));
    }
    objectContainer.commit();
    stopTimer("Deleted all objects");
}

```

Delete test: RAM disk

Store 90000 objects: 5973ms

Deleting 1 object of depth 3 on a RAM drive:

Deleted all objects: 5249ms

Delete test: hard drive

Store 90000 objects: 5043ms

Deleting 1 object of depth 3 on a hard drive:

Deleted all objects: 7475ms

The general rule is: the faster the drive is the better is the performance.

Client-Server

The comparison of delete performance in a local or networked database shows an obvious result: local mode is faster as it does not include network communication delays:

```

DeletePerformanceBenchmark.java: runClientServerTest
private void runClientServerTest() {
    System.out.println("Delete test: Client/Server environment");
    int objectsToDelete = 10;

    configureClientServer();

    init();
    clean();
    open();
    store();
    System.out.println("Delete " + objectsToDelete + " of " + _count
+ " objects [depth " + _depth + "] locally:");
    deleteAny(objectsToDelete);
    close();

    initForClientServer();
    clean();
    open();
    store();
    System.out.println("Delete " + objectsToDelete + " of " + _count
+ " objects [depth " + _depth + "] remotely:");
    deleteAny(objectsToDelete);
    close();
}

```

```

DeletePerformanceBenchmark.java: initForClientServer
private void initForClientServer() {
    _count = 10000;
    _depth = 3;
    _isClientServer = true;
    _host = "localhost";
}

DeletePerformanceBenchmark.java: configureClientServer
private void configureClientServer() {
    Configuration config = Db4o.configure();
    config.lockDatabaseFile(false);
    config.flushFileBuffers(false);
    config.clientServer().singleThreadedClient(true);
}

DeletePerformanceBenchmark.java: deleteAny
private void deleteAny(int i) {
    long time = 0;
    int counter = 0;
    while (counter++ < i) {
        long id = selectAny();
        time += deleteById(id);
    }
    System.out.println("Deleted " + i + " objects in: " + time + " ms.");
}

```

The approximate results on a test computer:

Delete test: Client/Server environment

Store 30000 objects: 1710ms

Delete 10 of 10000 objects [depth 3] locally:

Deleted 10 objects in: 1 ms.

Store 30000 objects: 2721ms

Delete 10 of 10000 objects [depth 3] remotely:

Deleted 10 objects in: 31 ms.

Cascade On Delete

If you are using cascadeOnDelete option and explicit deletion of one object causes cascaded delete on several others, you may expect slower execution:

```

DeletePerformanceBenchmark.java: runDeleteDepthTest
private void runDeleteDepthTest() {
    System.out.println("Delete test with objects of different depth");

    configureDepthTest();
    initShallowObject();
    System.out.println("Deleting 1000 of " + _count + " objects with "
+ _depth + " levels of embedded objects:");
    clean();
    open();
    store();
    System.out.println("Amount of objects left: " + countObjects());
    deleteAny(1000);
    System.out.println("Amount of objects left: " + countObjects());
    close();
}

```

```

        clean();
        init();
        System.out.println("Deleting 1000 of " + _count + " objects with "
+ _depth + " levels of embedded objects:");
        open();
        store();
        System.out.println("Amount of objects left: " + countObjects());
        deleteAny(1000);
        System.out.println("Amount of objects left: " + countObjects());
        close();

    }

DeletePerformanceBenchmark.java: configureDepthTest
private void configureDepthTest() {
    Configuration config = Db4o.configure();
    config.lockDatabaseFile(false);
    config.io(new MemoryIoAdapter());
    config.flushFileBuffers(false);
    config.objectClass(Item.class).cascadeonDelete(true);
}

DeletePerformanceBenchmark.java: initShallowObject
private void initShallowObject() {
    _count = 1000;
    _depth = 1;
    _isClientServer = false;

}

DeletePerformanceBenchmark.java: init
private void init() {
    _count = 1000;
    _depth = 3;
    _isClientServer = false;

}

```

Note that in the first part of the test each object deletion only includes a single object. Therefore, when we delete a 1000 objects the amount of the objects in the database is 1000 less. In the second part each object deletion will trigger deletion of all field objects (cascadeOnDelete option), therefore the amount of objects deleted from the database can be anywhere between 1000 (only lowest level objects) and 3000(all top level objects with their field objects).

The results from the test machine:

Delete test with objects of different depth

Deleting 1000 of 10000 objects with 1 levels of embedded objects:

Store 10000 objects: 528ms

Amount of objects left: 10000

Deleted 1000 objects in: 197 ms.

Amount of objects left: 9000

Deleting 1000 of 10000 objects with 3 levels of embedded objects:

Store 30000 objects: 1693ms

Amount of objects left: 39000

Deleted 1000 objects in: 340 ms.

Amount of objects left: 36978

Update Performance

Update performance is influenced by the similar factors as [Insert Performance](#). The main factors include: configuration, disk access times, complexity of objects.

The following chapters provide some simple tests showing the influence of the above-mentioned factors. The test results are provided for Toshiba Sattelite Pro A120 notebook with 1,5Gb RAM 120GB ATA drive running on Vista and may be different on a different environment.

The following Item class is used in most of the tests:

```
UpdatePerformanceBenchmark.java: Item
private void update(Object item) {
    objectContainer.store(item);
}
// end update

private void runDifferentObjectsTest() {
    System.out.println("Update test with different objects");
    int objectsToUpdate = 90;
    int updated = objectsToUpdate;

    initDifferentObjectsTest();

    clean();
    System.out.println(" - primitive object with int field");
    open(configure());
    storeSimplest();

    ObjectSet result = objectContainer.queryByExample(null);
    startTimer();
    for (int i = 0; i < objectsToUpdate; i++) {
        if (result.hasNext()) {
            SimplestItem item = (SimplestItem)result.next();
            item._id = 1;
            update(item);
        } else {
            updated = i;
            break;
        }
    }
    stopTimer("Updated " + updated + " items");
    close();

    clean();
    open(configure());
    System.out.println(" - object with String field");
    store();
    updated = objectsToUpdate;
    result = objectContainer.queryByExample(null);
    startTimer();
    for (int i = 0; i < objectsToUpdate; i++) {
        if (result.hasNext()) {
            Item item = (Item)result.next();
            item._name = "Updated";
            update(item);
        } else {
            updated = i;
            break;
        }
    }
}
```

```

}

    stopTimer("Updated " + updated + " items");
    close();

    clean();
    open(configure());
    System.out.println(" - object with StringBuffer field");
    storeWithStringBuffer();

    updated = objectsToUpdate;
    result = objectContainer.queryByExample(null);
    startTimer();
    for (int i = 0; i < objectsToUpdate; i++) {
        if (result.hasNext()) {
            ItemWithStringBuffer item = (ItemWithStringBuffer)result.next();
            item._name = new StringBuffer("Updated");
            update(item);
        } else {
            updated = i;
            break;
        }
    }
    stopTimer("Updated " + updated + " items");
    close();

    clean();
    open(configure());
    System.out.println(" - object with int array field");
    storeWithArray();
    updated = objectsToUpdate;
    result = objectContainer.queryByExample(null);
    startTimer();
    for (int i = 0; i < objectsToUpdate; i++) {
        if (result.hasNext()) {
            ItemWithArray item = (ItemWithArray)result.next();
            item._id = new int[] {1,2,3};
            update(item);
        } else {
            updated = i;
            break;
        }
    }
    stopTimer("Updated " + updated + " items");
    close();

    clean();
    open(configure());
    System.out.println(" - object with ArrayList field");
    storeWithArrayList();
    updated = objectsToUpdate;
    result = objectContainer.queryByExample(null);
    startTimer();
    for (int i = 0; i < objectsToUpdate; i++) {
        if (result.hasNext()) {
            ItemWithArrayList item = (ItemWithArrayList)result.next();
            item._ids = new ArrayList();
            update(item);
        } else {
            updated = i;
            break;
        }
    }
}

```

```

        }
    }
    stopTimer("Updated " + updated + " items");
    close();
}
// end runDifferentObjectsTest

private void runIndexTest() {
    System.out.println("Update test for objects with and without indexed fields");

    int objectsToUpdate = 100;
    init();
    System.out.println("Updating " + objectsToUpdate + " of " + _count + " objects");
    clean();
    open(configure());
    store();
    updateItems(objectsToUpdate);
    close();

    clean();
    init();
    System.out.println("Updating " + objectsToUpdate + " of " + _count + " objects with indexed field");
    open(configureIndexTest());
    store();
    updateItems(objectsToUpdate);
    close();
}
// end runIndexTest

private void init() {
    _count = 1000;
    _depth = 90;
    _isClientServer = false;

}
// end init

private void initDifferentObjectsTest() {
    _count = 1000;
    _depth = 1;
    _isClientServer = false;

}
// end initDifferentObjectsTest

private void initForClientServer() {
    _count = 1000;
    _depth = 90;
    _isClientServer = true;
    _host = "localhost";
}
// end initForClientServer

private void initForRamDriveTest() {
    _count = 30000;
    _depth = 1;
    _filePath = "r:\\performance.db4o";
}
// end initForRamDriveTest

```

```

_isClientServer = false;

}

// end initForRamDriveTest

private void initForHardDriveTest() {
    _count = 10000;
    _depth = 3;
    _filePath = "performance.db4o";
    _isClientServer = false;
}
// end initForHardDriveTest

private void initForCommitTest() {
    _count = 10000;
    _commitInterval = 1000;
    _depth = 3;
    _isClientServer = false;
}
// end initForCommitTest

private void clean() {
    new File(_filePath).delete();
}
// end clean

private Configuration configure() {
    Configuration config = Db4o.newConfiguration();
    // using MemoryIoAdapter improves the performance
    // by replacing the costly disk IO operations with
    // memory access
    config.io(new MemoryIoAdapter());
    return config;
}
// end configure

private Configuration configureTP() {
    Configuration config = Db4o.newConfiguration();
    // With Transparent Persistence enabled only modified
    // objects are written to disk. This allows to achieve
    // better performance
    config.objectClass(Item.class).cascadeOnUpdate(true);
    return config;
}
// end configureTP

private Configuration configureCascade() {
    Configuration config = Db4o.newConfiguration();
    // CascadeOnUpdate can be a performance-killer for
    // deep object hierarchies
    config.objectClass(Item.class).cascadeOnUpdate(true);
    return config;
}
// end configureCascade

private Configuration configureIndexTest() {
    Configuration config = Db4o.newConfiguration();
    config.io(new MemoryIoAdapter());
    config.objectClass(Item.class).objectField("_name").indexed(true);
    return config;
}

```

```

    }
    // end configureIndexTest

    private Configuration configureForCommitTest() {
        Configuration config = Db4o.newConfiguration();
        config.lockDatabaseFile(false);
        // the commit information is physically written
        // and in the correct order
        config.flushFileBuffers(true);
        return config;
    }
    // end configureForCommitTest

    private Configuration configureClientServer() {
        Configuration config = Db4o.newConfiguration();
        config.clientServer().singleThreadedClient(true);
        return config;
    }
    // end configureClientServer

    private Configuration configureDriveTest() {
        Configuration config = Db4o.newConfiguration();
        config.flushFileBuffers(true);
        return config;
    }
    // end configureDriveTest

    private void store() {
        startTimer();
        for (int i = 0; i < _count ;i++) {
            Item item = new Item("level" + i, null);
            for (int j = 1; j < _depth; j++) {
                item = new Item("level" + i + "/" + j, item);
            }
            objectContainer.store(item);
        }
        objectContainer.commit();
        stopTimer("Store " + totalObjects() + " objects");
    }
    // end store

    private void storeActivatableItems() {
        startTimer();
        for (int i = 0; i < _count ;i++) {
            ActivatableItem item = new ActivatableItem("level" + i, null);
            for (int j = 1; j < _depth; j++) {
                item = new ActivatableItem("level" + i + "/" + j, item);
            }
            objectContainer.store(item);
        }
        objectContainer.commit();
        stopTimer("Store " + totalObjects() + " objects");
    }
    // end storeActivatableItems

    private void storeInherited() {
        startTimer();
        for (int i = 0; i < _count ;i++) {
            ItemDerived item = new ItemDerived("level" + i, null);

```

```

        for (int j = 1; j < _depth; j++) {
            item = new ItemDerived("level" + i + "/" + j, item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store "+ totalObjects() + " objects");
}
// end storeInherited

private void storeWithStringBuffer() {
    startTimer();
    for (int i = 0; i < _count ;i++) {
        ItemWithStringBuffer item = new ItemWithStringBuffer(new StringBuffer("level" + i), null);
        for (int j = 1; j < _depth; j++) {
            item = new ItemWithStringBuffer(new StringBuffer("level" + i + "/" + j), item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store "+ totalObjects() + " objects");
}
// end storeWithStringBuffer

private void storeSimplest() {
    startTimer();
    for (int i = 0; i < _count ;i++) {
        SimplestItem item = new SimplestItem(i, null);
        for (int j = 1; j < _depth; j++) {
            item = new SimplestItem(i, item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store "+ totalObjects() + " objects");
}
// end storeSimplest

private void storeWithArray() {
    startTimer();
    int[] array = new int[] {1,2,3,4};
    for (int i = 0; i < _count ;i++) {
        int[] id = new int[] {1,2,3,4};
        ItemWithArray item = new ItemWithArray(id, null);
        for (int j = 1; j < _depth; j++) {
            int[] id1 = new int[] {1,2,3,4};
            item = new ItemWithArray(id1, item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store "+ totalObjects() + " objects");
}
// end storeWithArray

private void storeWithArrayList() {
    startTimer();
    ArrayList idList = new ArrayList();
    idList.add(1);

```

```

idList.add(2);
idList.add(3);
idList.add(4);
for (int i = 0; i < _count ;i++) {
    ArrayList ids = new ArrayList();
    ids.addAll(idList);
    ItemWithArrayList item = new ItemWithArrayList(ids, null);
    for (int j = 1; j < _depth; j++) {
        ArrayList ids1 = new ArrayList();
        ids1.addAll(idList);
        item = new ItemWithArrayList(ids1, item);
    }
    objectContainer.store(item);
}
objectContainer.commit();
stopTimer("Store "+ totalObjects() + " objects");
}
// end storeWithArrayList

private int totalObjects() {
    return _count * _depth;
}
// end totalObjects

private void open(Configuration config) {
    if(_isClientServer) {
        int port = TCP ? PORT : 0;
        String user = "db4o";
        String password = user;
        objectServer = Db4o.openServer(_filePath, port);
        objectServer.grantAccess(user, password);
        objectContainer = TCP ? Db4o.openClient(_host, port, user,
            password) : objectServer.openClient();
    } else {
        objectContainer = Db4o.openFile(config, _filePath);
    }
}
// end open

private void close() {
    objectContainer.close();
    if(_isClientServer) {
        objectServer.close();
    }
}
//end close

private void startTimer() {
    startTime = System.currentTimeMillis();
}
// end startTimer

private void stopTimer(String message) {
    long stop = System.currentTimeMillis();
    long duration = stop - startTime;
    System.out.println(message + ": " + duration + "ms");
}
// end stopTimer

```

