

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](#).

This documentation provides a collection of topics covering the majority of db4o features including utility projects like [dRS](#), [OME](#), db4oTool, [Sharpen](#) etc.

Online version provides information on all supported platforms (Java, .NET). Offline version, included in distribution, is specific to downloaded version.

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.

db4objects 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 mission to give developers a choice differentiated from out dated relational approaches when it comes to object persistence and thus make their life a lot easier. There is no mapping! No mapping annotation or XML mapping meta data. The db4o database is designed to be a universal, affordable product platform, that is easy to learn and easy to implement. 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.

More Reading:

- [Data Persistence](#)
- [OODBMS](#)
- [Db4o Position](#)
- [Why Choose Db4o](#)

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" (OR mismatch shortly). 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 relation, attribute, tuple, relation value and relation variable.

The object-relational mismatch has become enormously significant with the total adoption of OO technology. This resulted in the rapid development of so-called object-relational mappers (ORM), such as Hibernate or Toplink. This solution "cures" the symptoms of the OR mismatch by adding a layer into the software stack that automates the tedious task of linking objects to tables. However, this approach creates a huge drain on system performance, drives up software complexity, and increases the burden on software maintenance, thus resulting in higher cost of ownership. While the mapper solution may be feasible in large, administered datacenter environments, it is prohibitive in distributed and zero-administration architectures such as those required for embedded databases in client software, mobile devices, middleware or real-time systems.

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

Primary performance issues come from the fact that despite being called a "relational database", an RDBMS 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, first class -vs- second class objects, 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 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). OODBMS provides an ideal match with object oriented environments like Java and .NET reducing the cost of development, support and versioning and hence overall system costs.

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

- continuous refactoring;
- agile modeling;
- continuous regression testing;
- configuration management;
- developer "sandboxes".

(For more information about OODBMS technology refer to the [ODBMS.ORG website](#).)

Db4o Position

The db4o database came to the market in 2004 with a goal to become the mainstream persistence architecture for embedded applications (in which the database is invisible to the end user) in general, and for mobile and embedded devices running on Java or .NET, in particular. Versant's vision for db4o is to become the affordable, dominant, open source persistence solution of object oriented

developers of Java and .NET. In a very short time, the db4o team has achieved mainstream adoption with a fast growing user community currently boasting over 60,000 members. Community adoption is continually driven by db4o's efficient innovative technology, native queries, deployment in Java and .NET and its open source dual licensing business model.

The target environments for db4o are persistence architectures where there is no database administrator present and no RDBMS 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.

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, cross-platform applicability to Java and .NET, 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.

Why Choose Db4o

There are many advantages of using "native" object technology over RDBMS 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) with its Version 5.0 launched in November 2005 and since introduction of LINQ in .NET version 3.5. Db4o Native Queries provide an API that uses the programming languages Java or .NET to access the database. No time is spent on learning a separate data manipulation or query language. Unlike incumbent, string-based, non-native APIs (such as SQL, OQL, JDOQL and others) Native Queries and LINQ are 100% type-safe, 100% refactorable and 100% object-oriented, boosting developer productivity by up to 30%. In addition, the sophisticated modern programming development environments can be used to simplify the development and maintenance work even further.

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

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

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

For more information see our ["Choose db4o" presentation](#)

Getting Started

This topic will give you some start information about db4o and will help you to get your environment ready to work with db4o.

Download Contents

Java Platform

The db4o Java distribution comes as one zip file, db4o-7.12-java.zip. You will have to extract the contents to any folder before starting to use db4o.

db4o-7.12/doc/reference

contains reference documentation, which you are reading.

In addition to it you will find the following docs in your distribution:

db4o-7.12/doc/tutorial/index.htm

This is the interactive "formula-1" HTML tutorial. Examples can be run "live" against a db4o database from within the browser. In order to use the interactive functionality a Java JRE 1.3 or above needs to be installed and integrated into the browser. Java security settings have to allow applets to be run.

It is recommended to take a first quick "drive" with "formula-1" before studying other db4o documents.

db4o-7.12/doc/tutorial/db4o-7.12-tutorial.pdf

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

db4o-7.12/doc/api/index.htm

The API documentation for db4o is supplied as JavaDocs HTML files. While you read through this documentation it may be helpful to look into the API documentation occasionally.

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

The db4o Engine

The Java version of db4o object database engine consists of one single core jar file. In addition you

may want to use client/server library or optional components, which you can add separately or use "db4o-all" jar. **db4o-7.12-core-java1.1.jar**

will run with most Java JDKs that supply JDK 1.1.x functionality such as reflection and Exception handling. That includes many IBM J9 configurations, Symbian and Savaje. **db4o-7.12-core-java1.2.jar** is built for all Java JDKs between 1.2 and 1.4. **db4o-7.12-core-java5.jar** is built for Java JDK 5 and JDK 6 If you intend to use client/server version of db4o you will additionally need client/server library matching your JDK version: **db4o-7.12-cs-java1.1.jar** **db4o-7.12-cs-java1.2.jar** **db4o-7.12-cs-java5.jar** Some advanced functionality such as cluster support, platform-specific IO adapters, statistic tools etc can be added by including db4o optional library: **db4o-7.12-optional-java1.1.jar** **db4o-7.12-optional-java1.2.jar** **db4o-7.12-optional-java5.jar** You can also get all of the above in a single jar: **db4o-7.12-all-java1.1.jar** **db4o-7.12-all-java1.2.jar** **db4o-7.12-all-java5.jar**

[Security Requirements On Java Platform](#) reviews db4o jar security permissions requirements.

Installation

Java Installation

If you add one of the above db4o-*jar files to your CLASSPATH, db4o is installed. For beginners it is recommended to use "db4o-all" library to avoid confusion with the location of certain classes. In case you work with an integrated development environment like [Eclipse](#) you would copy the db4o-*jar to a /lib/ folder under your project and add db4o to your project as a library. (You only need to copy the one jar file for the distribution you are targeting.)

Here is how to add the db4o library to an Eclipse project

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

Please, note, that db4o can't be installed in JDK or JRE lib folder, the reasons are explained [further in the documentation](#).

Basic Concepts

This topic collection discusses concepts used in the foundation of the db4o system.

db4o is an object-oriented database. It provides all the benefits of an OO environment including data abstraction, inheritance, encapsulation. The object model simplifies maintenance and refactoring and seamlessly integrates with the modern programming languages (Java, .NET). One of the valuable benefits of the OO technology is Native Queries - database queries expressed in a native programming language.

In .NET version 3.5 the preferred alternative to Native Queries is LINQ - Language Integrates Queries. Though LINQ supports relational databases as well, in the db4o case, query execution does not include Object-Relational mapping which gives db4o better performance.

At the same time, db4o provides important database features like ACID transactions and concurrency control.

These and other technologies will be discussed in more details below (work in progress).

More Reading:

- [Evaluation Guide](#)
- [ACID Model](#)
- [Concurrency Control And Locking](#)
- [Object Identity](#)
- [Transaction](#)
- [Database Models](#)
- [Native Query Concepts](#)

Evaluation Guide

Database evaluation is one of the most important processes in a database application development. The wrong database choice can considerably affect the whole development process and even be one of the failure reasons. On the other hand the right database choice can save lots of development time and produce a more reliable and flexible solution.

The following topics provide some hints on how db4o should be evaluated for your future application, which factors are important and what benefits can be achieved by using db4o.

More Reading:

- [Db4o Applications](#)
- [Common Beginner Pitfalls](#)
- [Performance Benchmarks](#)
- [Scalability](#)

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. 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. Automatic refactoring can be another valuable factor, which allows to skip the job of updating the databases when a new version of object model is implemented.
Device applications	Device applications enjoy the same benefits as Mobile applications. In addition, smaller footprints can be achieved by using the minimal Micro edition.
Open-source software	GPL, open-source compatibility licence. Native to Java and .NET. Easily integrates with any Java and .NET open-source products.
Web-applications	open-source, reporting support from several Java open-source reporting frameworks and .NET reporting API

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

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

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

Common Beginner Pitfalls

The biggest trouble for users starting to use db4o is to realize that it is not a relational database. In db4o object identity and links between objects are managed thought the object model and are actually memory references to instantiated objects. The idea of a primary key is not necessary for db4o as any object can be present in memory only once and thus is identified by the variable holding the reference. For more information about the differences between db4o and relational databases see [Object And Relational Model Comparison](#).

Another concept that should be well understood by the db4o beginners is the concept of activation, i.e. object loading into the memory. When an object is loaded into the memory after the query was executed its field objects should be available to the user as well. However it might be a recipe for a trouble if an object has a deep structure, for example if it is a linked list. In this case it makes sense to load only the objects that will be accessed by the user. For more information about it see [Activation](#).

Related to activation is the problem of update. When an object is restored to the database, it is important to store the modifications in the field objects, as they might be the only values that were changed. However, it can produce a serious performance penalty if the actual modification was only in the top-level and the object contains a deep field structure (i.e. all objects in the graph will have to be updated). This problem is solved similarly to activation problem. For more information see [Transparent Persistence](#).

Another common beginner's mistake is overuse of commit. Commit is an important database operation, which ensures that all the changes were actually written to a physical storage and are safe there, however commit is also one of the most common performance bottlenecks. Db4o uses commit to physically write data to the physical storage. This operation requires several cycles of disk access, which ensure ACID transaction, and can be very slow depending on your storage hardware. For more information see [Commit Strategies](#).

Other common db4o pitfalls are listed in [Usage Pitfalls](#) chapter of db4o documentation.

Performance Benchmarks

The best way to evaluate the suitability of db4o for your next application is to write some performance tests. The Benchmarks part of reference documentation provides some simple benchmarks on insert, delete, update and query operations. You can use those tests and adjust them to your application environment.

To get the best picture of the expected performance, replace the classes used in the benchmark with your own object model classes. Populate the appropriate amount of instances to get the database size you expect and calculate the times of the most frequently used operations. For example, if the primary use of your application is caching, you will be mostly interested in insert performance (The cache might not be ever read after all). If you require rapid access to the data in the database - use query benchmarks to evaluate the general performance.