```

public static class Item {
    public String _name;
    public Item _child;

    public Item() {
    }

    public Item(String name, Item child) {
        _name = name;
        _child = child;
    }
}

```

More Reading:

- [Configuration](#)
- [Object Structure](#)
- [Commit Frequency](#)
- [Hard Drive Speed](#)
- [Client-Server](#)
- [Indexes](#)
- [Inheritance](#)

Configuration

db4o provides a wide range of configuration options to help you meet your performance and reliability requirements. The following example shows how different configurations affect update performance:

```

UpdatePerformanceBenchmark.java: runConfigurationTest
private void runConfigurationTest() {
    System.out.println("Update test with different configurations");

    //
    clean();
    init();
    System.out.println("Update test: default configurations");
    open(Db4o.newConfiguration());
    store();
    updateItems(90);
    close();
    //

    clean();
    System.out.println("Update test: memory IO adapter");
    open(configure());
    store();
    updateItems(90);
    close();
    //
    clean();
    System.out.println("Update test: cascade on update");
    open(configureCascade());
    store();
    updateTopLevelItems(90);
    close();
}

```

```

//  

clean();  

System.out.println("Update test: Transparent Persistence");  

open(configureTP());  

storeActivatableItems();  

updateActivatableItems(90);  

close();  

}  



---


UpdatePerformanceBenchmark.java: updateItems  

private void updateItems(int count) {  

    startTimer();  

    ObjectSet result = objectContainer.queryByExample(null);  

    for (int i = 0; i < count; i++) {  

        if (result.hasNext()) {  

            Item item = (Item)result.next();  

            item._name = "Updated";  

            update(item);  

        } else {  

            count = i;  

            break;  

        }
    }
    stopTimer("Updated " + count + " items");
}  



---


UpdatePerformanceBenchmark.java: updateTopLevelItems  

private void updateTopLevelItems(int count) {  

    startTimer();  

    Query query = objectContainer.query();  

    query.constrain(Item.class);  

    query.descend("_name").constrain("level0").startsWith(true);  

    ObjectSet result = query.execute();  

    for (int i = 0; i < count; i++) {  

        if (result.hasNext()) {  

            Item item = (Item)result.next();  

            item._name = "Updated";  

            update(item);  

        } else {  

            count = i;  

            break;  

        }
    }
    stopTimer("Updated " + count + " items");
}  



---


UpdatePerformanceBenchmark.java: updateActivatableItems  

private void updateActivatableItems(int count) {  

    startTimer();  

    Query query = objectContainer.query();  

    query.constrain(ActivatableItem.class);  

    query.descend("_name").constrain("level0").startsWith(true);  

    ObjectSet result = query.execute();  

    for (int i = 0; i < count; i++) {  

        if (result.hasNext()) {  

            ActivatableItem item = (ActivatableItem)result.next();

```

```

        item.setName("Updated");
        update(item);
    } else {
        count = i;
        break;
    }
}
stopTimer("Updated " + count + " items");
}

UpdatePerformanceBenchmark.java: configure
private Configuration configure() {
    Configuration config = Db4o.newConfiguration();
    // using MemoryIoAdapter improves the performance
    // by replacing the costly disk IO operations with
    // memory access
    config.io(new MemoryIoAdapter());
    return config;
}

UpdatePerformanceBenchmark.java: configureCascade
private Configuration configureCascade() {
    Configuration config = Db4o.newConfiguration();
    // CascadeOnUpdate can be a performance-killer for
    // deep object hierarchies
    config.objectClass(Item.class).cascadeOnUpdate(true);
    return config;
}

UpdatePerformanceBenchmark.java: configureTP
private Configuration configureTP() {
    Configuration config = Db4o.newConfiguration();
    // With Transparent Persistence enabled only modified
    // objects are written to disk. This allows to achieve
    // better performance
    config.objectClass(Item.class).cascadeOnUpdate(true);
    return config;
}

UpdatePerformanceBenchmark.java: ActivatableItem
public static class ActivatableItem implements Activatable {

    private String _name;
    public ActivatableItem _child;

    transient Activator _activator;

    public void bind(Activator activator) {
        if (_activator == activator) {
            return;
        }
        if (activator != null && _activator != null) {
            throw new IllegalStateException();
        }
        _activator = activator;
    }

    public void activate(ActivationPurpose purpose) {
        if (_activator == null) return;
        _activator.activate(purpose);
    }
}
```

```

public ActivatableItem() {
}

public ActivatableItem(String name, ActivatableItem child) {
    setName(name);
    _child = child;
}

public void setName(String _name) {
    this._name = _name;
}

public String getName() {
    return _name;
}

```

The results:

Update test with different configurations

Update test: default configurations

Store 90000 objects: 7869ms

Updated 90 items: 471ms

Update test: memory IO adapter

Store 90000 objects: 6622ms

Updated 90 items: 289ms

Update test: cascade on update

Store 90000 objects: 6848ms

Updated 90 items: 1531ms

Update test: Transparent Persistence

Store 90000 objects: 6604ms

Updated 90 items: 1297ms

From the results you can see that MemoryIoAdapter allows to improve performance, CascadeOnUpdate option results in a considerable drop of performance, and Transparent Persistence makes it better again.

Object Structure

Update performance is dependent upon the structure and complexity of objects. This is demonstrated in the following test:

```

UpdatePerformanceBenchmark.java: runDifferentObjectsTest
private void runDifferentObjectsTest() {
    System.out.println("Update test with different objects");
    int objectsToUpdate = 90;
    int updated = objectsToUpdate;

    initDifferentObjectsTest();

    clean();
    System.out.println(" - primitive object with int field");
    open(configure());

```

```

storeSimplest();

ObjectSet result = objectContainer.queryByExample(null);
startTimer();
for (int i = 0; i < objectsToUpdate; i++) {
    if (result.hasNext()) {
        SimplestItem item = (SimplestItem)result.next();
        item._id = 1;
        update(item);
    } else {
        updated = i;
        break;
    }
}
stopTimer("Updated " + updated + " items");
close();

clean();
open(configure());
System.out.println(" - object with String field");
store();
updated = objectsToUpdate;
result = objectContainer.queryByExample(null);
startTimer();
for (int i = 0; i < objectsToUpdate; i++) {
if (result.hasNext()) {
    Item item = (Item)result.next();
    item._name = "Updated";
    update(item);
} else {
    updated = i;
    break;
}
}
stopTimer("Updated " + updated + " items");
close();

clean();
open(configure());
System.out.println(" - object with StringBuffer field");
storeWithStringBuffer();

updated = objectsToUpdate;
result = objectContainer.queryByExample(null);
startTimer();
for (int i = 0; i < objectsToUpdate; i++) {
if (result.hasNext()) {
    ItemWithStringBuffer item = (ItemWithStringBuffer)result.next();
    item._name = new StringBuffer("Updated");
    update(item);
} else {
    updated = i;
    break;
}
}
stopTimer("Updated " + updated + " items");
close();

clean();
open(configure());

```

```

System.out.println(" - object with int array field");
storeWithArray();
updated = objectsToUpdate;
result = objectContainer.queryByExample(null);
startTimer();
for (int i = 0; i < objectsToUpdate; i++) {
if (result.hasNext()) {
    ItemWithArray item = (ItemWithArray)result.next();
    item._id = new int[] {1,2,3};
    update(item);
} else {
    updated = i;
    break;
}
}
stopTimer("Updated " + updated + " items");
close();

clean();
open(configure());
System.out.println(" - object with ArrayList field");
storeWithArrayList();
updated = objectsToUpdate;
result = objectContainer.queryByExample(null);
startTimer();
for (int i = 0; i < objectsToUpdate; i++) {
if (result.hasNext()) {
    ItemWithArrayList item = (ItemWithArrayList)result.next();
    item._ids = new ArrayList();
    update(item);
} else {
    updated = i;
    break;
}
}
stopTimer("Updated " + updated + " items");
close();
}

```

```

UpdatePerformanceBenchmark.java: SimplestItem
public static class SimplestItem {

    public int _id;
    public SimplestItem _child;

    public SimplestItem() {
    }

    public SimplestItem(int id, SimplestItem child) {
        _id = id;
        _child = child;
    }
}

```

```

UpdatePerformanceBenchmark.java: ItemWithStringBuffer
public static class ItemWithStringBuffer {

    public StringBuffer _name;
    public ItemWithStringBuffer _child;

    public ItemWithStringBuffer() {

```

```

    }

    public ItemWithStringBuffer(StringBuffer name,
ItemWithStringBuffer child) {
        _name = name;
        _child = child;
    }
}

UpdatePerformanceBenchmark.java: ItemWithArray
public static class ItemWithArray {

    public int[] _id;
    public ItemWithArray _child;

    public ItemWithArray() {
    }

    public ItemWithArray(int[] id, ItemWithArray child) {
        _id = id;
        _child = child;
    }
}

UpdatePerformanceBenchmark.java: ItemWithArrayList
public static class ItemWithArrayList {

    public ArrayList _ids;
    public ItemWithArrayList _child;

    public ItemWithArrayList() {
    }

    public ItemWithArrayList(ArrayList ids, ItemWithArrayList child) {
        _ids = ids;
        _child = child;
    }
}

```

The results:

Update test with different objects

- primitive object with int field

Store 1000 objects: 273ms

Updated 90 items: 185ms

- object with String field

Store 1000 objects: 166ms

Updated 90 items: 158ms

- object with StringBuffer field

Store 1000 objects: 199ms

Updated 90 items: 488ms

- object with int array field

Store 1000 objects: 78ms

Updated 90 items: 134ms

- object with ArrayList field

Store 1000 objects: 191ms

Updated 90 items: 647ms

In general update of a more complex object takes more time, however the exact result depends on the TypeHandler implementation.

Commit Frequency

Commit frequency has a direct impact on db4o performance. Commit is an expensive operation due to physical disk access. However, commit is also the only way to ensure that the whole transaction is stored safely on the disk and no data loss will occur in case of unexpected system failure.

The following test shows how commit frequency influences the performance on update:

```
UpdatePerformanceBenchmark.java: runCommitTest
private void runCommitTest() {
    System.out.println("Update test with different commit frequency");

    initForCommitTest();

    clean();
    System.out.println("Test update all:");
    open(configureForCommitTest());
    store();
    updateItems(_count);
    close();

    clean();
    System.out.println("Test update all with commit after each " +
(commitInterval + " objects:");
    open(configureForCommitTest());
    store();
    updateWithCommit(_count);
    close();

}

UpdatePerformanceBenchmark.java: ConfigureForCommitTest
private Configuration configureForCommitTest() {
    Configuration config = Db4o.newConfiguration();
    config.lockDatabaseFile(false);
    // the commit information is physically written
    // and in the correct order
    config.flushFileBuffers(true);
    return config;
}

UpdatePerformanceBenchmark.java: updateItems
private void updateItems(int count)  {
    startTimer();
    ObjectSet result = objectContainer.queryByExample(null);

    for (int i = 0; i < count; i++) {
        if (result.hasNext()) {
            Item item = (Item)result.next();
            item._name = "Updated";
            update(item);
        } else  {
            count = i;
```

```

        break;
    }
}
stopTimer("Updated " + count + " items");
}

UpdatePerformanceBenchmark.java: updateWithCommit
private void updateWithCommit(int count) {
    startTimer();
    ObjectSet result = objectContainer.queryByExample(null);
    int j = 0;
    for (int i = 0; i < count; i++) {
        if (result.hasNext()) {
            Item item = (Item)result.next();
            item._name = "Updated";
            update(item);
            if (j >= _commitInterval) {
                j = 0;
                objectContainer.commit();
            } else {
                j++;
            }
        } else {
            count = i;
            break;
        }
    }
    stopTimer("Updated " + count + " items");
}

```

The results:

Update test with different commit frequency

Test update all:

Store 30000 objects: 2661ms

Updated 10000 items: 1402ms

Test update all with commit after each 1000 objects:

Store 30000 objects: 2250ms

Updated 10000 items : 2812ms

Hard Drive Speed

Update db4o operation requires disk access and therefore is very dependent on the disk speed. To emulate a different drive speed we will use a RAMDISK utility, which creates an alternative storage media in the memory.

```

UpdatePerformanceBenchmark.java: runHardDriveTest
private void runHardDriveTest() {
    System.out.println("Update test: hard drive");
    int objectsToUpdate = 90;

    initForHardDriveTest();
    clean();
    open(configureDriveTest());
    store();
    System.out.println("Updating " + objectsToUpdate + " "
objects on a hard drive:");
    updateItems(objectsToUpdate);
}

```

```

        close();
    }

UpdatePerformanceBenchmark.java: initForHardDriveTest
private void initForHardDriveTest() {
    _count = 10000;
    _depth = 3;
    _filePath = "performance.db4o";
    _isClientServer = false;
}

UpdatePerformanceBenchmark.java: runRamDiskTest
private void runRamDiskTest() {
    System.out.println("Update test: RAM disk");
    int objectsToUpdate = 90;
    initForRamDriveTest();
    clean();
    open(configureDriveTest());
    store();
    System.out.println("Updating " + objectsToUpdate +
" objects on a RAM drive:");
    updateItems(objectsToUpdate);
    close();
}

UpdatePerformanceBenchmark.java: initForRamDriveTest
private void initForRamDriveTest() {
    _count = 30000;
    _depth = 1;
    _filePath = "r:\\performance.db4o";
    _isClientServer = false;

}

UpdatePerformanceBenchmark.java: configureDriveTest
private Configuration configureDriveTest() {
    Configuration config = Db4o.newConfiguration();
    config.flushFileBuffers(true);
    return config;
}

```

The results:

Update test: hard drive
Store 30000 objects: 2884ms
Updating 90 objects on a hard drive:
Updated 90 items: 250ms
Update test: RAM disk
Store 30000 objects: 1910ms
Updating 90 objects on a RAM drive:
Updated 90 items: 105ms

The test shows that the faster media (RAMDISK) shows better performance on update.

Client-Server

Client/server performance is a bit slower than local performance. This is illustrated with the following test:

```

UpdatePerformanceBenchmark.java: runClientServerTest
private void runClientServerTest() {

```

```

System.out.println("Update test: Client/Server environment");
int objectsToUpdate = 30;

init();
clean();
open(configureClientServer());
store();
System.out.println("Update " + objectsToUpdate + " of " + _count
+ " objects locally:");
updateItems(objectsToUpdate);
close();

initForClientServer();
clean();
open(configureClientServer());
store();
System.out.println("Update " + objectsToUpdate + " of " + _count
+ " objects remotely:");
updateItems(objectsToUpdate);
close();
}

UpdatePerformanceBenchmark.java: initForClientServer
private void initForClientServer() {
    _count = 1000;
    _depth = 90;
    _isClientServer = true;
    _host = "localhost";
}

UpdatePerformanceBenchmark.java: init
private void init() {
    _count = 1000;
    _depth = 90;
    _isClientServer = false;
}

UpdatePerformanceBenchmark.java: configureClientServer
private Configuration configureClientServer() {
    Configuration config = Db4o.newConfiguration();
    config.clientServer().singleThreadedClient(true);
    return config;
}

```

The results:

Update test: Client/Server environment

Store 90000 objects: 7935ms

Update 30 of 1000 objects locally:

Updated 30 items: 404ms

Store 90000 objects: 11421ms

Update 30 of 1000 objects remotely:

Updated 30 items: 436ms

You can see that the performance drop is quite insignificant in this case, however it can be much worse on slow or unreliable networks.

Indexes

Updating indexed fields always takes longer as the index should be updated as well. This is shown in the following test:

```
UpdatePerformanceBenchmark.java: runIndexTest
private void runIndexTest() {
    System.out.println("Update test for objects with and without indexed fields");

    int objectsToUpdate = 100;
    init();
    System.out.println("Updating " + objectsToUpdate + " of " +
_count + " objects");
    clean();
    open(configure());
    store();
    updateItems(objectsToUpdate);
    close();

    clean();
    init();
    System.out.println("Updating " + objectsToUpdate + " of " +
_count + " objects with indexed field");
    open(configureIndexTest());
    store();
    updateItems(objectsToUpdate);
    close();
}

UpdatePerformanceBenchmark.java: init
private void init() {
    _count = 1000;
    _depth = 90;
    _isClientServer = false;

}

UpdatePerformanceBenchmark.java: configureIndexTest
private Configuration configureIndexTest() {
    Configuration config = Db4o.newConfiguration();
    config.io(new MemoryIoAdapter());
    config.objectClass(Item.class).objectField("_name").indexed(true);
    return config;
}
```

The results:

Update test for objects with and without indexed fields

Updating 100 of 1000 objects

Store 90000 objects: 7466ms

Updated 100 items: 295ms

Updating 100 of 1000 objects with indexed field

Store 90000 objects: 6839ms

Updated 100 items: 441ms

Inheritance

Inherited objects take longer to store as their parent indexes need to be updated too.

```

UpdatePerformanceBenchmark.java: runInheritanceTest
private void runInheritanceTest() {
    System.out.println("Update test: objects with deep inheritance");

    int objectsToUpdate = 30;
    init();
    clean();
    open(configure());
    store();
    System.out.println("Updating " + objectsToUpdate + " objects");
    updateItems(objectsToUpdate);
    close();

    clean();
    open(configure());
    storeInherited();
    System.out.println("Updating " + objectsToUpdate + " inherited objects");
    updateItems(objectsToUpdate);
    close();

}

UpdatePerformanceBenchmark.java: configure
private Configuration configure() {
    Configuration config = Db4o.newConfiguration();
    // using MemoryIoAdapter improves the performance
    // by replacing the costly disk IO operations with
    // memory access
    config.io(new MemoryIoAdapter());
    return config;
}

UpdatePerformanceBenchmark.java: init
private void init() {
    _count = 1000;
    _depth = 90;
    _isClientServer = false;

}

UpdatePerformanceBenchmark.java: ItemDerived
public static class ItemDerived extends Item {

    public ItemDerived(String name, ItemDerived child) {
        super(name, child);
    }
}

```

The results:

Update test: objects with deep inheritance
Store 90000 objects: 6312ms
Updating 30 objects
Updated 30 items: 272ms
Store 90000 objects: 5657ms
Updating 30 inherited objects
Updated 30 items: 436ms

Query Performance

Query Performance is one of the most important characteristics of a database system. In the same time it is probably the one that can vary the most. In general query performance can be dependent on the following list of factors and their combinations:

- hardware resources (free RAM, processor speed, disk access)
- database model & complexity of objects
- query structure
- query configuration
- amount of objects in the database
- connection speed
- indexes
- query optimization
- etc

The following set of tests shows some performance dependencies. A simple Item object with a string field is used in most of the tests:

```
QueryPerformanceBenchmark.java: Item
//  
    System.out.println("Native Query:");
    startTimer();
    List<Item> result = objectContainer.query(new Predicate<Item>()  {
        public boolean match(Item item)  {
            return item._name.equals("level1/1");
        }
    });
    item = result.queryByExample(0);
    stopTimer("Select 1 object NQ1: " + item._name);
    close();

//  
open(configureUnoptimizedNQ());
System.out.println("Native Query Unoptimized:");
startTimer();
result = objectContainer.query(new Predicate<Item>()  {
    public boolean match(Item item)  {
        return item._name.equals("level1/1");
    }
});
item = result.queryByExample(0);
stopTimer("Select 1 object NQ: " + item._name);

close();
}

// end runDifferentQueriesTest

private void runRamDiskTest()  {
```

¹Native Query

```

initForHardDriveTest();
clean();
System.out.println("Storing " + _count + " objects of depth " + _depth
    + " on a hard drive:");
open(configureRamDrive());
store();
close();
open(configureRamDrive());
startTimer();
Query query = objectContainer.query();
query.constrain(Item.class);
query.descend("_name").constrain("level1/1");
Item item = (Item) query.execute().next();
stopTimer("Select 1 object: " + item._name);
close();

initForRamDriveTest();
clean();
System.out.println("Storing " + _count + " objects of depth " + _depth
    + " on a RAM disk:");
open(configureRamDrive());
store();
close();
open(configureRamDrive());
startTimer();
query = objectContainer.query();
query.constrain(Item.class);
query.descend("_name").constrain("level1/1");
item = (Item) query.execute().next();
stopTimer("Select 1 object: " + item._name);
close();
}

// end runRamDiskTest

private void runClientServerTest() {

    initForClientServer();
    clean();
    System.out.println("Storing " + _count + " objects of depth " + _depth
        + " remotely:");
    open(configureClientServer());
    store();
    close();
    open(configureClientServer());
    startTimer();
    Query query = objectContainer.query();
    query.constrain(Item.class);
    query.descend("_name").constrain("level1/1");
    Item item = (Item) query.execute().next();
    stopTimer("Select 1 object: " + item._name);
    close();

    init();
    clean();
    System.out.println("Storing " + _count + " objects of depth " + _depth
        + " locally:");
    open(configureClientServer());
    store();
    close();
}

```

```

open(configureClientServer());
startTimer();
query = objectContainer.query();
query.constrain(Item.class);
query.descend("_name").constrain("level1/1");
item = (Item) query.execute().next();
stopTimer("Select 1 object: " + item._name);
close();
}

// end runClientServerTest

private void runInheritanceTest() {
    init();
    clean();
    System.out.println("Storing " + _count + " objects of depth " + _depth);
    open(configure());
    store();
    close();
    open(configure());
    startTimer();
    Query query = objectContainer.query();
    query.constrain(Item.class);
    query.descend("_name").constrain("level1/1");
    Item item = (Item) query.execute().next();
    stopTimer("Select 1 object: " + item._name);
    close();

    clean();
    System.out.println("Storing " + _count + " inherited objects of depth "
        + _depth);
    open(configure());
    storeInherited();
    close();
    open(configure());
    startTimer();
    // Query for item, inheriting objects should be included in the result
    query = objectContainer.query();
    query.constrain(Item.class);
    query.descend("_name").constrain("level1/1");
    item = (Item) query.execute().next();
    stopTimer("Select 1 object: " + item._name);
    close();
}

// end runInheritanceTest

private void runDifferentObjectsTest() {

    init();
    System.out.println("Storing " + _count + " objects with " + _depth
        + " levels of embedded objects:");

    clean();
    System.out.println();
    System.out.println("- primitive object with int field");
    open(configure());
    storeSimplest();
    close();
    open(configure());
}

```

```

startTimer();
Query query = objectContainer.query();
query.constrain(SimplestItem.class);
query.descend("_id").constrain(1);
List result = query.execute();
SimplestItem simplestItem = (SimplestItem) result.queryByExample(0);
stopTimer("Querying SimplestItem: " + simplestItem._id);
close();

open(configure());
System.out.println();
System.out.println(" - object with String field");
store();
close();
open(configure());
startTimer();
query = objectContainer.query();
query.constrain(Item.class);
query.descend("_name").constrain("level1/2");
result = query.execute();
Item item = (Item) result.queryByExample(0);
stopTimer("Querying object with String field: " + item._name);
close();

clean();
open(configure());
System.out.println();
System.out.println(" - object with StringBuffer field");
storeWithStringBuffer();
close();
open(configure());
startTimer();
query = objectContainer.query();
query.constrain(ItemWithStringBuffer.class);
query.descend("_name").constrain(new StringBuffer("level1/2"));
result = query.execute();
ItemWithStringBuffer itemWithSB = (ItemWithStringBuffer) result.queryByExample(0);
stopTimer("Querying object with StringBuffer field: "
        + itemWithSB._name);
close();

clean();
open(configure());
System.out.println();
System.out.println(" - object with int array field");
storeWithArray();
close();
open(configure());
startTimer();
query = objectContainer.query();
query.constrain(ItemWithArray.class);
Query idQuery = query.descend("_id");
idQuery.constrain(new Integer(1));
idQuery.constrain(new Integer(2));
idQuery.constrain(new Integer(3));
idQuery.constrain(new Integer(4));
result = query.execute();

ItemWithArray itemWithArray = (ItemWithArray) result.queryByExample(0);
stopTimer("Querying object with Array field: [" + itemWithArray._id[0]

```

```

        + ", " + +itemWithArrayList._id[1] + ", " + +itemWithArrayList._id[2]
        + ", " + +itemWithArrayList._id[0] + "]");
close();

clean();
open(configure());
System.out.println();
System.out.println(" - object with ArrayList field");
storeWithArrayList();
close();
open(configure());
startTimer();
query = objectContainer.query();
query.constrain(ItemWithArrayList.class);
query.descend("_ids").constrain(1).contains();
result = query.execute();
ItemWithArrayList itemWithArrayList = (ItemWithArrayList) result.queryByExample(0);
stopTimer("Querying object with ArrayList field: "
        + itemWithArrayList._ids.toString());
close();

}

// end runDifferentObjectsTest

private void runIndexTest() {

init();
System.out.println("Storing " + _count + " objects with " + _depth
        + " levels of embedded objects:");

clean();
System.out.println(" - no index");
open(configure());
store();
close();
open(configure());
startTimer();
Query query = objectContainer.query();
query.constrain(Item.class);
query.descend("_name").constrain("level1/2");
List result = query.execute();
Item item = (Item) result.queryByExample(0);
stopTimer("Querying object with String field: " + item._name);
close();

System.out.println(" - index on String field");
// open to create index
open(configureIndex());
close();
open(configure());
startTimer();
query = objectContainer.query();
query.constrain(Item.class);
query.descend("_name").constrain("level1/2");
result = query.execute();
item = (Item) result.queryByExample(0);
stopTimer("Querying object with String field: " + item._name);
close();
}

```

```
}

// end runIndexTest

private void init() {
    _filePath = "performance.db4o";
    // amount of objects
    _count = 10000;
    // depth of objects
    _depth = 3;
    _isClientServer = false;
}

// end init

private void initLargeDb() {
    _filePath = "performance.db4o";
    _count = 100000;
    _depth = 3;
    _isClientServer = false;
}

// end initLargeDb

private void initForClientServer() {
    _filePath = "performance.db4o";
    _isClientServer = true;
    _host = "localhost";
}

// end initForClientServer

private void initForRamDriveTest() {
    _count = 30000;
    _depth = 3;
    _filePath = "r:\\performance.db4o";
    _isClientServer = false;
}

// end initForRamDriveTest

private void initForHardDriveTest() {
    _count = 30000;
    _depth = 3;
    _filePath = "performance.db4o";
    _isClientServer = false;
}

// end initForHardDriveTest

private void clean() {
    new File(_filePath).delete();
}
```

```

// end clean

private Configuration configure() {
    Configuration config = Db4o.newConfiguration();
    return config;
}

// end configure

private Configuration configureUnoptimizedNQ() {
    Configuration config = Db4o.newConfiguration();
    config.optimizeNativeQueries(false);
    return config;
}
// end configureUnoptimizedNQ

private Configuration configureIndex() {
    Configuration config = Db4o.newConfiguration();
    config.objectClass(Item.class).objectField("_name").indexed(true);
    return config;
}

// end configureIndex

private Configuration configureClientServer() {
    Configuration config = Db4o.newConfiguration();
    config.queries().evaluationMode(QueryEvaluationMode.IMMEDIATE);
    config.clientServer().singleThreadedClient(true);
    return config;
}

// end configureClientServer

private Configuration configureRamDrive() {
    Configuration config = Db4o.newConfiguration();
    config.flushFileBuffers(true);
    return config;
}

// end configureRamDrive

private void store() {
    startTimer();
    for (int i = 0; i < _count; i++) {
        Item item = new Item("level" + i, null);
        for (int j = 1; j < _depth; j++) {
            item = new Item("level" + i + "/" + j, item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store " + totalObjects() + " objects");
}
// end store

private void storeInherited() {
    startTimer();
    for (int i = 0; i < _count; i++) {
        ItemDerived item = new ItemDerived("level" + i, null);
        for (int j = 1; j < _depth; j++) {

```

```

        item = new ItemDerived("level" + i + "/" + j, item);
    }
    objectContainer.store(item);
}
objectContainer.commit();
stopTimer("Store " + totalObjects() + " objects");
}

// end storeInherited

private void storeWithStringBuffer() {
    startTimer();
    for (int i = 0; i < _count; i++) {
        ItemWithStringBuffer item = new ItemWithStringBuffer(
            new StringBuffer("level" + i), null);
        for (int j = 1; j < _depth; j++) {
            item = new ItemWithStringBuffer(new StringBuffer("level" + i
                + "/" + j), item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store " + totalObjects() + " objects");
}

// end storeWithStringBuffer

private void storeSimplest() {
    startTimer();
    for (int i = 0; i < _count; i++) {
        SimplestItem item = new SimplestItem(i, null);
        for (int j = 1; j < _depth; j++) {
            item = new SimplestItem(i, item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store " + totalObjects() + " objects");
}

// end storeSimplest

private void storeWithArray() {
    startTimer();
    int[] array = new int[] { 1, 2, 3, 4 };
    for (int i = 0; i < _count; i++) {
        int[] id = new int[] { 1, 2, 3, 4 };
        ItemWithArray item = new ItemWithArray(id, null);
        for (int j = 1; j < _depth; j++) {
            int[] id1 = new int[] { 1, 2, 3, 4 };
            item = new ItemWithArray(id1, item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store " + totalObjects() + " objects");
}

// end storeWithArray

```

```

private void storeWithArrayList() {
    startTimer();
    ArrayList idList = new ArrayList();
    idList.add(1);
    idList.add(2);
    idList.add(3);
    idList.add(4);
    for (int i = 0; i < _count; i++) {
        ArrayList ids = new ArrayList();
        ids.addAll(idList);
        ItemWithArrayList item = new ItemWithArrayList(ids, null);
        for (int j = 1; j < _depth; j++) {
            ArrayList ids1 = new ArrayList();
            ids1.addAll(idList);
            item = new ItemWithArrayList(ids1, item);
        }
        objectContainer.store(item);
    }
    objectContainer.commit();
    stopTimer("Store " + totalObjects() + " objects");
}

// end storeWithArrayList

private int totalObjects() {
    return _count * _depth;
}

// end totalObjects

private void open(Configuration configure) {
    if (_isClientServer) {
        int port = TCP ? PORT : 0;
        String user = "db4o";
        String password = user;
        objectServer = Db4o.openServer(configure, _filePath, port);
        objectServer.grantAccess(user, password);
        objectContainer = TCP ? Db4o
            .openClient(configure, _host, port, user, password) : objectServer
            .openClient(configure);
    } else {
        objectContainer = Db4o.openFile(configure, _filePath);
    }
}

// end open

private void close() {
    objectContainer.close();
    if (_isClientServer) {
        objectServer.close();
    }
}

// end close

private void startTimer() {
    startTime = System.currentTimeMillis();
}

```

```

// end startTimer

private void stopTimer(String message) {
    long stop = System.currentTimeMillis();
    long duration = stop - startTime;
    System.out.println(message + ": " + duration + "ms");
}

// end stopTimer

public static class Item {

    public String _name;
    public Item _child;

    public Item() {

    }

    public Item(String name, Item child) {
        _name = name;
        _child = child;
    }
}

```

Different Query Types

db4o provides different query syntaxes: Query By Example, SODA, Native Queries and LINQ (for .NET 3.5 and higher). Under the hood all these syntaxes are converted to SODA. The conversion can be very straightforward (as in case of **QBE**¹), or pretty sophisticated (some Native queries). The fact that conversion takes place and can be more or less complex affects the performance of queries expressed with different syntax. Another factor affecting the performance can be using unoptimized Native Queries: this can happen if the query is too complex to analyze or when optimization is disabled through configuration. Optimization is also applicable to LINQ queries, i.e. some of LINQ queries are currently too complex to analyze and optimize. In cases when optimization does not happen, the query is run against all instances of an object in the database, which is quite slow and consumes a lot of RAM.