It is also a good idea to run benchmarks on the hardware, which is going to be used in production. That might be not so critical for normal desktop application, where the characteristics of the hardware can be easily improved, i.e. memory added, hard disk drive replaces for a faster one, etc. But for embedded device a simple memory upgrade can mean a considerable additional cost (multiplied by the amount of devices used in production). To simplify the decision-making process for such cases db4o provides a special IO benchmark, which can be configured to emulate the slower IO operations on an embedded device. For more information see [IO Benchmark Tools](#).

Other valuable information and tips for fine-tuning and improving db4o performance for different environments can be found in [Tuning](#) chapter. It also reveals the common mistakes that lead to poor performance.

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. For more information, please visit: <http://www.versant.com/>

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
- Use database clusters to spread the load between several database files.
- Use client-server version to enable many clients using data at the same time
- Use in-memory database to improve performance

Another important factor for performance and scalability is fragmentation. Fragmentation occurs when your objects grow in size regularly or when your application deletes lots of objects through

normal processing. That is why it is important to run db4o [Defragment](#) regularly. The physical disk fragmentation also plays an important role, so disk defragment should be also scheduled on a regular basis to improve the performance. The db4o defragmentation tool is an off-line tool. If you require online administration of actives such as defragmentation, you should probably be considering the Versant object database. For information, please visit: <http://www.versant.com/>

ACID Model

The ACID model is one of the oldest and most important concepts of database theory. It sets out the requirements for the database reliability: atomicity, consistency, isolation and durability. Without these requirements met, a database cannot be considered reliable.

Let's have a closer look at each of its components:

More Reading:

- [Atomicity](#)
- [Consistency](#)
- [Isolation](#)
- [Durability](#)
- [ACID Properties For Db4o](#)
- [Isolation Level For Db4o](#)

Atomicity

Atomicity states the mode, in which either all or no modifications are written to the database. In this mode each transaction is said to be "atomic". If any part of the transaction fails, the whole transaction will fail too. For example, in a bank transfer transaction there are 2 parts: 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.

Some of the atomicity features:

- A transaction is a unit of operation - either all the transaction's actions are completed or none are
- atomicity is maintained in the presence of deadlocks
- atomicity is maintained in the presence of database software failures
- atomicity is maintained in the presence of application software failures
- atomicity is maintained in the presence of operation system failures
- atomicity is maintained in the presence of CPU failures
- atomicity is maintained in the presence of disk failures

Consistency

Consistency imposes a rule that only valid data can be written to a database. When an update is executed, the database state will change only if all the database consistency rules are obeyed, otherwise the transaction will be rolled back and the database restored to its consistent state. On the other hand, if all the updates are consistent, the database will be taken to a new consistent state.

Consistency rules can include: data type correctness, relation correctness, uniqueness of data, etc.

Isolation

Isolation imposes rules, which ensure that transactions do not interfere with each other even if they are executed at the same time. If 2 (or more) transactions are executed at the same time, they must be executed in a way so that transaction N won't be impacted by the intermediate data of transaction M. 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 read 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 ("WHERE" clause in SQL). 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.

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.

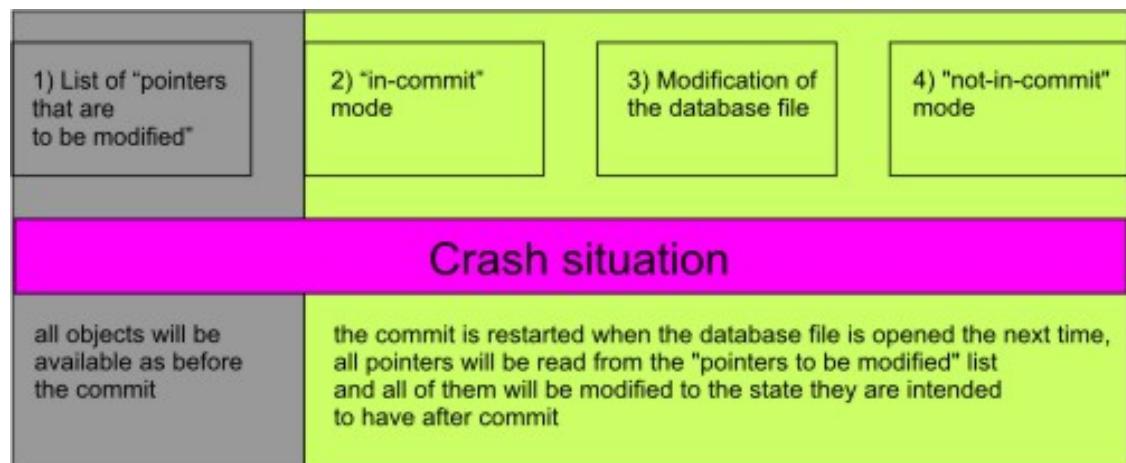
Durability

Durability property ensures that any committed transaction won't be lost. This is done by using database backups and transaction logs. Even in a case of a system or hardware failure, all the committed data should be restorable.

ACID Properties For Db4o

As any reliable database system db4o ensures ACID transactions. When a commit is executed, the following order of disc writes is ensured:

1. a list of "pointers that are to be modified" is written to the database file;
2. the database file is switched into "in-commit" mode;
3. the pointers are actually modified in the database file;
4. the database file is switched to "not-in-commit" mode.



As you can see from the picture above if a fatal failure occurs after the stage 1, the database will be restored to its pre-transaction consistent state. If the failure occurs after any other stage, the transaction information will be available from the transaction commit record and the commit will be restarted when the database will be open again.

The isolation in db4o database is at degree one, which means that a transaction does not overwrite "dirty data" of the other transactions and does not commit any writes until it completes all its writes (transaction commit record).

There are 2 settings that can effect ACID behavior in db4o:

1. Java:

```
RandomAccessFileAdapter randomAccessFileAdapter = new RandomAccessFileAdapter();
    NonFlushingIoAdapter nonFlushingIoAdapter =
        new NonFlushingIoAdapter(randomAccessFileAdapter);
    CachedIoAdapter cachedIoAdapter =
        new CachedIoAdapter(nonFlushingIoAdapter);
```

this setting can be potentially dangerous on systems using in-memory file caching. Instead of carrying out all writes immediately, the kernel stores data temporally in the buffer cache, waiting to see if it is possible to group several writes together. Cached file changes can also be reversed, which can break the ACID model.

Java:

```
configuration.disableCommitRecovery()
```

2. This setting disables commit recovery when the fatal failure occurs on stage 2-3 of the commit process. 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.

Isolation Level For Db4o

The default isolation level for db4o is read-committed. In your transaction, you can see the committed changes from the other transactions. However, don't forget that db4o uses object reference cache to speed up the retrieval. This means that if the object was already retrieved in the current transaction you will need to call `extObjectContainer.refresh(object)` to obtain the most recent value of the object.

Another interesting feature to point out: when using Lazy or Snapshot queries phantom reads can occur if the other transaction commits during the current transaction query execution. For more information see [Query Modes](#).

Please, note that the isolation level is only applicable for client-server version of db4o. If you are sharing the same object container between several users, you are actually working within one transaction. If you need to have isolation, you can implement it on your application level.

Concurrency Control And Locking

Concurrency control and locking is a mechanism used by DBMS to ensure that database transactions are executed in a safe manner. Atomicity, consistency, and isolation are achieved through concurrency control and locking. See [ACID Model](#).

When data is accessed from more than one transaction concurrently, it is usually necessary to ensure that only one transaction at a time can change a data item. Locking is a way to do this. Because of locking, all changes to a particular data item will be made in the correct order in a transaction. See [Isolation](#).

The following types of locking are usually distinguished:

More Reading:

- [Pessimistic Locking](#)
- [Optimistic Locking](#)
- [Overly Optimistic Locking](#)
- [Concurrency Control In Db4o](#)
- [Types Of Locks](#)
- [Locks In Db4o](#)

Pessimistic Locking

In pessimistic locking approach an entity is locked for the entire time the entity is in application memory (often in the form of an object). A read lock indicates that the object can be read but not modified or deleted by the other transactions. A write lock indicates that the object is locked exclusively and only the current transaction can read, modify or delete the object.

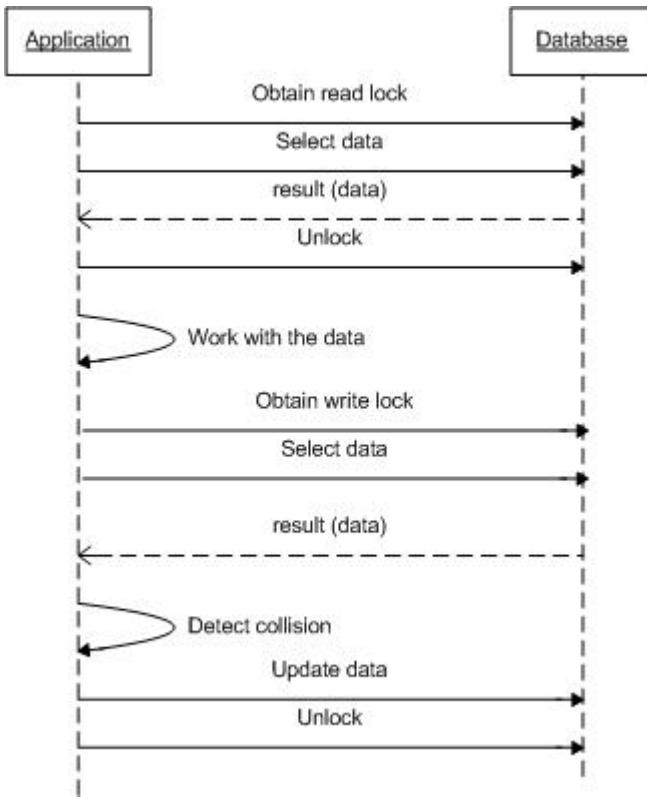
Advantage: pessimistic locking is easy to implement and guarantees that your changes to the database are made consistently and safely.

Disadvantage: pessimistic locking is not scalable; in systems with many users or long-running transactions, the waiting time for a lock to be released can be too long.

Optimistic Locking

In the real world there are many systems where collisions are not very frequent. For example if 2 users are processing bank transfers they may work with account objects but the accounts are different, so that they do not collide.

For the situation described above, optimistic locking will be a reasonable solution. When optimistic locking is used, it is accepted that collisions may occur, but instead of trying to prevent them, the system tries to detect and resolve them. Optimistic locking process flow is shown on the figure



below.

1. Data is retrieved with a read lock. The lock is released immediately after the retrieval.
2. Data modifications are done on unlocked data.
3. Before committing the modifications, the system tries to detect a collision. If a collision is detected, it should be handled according to the system rules (for example: the collision information can be logged or displayed to the user, transaction modifications can be merged, rejected or accepted).
4. The data is committed to the database or rolled back. The lock is released.

There are different ways to detect a collision. For example, you can mark the retrieved objects with a unique identifier, timestamp or username. Alternatively, you can store a copy of the originally retrieved object and compare it to the object from the database before trying to update.

Overly Optimistic Locking

Overly Optimistic Locking is a strategy, which assumes that the collision will never occur, therefore the system does not try to prevent a collision or to detect it. It is important to understand that this system can be only used in a single-user mode. Multi-user mode will require additional concurrency control strategy to be applied.

Concurrency Control In Db4o

Db4o uses overly optimistic concurrency control: an object is locked for read and write, but no collision detection is used. In the same time, db4o provides you with all means to implement a suitable

for you concurrency control strategy in your application.

In a client-server environment each client obtains a transaction from a client transaction pool. The changes are written to a temporary transactional space, which can be committed/rolled back when a commit/rollback message is received from the client.

When a client object container needs to write (or read) something a message with the task is created. If the batchMessages configuration setting is set to true, the messages will be collected in a batch of certain size before being sent to the server. The client write message process is synchronized, i.e. the only one client process can write to the server socket at a time.

On the server a message dispatcher loop gets messages from the socket and processes them depending on the type of message.

When a client transaction is committed a lock on the database file is obtained for physical write of data. The other client transactions have to wait till the write is done. To notify the other clients about the changes made by the current client commit [Commit-Time Callbacks](#) are used

For more information see [Concurrency Control](#).

Types Of Locks

Locking can also be classified by the entities that are being locked. Usually the following types are distinguished:

- Page locking: all the data on a specific memory page is locked.
- Cluster locking: all the objects in a cluster are locked (applies only to cluster-enabled object databases).
- Class or table locking: all objects of a class (OODBMS) or all rows in a table (RDBMS) will be locked.
- Object or instance locking: a single object (OODBMS) or a single relational tuple (RDBMS) will be locked.

Locks In Db4o

In db4o locks can be implemented with the help of [Semaphores](#). Though you cannot implement page or cluster locking, you can still vary the range of locking by using different semaphore names. For example:

1. The following semaphore will lock all the objects of a class:

Java:

```
extObjectContainer.setSemaphore(Pilot.class.getName, 3000)
```

2. This semaphore will lock a single object

Java:

```
extObjectContainer.setSemaphore("LOCK_"+objectContainer.ext().getID(pilot), 3000)
```

.NET:

```
extObjectContainer.SetSemaphore("LOCK_"+objectContainer.Ext().GetID(pilot), 3000)
```

Object Identity

Db4o keeps references to all persistent objects that are currently held in RAM, whether they were retrieved, created or activated in this session. The main role of the reference system is to provide access to the required data with the best speed and lowest memory consumption. Performance and usability of the reference system depend much on how the system manages objects identities.

More Reading:

- [Unique identity concept](#)
- [Identity Vs Equals](#)
- [Binding objects](#)
- [Weak References](#)

Unique identity concept

Db4o uses the concept of uniqueness of each object in reference cache. If an object is accessed by multiple queries or through multiple navigation access paths, db4o will always return the one single object, helping you to put your object graph together exactly the same way as it was when it was stored, without having to use IDs. You can simply use '==' to check the identity of two database objects.

```
IdentityExample.java: checkUniqueness
private static void checkUniqueness() {
    setObjects();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        ObjectSet cars = container.query(Car.class);
        Car car = (Car)cars.queryByExample(0);
        String pilotName = car.getPilot().getName();
        ObjectSet pilots = container.queryByExample(new Pilot(pilotName));
        Pilot pilot = (Pilot)pilots.queryByExample(0);
        System.out.println("Retrieved objects are identical: " + (pilot == car.getPilot()));
    } finally {
        container.close();
    }
}
```

```
    }  
}
```

How does db4o realize such behavior? Each object is loaded into reference cache only once in the session: db4o will return a new object only if it is not present in the cache yet, otherwise it will give you a reference to the object already in cache. This helps db4o to distinguish between objects that are to be updated and those ones that are to be created. All "known" objects are the subjects of update whereas "unknown" should be created. (Note that the reference system will only be in place as long as an ObjectContainer is open. Closing and reopening an ObjectContainer will clean the references system of the ObjectContainer and all objects in RAM will be treated as "new" afterwards.).

```
IdentityExample.java: checkReferenceCache  
private static void checkReferenceCache() {  
    setObjects();  
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);  
    try {  
        ObjectSet pilots = container.query(Pilot.class);  
        Pilot pilot = (Pilot)pilots.queryByExample(0);  
        String pilotName = pilot.getName();  
        pilot.setName("new name");  
        System.out.println("Retrieving pilot by name: " + pilotName);  
        ObjectSet pilots1 = container.queryByExample(new Pilot(pilotName));  
        listResult(pilots1);  
    } finally {  
        container.close();  
    }  
}
```

In the example *pilot* object is retrieved from the database (placed into cache) and changed, but not saved. The following retrieval uses pilot's name to retrieve the object from the database, but that object was already instantiated, so its cached (and modified) instance is actually returned.

Such behavior can be sometimes undesirable - you may expect to get object as it saved in the database instead of its modified instance in cache. One of the ways to do that is to use [ExtObjectContainer#peekPersisted\(object\)](#) method, which will give you a disconnected copy of a database object.

Another way is to purge objects from the cache before re-retrieving them.

You can use the following methods:

- ExtObjectContainer#isCached(object) shows if the object is present in reference cache
- ExtObjectContainer#purge(object) removes the object from the cache.

Let's look at our previous example extended with these methods:

```

IdentityExample.java: checkReferenceCacheWithPurge
private static void checkReferenceCacheWithPurge() {
    setObjects();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        ObjectSet pilots = container.query(Pilot.class);
        Pilot pilot = (Pilot)pilots.queryByExample(0);
        String pilotName = pilot.getName();
        pilot.setName("new name");
        System.out.println("Retrieving pilot by name: " + pilotName);
        long pilotID = container.ext().getID(pilot);
        if (container.ext().isCached(pilotID)) {
            container.ext().purge(pilot);
        }
        ObjectSet pilots1 = container.queryByExample(new Pilot(pilotName));
        listResult(pilots1);
    } finally {
        container.close();
    }
}

```

Now the second retrieval re-instantiates Pilot object from the database.

An object removed with ExtObjectContainer#purge(object) becomes "unknown" to the Object-Container, so this method may also be used to create multiple copies of objects:

```

IdentityExample.java: testCopyingWithPurge
private static void testCopyingWithPurge() {
    setObjects();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        ObjectSet pilots = container.query(Pilot.class);
        Pilot pilot = (Pilot)pilots.queryByExample(0);
        container.ext().purge(pilot);
        container.store(pilot);
        pilots = container.query(Pilot.class);
        listResult(pilots);
    } finally {
        container.close();
    }
}

```

Each reference in db4o works directly with the object. As only one instance of the object exists in cache there is no problem with object locks.

You can see an example of another concept used in [JDO system](#).

Actually db4o reference is a pointer to the object in the database file. It means that the size of the database does not affect query time: the object is retrieved from the known position without any necessity to traverse values.

Identity Vs Equals

One of the most common questions of db4o users is: why does not db4o allow to use equals() and hashCode to identify objects in the database. From the first glance it seems like a very attractive contract - 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 Pilot{string name} and Car{Pilot pilot} classes, and their equals method is based on comparing field values:

1. Store a pilot1(name="name1") and car1(pilot=pilot1) to the database
2. Retrieve pilot1
3. change pilot1(name = "name1") to pilot1(name="name2"). Note that though it is the same object from the runtime point of view, these are 2 different objects for the database based on equals comparison.
4. Now let's try to retrieve the car object, which has pilot = pilot1. We will get no results as the initial pilot stored with the database is not equal to the pilot1(name="name2"), and there is no car for the updated pilot anymore!

Now, this was a simple example, and can be solved by updating the car object together with the pilot. But what happens if there are thousands of objects referencing this pilot instance? They will all have to be retrieved and updated. Further, those objects can be also referenced somewhere and potentially a single update in a pilot object can trigger the re-write of the whole database.

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.

Binding objects

Db4o adds additional flexibility to its reference system allowing the user to re-associate an object with its stored instance or to replace an object in database:

Java:

```
ExtObjectContainer#bind(object,id)
```

Typical usecases could be:

- [enums and static fields](#)
- working on objects disconnected from the database
- refactoring

The following requirements should be met:

- The ID needs to be a valid internal object ID, previously retrieved with ExtObjectContainer#getId(object)
- The object parameter needs to be of the same class as the stored object.

Calling ExtObjectContainer#bind(object,id) does not have any impact on persisted objects. It only attaches the new object to the database identity. ObjectContainer#set(object) should be used to persist the change.

Let's look how it works in practice.

```
IdentityExample.java: testBind
private static void testBind() {
    setObjects();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        Query q = container.query();
        q.constrain(Car.class);
        q.descend("model").constrain("Ferrari");
        ObjectSet result = q.execute();
        Car car1 = (Car)result.queryByExample(0);
        long IdCar1 = container.ext().getID(car1);
        Car car2 = new Car("BMW", new Pilot("Rubens Barrichello"));
        container.ext().bind(car2,IdCar1);
        container.store(car2);

        result = container.query(Car.class);
        listResult(result);
    } finally {
        container.close();
    }
}
```

So this method gives you control over internal object storage. But its usage is potentially dangerous and normally should be avoided. Let's look at an example how `bind` can damage your object consistency:

Imagine three objects referencing each other:

`a1 => b1 => c1`

Now if you call `#bind()` to replace `b1` with `b2` in memory you will get the following:

`a1 => b1 => c1 b2 => c1`

`b2` will be the new in-memory copy of the persistent object formerly known as `b1`. `a1` will still point to `b1` which is now a transient object. If you now store `a1`, you will get a duplicate copy of `b1` stored.

Please, remember this scenario and use `ExtObjectContainer#bind(object,id)` only for short-lived objects and in controlled situations where no other references exist.