```

QueryPerformanceBenchmark.java: runDifferentQueriesTest
private void runDifferentQueriesTest() {
    init();

    clean();
    System.out.println("Storing objects as a bulk:");
    open(configure());
    store();
    close();

    open(configure());
    //
    System.out.println("Query by example:");
    startTimer();
    Item item = (Item) objectContainer.queryByExample(
        new Item("level1/1", null)).next();
    stopTimer("Select 1 object QBE: " + item._name);
}

```

¹Query By Example

```

// 
System.out.println("SODA:");
startTimer();
Query query = objectContainer.query();
query.constrain(Item.class);
query.descend("_name").constrain("level1/1");
item = (Item) query.execute().next();
stopTimer("Select 1 object SODA: " + item._name);

// 
System.out.println("Native Query:");
startTimer();
List<Item> result = objectContainer.query(new Predicate<Item>() {
    public boolean match(Item item) {
        return item._name.equals("level1/1");
    }
});
item = result.queryByExample(0);
stopTimer("Select 1 object NQ: " + item._name);
close();

// 
open(configureUnoptimizedNQ());
System.out.println("Native Query Unoptimized:");
startTimer();
result = objectContainer.query(new Predicate<Item>() {
    public boolean match(Item item) {
        return item._name.equals("level1/1");
    }
});
item = result.queryByExample(0);
stopTimer("Select 1 object NQ: " + item._name);

close();
}

QueryPerformanceBenchmark.java: init
private void init() {
    _filePath = "performance.db4o";
    // amount of objects
    _count = 10000;
    // depth of objects
    _depth = 3;
    _isClientServer = false;
}

QueryPerformanceBenchmark.java: configure
private Configuration configure() {
    Configuration config = Db4o.newConfiguration();
    return config;
}

```

The following results were obtained on a test machine:

Storing objects as a bulk:

Store 30000 objects: 5337 ms

Query by example:

Select 1 object QBE: level1/1: 1021 ms

SODA:

Select 1 object SODA: level1/1: 809 ms

LINQ:

Select 1 object LINQ: level1/1: 915 ms

Native Query:

Select 1 object NQ¹: level1/1: 1604 ms

Native Query Unoptimized:

Select 1 object NQ: level1/1: 5008 ms

You can see that SODA query shows the best performance. The other query types are less performant due to conversion, however they can be easier to support and refactor. The worst performance is shown in the case of unoptimized Native Query: in this case all the objects from the database were instantiated and tested against the constraint.

Query Structure

More complex queries take longer to execute, as they include more constraints and can impose some additional operations as sorting, aggregate, negation etc.

```
QueryPerformanceBenchmark.java: runQueryStructureTest
private void runQueryStructureTest() {
    init();

    clean();
    System.out.println("Storing objects as a bulk:");
    open(configure());
    store();
    close();

    //
    open(configure());
    System.out.println("Simple SODA query:");
    startTimer();
    Query query = objectContainer.query();
    query.constrain(Item.class);
    query.descend("_name").constraint("level1/1");
    Item item = (Item) query.execute().next();
    stopTimer("Select 1 object SODA: " + item._name);
    close();

    //
    open(configure());
    System.out.println("Sorted SODA query:");
    startTimer();
    query = objectContainer.query();
    query.constrain(Item.class);
    query.descend("_name").orderDescending();
    item = (Item) query.execute().next();
    stopTimer("Select 1 object SODA: " + item._name);
    close();

    //
    open(configure());
```

¹Native Query

```

        System.out.println("SODA query with joins:");
        startTimer();
        query = objectContainer.query();
        query.constrain(Item.class);
        Constraint con = query.constrain("level2/1");
        query.descend("_name").orderDescending().constrain("level1/1").or(con);
        List result = query.execute();
        stopTimer("Selected " + result.size() + " objects SODA");
        close();
    }

QueryPerformanceBenchmark.java: init
private void init() {
    _filePath = "performance.db4o";
    // amount of objects
    _count = 10000;
    // depth of objects
    _depth = 3;
    _isClientServer = false;
}

QueryPerformanceBenchmark.java: configure
private Configuration configure() {
    Configuration config = Db4o.newConfiguration();
    return config;
}

```

Results from the test machine:

Storing objects as a bulk:

Store 30000 objects: 3049ms

Simple SODA query:

Select 1 object SODA: level1/1: 440ms

Sorted SODA query:

Select 1 object SODA: level9999/2: 1509ms

SODA query with joins:

Selected 1 objects SODA: 1735ms

From the test results you can see that sorting makes the query slower, and adding additional constraints slows things down even more.

Database Size

Query performance can degrade with the amount of objects of the queried type:

```

QueryPerformanceBenchmark.java: runQueryAmountOfObjectsTest
private void runQueryAmountOfObjectsTest() {
    init();
    clean();
    System.out.println("Storing " + _count + " of objects of depth " + _depth);
    open(configure());
    store();
    close();

    //
    open(configure());
}

```

```

startTimer();
Query query = objectContainer.query();
query.constrain(Item.class);
query.descend("_name").constrain("level1/1");
Item item = (Item) query.execute().next();
stopTimer("Select 1 object SODA: " + item._name);

System.out.println(
"Add some objects of another type and check the query time again:");
storeWithArray();
close();
//
open(configure());
startTimer();
query = objectContainer.query();
query.constrain(Item.class);
query.descend("_name").constrain("level1/1");
item = (Item) query.execute().next();
stopTimer("Select 1 object SODA: " + item._name);
close();

// Add many objects of the same type
initLargeDb();
clean();
System.out.println("Storing " + _count + " of objects of depth " + _depth);
open(configure());
store();
close();

//
open(configure());
startTimer();
query = objectContainer.query();
query.constrain(Item.class);
query.descend("_name").constrain("level1/1");
item = (Item) query.execute().next();
stopTimer("Select 1 object SODA: " + item._name);
close();
}

QueryPerformanceBenchmark.java: init
private void init() {
    _filePath = "performance.db4o";
    // amount of objects
    _count = 10000;
    // depth of objects
    _depth = 3;
    _isClientServer = false;
}

QueryPerformanceBenchmark.java: initLargeDb
private void initLargeDb() {
    _filePath = "performance.db4o";
    _count = 100000;
    _depth = 3;
    _isClientServer = false;
}

```

```

QueryPerformanceBenchmark.java: configure
private Configuration configure() {
    Configuration config = Db4o.newConfiguration();
    return config;
}

```

However, the general size of the database, i.e. amount of other type of objects in the database should not have any impact on the query performance.

Results from the test machine:

```

Storing 10000 of objects of depth 3
Store 30000 objects: 3305ms
Select 1 object SODA: level1/1: 464ms
Storing 100000 of objects of depth 3
Store 300000 objects: 21338ms
Select 1 object SODA: level1/1: 5316ms

```

Complexity Of Objects

More complex objects are usually more difficult not only to store, but also to query and instantiate. The following test demonstrates how query performance depends on class structure, complexity and depth:

```

QueryPerformanceBenchmark.java: runDifferentObjectsTest
private void runDifferentObjectsTest() {

    init();
    System.out.println("Storing " + _count + " objects with " + _depth
        + " levels of embedded objects:");

    clean();
    System.out.println();
    System.out.println(" - primitive object with int field");
    open(configure());
    storeSimplest();
    close();
    open(configure());
    startTimer();
    Query query = objectContainer.query();
    query.constrain(SimplestItem.class);
    query.descend("_id").constrain(1);
    List result = query.execute();
    SimplestItem simplestItem = (SimplestItem) result.queryByExample(0);
    stopTimer("Querying SimplestItem: " + simplestItem._id);
    close();

    open(configure());
    System.out.println();
    System.out.println(" - object with String field");
    store();
    close();
    open(configure());
    startTimer();
    query = objectContainer.query();
    query.constrain(Item.class);
    query.descend("_name").constrain("level1/2");
    result = query.execute();
    Item item = (Item) result.queryByExample(0);
}

```

```

stopTimer("Querying object with String field: " + item._name);
close();

clean();
open(configure());
System.out.println();
System.out.println(" - object with StringBuffer field");
storeWithStringBuffer();
close();
open(configure());
startTimer();
query = objectContainer.query();
query.constrain(ItemWithStringBuffer.class);
query.descend("_name").constrain(new StringBuffer("level1/2"));
result = query.execute();
ItemWithStringBuffer itemWithSB = (ItemWithStringBuffer) result.queryByExample(0);
stopTimer("Querying object with StringBuffer field: "
        + itemWithSB._name);
close();

clean();
open(configure());
System.out.println();
System.out.println(" - object with int array field");
storeWithArray();
close();
open(configure());
startTimer();
query = objectContainer.query();
query.constrain(ItemWithArray.class);
Query idQuery = query.descend("_id");
idQuery.constrain(new Integer(1));
idQuery.constrain(new Integer(2));
idQuery.constrain(new Integer(3));
idQuery.constrain(new Integer(4));
result = query.execute();

ItemWithArray itemWithArray = (ItemWithArray) result.queryByExample(0);
stopTimer("Querying object with Array field: [" + itemWithArray._id[0]
        + ", " + +itemWithArray._id[1] + ", " + +itemWithArray._id[2]
        + ", " + +itemWithArray._id[0] + "]");
close();

clean();
open(configure());
System.out.println();
System.out.println(" - object with ArrayList field");
storeWithArrayList();
close();
open(configure());
startTimer();
query = objectContainer.query();
query.constrain(ItemWithArrayList.class);
query.descend("_ids").constrain(1).contains();
result = query.execute();
ItemWithArrayList itemWithArrayList = (ItemWithArrayList) result.queryByExample(0);
stopTimer("Querying object with ArrayList field: "
        + itemWithArrayList._ids.toString());
close();

```

```
}

QueryPerformanceBenchmark.java: init
private void init() {
    _filePath = "performance.db4o";
    // amount of objects
    _count = 10000;
    // depth of objects
    _depth = 3;
    _isClientServer = false;

}

QueryPerformanceBenchmark.java: configure
private Configuration configure() {
    Configuration config = Db4o.newConfiguration();
    return config;
}

QueryPerformanceBenchmark.java: SimplestItem
public static class SimplestItem {

    public int _id;
    public SimplestItem _child;

    public SimplestItem() {
    }

    public SimplestItem(int id, SimplestItem child) {
        _id = id;
        _child = child;
    }
}

QueryPerformanceBenchmark.java: ItemWithStringBuffer
public static class ItemWithStringBuffer {

    public StringBuffer _name;
    public ItemWithStringBuffer _child;

    public ItemWithStringBuffer() {
    }

    public ItemWithStringBuffer(StringBuffer name,
        ItemWithStringBuffer child) {
        _name = name;
        _child = child;
    }
}

QueryPerformanceBenchmark.java: ItemWithArray
public static class ItemWithArray {

    public int[] _id;
    public ItemWithArray _child;

    public ItemWithArray() {
    }

    public ItemWithArray(int[] id, ItemWithArray child) {
        _id = id;
        _child = child;
    }
}
```

```

        }
    }

QueryPerformanceBenchmark.java: ItemWithArrayList
public static class ItemWithArrayList {
    public ArrayList _ids;
    public ItemWithArrayList _child;

    public ItemWithArrayList() {
    }

    public ItemWithArrayList(ArrayList ids, ItemWithArrayList child) {
        _ids = ids;
        _child = child;
    }
}

```

Results from the test machine:

- *primitive object with int field*

Store 30000 objects: 1878ms

Querying SimplestItem: 1: 425ms

- *object with String field*

Store 30000 objects: 2599ms

Querying object with String field: level1/2: 436ms

- *object with StringBuffer field*

Store 30000 objects: 5658ms

Querying object with StringBuffer field: level1/2: 3489ms

- *object with int array field*

Store 30000 objects: 2487ms

Querying object with Array field: [1, 2, 3, 1]: 1777ms

- *object with ArrayList field*

Store 30000 objects: 5302ms

Querying object with ArrayList field: [1, 2, 3, 4]: 3796ms

Inherited Objects

You can use a class base to query for inherited objects. This makes a query path a bit more complex and may result in a small performance degrade.