For the scenarios, which merging disconnected transient object, please refer to [Merge Module](#) project suggested design.

Weak References

Each retrieved or created object is automatically placed into reference system. Of course you have control over it and can purge or deactivate retrieved objects to prevent ever-growing memory consumption. However this requires a lot of attention and coding effort. Luckily, this is not necessary as db4o offers much easier way to manage the memory - WeakReferences.

Any object is kept in the memory while application has references to it otherwise it becomes eligible for garbage collection.

In the default configuration db4o uses weak references and a dedicated thread to clean them up after objects have been garbage collected by the VM. Weak references need extra resources and the cleanup thread will have a considerable impact on performance since it has to be synchronized with the normal operations within the ObjectContainer.

Transaction

All work within db4o ObjectContainer is transactional. A transaction is implicitly started when you open a container, and the current transaction is implicitly committed when you close it again. db4o transaction is tied to an open object container and only one transaction is allowed per object container instance.

Commit And Rollback

You may choose to make a commit explicit or you may leave it for the `#close()` call:

```
TransactionExample.java: storeCarCommit
private static void storeCarCommit(ObjectContainer container)  {
    Pilot pilot=new Pilot("Rubens Barrichello",99);
    Car car=new Car("BMW");
    car.setPilot(pilot);
    container.store(car);
    container.commit();
}
```

```
TransactionExample.java: listAllCars
private static void listAllCars(ObjectContainer container)  {
    ObjectSet result=container.queryByExample(Car.class);
    listResult(result);
}
```

Before transaction is committed all the modifications to a database are written to a [temporary memory storage](#). Commit (explicit or implicit) writes the modifications to the disk.

Please, remember to always commit or close your ObjectContainer when the work is done, to make sure that the data is saved to the permanent storage. [Commit Strategies](#) contains some important information on when and how commit should be used to achieve the best performance.

If you do not want to save changes to the database, you can call rollback, resetting the state of our database to the last commit point.

```
TransactionExample.java: storeCarRollback
private static void storeCarRollback(ObjectContainer container)  {
    Pilot pilot=new Pilot("Michael Schumacher",100);
    Car car=new Car("Ferrari");
    car.setPilot(pilot);
    container.store(car);
    container.rollback();
}
```

Refresh Live Objects

There is one thing that you should remember when rolling back: the `#rollback()` method will cancel the modifications, but it won't change back the state of the objects in your reference cache.

```
TransactionExample.java: carSnapshotRollback
private static void carSnapshotRollback(ObjectContainer container)  {
```

```

ObjectSet result=container.queryByExample(new Car("BMW"));
Car car=(Car)result.next();
car.snapshot();
container.store(car);
container.rollback();
System.out.println(car);
}

```

You have to explicitly refresh your live objects when their state might become different from the state in the database:

```

TransactionExample.java: carSnapshotRollbackRefresh
private static void carSnapshotRollbackRefresh(ObjectContainer container)  {
    ObjectSet result=container.queryByExample(new Car("BMW"));
    Car car=(Car)result.next();
    car.snapshot();
    container.store(car);
    container.rollback();
    container.ext().refresh(car,Integer.MAX_VALUE);
    System.out.println(car);
}

```

The `#refresh()` method might be also helpful when the changes to the database are done from different threads. See [Client-Server](#) for more information.

Database Models

This section covers object and relational database models and discusses them in comparison. More Reading:

- [Object Database Model](#)
- [Relational Database Model](#)
- [Object And Relational Model Comparison](#)
- [Object-Relational How To](#)

Object Database Model

Object model is characterized by:

[Data Abstraction](#)

[Encapsulation](#)

[Inheritance](#)

Data Abstraction

Data Abstraction enables isolation of the object's implementation details from the way it is used. Data Abstraction groups the pieces of data that describe some entity, so that programmers can manipulate that data as a unit. It helps a programmer to cope with the complexity of data by hiding the details.

In the object database model, each single entity is called an object instance. Each object has a unique unchanging identity. Object identity is characterized by the following features:

- object identity is independent from the data contained in the object - internal data values are not used to generate an identity;
- object identities are generated by the object system and are not controlled by a programmer or a database system;
- object identity lives as long as the object itself, it does not change with the modifications of the objects data content.

Objects are described by classes. A class defines a structure and attributes (fields) of an object. Classes are also used to define hierarchical properties: child and parent relationships.

Object model directly supports references. Note that references establish a connection between objects using their identities. For example:

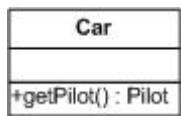


In this example Pilot object is referenced from a Car object.

Encapsulation

Encapsulation is an object model concept, which allows to hide specific behavior or processing abilities within object instances defined by a class. Method definitions within a class are an integral part of encapsulation, which allows to store data and code together.

For example:

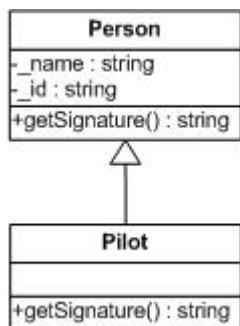


getPilot method can be "attached to" a Car object or encapsulated. Alternatively, getPilot method can be stored in the application or a separate library and distributed with the object database. However, encapsulation approach has an important advantage: method code cannot be lost or outdated as in the case of an application/library storage. Object system recognizes which methods belong to which data. The process of the correct method linking to the object is called **dispatching**.

Inheritance

Inheritance is a way to form new classes based on the classes that have already been defined.

For example:

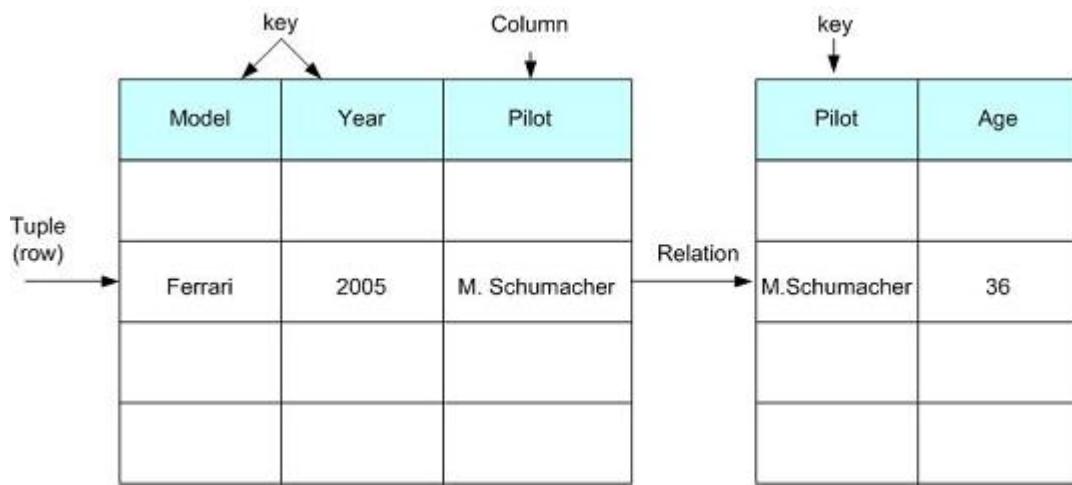


The derived class inherits all the fields and methods of the base class. Note that in the object model, there is no distinction between using system and user-defined types, so you can define sub-types of the system types. This feature is known as extensibility.

In the example above `getSignature` method is overridden in the derived class - this is known as **polymorphism**. The ability of the object model to execute the correct method based on the type of a class is called **dispatching**.

Relational Database Model

Relational data model is based on the concept of a relation or a table. An entity is presented by a tuple or row. Table fields are used to describe entity attributes (Model, Year, Pilot, Age in the example below). Each entity should be identified by a primary key, which is constructed from the data in the row. In the example below the key is created by combining the values of "Model" and "Year" fields.



In addition to the primary key relational data model supports foreign keys: a primary key of one table used in another table to allow joining of data. In the example above the primary key from the Pilots table ("Pilot" field) is used in the Cars table; it allows to associate a car and a pilot.

Object And Relational Model Comparison

Object and relational models though serving the same goal are very different in their concepts. There is no solid answer which technology is better or superior. In most cases when deciding between the 2 approaches you will have to base your decision on a close requirements analysis. See [RDBMS And OODBMS Application](#) for a list of environments and requirements where OODBMS is preferred. More Reading:

- [Basic Terms](#)
- [RDBMS And OODBMS Application](#)

Basic Terms

While relational and object databases use different sets of terms, it is possible to draw parallels between some of the terms:

OODBMS	RDBMS	Similar	Different
Class	Table	Define an entity data structure.	Class can define fields (static data) and methods, while table can only define static data.
Object instance	Tuple	Represent an instance of data to other objects.	Object can hold data of different visibility and references defined in a class or a table.
Attribute	Column	Define one of the fields in the data definition.	RDBMS has a preset collection of types, which can be used to define the type of the data in the column. OODBMS can use user-defined types.
Method	Stored Procedure	Define a piece of functionality.	Method is a characteristic of a class. Stored procedure is a separate object in RDBMS.
Identity	Key	Identify a single object or tuple.	Object identity makes any object unique independently of the object content. Key imposes a uniqueness of a database object based on the uniqueness of the column values.
Reference	Relation	Link different objects within a database.	In OODBMS direct object links are used. In RDBMS objects are linked through foreign keys, i.e. identical values in different rows.

Note, that the above-mentioned terms are similar only to a certain degree; there are some fundamental differences between them.

RDBMS And OODBMS Application

Relational data systems are proven market leaders in persistence solutions. They are mature and reliable choice for a general purpose solution. Moreover, RDBMS is supported by a huge number of applications and components and a large amount of human resources experienced in the technology.

However, OODBMS has some valuable advantages, which can't be underestimated:

- native Object Oriented language support (no object-relational impedance mismatch);
- native complex object structures and hierarchies support;
- easy refactoring and product evolution;
- zero-administration;
- continuous regression testing.

Some of the most obvious usecases and advantages of OODBMS are listed below.

Object Oriented Development

The fact that OODBMS technology is the perfect match for using in OO development environment is obvious from the very name. Though RDBMS has a wide support in tools and methods of using in OO environment there will always be an overhead of the translation from the object to relational world, commonly known as object-relational impedance mismatch.

Embedded Applications

Embedded device applications need a small-footprint zero-administration database, which makes a perfect match with an OODBMS. In addition, they usually need a quick response-time, which can be better achieved by a native OO technology, when no object-relational conversion is needed.

Complex Object Structures And Hierarchies

In applications having to deal with deep object structures (tree or graph), OODBMS offers a much easier native way to store them. Trees or graphs cannot be easily translated to RDBMS structure, so that the application has to deal with often complex conversion algorithms, which are difficult to maintain and refactor. On the other hand, OODBMS can store these structures transparently without any additional coding.

The same is true for difficult inter-objects relationships. In RDBMS they are realized with foreign keys. In some cases, you will need to fetch object by object until you will reach the final relation. In OODBMS all you need is to specify an [activation depth](#) or use [transparent activation](#) to reach all the related objects through the top-object fields.

From the said above you can conclude that the applications using objects with collection members (one-to-many relationships) will also benefit from OODBMS technology.

Refactoring And Schema Evolution

Most applications evolve as the time goes. It means that their class and data structure (schema) can change. In applications using relational databases schema evolution is quite work-consuming process: the schema must be changed, the class structure must be changed and the query collection must be changed too. In OODBMS applications, only class structure is a subject of change and the process can be highly automated by using modern refactoring techniques available from IDEs.

Agile Development

OODBMS makes it possible to implement agile development process in your team:

- continuous refactoring;
- agile modeling;
- continuous regression testing;
- configuration management;
- developer "sandboxes".

The use of RDBMS technology complicates the adoption of these techniques due to the technical impedance mismatch, the cultural impedance mismatch, and the current lack of tool support.

Object-Relational How To

This chapter is intended to help people moving from a relational database to db4o. The information here will try to draw parallels between object and relational concepts, explain how to achieve similar effect in both technologies and what practices are best to be used with the object database.

More Reading:

- [Identity And Primary Keys](#)
- [Foreign Keys](#)
- [Referential Integrity](#)
- [Triggers](#)
- [Unique Constraints](#)
- [Indexes](#)
- [Concurrency Control](#)

Identity And Primary Keys

One of the most important concepts to understand working with a database system is the identification strategy.

In RDBMS separate entities are distinguished with the help of primary keys. Effectively, objects are compared on their field contents and a constraint is used prohibiting 2 different rows in a table to have the same value in a column marked as a primary key.

In the object world, the concept is quite different - any object is unique, independently of the data it holds. For example:

```
class Car  
  
    model  
  
    year  
  
car1 = new Car("BMW", "1999")  
  
car2 = new Car("BMW", "1999")
```

`car1` and `car2` objects consist of the same data, but they are not equal.

On the other hand we can create any amount of references to the object and they will all be equal:

```
car1 = new Car("BMW", "1999")  
  
car2 = car1  
  
car3 = car1
```

In this example `car2` is equal to `car1` and `car3` and all these variables actually reference the same object.

Object database task is to ensure that independently of the way the object was accessed (by different queries or through different navigation access paths) the database must return a reference to the one single object. For a detailed explanation how it is achieved in db4o, please see [Unique identity concept](#).

If you are moving from a relational database to db4o it is very important to understand db4o identity concept. Instead of tying objects to the database reference key you should use the objects, obtained from the database directly without additional worries about their identity.

Unfortunately, there still are some cases, when additional identification system is necessary. This can happen when an object must be referenced out of the application memory boundaries: objects referenced between sessions, objects passed to another application, partially connected applications etc. In these cases, you are encouraged to use db4o internal IDs or Unique Universal IDs (UUID). Internal IDs are unique within one database file and do not change after defragment. UUIDs are unique between all db4o databases and do not change. For more information see [IDs and UUIDs](#).

Foreign Keys

Foreign keys issue is closely connected to the issue of [object identity](#).

In RDBMS foreign keys are created by designating a special column (or set of columns) in 2 different tables to hold the same information. The data in a row of one table is related to the data in a row of another table if the values in those columns match. Usually primary key is used on one side of the equation (can be replaced by unique key). Foreign keys are also called [referential constraints](#).

In db4o there is no concept of a primary key, therefore no foreign key as well. Object relationships are established through direct object references, which are kept in object fields. This approach is perfectly straightforward and intuitive. It considerably simplifies the navigation: instead of issuing more and more queries to access child (referenced) objects, you can just get them by the reference in the object field. This is definitely much more productive and requires less coding. However, object references fulfill only part of foreign key responsibilities: in addition to referencing another data entity, foreign key ensures the referential integrity. The [following chapter](#) discusses how referential integrity can be implemented in db4o.

Referential Integrity

Referential integrity (RI) is an important feature of RDBMS. It helps to protect data from misuse and corruption. However, in the modern world of multi-tier and distributed technologies it becomes questionable if referential integrity should be realized on a database level or is it a responsibility of a business layer. The answer to this question depends on the system design. Usually database RI makes lots of sense in a data-centric application with one main database. More complex, distributed in space and time referential constraints can be better implemented within a business object framework.

db4o database does not provide full referential integrity support, rather it gives a user possibilities to implement RI on the application level. For an example of a referential integrity solution, see [Referential Integrity](#).

Triggers

Triggers are widely used by RDBMS to automate some work as a response to a certain event. Similar behavior can be easily implemented in db4o with the help of callbacks. See [db4o callbacks](#) for more information.

Unique Constraints

Unique constraint in RDBMS is a field or combination of fields that uniquely defines a record. In an object database world unique constraints will uniquely define an object. In db4o unique constraints were first introduced in version 6.2 and are still work in progress. Current version supports unique constraints over only one field. For more information see [Unique Constraints](#)

Indexes

Indexes play an important role in database performance optimization. An index is a special data structure that helps to make searches more effective. In db4o, as in many relational databases,

indexes are based on B-Trees. You can define an index for any object field (including private ones).

Several indexes on different fields of the same class will result in better searching performance. Unlike relational databases db4o does not allow indexes over a set of fields. For more information see [Indexing](#) and [Enable Field Indexes](#).

Concurrency Control

Concurrency control is a vital feature for any database. While most popular relational databases give you a choice of isolation levels and locking types, db4o goes with a default read-committed overly optimistic locking strategy. For more information see [Concurrency Control And Locking](#) and [Isolation Level For Db4o](#). [Locks In Db4o](#) explains how applications can implement customary locks in db4o.

Native Query Concepts

Database query is a language used to communicate with the database. For a long time databases stood the central and commanding place in software products: the applications were literally written to support the needs of a main central database. However, in the modern high-speed world of distribute computing the transfer and modification of the information plays increasingly important role and the databases more and more often take a place of a temporary storage or a point of exchange.

This evolutionary change results in some practical modifications: more attention is paid to the business logic and tools that are used for its development, language structures are automatically produced and checked by development environments and string-based logic is getting outdated and not enough efficient. Database query language is expected to follow the features of an object-oriented language: type-safe, easy to refactor and test. String-based languages like SQL, OQL, JDOQL do not meet these requirements. To fill this gap and provide developers with sufficient database query tools a new concept was developed by W.Cook and C. Rosenberger which was given a name [Native Queries](#).

In the following paragraphs, we will review the concepts of Native Queries (NQ). We will use Pilot class for all examples suggested further:

```
Pilot.java
/**/* Copyright (C) 2007 Versant Inc. http://www.db4o.com */
package com.db4odoc.nqconcepts;

public class Pilot {
    private String name;
    private int points;

    public Pilot(String name, int points) {
```

```

        this.name=name;
        this.points =points;
    }

    public String getName()  {
        return name;
    }

    public int getPoints()  {
        return points;
    }
}

```

More Reading:

- [Problems Of String Based Query Languages](#)
- [Native Query Characteristics](#)
- [Native Query Implementation](#)
- [Native Query Optimization](#)
- [Code Inside](#)
- [Sorting Results](#)

Problems Of String Based Query Languages

Let's look how a query can be expressed against the Car class in some of the object querying languages:

OQL

```

String oql = "select * from pilot in AllPilots where pilot.points < 100";
OQLQuery query = new OQLQuery(oql);
Object pilots = query.execute();

```

JDOQL

```

Query query = persistenceManager.newQuery(Pilot.class, "points < 100");
Collection pilots = (Collection)query.execute();

```

db4o SODA, using C#

```

Query query = database.Query();
query.Constrain(typeof(Pilot));
query.Descend("points").Constrain(100).Smaller();

```

```
IList pilots = query.Execute();
```

As you can see, query parameters ("points") and constraints ("<100") are expressed as strings, which results in the following problems:

- Modern development environments do not check embedded strings for syntactic and semantic correctness. A small typo in a query (for example 10 instead of 100) will be very difficult to trace, moreover it can pass unnoticed to a production version.
- Embedded strings are not affected by refactoring tools. As the application evolves and the changes are made to the field variables, the string-based queries will become obsolete and will need to be changed by hand.
- String queries address fields directly instead of using publicly accessible methods/attributes, breaking encapsulation principle.
- Embedded strings can be on the way of modern agile techniques, which encourage constant refactoring. Since string queries are difficult to refactor and maintain they will delay a decision to refactor and result in a lower-quality code.
- Working with a string query language inside an OO language requires a developer to learn both and switch between them in the development cycle.
- Reusability support of OO languages (method calls, polymorphism, overriding) are not accessible to string-based queries.
- Embedded strings can be subject to injection attacks.

Native Query Characteristics

The [previous paragraph](#) reveals the need of a new querying technology native to modern development languages. This new technology was developed by [W.Cook and C.Rosenberger](#) and named Native Queries.

Native Queries are characterized by the following:

- 100% native: Queries are completely expressed in the implementation language (Java or c#), and they fully obey all language semantics.
- 100% object-oriented: Queries are runnable in the language itself, to allow unoptimized execution against plain collections without custom preprocessing.
- 100% type-safe: Queries are fully accessible to modern IDE features like syntax checking, type checking, refactoring, etc.
- optimizable: It is possible to translate a native query to a persistence architecture's query language or API for performance optimization. This is done at compile time or at load time by source code or bytecode analysis and translation. However, not every construct within a Native Query can be optimized.

Native Query Implementation

So how are Native Queries realized in practice?

Let's return to the [Pilot](#) class. Suppose we need to find all pilots with the name starting from "M" and points over 100. In a native OO language it would be expressed as:

Java:

```
pilot.getName().startsWith("M") && pilot.getPoints() > 100
```

In order to pass this condition to a database, collection or other query processor a special object is used. In .NET2 it is a delegate, in Java5 it is a named method.