```

QueryPerformanceBenchmark.java: runInheritanceTest
private void runInheritanceTest() {
    init();
    clean();
    System.out.println("Storing " + _count + " objects of depth " + _depth);
    open(configure());
    store();
    close();
    open(configure());
    startTimer();
    Query query = objectContainer.query();
    query.constrain(Item.class);
}

```

```
query.descend("_name").constrain("level1/1");
Item item = (Item) query.execute().next();
stopTimer("Select 1 object: " + item._name);
close();

clean();
System.out.println("Storing " + _count + " inherited objects of depth "
+ _depth);
open(configure());
storeInherited();
close();
open(configure());
startTimer();
// Query for item, inheriting objects should be included in the result
query = objectContainer.query();
query.constrain(Item.class);
query.descend("_name").constrain("level1/1");
item = (Item) query.execute().next();
stopTimer("Select 1 object: " + item._name);
close();
}
```

```
QueryPerformanceBenchmark.java: ItemDerived
public static class ItemDerived extends Item {

    public ItemDerived(String name, ItemDerived child) {
        super(name, child);
    }
}
```

```
QueryPerformanceBenchmark.java: init
private void init() {
    _filePath = "performance.db4o";
    // amount of objects
    _count = 10000;
    // depth of objects
    _depth = 3;
    _isClientServer = false;

}
```

```
QueryPerformanceBenchmark.java: configure
private Configuration configure() {
    Configuration config = Db4o.newConfiguration();
    return config;
}
```

Results from the test machine:

Storing 10000 objects of depth 3

Store 30000 objects: 2236ms

Select 1 object: level1/1: 457ms

Storing 10000 inherited objects of depth 3

Store 30000 objects: 2790ms

Select 1 object: level1/1: 595ms

Hardware Resources

Effective querying requires enough operating memory and quick hard drive access. Hard drive access time is important as the object will be read from the physical location into the operating memory. However, hard drive speed is not so critical for querying as it is for [inserting](#).

The following test uses a RAM drive to compare test results with the hard drive:

```
QueryPerformanceBenchmark.java: runRamDiskTest
private void runRamDiskTest()  {

    initForHardDriveTest();
    clean();
    System.out.println("Storing " + _count + " objects of depth " + _depth
        + " on a hard drive:");
    open(configureRamDrive());
    store();
    close();
    open(configureRamDrive());
    startTimer();
    Query query = objectContainer.query();
    query.constrain(Item.class);
    query.descend("_name").constrain("level1/1");
    Item item = (Item) query.execute().next();
    stopTimer("Select 1 object: " + item._name);
    close();

    initForRamDriveTest();
    clean();
    System.out.println("Storing " + _count + " objects of depth " + _depth
        + " on a RAM disk:");
    open(configureRamDrive());
    store();
    close();
    open(configureRamDrive());
    startTimer();
    query = objectContainer.query();
    query.constrain(Item.class);
    query.descend("_name").constrain("level1/1");
    item = (Item) query.execute().next();
    stopTimer("Select 1 object: " + item._name);
    close();
}

QueryPerformanceBenchmark.java: initForHardDriveTest
private void initForHardDriveTest()  {
    _count = 30000;
    _depth = 3;
    _filePath = "performance.db4o";
    _isClientServer = false;

}

QueryPerformanceBenchmark.java: configureRamDrive
private Configuration configureRamDrive()  {
    Configuration config = Db4o.newConfiguration();
    config.flushFileBuffers(true);
    return config;
}
```

Test results:

Storing 30000 objects of depth 3 on a hard drive:

Store 90000 objects: 6019ms

Select 1 object: level1/1: 1515ms

Storing 30000 objects of depth 3 on a RAM disk:

Store 90000 objects: 5264ms

Select 1 object: level1/1: 1518ms

You can see that the difference in query performance is negligible.

Client-Server

In client-server use the connection speed can play an important role in the query performance.

```
QueryPerformanceBenchmark.java: runClientServerTest
private void runClientServerTest()  {

    initForClientServer();
    clean();
    System.out.println("Storing " + _count + " objects of depth " + _depth
        + " remotely:");
    open(configureClientServer());
    store();
    close();
    open(configureClientServer());
    startTimer();
    Query query = objectContainer.query();
    query.constrain(Item.class);
    query.descend("_name").constrain("level1/1");
    Item item = (Item) query.execute().next();
    stopTimer("Select 1 object: " + item._name);
    close();

    init();
    clean();
    System.out.println("Storing " + _count + " objects of depth " + _depth
        + " locally:");
    open(configureClientServer());
    store();
    close();
    open(configureClientServer());
    startTimer();
    query = objectContainer.query();
    query.constrain(Item.class);
    query.descend("_name").constrain("level1/1");
    item = (Item) query.execute().next();
    stopTimer("Select 1 object: " + item._name);
    close();
}

QueryPerformanceBenchmark.java: initForClientServer
private void initForClientServer()  {
    _filePath = "performance.db4o";
    _isClientServer = true;
    _host = "localhost";
}

QueryPerformanceBenchmark.java: configureClientServer
private Configuration configureClientServer()  {
    Configuration config = Db4o.newConfiguration();
    config.queries().evaluationMode(QueryEvaluationMode.IMMEDIATE);
```

```

        config.clientServer().singleThreadedClient(true);
        return config;
    }
}

```

Results from the test machine:

Storing 30000 objects of depth 3 remotely:

Store 90000 objects: 10725ms

Select 1 object: level1/1: 1763ms

Storing 10000 objects of depth 3 locally:

Store 30000 objects: 2904ms

Select 1 object: level1/1: 630ms

In order to improve the performance use [Lazy or Snapshot evaluation modes](#).

Indexing

Using indexes is always a good idea to improve query performance. The following test illustrates index performance impact:

```

QueryPerformanceBenchmark.java: runIndexTest
private void runIndexTest() {

    init();
    System.out.println("Storing " + _count + " objects with " + _depth
        + " levels of embedded objects:");

    clean();
    System.out.println(" - no index");
    open(configure());
    store();
    close();
    open(configure());
    startTimer();
    Query query = objectContainer.query();
    query.constrain(Item.class);
    query.descend("_name").constrain("level1/2");
    List result = query.execute();
    Item item = (Item) result.queryByExample(0);
    stopTimer("Querying object with String field: " + item._name);
    close();

    System.out.println(" - index on String field");
    // open to create index
    open(configureIndex());
    close();
    open(configure());
    startTimer();
    query = objectContainer.query();
    query.constrain(Item.class);
    query.descend("_name").constrain("level1/2");
    result = query.execute();
    item = (Item) result.queryByExample(0);
    stopTimer("Querying object with String field: " + item._name);
    close();
}
}

```

```

QueryPerformanceBenchmark.java: initForHardDriveTest
private void initForHardDriveTest() {
    _count = 30000;
    _depth = 3;
    _filePath = "performance.db4o";
    _isClientServer = false;

}

QueryPerformanceBenchmark.java: configureRamDrive
private Configuration configureRamDrive() {
    Configuration config = Db4o.newConfiguration();
    config.flushFileBuffers(true);
    return config;
}

```

Results from the test machine:

Storing 10000 objects with 3 levels of embedded objects:

- no index

Store 30000 objects: 2228ms

Querying object with String field: level1/2: 461ms

- index on String field

Querying object with String field: level1/2: 460ms

IO Benchmark Tools

I/O access times play a crucial role in the overall performance of a database. To make their measurements easy and user-friendly we introduce two new tools to

1. measure the actual I/O performance of a system as seen by db4o
2. simulate the behaviour of a slower system on a faster one

All the code presented in this article can be found in the db4otools project / com.db4o.bench package.

SVN: <https://source.db4o.com/db4o/trunk/db4otools/>

Of course, you will also get it in your distribution. The code can be compiled with JDK 1.3 and higher.

More Reading:

- [First Steps](#)
- [Using Your Application To Generate The IO Pattern](#)
- [Simulating Slow IO On A Fast Machine](#)
- [IO Log File Statistics](#)

First Steps

The main class of the benchmark is com.db4o.bench.IoBenchmark. Let's have a look at its run method to see what it does.

```

private void run(IoBenchmarkArgumentParser argumentParser) throws IOException {
    runTargetApplication(argumentParser.objectCount());
    prepareDbFile(argumentParser.objectCount());
    runBenchmark(argumentParser.objectCount());
}

```

As you can see from this code, the benchmark consists of 3 stages:

1. Run a target application and log its I/O access pattern
2. Replay the recorded I/O operations once to prepare a database file. This step is necessary to ensure that during the grouped replay in the next step, none of the accesses will go beyond the currently existing file.
3. Replay the recorded I/O operations a second time. Operations are grouped by command type (read, write, seek, sync), and the total time executing all operations of a specific command type is measured. Grouping is necessary to avoid micro-benchmarking effects and to get time values above timer resolution.

We divide the numbers collected in stage 3 by the respective number of operations and we calculate the average time a particular command takes on the given system. But enough of the theory for the moment, let's see how you can run the benchmark. For this purpose there is the pair of an Ant script and a properties file:

- IoBenchmark.xml: The Ant script
- IoBenchmark.properties: Holding configurations for the Ant script

Both files are located in the root of db4otools. To be able to run the benchmark from the Ant script, you have to put a db4o JAR file in the lib folder of the db4otools project. Insert the name of the JAR in the db4o.jar property in the property file, e.g.

```
db4o.jar=db4o-7.1.27.9109-java5.jar
```

and you are ready to go! To give it a first try, you can run the run.benchmark.small target of the Ant script, which is also the default target. You should get output similar to this:

```
===== Running db4o IoBenchmark =====
Running target application ...
Preparing DB file ...
Running benchmark ...

-----
db4o IoBenchmark results with 1000 items
Statistics written to db4o-IoBenchmark-results-1000.log
----- Results for READ > operations executed: 143
```

As the output indicates, the results of this benchmark run will also be written to a file called db4o-IoBenchmark-results-1000.log. You can find this file in the db4otools directory. The ns (nanosecond) values are our benchmark standard for the respective operation. Smaller numbers are better. Note: It may be possible, that you get some zero values for time elapsed, and therefore infinity for operations per ms. This can occur if your machine is fast enough to execute all operations under 1ms. To overcome this you can run the run.benchmark.medium target which operates with more objects and takes longer to complete.

Using Your Application To Generate The IO Pattern

When you execute IoBenchmark, it uses a simple CRUD (create, read, update, delete) application as the target application. This application is located in the com.db4o.bench.crud package. If you want to use your own application for generating the I/O access patterns, here's what you have to do:

- Use a LoggingIoAdapter, delegating to your default IoAdapter:

```
RandomAccessFileAdapter rafAdapter =
new RandomAccessFileAdapter();
IoAdapter ioAdapter =
new LoggingIoAdapter(rafAdapter, "filename.log");
Configuration config = Db4o.cloneConfiguration();
config.io(ioAdapter);
You'll also find this code in
```

```
com.db4o.bench.crud.CrudApplication#prepare().
```

- Change IoBenchmark to call your application by modifying the runTargetApplication() method. You also have to exchange the calls to CrudApplication.logFileName(itemCount) in prepareDbFile() and runBenchmark with the file name of the

log containing the I/O access pattern of your application. Using the code from above, this log file will be called "filename.log".

If you want to generate your log by interacting with your application, rather than having IoBenchmark calling it, do as follows:

- Use a LoggingIoAdapter in your application
- Interact with your application to create the log
- Remove the stage 1 from IoBenchmark and make it start in stage 2 with your log.

If you are using your own application to generate the I/O log file, check out the [IO Log File Statistics](#) section further down.

Simulating Slow IO On A Fast Machine

The code for this section is located in the com.db4o.bench.delaying package. To run delaying, the `System.nanoTime()` is needed. This method was introduced with Java 5. If you only have older versions installed, get the latest here: <http://java.sun.com/javase/downloads/> You also need a java5 db4o JAR file, otherwise you'll see a `NotImplementedException` when the benchmark tries to access `nanoTime()`. Think of the following scenario: You develop software with db4o for a target system, that has much slower I/O than your developer system (e.g. an embedded device). Wouldn't it sometimes be nice getting a feel for the expected speed your application will work with on the target system without having to deploy to it? In particular, if you want to profile your system with a profiler like [JProbe](#), simulating the expected slow I/O on a device will help you identifying the bottlenecks in your application. This is where the results of IoBenchmark and a DelayingIoAdapter enter the arena. If you run IoBenchmark on both the embedded device and your developer machine you get two results files. Copy the file from the slower device to the db4otools folder on the faster machine and set both filenames in IoBenchmark.properties:

```
results.file.1=db4o-IoBenchmark-results-30000_faster.log  
results.file.2=db4o-IoBenchmark-results-30000_slower.log
```

It's not necessary that `results.file.1` holds the faster log, any order will work. You are now set to run the benchmark in delayed mode. The expected result of such a run is, that the results of a delayed run on the faster machine should be close to those on the slow device. To do a delayed run execute one of the `run.delayed.benchmark.*` targets of the Ant script. At the beginning of the output - prior to the benchmark results - you'll notice additional information about the delaying:

```
=====      Running db4o IoBenchmark      =====  
=====          Delaying:          =====  
> machine1 (db4o-IoBenchmark-results-30000_faster.log) is faster!  
> Required delays:  
> [delays in nanoseconds] read: 8195 | write: 10669 | seek: 10098 | sync: 215121  
> Adjusting delay timer to match required delays...  
> Adjusted delays:  
> [delays in nanoseconds] read: 4934 | write: 7387 | seek: 6849 | sync: 202203  
Running target application ...      Preparing DB file ...      Running benchmark ...  
[...]
```

Let's have a look at what exactly is going on when setting up delaying. First there is a check for the validity of the two result files for delaying. To pass this check, one of the two supplied benchmark results file must contain the better values for all the 4 operations. This constraint exists because it's not possible to speed things up, only slowing them down. Once this check is passed, the delays are calculated by simply subtracting the numbers found in the result files. The resulting numbers tell us, how long each I/O operation should be delayed on the faster machine to get the same behaviour as on the slower one. The problem is now that just simply waiting for the calculated amount of time will make us wait for too long. This is due to additional setup time for each wait (method calls) and the "at least" semantics of the wait method itself. To cope with this limitation there is a delay adjustment logic. It tries to find the actual delay to wait for such that the overall waiting time, including the setup method calls, matches the desired delay time. However, there's a catch to this adjustment logic: On each machine there's a minimum delay that can be

achieved with waiting, and this delay is not equal to zero (e.g. 400ns)! If the performance of the two machines is too close together, it is possible that when trying to adjust a delay, the outcome is below the minimum delay achievable. In this case you'll see output like this:

```
>> Smallest achievable delay: 400
>> Required delay setting: 260
>> Using delay(0) to wait as short as possible.
>> Results will not be accurate.
```

To find out which delay actually was too small, and hence which results won't be accurate, take a look at the adjusted delays:

```
> Adjusted delays:
> [delays in nanoseconds] read: 0 | write: 7387 | seek: 6849 | sync: 202203
```

Here the read delay was too small and therefore the results for read are expected to be slower than targeted. Once the delays are adjusted, they can be fed to the DelayingIoAdapter (as done in IoBenchmark#delayingIoAdapter):

```
IoAdapter rafFactory = new RandomAccessFileAdapter();
IoAdapter delFactory = new
DelayingIoAdapter(rafFactory, _delays);
IoAdapter io = delFactory.open(dbFileName, false, 0, false);
```

If you now configure db4o with the IoAdapter io from above, each I/O operation will be delayed by the respective delay stored in _delays! The above IoAdapter setup is also exactly what you need in your own application to simulate the slower I/O of your target device on your faster machine.

IO Log File Statistics

To get statistically meaningful results from the benchmark it is necessary that the I/O log file contains enough operations of each type. To get an overview on how well your I/O log file represents each operation, you can use the class LogStatistics in com.db4o.bench.logging.statistics. Given the file name of an I/O log file LogStatistics will produce an HTML file that contains a table with statistics:

	Count	%	Bytes	%
Reads	664'631	30.64	52'214'278	70.83
Writes	360'001	16.6	21'508'576	29.17
Seeks	1'024'632	47.23		
Syncs	120'005	5.53		
Total	2'169'269		73'722'854	

Average byte count per read: 78 Average byte count per write: 59

This is the output of LogStatistics when run with the file generated by CrudApplication with 30k objects, which is the default setting for the Ant target run.benchmark.medium. Typically sync operations are much rarer than seek operations. If you look at the source of CrudApplication you'll see that extra commit calls when deleting objects. These were inserted to to get a higher sync count in the I/O log. It's possible that you also have to "tune" your application in a similar way to get good statistics.

Selective Persistence

Sometimes your persistent classes may have fields, which are useless or even undesirable to store. References to classes, which objects are constructed at runtime, can be an example.

How to avoid saving these fields to db4o?

More Reading:

- [Transient Fields In Java](#)
- [Transient Classes](#)
- [Storing Transient Fields](#)

Transient Fields In Java

You can use the *transient* keyword to indicate that a field is not part of the persistent state of an object:

```
Test.java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */

package com.db4odoc.selectivepersistence;

public class Test {
    transient String transientField;

    String persistentField;

    public Test(String transientField, String persistentField) {
        this.transientField = transientField;
        this.persistentField = persistentField;
    }

    public String toString() {
        return "Test: persistent: " + persistentField
            + ", transient: " + transientField;
    }
}
```

The following example demonstrates the effect of transient keyword on db4o:

```
MarkTransientExample.java: saveObjects
private static void saveObjects(Configuration configuration) {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4o.openFile(configuration, DB4O_FILE_NAME);
    try
    {
        Test test = new Test("Transient string","Persistent string");
        container.store(test);
    }
    finally
    {
        container.close();
    }
}

MarkTransientExample.java: retrieveObjects
private static void retrieveObjects()
{
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try
    {
        ObjectSet result = container.query(Test.class);
        listResult(result);
    }
    finally
    {
        container.close();
    }
}
```

```
    }  
}
```

Transient Classes

Some of the classes are not supposed to be persistent. Of course you can avoid saving their instances in your code and mark all their occurrences in another classes as transient ([Java](#)/[.NET](#)). But that needs some attention and additional coding. You can achieve the same result in an easier way using TransientClass interface:

Java:

```
com.db4o.types.TransientClass
```

TransientClass is a marker interface, which guarantees that the classes implementing it will never be added to the class metadata. In fact they are just skipped silently by db4o persistence mechanism.

An example of the TransientClass implementation is db4o object container (we do not need to save a database into itself).

Let's look how it works on an example. We will create a simplest class implementing TransientClass interface:

```
NotStorable.java  
/* */ Copyright (c) 2004 - 2007 Versant Inc. http://www.db4o.com */  
  
package com.db4odoc.selectivepersistence;  
  
import com.db4o.types.TransientClass;  
  
public class NotStorable implements TransientClass {  
  
    public String toString() {  
        return "NotStorable class";  
    }  
}
```

NotStorable class will be used as a field in two test objects: [Test1](#) and [Test2](#).

In our example we will use the default configuration and save Test1 and Test2 objects just as usual:

```
TransientClassExample.java: saveObjects  
private static void saveObjects() {  
    new File(DB4O_FILE_NAME).delete();  
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);  
    try {  
        // Save Test1 object with a NotStorable class field  
        Test1 test1 = new Test1("Test1", new NotStorable());  
        container.store(test1);  
        // Save Test2 object with a NotStorable class field  
        Test2 test2 = new Test2("Test2", new NotStorable(), test1);  
        container.store(test2);  
    } finally {  
        container.close();  
    }  
}
```

Now let's try to retrieve the saved objects:

```

TransientClassExample.java: retrieveObjects
private static void retrieveObjects() {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        // retrieve the results and check if the NotStorable
        // instances were saved
        ObjectSet result = container.queryByExample(null);
        listResult(result);
    } finally {
        container.close();
    }
}

```

If you will run the example code you will see that all the instances of NotStorable class are set to null.

Test1

```

public class Test1 {
    private String name;
    private NotStorable transientClass;
    public Test1(String name, NotStorable transientClass) {
        this.name = name;
        this.transientClass = transientClass;
    }
    public String toString() {
        return name + "/" + transientClass;
    }
}

```

Test2

```

Test2.java
public class Test2 {
    private Test1 test1;
    private String name;
    private NotStorable transientClass;
    public Test2(String name, NotStorable transientClass,
                Test1 test1) {
        this.name = name;
        this.transientClass = transientClass;
        this.test1 = test1;
    }
    public String toString() {
        return name + "/" + transientClass + "; test1: " + test1;
    }
}

```

Performance Hints

The following is an overview over possible tuning switches that can be set when working with db4o. Users that do not care about performance may like to read this chapter also because it provides a side glance at db4o features with *Alternate Strategies* and some insight on how db4o works.

More Reading:

- [Play with the different prefetching options on the client configuration.](#)
- [Enable Field Indexes](#)

- [Discarding Free Space](#)
- [Calling constructors](#)
- [Defragment](#)
- [Turning Off Weak References](#)
- [No callbacks](#)
- [No schema changes](#)
- [No lock file thread](#)
- [No test instances](#)
- [Increasing The Maximum Database File Size](#)
- [B-Tree tuning](#)
- [Inheritance hierarchies](#)
- [Persistent and transient fields](#)
- [Activation strategies](#)
- [Automatic Shutdown](#)
- Chose an appropriate [string encoding](#).
- [No Class Index](#)
- [Change the Storage mechanism used by db4o](#)
- [Commit Strategies](#)
- [Database Size](#)
- [Change the underlying Storage](#)

Discarding Free Space

Java: `configuration.file().freespace().discardSmallerThan(byteCount)`

Configures the minimum size of free space slots in the database file that are to be reused.

2 extremes for byteCount value:

- `Integer.MAX_VALUE` - discard all free slots for the best possible startup time. The downside: database files will necessarily grow faster
- 0 - default setting, all freespace is reused. The downside: increased memory consumption and performance loss for maintenance of freespace lists in RAM

Advantage

Allows fine-tuning of performance/size relation for your environment.

Effect

When objects are updated or deleted, the space previously occupied in the database file is marked as "free", so it can be reused. db4o maintains two lists in RAM, sorted by address and by size. Adjacent entries are merged. After a large number of updates or deletes have been executed, the lists can become large, causing RAM consumption and performance loss for maintenance. With this method you can specify an upper bound for the byte slot size to discard.

Alternate Strategies

Regular defragment will also keep the number of free space slots small. See:

Java: `com.db4o.defragment.Defragment`

If defragment can be frequently run, it will also reclaim lost space and decrease the database file to the minimum size. Therefore `#discardSmallerThan(maxValue)` may be a good tuning mechanism for setups with frequent defragment runs.

Defragment

```
Java: Defragment.defrag("sample.db4o")
```

Advantage

It is recommended to run Defragment frequently to reduce the database file size and to remove unused fields and freespace slots.

Effect

db4o does not discard fields from the database file that are no longer being used. Within the database file quite a lot of space is used for transactional processing. Objects are always written to a new slot when they are modified. Deleted objects continue to occupy 8 bytes until the next Defragment run. Defragment cleans all this up by writing all objects to a completely new database file. The resulting file will be smaller and faster.

Alternate Strategies

Instead of deleting objects it can be an option to mark objects as deleted with a "deleted" boolean field and to clean them out (by not copying them to the new database file) during the Defragment run. Two advantages:

1. Deleted objects can be restored.
2. In case there are multiple references to a deleted object, none of them would point to null.

Activation strategies

Java:

```
configuration.common().activationDepth(activationDepth);
```

Advantage

Db4o default activation depth is 5. This setting gives you control over activation depth level depending on your application requirements.

Effect

The two extremes:

- `activationDepth = maximum integer value` - will pop the whole object graph into the memory on every retrieved object. Can be a reasonable solution for shallow objects' design. No need to bother about manual activation;
- `activationDepth = 0` - will reduce memory consumption to the lowest level though leaving all the activation logic for your code.

Alternate strategies

If your object is not fully activated due to the default configuration settings you can activate it manually:

```
Java: ObjectContainer#activate(object,depth)
```

or use specific object settings:

```
Java:
```

```
configuration.  
objectClass("yourClass").minimumActivationDepth(minimumDepth);  
configuration.common().objectClass("yourClass").  
maximumActivationDepth(maximumDepth);  
configuration.common().objectClass("yourClass").  
cascadeOnActivate(bool);  
configuration.common().objectClass("yourClass").  
objectField("field").cascadeOnActivate(bool);
```

For more information on activation strategies see [Activation chapter](#).

No Class Index

You can disable the class-index: See "Disable Class Index" on page 109

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();  
configuration.common().objectClass(Person.class).indexed(false);
```

ObjectConfigurationExamples.java: Disable class index

Advantage

Allows to improve the performance to delete and create objects of a class.

Effect

db4o maintains an index for each class to be able to deliver all instances of a class in a query. In some cases class index is not necessary:

- the application always works with subclasses or superclasses;
- there are convenient field indexes that will always find instances of a class;
- the application always works with IDs.

`Indexed(false)` setting will save resources on maintaining the class index on create and delete of the class objects.

Alternate Strategies

Object creation performance can be improved using [configuration.callConstructors\(true\)](#) setting.

Commit Strategies

Java:

```
objectContainer.commit();
```

Objects created or instantiated within one db4o transaction are written to a temporary transaction area in the database file and are only durable after the transaction is committed.

Transactions are committed implicitly when the object container is closed.

Java:

```
objectContainer.close();
```

Advantage

Committing a transaction makes sure that all the changes are effectively written to a storage location. Commit uses a special sequence of actions, which ensures ACID transactions. The following operations are done during commit:

- flushing modified class indexes
- flushing changes of in-memory field indexes to file-based indexes

- writing all intended pointer changes as a "pre-log" to the file
- writing all pointer changes
- reorganizing the free-space system
- deleting the "pre log"

Effect

Commit is a costly operation as it includes disk writes and flushes of the operating system disk cache. 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 done get permanently stored to the database.
- If 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 and slowing down the operations due to memory flushes to disk. The exact amount of inserts that can be done safely and effectively within one transaction should be calculated for the concrete system and will depend on available system resources and 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.

Increasing The Maximum Database File Size

Java:

```
EmbeddedConfiguration           configuration=           Db4oEmbedded.newConfiguration();
configuration.file().blockSize(newBlockSize);
DefragmentConfig      defragConfig      =           new      DefragmentConfig("database.db4o");
defragConfig.db4oConfig(configuration);
Defragment.defrag(defragConfig);
```

Advantage

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

Effect

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

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

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

Alternate Strategies

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

Database Size

If you are concerned about the size of your database file, it is important to understand what contributes to it and what are the strategies to keep it down.

Object Overhead

When you create a new db4o database file - it contains only the header and has a fixed size. As soon as you start storing the information the file will grow. The size overhead per object depends on the type-handler implementation.

In general the object consists of internal ID and value types, i.e. integers, arrays, enums etc. Overhead per object type is ID, which is integer. The overhead for value type is an integer value showing which value type is it, i.e. int or string etc. For variable length value types, there is a long value to store the length. If object contains another complex object - the id of another object is referenced in the top-level object. If you decide to use UUIDs and version number for your objects, you will get an additional overhead:

UUID = 35 bytes (signature part) + 8 bytes (long part) version number = 8 bytes.

Additional overhead per object will appear from using indexes and will depend on the amount of indexes fields and indexes value types.

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 were 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.

Inheritance hierarchies

Do not create inheritance hierarchies, if you don't need them.

Advantage

Avoiding inheritance hierarchies will help you to get better performance as only actual classes will be kept in the class index and in the database.

Effect

Every class in the hierarchy requires db4o to maintain a class index. It is also true for abstract classes and interfaces since db4o has to be able to run a query against them.

Alternate strategies

Class hierarchies and interfaces may be valuable for your application design. You can also use interface/superclass to query for implementations/subclasses.

Persistent and transient fields

Do not create fields that you don't need for persistence

Advantage

Storing only needed information will help to keep your database footprint as small as possible.

Effect

If your persistent class contains fields that do not need to be stored you should mark them as transient to prevent them from being stored:

```
Java: public class NotStorable { private transient int length; . . . }
```

You can use [Callbacks](#) or [Translators](#) to set transient fields on retrieval.

Also avoid storing classes having only transient information - their indexes' maintenance will produce unnecessary performance overhead.

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 282

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.

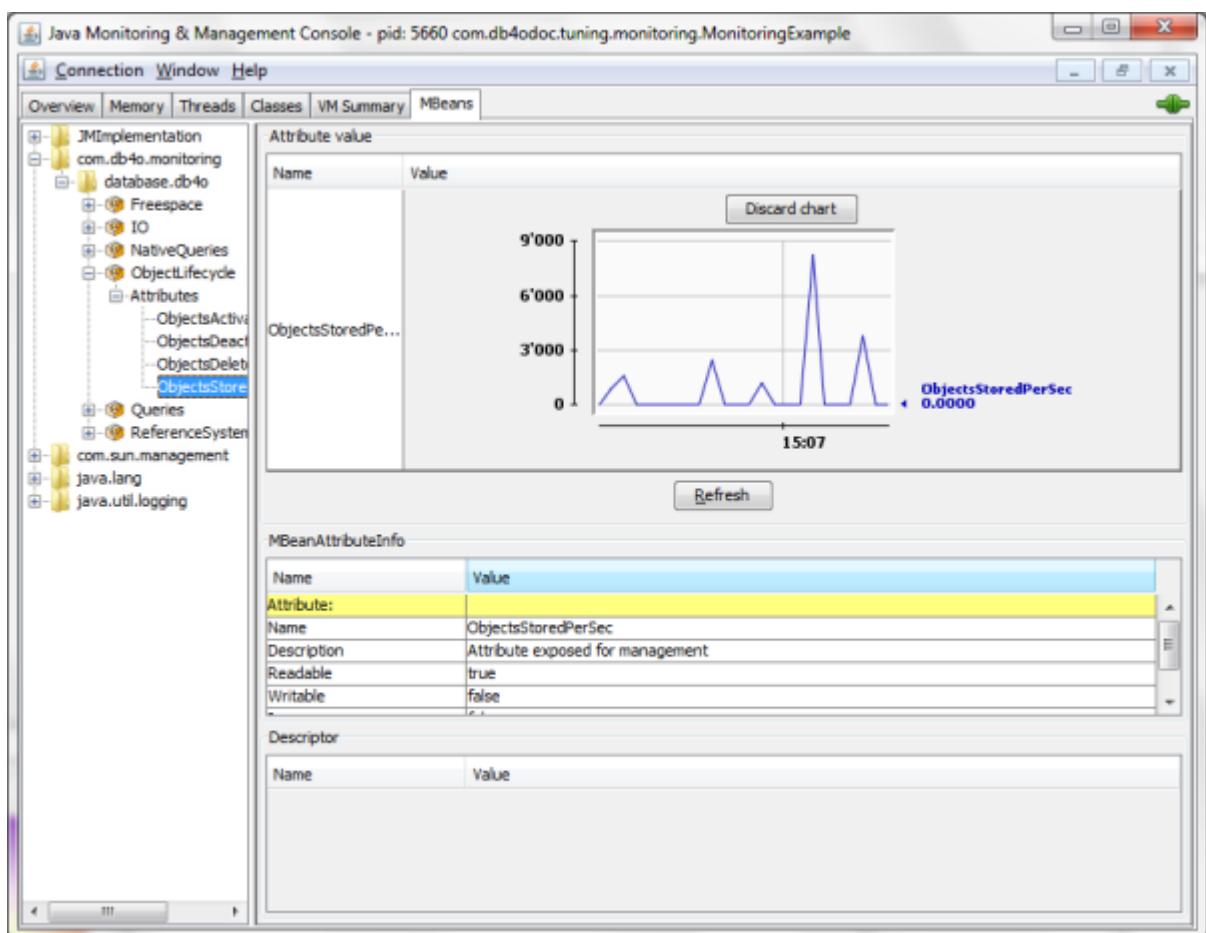
Monitor With JConsole

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 your query-performance. For example by adding additional indexes. [See "Indexing" on page 75](#)

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 75](#)

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 75](#)

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 spent 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 issued per second.