Java:

```
public abstract class Predicate <ExtentType> {  
    public <ExtentType> Predicate (){}  
    public abstract boolean match (ExtentType candidate);  
}  
  
new Predicate <Pilot> () {  
    public boolean match(Pilot pilot){  
        return pilot.getName().contains("M") && pilot.getPoints() > 100;  
    }  
}
```

For more information about NQ implementations in the other Java and .NET versions see [Native Query Syntax](#).

Native Query Optimization

Though Native Query API discussed in the [previous paragraph](#) is simple and straightforward, the real challenge is to provide a performant solution.

If the NQ code is run as is, it requires instantiation of all the members of a class. This is very slow in most cases. In order to improve the performance a special optimizer is used by db4o. The idea of the optimization is to analyze the code in a Native Query and provide an alternative in a database query language. This can be done in runtime or build time.

Obviously, optimization is not possible in cases, when a native query does not have a database query alternative. To reveal those cases [db4o Diagnostic](#) system should be used.

For more information see [Native Query Optimization](#).

Code Inside

Native Query expression is dealt as a normal piece of code. Therefore, any language construction is legible inside:

- variables;
- temporary objects created within a query;
- static calls;
- exception handling.

However, some restrictions do apply:

- NQ should not be used to modify the database to prevent loops;
- NQ should not use threads, as NQ are expected to be triggered in large numbers;
- NQ should be fast, therefore they should not interact with the GUI;
- NQ should follow security restrictions, as they are executed in the server and potentially can create malicious behavior there.

In a case of uncaught exception within a Native Query a null result is returned.

For more info see [Native Query Collection](#)

Sorting Results

Native Queries also provide a native interface for sorting of the results using Comparator/IComparer:

Java:

```
ObjectSet query(Predicate predicate, Comparator comparator);
```

For more information see [Native Query Sorting](#).

Object Lifecycle

This topic set explains the lifecycle of an object within db4o database. Reading through the topics you will learn to open and close a database, prepare your objects for persistence, store them to db4o, and retrieve using differing querying strategies.

More Reading:

- [Object Container](#)
- [Simple Persistence](#)
- [Querying](#)
- [Transparent Persistence](#)
- [Working With Structured Objects](#)
- [Activation](#)
- [Update Depth](#)
- [Delete Behavior](#)
- [Object Construction](#)

Object Container

db4o gives you a simple and straightforward interface to object persistence - ObjectContainer. In .NET versions a conventional name IObjectContainer is used.

Accessing A Database

ObjectContainer is your db4o database.

Java:

```
ObjectContainer container = Db4o.openFile(filename);
```

filename is the path to the file, in which you want to store your objects. Normally you should open an ObjectContainer when the application starts and close it, when the session is finished to persist the changes to a physical storage location:

Java:

```
container.close();
```

nly the first call against a file can be successful. Subsequent calls that request to open a database file that is already open will get a `DatabaseFileLockedException`.

Working With Objects

`ObjectContainer` interface gives you all the basic functionality to work with persistent objects. Normally you can save a new or updated object of any class using `objectContainer.store(object)`

Deletion is done with the following method:

Java:

```
container.delete(object)
```

Through `ObjectContainer#get` and `ObjectContainer#query` you get access to objects retrieval functionality.

Object Container Features

The characteristic features of an Object Container are:

- An Object Container can either be a database in a single-user mode or a client connection to a db4o server.
- Every Object Container owns one transaction. All work is transactional. When you open an Object Container, you are in a transaction, when you `commit()` or `rollback()`, the next transaction is started immediately.
- Every Object Container maintains its own references to stored and instantiated objects. In doing so, it manages object identities, and is able to achieve a high level of performance.
- ObjectContainers are intended to be kept open as long as you work against them. When you close an Object Container, all database references to objects in RAM will be discarded.

Basically Object Container supplies functionality, which is enough for the most common usage of db4o database.

Extended Object Container Interface

Additional db4o features are provided by an interface extending Object Container - `ExtObjectContainer/IExtObjectContainer`.

The idea of splitting basic and advanced functionality between 2 interfaces is:

- Keep the root package/namespace very small and well readable.
- Separate vital and optional functionality.
- Make it easy for other products to implement the basic db4o interface.

- Show an example of how a lightweight version of db4o could look.

Every Object Container object is also an ExtObjectContainer. You can cast it to ExtObjectContainer or you can use ext method to get to the advanced features.

Simple Persistence

db4o makes your work with persistent objects very simple and straightforward. The only store(object) method is used for both saving and modification of any object that exists in your model.

```
QueryExample.java: storePilot
public static void storePilot() {
    new File(YAPFILENAME).delete();
    ObjectContainer db=Db4o.openFile(YAPFILENAME);
    try {
        Pilot pilot=new Pilot("Michael Schumacher",0);
        db.store(pilot);
        System.out.println("Stored "+pilot);
        // change pilot and resave updated
        pilot.addPoints(10);
        db.store(pilot);
        System.out.println("Stored "+pilot);
    } finally {
        db.close();
    }
    retrieveAllPilots();
}
```

```
QueryExample.java: updatePilotWrong
public static void updatePilotWrong() {
    storePilot();
    ObjectContainer db=Db4o.openFile(YAPFILENAME);
    try {
        // Even completely identical Pilot object
        // won't work for update of the saved pilot
        Pilot pilot = new Pilot("Michael Schumacher",10);
        pilot.addPoints(10);
        db.store(pilot);
        System.out.println("Added 10 points for "+pilot);
    } finally {
        db.close();
    }
    retrieveAllPilots();
}
```

```
QueryExample.java: updatePilot
public static void updatePilot() {
    storePilot();
    ObjectContainer db=Db4o.openFile(YAPFILENAME);
    try {
        // first retrieve the object from the database
```

```

ObjectSet result=db.queryByExample(new Pilot("Michael Schumacher",10));
Pilot found=(Pilot)result.next();
found.addPoints(10);
db.store(found);
System.out.println("Added 10 points for "+found);
} finally {
db.close();
}
retrieveAllPilots();
}

```

Deletion is just as easy:

```

QueryExample.java: deletePilot
public static void deletePilot()  {
    storePilot();
    ObjectContainer db=Db4o.openFile(YAPFILENAME);
    try  {
//        first retrieve the object from the database
        ObjectSet result=db.queryByExample(new Pilot("Michael Schumacher",10));
        Pilot found=(Pilot)result.next();
        db.delete(found);
        System.out.println("Deleted "+found);
    } finally {
        db.close();
    }
    retrieveAllPilots();
}

```

The objects are identified by their references in an application cache. You do not need to implement any additional identification systems (like primary keys in RDBMS). See [Identity chapter](#) for details. The uniqueness of an object is defined only by its reference, if you will create 2 objects of the same class with exactly the same fields and save them to db4o - you will get 2 objects in your database. As you can see from the examples an object instance should be retrieved from the database before updating or deleting or you can use the newly created object if it was stored in the same session. Creating a new instance identical to the object in the database and saving it, will create a new object in the database.

Db4o does all the "dirty" work of objects transition between your classes and persistent state using [Reflection](#). No mappings or additional coding is needed from your side. If you will need to change your application model for the next version you will also be surprised with the simplicity: all the changes are done in one place - your code, and the most common operations are done completely automatically (see [Refactoring And Schema Evolution](#) chapter for details).

Please, remember that all db4o work is done within [Transaction](#), which can be committed or rolled back depending on the result you want to achieve.

Querying

db4o supplies three querying systems, Query-By-Example (QBE),Native Queries (NQ), and the SODA Query API.

[Queries-By-Example](#) (QBE) 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.

[Native Queries](#) (NQ) are the main db4o query interface, recommended for general use.

[LINQ](#) queries are a convenient alternative to .NET users.

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 NQ are too strongly typed. There may be queries that will execute faster in SODA style, so it can be used to tune applications.

Of course, you can mix these strategies as needed.

For more information on custom query comparators see [Custom Query Comparator](#).

More Reading:

- [Query By Example](#)
- [Native Queries](#)
- [SODA Query](#)
- [LINQ](#)
- [Query Modes](#)
- [SODA Evaluations](#)
- [Sorting Query Results](#)
- [Custom Query Comparator](#)

Query By Example

When using *Query By Example*(QBE) you provide db4o with a template object. db4o will return all of the objects which match all non-default field values. This is done via reflecting all of the fields and building a query expression where all non-default-value fields are combined with AND expressions. Here's a simple example:

```
PersistentExample.java: retrievePilotByName
```

```

private static void retrievePilotByName(ObjectContainer container) {
    Pilot proto = new Pilot("Michael Schumacher", 0);
    ObjectSet result = container.queryByExample(proto);
    listResult(result);
}

```

Querying this way has some obvious limitations:

- db4o must reflect all members of your example object.
- You cannot perform advanced query expressions. (AND, OR, NOT, etc.)
- You cannot constrain on values like 0 (integers), "" (empty strings), or nulls (reference types) because they would be interpreted as unconstrained.
- You need to be able to create [objects without initialized fields](#). That means you can not initialize fields where they are declared.
- You need a [constructor](#) to create objects without initialized fields.

For more information see [QBE Limitations](#). To get around all of these constraints, db4o provides the [Native Query](#) (NQ) system.

Pilot2

```

Pilot2.java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */

package com.db4odoc.qbe;

public class Pilot2 {
    private String name;

    private int points = 100;

    public Pilot2(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

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

    public int getPoints() {
        return points;
    }
}

```

```

    public String toString() {
        return name + "/" + points;
    }
}

```

QBE Limitations

As it was mentioned [before](#) QBE has some serious limitations.

Query-By-Example evaluates all non-null fields and all simple type variables that do not hold their default values against the stored objects. Check to make sure that you are not constraining the resultset by accidentally initializing variables on your template objects. Typical places could be:

- Constructors
- Constructors in ancestors of your class
- Static initialization
- Static initialization in ancestors of your class.

The following classes provide an example of classes that cannot be used with QBE:

- [Pilot1](#)
- [Pilot1Derived](#)
- [Pilot2](#)
- [Pilot2Derived](#)

The following examples show the results of QBE usage with the classes above. Note, that there are some differences between Java and .NET behavior:

- in Java QBE does not return any results when static initialization or initialization in a constructor is present;
- in .NET QBE will return results in the above-mentioned cases, but it can't guarantee that the result will be correct if the value was changed in the constructor.

1. QBE used against a class that has arbitrary member initialization in the constructor:

```

QBEExample.java: test1
private static void test1() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            // Pilot1 contains initialisation in the constructor
            Pilot1 pilot = new Pilot1("Kimi Raikkonen");
            container.store(pilot);
            // QBE does not return any results
            ObjectSet result = container.queryByExample(new Pilot1("Kimi Raikonen"));
            System.out.println(
"Test QBE on class with member initialization in constructor");
            listResult(result);
        }
    }
}

```

```

        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}

```

2. This example is similar to the previous one, but uses a class derived from the class in test1:

```

QBEExample.java: test2
private static void test2() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            // Pilot1Derived derives the constructor with initialisation
            Pilot1Derived pilot = new Pilot1Derived("Kimi Raikkonen");
            container.store(pilot);
            // QBE does not return any results
            ObjectSet result = container.queryByExample(
                new Pilot1Derived("Kimi Raikonen"));
            System.out.println("Test QBE on class with member " +
                " initialization in ancestor constructor");
            listResult(result);
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}

```

3. This example uses QBE against a class with static member initialization:

```

QBEExample.java: test3
private static void test3() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            // Pilot2 uses static initialization of points member
            Pilot2 pilot = new Pilot2("Kimi Raikkonen");
            container.store(pilot);
            // QBE does not return any results
            ObjectSet result = container.queryByExample(new Pilot2("Kimi Raikonen"));
            System.out.println("Test QBE on class with static member initialization");
            listResult(result);
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        }
    }
}

```

```

        } finally {
            closeDatabase();
        }
    }
}

```

4. This example is similar to test3, but a derived class is used:

```

QBEExample.java: test4
private static void test4()  {
    ObjectContainer container = database();
    if (container != null)  {
        try {
            // Pilot2Derived is derived from class with static initialization of points member
            Pilot2Derived pilot = new Pilot2Derived("Kimi Raikkonen");
            container.store(pilot);
            // QBE does not return any results
            ObjectSet result = container.queryByExample(
new Pilot2Derived("Kimi Raikonen"));
            System.out.println(
"Test QBE on class derived from a class with static member initialization");
            listResult(result);
        } catch (Exception ex)  {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}

```

Pilot2

```

Pilot2.java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */

package com.db4odoc.qbe;

public class Pilot2  {
    private String name;

    private int points = 100;

    public Pilot2(String name)  {
        this.name = name;
    }

    public String getName()  {
        return name;
    }
}

```

```

    }

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

    public int getPoints()  {
        return points;
    }

    public String toString()  {
        return name + "/" + points;
    }

}

```

Pilot2Derived

```

Pilot2Derived.java
package com.db4odoc.qbe;

public class Pilot2Derived extends Pilot2  {

    public Pilot2Derived(String name)  {
        super(name);
    }

}

```

Pilot1Derived

```

Pilot1Derived.java
package com.db4odoc.qbe;

public class Pilot1Derived extends Pilot1  {

    public Pilot1Derived(String name)  {
        super(name);
        // TODO Auto-generated constructor stub
    }

}

```

/p>

Pilot1

```

Pilot1.java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */

package com.db4odoc.qbe;

```

```

public class Pilot1 {
    private String name;

    private int points;

    public Pilot1(String name) {
        this.name = name;
        this.points = 100;
    }

    public String getName() {
        return name;
    }

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

    public int getPoints() {
        return points;
    }

    public String toString() {
        return name + "/" + points;
    }
}

```

Native Queries

Wouldn't it be nice to pose queries in the programming language that you are using? Wouldn't it be nice if all your query code was 100% typesafe, 100% compile-time checked and 100% refactorable? Wouldn't it be nice if the full power of object-orientation could be used by calling methods from within queries? All mentioned above is achievable by using Native Queries or [LINQ](#) (if you are developing in .NET3.5)

Native queries are the main db4o query interface and they are the recommended way to query databases from your application for all platforms except .NET3.5 where LINQ is preferable. 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.

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

Principle

Native Queries provide the ability to run one or more lines of code against all instances of a class. Native query expressions should return true to mark specific instances as part of the result set. db4o will attempt to [optimize native query](#) expressions and use [internal query processor](#) to run them against indexes and without instantiating actual objects, where this is possible.

Simple Example

Let's look at how a simple native query will look like in some of the programming languages and dialects that db4o supports:

Java5:

```
NQExample.java: primitiveQuery
private static void primitiveQuery(ObjectContainer container) {
    List<Pilot> pilots = container.query(new com.db4o.query.Predicate<Pilot>() {
        public boolean match(Pilot pilot) {
            return pilot.getPoints() == 100;
        }
    });
}
```

Java1.2-1.4:

```
PrimitiveExample.java: primitiveQuery
public static void primitiveQuery(ObjectContainer db) {
    List pilots = db.query(new Predicate() {
        public boolean match(Pilot pilot) {
            return pilot.getPoints() == 100;
        }
    });
}
```

Java1.1:

```
PrimitiveExample.java: primitiveQuery1
public static void primitiveQuery1(ObjectContainer db) {
    List pilots = db.query(new PilotHundredPoints());
}
```

```
PilotHundredPoints.java
/** Copyright (C) 2004 - 2006 Versant Inc. http://www.db4o.com */
import com.db4o.query.Predicate;

public class PilotHundredPoints extends Predicate {
    public boolean match(Pilot pilot) {
        return pilot.getPoints() == 100;
    }
}
```

A side note on the above syntax:

For all dialects without support for generics, Native Queries work by convention. A class that extends the Predicate class is expected to have a boolean #match() or #Match() method with one parameter to describe the class extent:

Java:

```
boolean match(Pilot candidate);
```

When using native queries, don't forget that modern integrated development environments (IDEs) can do all the typing work around the native query expression for you, if you use templates and autocompletion.

Here is how to configure a Native Query template with Eclipse 3.1:

From the menu, choose Window + Preferences + Java + Editor + Templates + New

As the name type "nq". Make sure that "java" is selected as the context on the right. Paste the following into the pattern field:

```
List <${extent}> list = db.query(new Predicate <${extent}> () { public boolean match(${extent} candidate){ return true; }});
```

Now you can create a native query with three keys: n + q + Control-Space.

Similar features are available in most modern IDEs.

For more information see [Native Query Syntax](#).

Advanced Example

For complex queries, the native syntax is very precise and quick to write. Let's compare to a SODA query that finds all pilots with a given name or a score within a given range:

```
NQExample.java: storePilots
private static void storePilots(ObjectContainer container) {
    container.store(new Pilot("Michael Schumacher", 100));
    container.store(new Pilot("Rubens Barrichello", 99));
}
```

```
NQExample.java: retrieveComplexSODA
private static void retrieveComplexSODA(ObjectContainer container) {
    Query query = container.query();
    query.constrain(Pilot.class);
    Query pointQuery = query.descend("points");
```

```

query.descend("name").constrain("Rubens Barrichello").or(
    pointQuery.constrain(new Integer(99)).greater().and(
        pointQuery.constrain(new Integer(199))
        .smaller())));
ObjectSet result = query.execute();
listResult(result);
}

```

Here is how the same query will look like with native query syntax, fully accessible to auto-completion, refactoring and other IDE features, fully checked at compile time:

```

NQExample.java: advancedQuery
private static void advancedQuery(ObjectContainer container)  {
    List<Pilot> result = container.query(new com.db4o.query.Predicate<Pilot>()  {
        public boolean match(Pilot pilot)  {
            return pilot.getPoints() > 99
                && pilot.getPoints() < 199
                || pilot.getName().equals(
                    "Rubens Barrichello");
        }
    });
}

```

Arbitrary Code

Basically that's all there is to know about native queries to be able to use them efficiently. In principle you can run arbitrary code as native queries, you just have to be very careful with side effects - especially those that might affect persistent objects.

Let's run an example that involves some more of the language features available.

```

NQExample.java: retrieveArbitraryCodeNQ
private static void retrieveArbitraryCodeNQ(
    ObjectContainer container)  {
    final int[] points =  { 1, 100 };
    ObjectSet result = container.query(new com.db4o.query.Predicate<Pilot>()  {
        public boolean match(Pilot pilot)  {
            for (int i = 0; i < points.length; i++)  {
                if (pilot.getPoints() == points[i])  {
                    return true;
                }
            }
            return pilot.getName().startsWith("Rubens");
        }
    });
    listResult(result);
}

```

Native Query Performance

One drawback of native queries has to be pointed out: under the hood db4o tries to analyze native queries to convert them to SODA. This is not possible for all queries. For some queries it is very difficult to analyze the flowgraph. In this case db4o will have to instantiate some of the persistent objects to actually run the native query code. db4o will try to analyze parts of native query expressions to keep object instantiation to the minimum.

The development of the native query optimization processor will be an ongoing process in a close dialog with the db4o community. Feel free to contribute your results and your needs by providing feedback to our db4o forums.

The current state of the query optimization process is detailed in the chapter on [Native Query Optimization](#)

With the current implementation, all above examples will run optimized, except for the "Arbitrary Code" example - we are working on it.

Note:

- on Java Native Query optimization requires bloat.jar, db4o-nqopt.jar and db4o-instrumentation.jar to be present in the classpath;
- on .NET Native Query optimization requires a reference to Db4obects.Db4o.Instrumentation.dll and Db4obects.Db4o.NativeQueries.dll in your project.

Native Query Syntax

Native queries give you a choice of several implementations. This topic will explain how and where each implementation should be used.