SyncsPerSecond: Tells you how many sync-operations are done per second. The sync-operation is a part of the commit-process. It 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 needs to wait on the physical disk to finish all write operations

WritesPerSecond: Tells you how many 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 high 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.

Debugging db4o

Debugging is, in general, a cumbersome and tiring task. And it tends to be harder when various sub-systems are tightly coupled, like db4o library and your application. How can db4o help you with the process?

More Reading:

- [Debug Messaging System](#)
- [Using DTrace](#)
- [Reading Db4o File](#)

Debug Messaging System

You can turn on the [db4o debug messages](#), which show more information about what db4o is doing.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().messageLevel(4);
```

CommonConfigurationExamples.java: Change the message-level

These messages show you in detail what db4o is doing and processing. You can redirect the message [output to any output stream](#). By default the console output stream is used.

```
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

Using DTrace

If you are interested in internal db4o debugging you can make use of com.db4o.DTrace class. DTrace is basically a logging agent, which gets called for db4o core events, which can provide information valuable for debugging. The list of currently available events can be found in DTrace class, they are represented by DTrace type static variables, for example:

```
public static DTrace ADD_TO_CLASS_INDEX;
public static DTrace BEGIN_TOP_LEVEL_CALL;
```

DTrace information is not enabled by default, as it can make the system really slow. In order to enable DTrace make the following changes in DTrace class and rebuild the core:

- set public static final boolean enabled = true
- modify the `configure` method to tell DTrace that you are interested in all ranges and IDs
 - `addRangeWithLength(0, Integer.MAX_VALUE);`
- make sure that no `turnAllOffExceptFor` method is called in the `configure()` method

With this DTrace setup you will basically see everything that is happening logged to the console in detail. However, this information can be excessive and difficult to handle. That is why DTrace provides different configurations, allowing to limit the range of information you are interested in.

1. `turnAllOffExceptFor(DTrace[] these)`

This method allows you to pass an array of DTrace events, which you want to see in the console. For example:

```
turnAllOffExceptFor(new DTrace[] { ADD_TO_CLASS_INDEX , BEGIN_TOP_LEVEL_CALL })
```

2. `addRange(long)`

```
addRangeWithEnd(long start, long end)
addRangeWithLength(long start, long length)
```

These methods allow to specify a range of addresses in a database file that you are interested in. `addRange` methods are especially useful for debugging a database file structure. Note, that in db4o internal object ID corresponds to the object's address in the file.

3. `trackEventsWithoutRange()`

These method will allow all events with no range specified to log their information.

The format of the output message is the following:

: [event number] : [start address] : [start address] :[information]

event number - sequential event number

start address - start of the event address range (optional)

end address - end of the event address range (optional)

information - informational message from the event

If DTrace log messages are not enough for you to track the problem, you can use DTrace in debug mode. Use `breakOnEvent(1long)` method to specify on which address DTrace must break and put a breakpoint inside `breakPoint()` method. As it was mentioned before DTrace events are already created in the most important execution points of db4o core. However, if you need more events, feel free to add them, encapsulating the calls with `if (DTrace.enabled)` to make sure that your code is removed from distributions by the compiler.

Reading Db4o File

For debugging, learning and teaching purposes, the db4o file format can be modified to be (nearly ?) human readable.

To do this, simply compile the sources with `Deploy.debug` set to true, run an application that creates a db4o database file and look at the file with any editor.

With the `Deploy.debug` setting all pointers in the database file will be readable with their physical address as a readable number.

All other slots will be identifiable by a single character at the beginning. An index that explains the character constants can be found in `com.db4o.internal.Const4`.

To understand the format best, you may want to look at the [File Header](#) structure and at the `#readThis()` methods of classes derived from `PersistentBase`, like `ClassMetadataRepository` for instance.

This functionality proved to be very useful when db4o was originally written. By marking freespace with XXXXes a bug in the format could be spotted immediately by visual inspection of a database file.

To navigate through a database file in your favourite editor, it will work best if you write a macro for this editor that allows you to mark and select a number in the database file and to hit a button in the editor to jump to the corresponding offset in the database file (number of characters from the beginning).

Such macro for Microsoft Word is presented below:

```
OffsetNavigator.Vb
Sub SearchOffset()
    Dim pos As Integer
    pos = Val(Selection.Text)
    If pos = 0 Then
        MsgBox ("The selection is not a number")
    Else
        ActiveDocument.Content.Characters(pos).Select
    End If
End Sub
```

To make use of it, open Visual Basic Macro editor within your Word environment, create a new Macro in the Normal template and paste the code above. In order to make its usage easy assign a key sequence to call the macro command:

- open Tools/Customize/Commands/Keyboard;
- select "Macros" as a Category and the newly-created macro name in the Commands list;
- press a new key sequence for the command and press "Assign".

Now you can navigate through the human-readable db4o file using the selected key sequence.

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 existing 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

Utility Methods

In this chapter we will review utility methods provided by extended object container interface. Use API documentation for more information on ExtObjectContainer methods.

More Reading:

- [PeekPersisted](#)
- [IsActive](#)
- [IsStored](#)
- [Descend](#)

PeekPersisted

Db4o loads each object into reference cache only once in the session, thus ensuring that independently of the way of retrieving, you will always get a reference to the same object. This concept certainly makes things clearer, but in some cases you will want to operate on the copy of an object.

Typical usecases can be:

- comparing object's changes in a running transaction with the original object in a database;
- safely changing an object without making changes to the database;
- modifying an object in several threads independently, writing the changes to the database after conflict resolution.

Db4o helps you with these tasks providing the following method:

Java: ExtObjectContainer.peekPersisted(object, depth, committed)

This method creates a copy of a database object in memory instantiating its members up to depth parameter value. The object has no connection to the database.

Committed parameter defines whether committed or set values are to be returned. Let's see how you can use it.

We will use 2 threads measuring temperature independently in different parts of the car: somewhere in the cabin (getCabinTemperature) and on the conditioner unit (getConditionerTemperature). After some period of time the average measured value will be written to the database.

```
PeekPersistedExample.java: measureCarTemperature
private static void measureCarTemperature() {
    setObjects();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        ObjectSet result = container.query(Car.class);
        if (result.size() > 0) {
            Car car = (Car) result.queryByExample(0);
            Car car1 = (Car) container.ext().peekPersisted(car,
                5, true);
            Change1 ch1 = new Change1();
            ch1.init(car1);
            Car car2 = (Car) container.ext().peekPersisted(car,
                5, true);
            Change2 ch2 = new Change2();
            ch2.init(car2);
            try {
                Thread.sleep(300);
            } catch (InterruptedException e) {
            }
            // We can work on the database object at the same time
            car.setModel("BMW M3Coupe");
            container.store(car);
            ch1.stop();
            ch2.stop();
            System.out.println("car1 saved to the database: "
                + container.ext().isStored(car1));
            System.out.println("car2 saved to the database: "
                + container.ext().isStored(car1));
            int temperature = (int) ((car1.getTemperature() + car2
                .getTemperature()) / 2);
            car.setTemperature(temperature);
            container.store(car);
        }
    } finally {
        container.close();
    }
    checkCar();
}
```

peekPersisted method gives you an easy way to work with database objects' clones. Remember that these clones are totally disconnected from the database. If you will try to save such object you will get a new object in the database.

IsActive

ExtObjectContainer.isActive method provides you with means to define if the object is active.

```

UtilityExample.java: checkActive
private static void checkActive() {
    storeSensorPanel();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        container.ext().configure().activationDepth(2);
        System.out
            .println("Object container activation depth = 2");
        ObjectSet result = container.queryByExample(new SensorPanel(1));
        SensorPanel sensor = (SensorPanel) result.queryByExample(0);
        SensorPanel next = sensor.next;
        while (next != null) {
            System.out.println("Object " + next + " is active: "
                + container.ext().isActive(next));
            next = next.next;
        }
    } finally {
        container.close();
    }
}

```

This method can be useful in applications with deep object hierarchy if you prefer to use manual activation.

IsStored

ExtObjectContainer#isStored helps you to define if the object is stored in the database. The following example shows how to use it:

```

UtilityExample.java: checkStored
private static void checkStored() {
    // create a linked list with length 10
    SensorPanel list = new SensorPanel().createList(10);
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        // store all elements with one statement, since all
        // elements are new
        container.store(list);
        Object sensor = (Object) list.sensor;
        SensorPanel sp5 = list.next.next.next.next;
        System.out.println("Root element " + list + " isStored: "
            + container.ext().isStored(list));
        System.out.println("Simple type " + sensor
            + " isStored: "
            + container.ext().isStored(sensor));
        System.out.println("Descend element " + sp5
            + " isStored: " + container.ext().isStored(sp5));
        container.delete(list);
        System.out.println("Root element " + list + " isStored: "
            + container.ext().isStored(list));
    } finally {
        container.close();
    }
}

```

```

UtilityExample.vb: CheckStored
Public Shared Sub CheckStored()

```

```

' create a linked list with length 10
Dim list As SensorPanel = New SensorPanel().CreateList(10)
File.Delete(Db4oFileName)
Dim db As IObjectContainer = Db4oFactory.OpenFile(Db4oFileName)
Try
    ' store all elements with one statement,
    ' since all elements are new
    db.Store(list)
    Dim sensor As Object = CType(list.Sensor, Object)
    Dim sp5 As SensorPanel = list.NextSensor.NextSensor. _
NextSensor.NextSensor
    System.Console.WriteLine("Root element " + list.ToString() _ 
+ " isStored: " + db.Ext().IsStored(list).ToString())
    System.Console.WriteLine("Simple type " + sensor.ToString() _ 
+ " isStored: " + db.Ext().IsStored(sensor).ToString())
    System.Console.WriteLine("Descend element " + sp5.ToString() _ 
+ " isStored: " + db.Ext().IsStored(sp5).ToString())
    db.Delete(list)
    System.Console.WriteLine("Root element " + list.ToString() _ 
+ " isStored: " + db.Ext().IsStored(list).ToString())
Finally
    db.Close()
End Try
End Sub

```

Descend

ExtObjectContainer#descend method allows you to navigate from a persistent object to its members without activating or instantiating intermediate objects.

```

UtilityExample.java: testDescend
private static void testDescend() {
    storeSensorPanel();
    Configuration configuration = Db4o.newConfiguration();
    configuration.activationDepth(1);
    ObjectContainer container = Db4o.openFile(configuration, DB4O_FILE_NAME);
    try {
        System.out
            .println("Object container activation depth = 1");
        ObjectSet result = container.queryByExample(new SensorPanel(1));
        SensorPanel spParent = (SensorPanel) result.queryByExample(0);
        SensorPanel spDescend = (SensorPanel) container.ext()
            .descend(
                (Object) spParent,
                new String[] { "next", "next", "next",
                    "next", "next" });
        container.ext().activate(spDescend, 5);
        System.out.println(spDescend);
    } finally {
        container.close();
    }
}

```

Navigating in this way can save you resources on activating only the objects you really need.

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 294](#)

When updates are not stored then you have issues with the update-depth. [See "Update Depth Pitfall" on page 295](#)

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 properly. [See "UUID Support" on page 125](#).

For big math types you need some additional configuration [See "Storing BigDecimal And BigInteger" on page 296](#). Also be careful with collections and avoid SQL-Types. [See "Special Type Handling" on page 296](#)

In more complex Java environments you may run into class loading issues. [See "Classloader And Generic Classes" on page 297](#)

There are a few dangerous practices which you really should avoid. [See "Dangerous Practices" on page 297](#)

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 298](#)

The Activation Pitfall

In order to work effectively with db4o you must understand the concept of [Activation](#). Activation controls the amount of referenced objects loaded into the memory. There are two main pitfalls that you must be aware about.

Accessing Not Activated Objects

One common pitfall is to access not activated 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 object up the a depth of 5. This means that when you load a object, that object and all object 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 stored. 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 completely.

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 143](#)

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. Unfortunately this isn't true for the java.sql.Date-type. This also means that the SQL-date type cannot be stored correctly. When you retrieve a SQL-Date type it will contain the wrong date.

The solution is to use only the normal JDK-Date. There's 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 your 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 159
- 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 158

Persistent Hashtables

Hashtable or Hashmap is a data structure that associates keys to values. Hashtable uses a hash function to quickly navigate to a specific value. Hash function returns an integer value based on a specific algorithm which is based on the contents of the object. Different hash algorithms can be used to produce hash codes for different objects. The general requirements for hash code are the following:

- Hash function must return the same result for the same object during the lifetime of the application.
- Hash function must produce the same results for the objects that are equal according to the equals(object) function
- If 2 objects are unequal according to the equals(object) function it is not required that the hash function produce distinct results

As you can see from the last point there can be more than one distinct key object in a hashtable that have the same hash code. Special methods called collision resolution are used to find the correct value for the specific key. Usually a separate storage - a bucket - is used for all keys with the same hash code. In this case a bucket is located by the hash code and then the right key is searched within the bucket, which allows to get a good enough performance. This works good enough for an in-memory hashtable as the hash values are not changed during application lifetime. However, it gets more difficult with a persistent hashtable.

When a hashtable is stored to a database - the hash values are not stored. As we know from the definition, the hash value of an object is only required to stay the same during the application lifetime, which means that if the hashtable will be loaded into memory from the database in another application or in another session, the hash values of the keys can differ from their initial value. We will still be able to retrieve values by their key objects if equals and hashCode functions are based on the object contents. However the consistency of the hashtable can potentially be broken. This can happen if the key objects from different buckets will obtain the same hash value as the result of re-instantiation from the database.

The simplest way to avoid the inconsistency of the persisted hash table use object content-based hash code functions for your key objects.

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](#).

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 sub-folders in the root folder of your Eclipse installation.

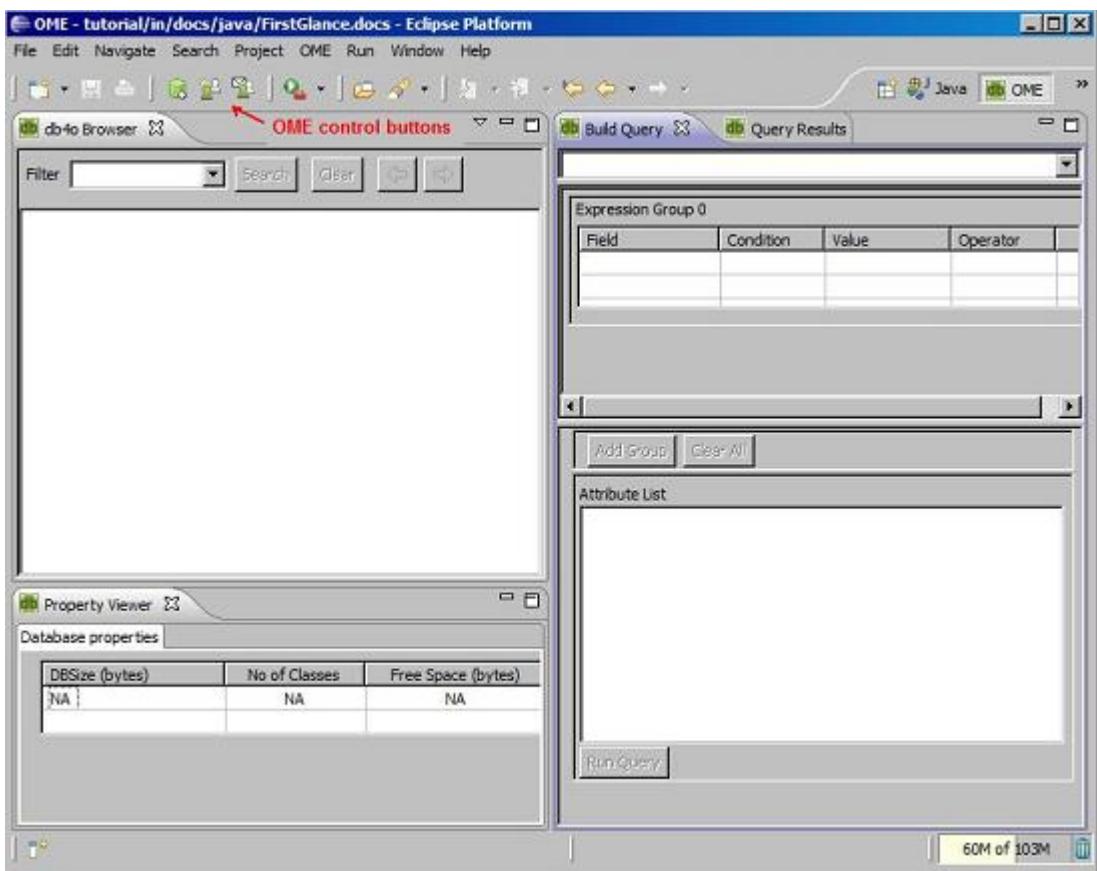
More Reading:

- [OME Interface](#)
- [Browsing A Database](#)
- [Querying](#)

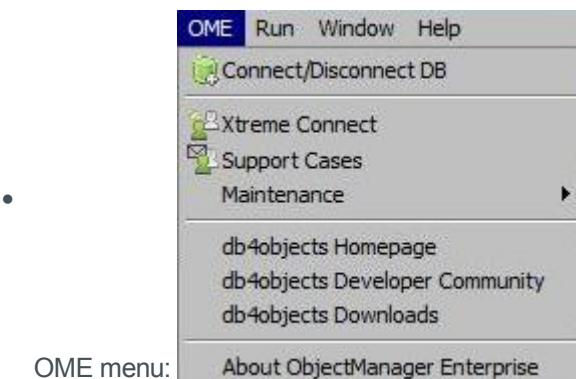
OME Interface

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:

¹Object Manager, a tool to view and edit a db4o database
²Object Manager, a tool to view and edit a db4o database



In the OME perspective you can see:

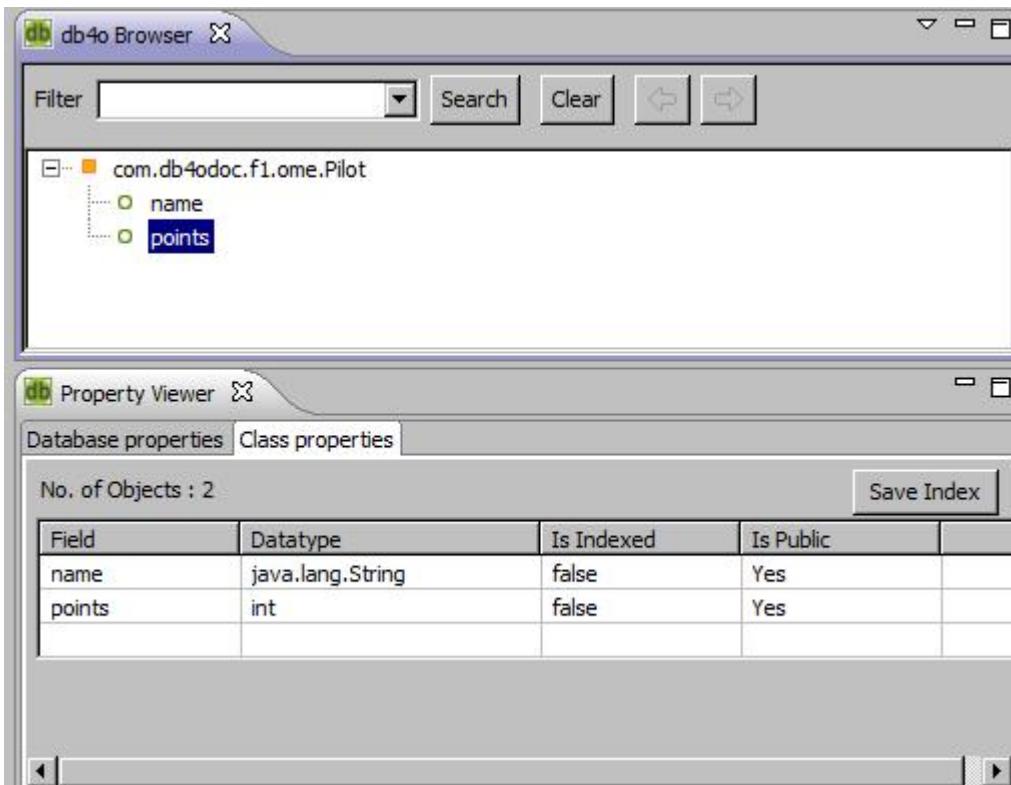


- OME toolbar buttons to access the frequently used functionality fast
- Db4o Browser: window displaying the contents of the open db4o database
- Property Viewer: window displaying the properties of the open database or the properties of the selected database class
- Build Query: windows allowing to 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:

¹Object Manager, a tool to view and edit a db4o database

com.db4odoc.f1.ome.Pilot

Row Id	name	points
1	Michael Schumacher	100
2	Rubens Barrichello	99

Object 2

Field	Value
com.db4odoc.f1.ome.Pilot	(G) com.db4odoc.f1.ome.Pilot
name	Rubens Barrichello
points	99

Save

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:

Filter Search Clear

com.db4odoc.f1.ome.Pilot

- name
- points

Property Viewer

No. of Objects : 2

Field	Datatype	Is Indexed
name	java.lang.String	false
points	int	false

Build Query

Query Results

Expression Group 0

Field	Condition	Value	Operator
-------	-----------	-------	----------

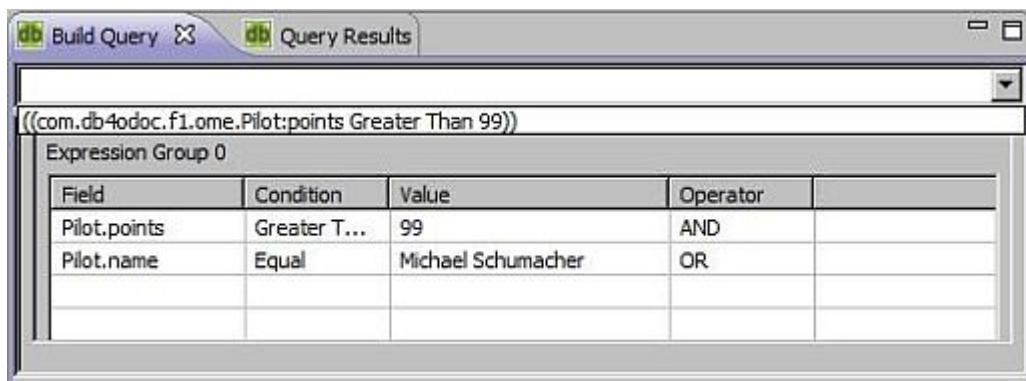
Add Group Clear All

Attribute List

Run Query

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.

¹Object Manager, a tool to view and edit a db4o database

Community

Participate in the db4o community and help to improve db4o. The main community website for db4o is <http://developer.db4o.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://developer.db4o.com/Forums.aspx>

Additional Resources on the Web

There's a overview of additional resources here: <http://developer.db4o.com/Resources.aspx>. To stay informed about new features, community activity and other news subscribe to the different db4o blogs: <http://developer.db4o.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 306

Report Bugs

Find a bug in db4o? Here are a few tips and tricks for reporting bugs.

Where Do I Report Bugs?

The best way to report bugs is to directly report the bug in our issue [tracking system here](#). 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 where you are not sure if the issue is an bug or another problem you maybe want to first 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. Those items are important:

- Steps to reproduce the bug. In order to fix the bug a step by step instructions on how to reproduce the issue are essential.
- 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.

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

Sharpen¹ is a 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

Additionally **Sharpen**² 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:

¹Sharpen is a tool to translate Java source code to C# source code

²Sharpen is a tool to translate Java source code to C# source code

```

<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.jar"/>
</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.

```
#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.

```
<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</resource>
        <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:

¹Sharpen is a tool to translate Java source code to C# source code

```
-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

Sharpen¹ 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 identifiers 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 Sharp- en.Runtime.ToJavaMilliseconds
-typeMapping	Maps a java class to .NET type:

¹Sharpen is a tool to translate Java source code to C# source code

	-typeMapping com.db4o.Db4o Db4oObjects.Db4o.Db4oFactory
-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

Sharpen¹ 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: <pre>/* * @sharpen.private */ public List4 _first;</pre>
@sharpen.internal	Specifies that the element must be declared internal in the con-

¹Sharpen is a tool to translate Java source code to C# source code

	<p>verted file:</p> <pre>/** * @sharpen.internal */ public abstract int size();</pre>
@sharpen.protected	<p>Specifies that the element must be declared protected in the converted file:</p> <pre>/** * @sharpen.protected */ public abstract int size();</pre>
@sharpen.new	Adds the C#-'new' modifier to the translated code.
@sharpen.event	<p>Links an event to its arguments. For example:</p> <p>Java:</p> <pre>/* * @sharpen.event com.db4o.events.QueryEventArgs */ public Event4 queryStarted();</pre> <p>is converted to:</p> <pre>public delegate void QueryEventHandler(object sender, Db4objects.Db4o.Events.QueryEventArgs args); event Db4objects.Db4o.Events.QueryEventHandler QueryStarted;</pre>
@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: <code>@sharpen.if <expression></code>
@sharpen.property	<p>Convert a java method as a property:</p> <pre>/** * @sharpen.property */ public abstract int size();</pre>
@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	<p>Adds an extends clause to the converted class definition. For example:</p> <p>Java:</p> <pre>/** * @sharpen.extends System.Collections.IList */ public interface ObjectSet {...</pre>

	converts to
@sharpen.partial	Marks the converted class as partial
@sharpen.remove	Marks a method invocation that should be removed
@sharpen.remove.first	<p>Removes the first line of the method/constructor when converting to C#:</p> <pre>/* * @sharpen.remove.first */ public void doSomething(){ System.out.println("Java"); NextMethod(); }</pre> <p>converts to:</p> <pre>public void DoSomething(){ NextMethod(); }</pre>
@sharpen.struct	Marks class to be converted as c# struct
@sharpen.unwrap	<p>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#.</p> <pre>/* * @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 }</pre> <p>Is converted to:</p> <pre>public IEnumerable ToIterable(object[] array){ return Arrays.AsList(array); } public void doSomething(object[] objs){ Iterable iterable = objs; // do something with the iterable }</pre>
@sharpen.attribute	<p>Adds an attribute to the converted code:</p> <pre>/* * @sharpen.attribute TheAttribute */ public void doSomething(){} Will be converted to: [TheAttribute] public void DoSomething{}} </pre>
@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 convenient 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.

¹OSGi framework, a module system and service platform

- 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.
- 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-contrib.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

If you want to contribute to the core code, you must follow our [contribution policy](#).

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.

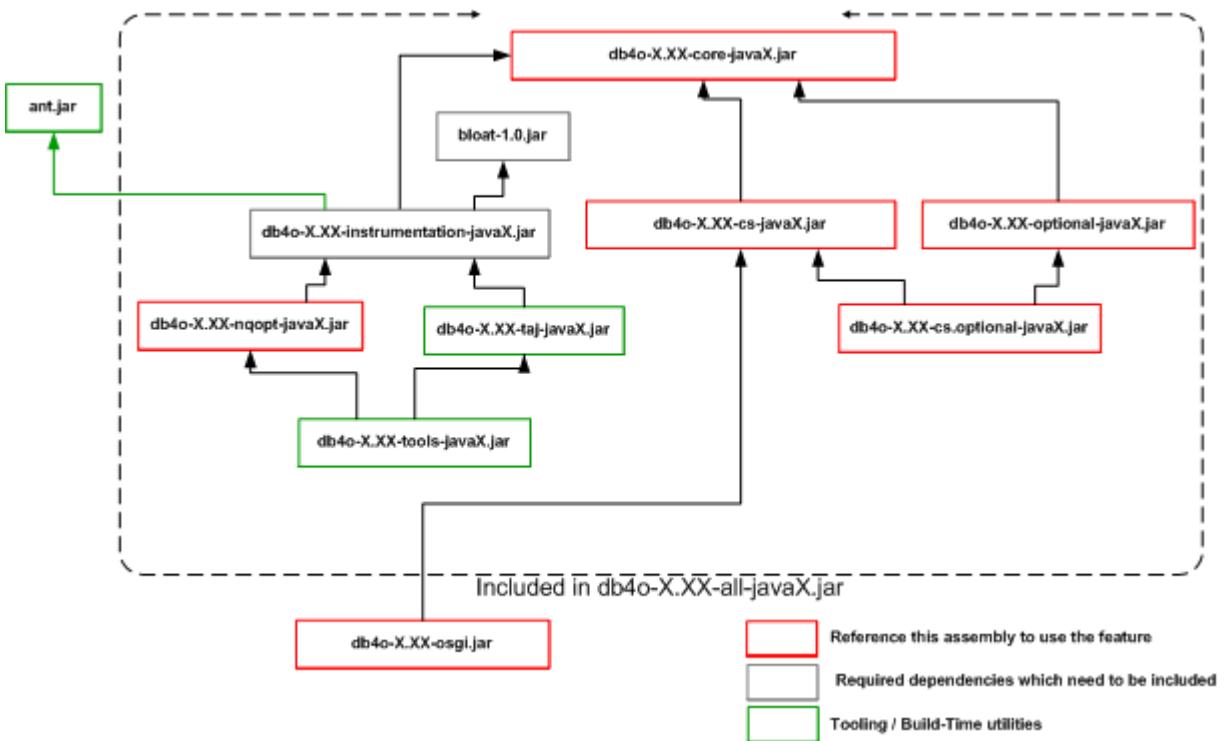
If you are using [Subversive](#) plugin you can use the following steps:

- 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.

¹OSGi 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. Preoptimized 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. And take a look at additional resources on [the community website](#). If you want to stay informed, subscribe to [db4o blogs](#).

Address and Contact Information

Versant Corporation

255 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