More Reading:

- [Java Syntax](#)

Java Syntax

1. The following example shows how to use Native Query to retrieve all the objects of the specified type. This syntax can be used with or without generics.

```
NQSyntaxExamples.java: querySyntax1
private static void querySyntax1()  {
    ObjectContainer container = database();
    if (container != null)  {
        try  {
            List<Pilot> result = container.query(Pilot.class);
            container.ext().configure().freespace();
```

```

        listResult(result);
    } catch (Exception ex) {
        System.out.println("System Exception: " + ex.getMessage());
    } finally {
        closeDatabase();
    }
}
}
}

```

2. In this example an anonymous predicate class is used to specify the query parameter:

```

NQSyntaxExamples.java: querySyntax2
private static void querySyntax2() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            List<Pilot> result = container.query(new Predicate<Pilot>() {
                public boolean match(Pilot pilot) {
                    // each Pilot is included in the result
                    return true;
                }
            });
            listResult(result);
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}

```

3. This example shows how to use NQ to sort the query results; anonymous predicate and anonymous comparator are used:

```

NQSyntaxExamples.java: querySyntax3
private static void querySyntax3() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            List<Pilot> result = container.query(new Predicate<Pilot>() {
                public boolean match(Pilot pilot) {
                    // each Pilot is included in the result
                    return true;
                }
            }, new Comparator<Pilot>() {
                public int compare(Pilot pilot1, Pilot pilot2) {
                    return pilot1.getPoints() - pilot2.getPoints();
                }
            });
            listResult(result);
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        }
    }
}

```

```

        } finally {
            closeDatabase();
        }
    }
}

```

4. The following example shows a Native Query using external comparator and predicate. This can be useful when comparator and predicate are widely used and logically do not belong to the querying class:

```
NQSyntaxExamples.java: PilotPredicate
private static class PilotPredicate extends Predicate<Pilot> {
    public boolean match(Pilot pilot) {
        // each Pilot is included in the result
        return true;
    }
}
```

```
NQSyntaxExamples.java: PilotComparator
private static class PilotComparator implements Comparator<Pilot> {
    public int compare(Pilot pilot1, Pilot pilot2) {
        return pilot1.getPoints() - pilot2.getPoints();
    }
}
```

```
NQSyntaxExamples.java: querySyntax4
private static void querySyntax4() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            List<Pilot> result = container.query(new PilotPredicate(),
                new PilotComparator());
            listResult(result);
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}
```

5.
6. In java versions without generics syntax is similar:

```
NQSyntaxExamples.java: querySyntax5
private static void querySyntax5() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            List result = container.query(new Predicate() {
                public boolean match(Object obj) {
                    // each Pilot is included in the result

```

```
        if (obj instanceof Pilot)  {
            return true;
        }
        return false;
    }
}, new Comparator()  {
    public int compare(Object object1, Object object2)  {
        return ((Pilot) object1).getPoints()
            - ((Pilot) object2).getPoints();
    }
});
listResult(result);
} catch (Exception ex)  {
    System.out.println("System Exception: " + ex.getMessage());
} finally  {
    closeDatabase();
}
}
}
```

7. For java versions that do not provide Comparator interface (<JDK1.2) db4o provides QueryComparator interface with the same functionality:

```
NQSyntaxExamples.java: querySyntax6
private static void querySyntax6()  {
    // this example will only work with java versions without
    // generics support
    ObjectContainer container = database();
    if (container != null)  {
        try  {
            List result = container.query(new Predicate()  {
                public boolean match(Object obj)  {
                    // each Pilot is included in the result
                    if (obj instanceof Pilot)  {
                        return true;
                    }
                    return false;
                }
            }, new QueryComparator()  {
                public int compare(Object pilot1, Object pilot2)  {
                    return ((Pilot) pilot1).getPoints()
                        - ((Pilot) pilot2).getPoints();
                }
            });
            listResult(result);
        } catch (Exception ex)  {
            System.out.println("System Exception: " + ex.getMessage());
        } finally  {
            closeDatabase();
        }
    }
}
```

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](#) or [LINQ](#) 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 with a single `constrain` call
- [Native Queries](#) use bytecode and IL analysis to [convert](#) to SODA
- [LINQ](#) also uses [IL analysis](#)

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

More Reading:

- [Building SODA Queries](#)
- [SODA Query Graph](#)
- [SODA Query API](#)
- [SODA Query Engine](#)

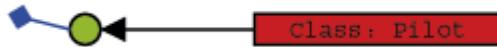
Building SODA Queries

Let's see how simple QBE queries are expressed with SODA. A new `Query` object is created through the `query` method of the `ObjectContainer` and we can add `Constraint` instances to it. To find all `Pilot` instances, we constrain the query with the `Pilot` class object.

```
QueryExample.java: retrieveAllPilots
private static void retrieveAllPilots()  {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try  {
        Query query = container.query();
        query.constrain(Pilot.class);
        ObjectSet result = query.execute();
        listResult(result);
    } finally {
        container.close();
    }
}
```

Basically, we are exchanging our 'real' prototype for a meta description of the objects we'd like to hunt down: a **query graph** made up of query nodes and constraints. A query node is a placeholder for a candidate object, a constraint decides whether to add or exclude candidates from the result.

Our first simple graph looks like this.



We're just asking any candidate object (here: any object in the database) to be of type Pilot to aggregate our result.

To retrieve a pilot by name, we have to further constrain the candidate pilots by descending to their name field and constraining this with the respective candidate String.

```
QueryExample.java: retrievePilotByName
private static void retrievePilotByName(ObjectContainer container) {
    Query query = container.query();
    query.constrain(Pilot.class);
    query.descend("name").constrain("Michael Schumacher");
    ObjectSet result = query.execute();
    listResult(result);
}
```

What does 'descend' mean here? Well, just as we did in our 'real' prototypes, we can attach constraints to child members of our candidates.



So a candidate needs to be of type Pilot and have a member named 'name' that is equal to the given String to be accepted for the result.

Note that the class constraint is not required: If we left it out, we would query for all objects that contain a 'name' member with the given value. In most cases this will not be the desired behavior, though.

Finding a pilot by exact points is analogous. We just have to cross the Java primitive/object divide.

```
QueryExample.java: retrievePilotByExactPoints
private static void retrievePilotByExactPoints(
    ObjectContainer container) {
    Query query = container.query();
```

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

SODA Query Graph

SODA allows to create a query graph of any complexity by joining field object constraints. SODA usage is very generic and can be applied to any objects and conditions. The following 5 steps can be used (all steps are optional and can be repeated logically):

- create a root of a query object

```
Java: Query queryRootNode = db.query();
```

- add constraints to any node anywhere

```
Java: queryRootNode.constrain(Foo.class);
```

- navigate from any query node to any subordinate node

```
Java: Query barNode = queryRootNode.descend("bar");
```

- add further constraints to any node

```
Java: Constraint barConstraint = barNode.constrain(5);
```

- set the evaluation mode of a node

```
Java: barConstraint().greater();
```

The API is very powerful with a small number of method calls.

The "backward" order to add constraints first and to specify the evaluation mode as a second step allows plugging complex objects into a query.

SODA Query API

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

Not

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

```
QueryExample.java: retrieveByNegation
private static void retrieveByNegation(ObjectContainer container) {
    Query query = container.query();
    query.constrain(Pilot.class);
    query.descend("name").constrain("Michael Schumacher").not();
    ObjectSet result = query.execute();
    listResult(result);
}
```

And

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

```
QueryExample.java: retrieveByConjunction
private static void retrieveByConjunction(ObjectContainer container) {
    Query query = container.query();
    query.constrain(Pilot.class);
    Constraint constr = query.descend("name").constrain(
        "Michael Schumacher");
    query.descend("points").constrain(new Integer(10))
        .and(constr);
    ObjectSet result = query.execute();
    listResult(result);
}
```

Or

```
QueryExample.java: retrieveByDisjunction
private static void retrieveByDisjunction(ObjectContainer container) {
    Query query = container.query();
    query.constrain(Pilot.class);
    Constraint constr = query.descend("name").constrain(
        "Michael Schumacher");
    query.descend("points").constrain(new Integer(99)).or(constr);
    ObjectSet result = query.execute();
    listResult(result);
}
```

Greater, Smaller, Equal <=>

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

Return pilots with more than 99 points:

```
QueryExample.java: retrieveByComparison
private static void retrieveByComparison(ObjectContainer container)  {
    Query query = container.query();
    query.constrain(Pilot.class);
    query.descend("points").constrain(new Integer(99)).greater();
    ObjectSet result = query.execute();
    listResult(result);
}
```

Using Default Values

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

```
QueryExample.java: retrieveByDefaultFieldValue
private static void retrieveByDefaultFieldValue(
    ObjectContainer container)  {
    Pilot somebody = new Pilot("Somebody else", 0);
    container.store(somebody);
    Query query = container.query();
    query.constrain(Pilot.class);
    query.descend("points").constrain(new Integer(0));
    ObjectSet result = query.execute();
    listResult(result);
    container.delete(somebody);
}
```

String Comparisons

Like

This is an equivalent to SQL "like" operator:

```
SodaExample.java: testLike
public static void testLike()  {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database();
    if (container != null)  {
        try {
            Pilot pilot = new Pilot("Test Pilot1", 100);
            container.store(pilot);
```

```

pilot = new Pilot("Test Pilot2", 102);
container.store(pilot);

// Simple query
Query query1 = container.query();
query1.constrain(Pilot.class);
query1.descend("name").constrain("est");
ObjectSet result = query1.execute();
listResult(result);

// Like query
Query query2 = container.query();
query2.constrain(Pilot.class);
// All pilots with the name containing "est" will be retrieved
query2.descend("name").constrain("est").like();
result = query2.execute();
listResult(result);

} catch (Db4oException ex) {
    System.out.println("Db4o Exception: " + ex.getMessage());
} catch (Exception ex) {
    System.out.println("System Exception: " + ex.getMessage());
} finally {
    closeDatabase();
}
}

}
}

```

startsWith, endsWith

Compares a beginning or ending of a string:

```

SodaExample.java: testStartsEnds
public static void testStartsEnds()  {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database();
    if (container != null)  {
        try  {
            Pilot pilot = new Pilot("Test Pilot0", 100);
            container.store(pilot);
            pilot = new Pilot("Test Pilot1", 101);
            container.store(pilot);
            pilot = new Pilot("Test Pilot2", 102);
            container.store(pilot);

            Query query = container.query();
            query.constrain(Pilot.class);
            query.descend("name").constrain("T0").endsWith(false).not();
            // query.descend("name").constrain("Pil").startsWith(true);
            ObjectSet result = query.execute();
            listResult(result);
        } catch (Db4oException ex)  {

```

```

        System.out.println("Db4o Exception: " + ex.getMessage());
    } catch (Exception ex) {
        System.out.println("System Exception: " + ex.getMessage());
    } finally {
        closeDatabase();
    }
}
}
}

```

Case Insensitive Queries

By default all string querying functions use case-sensitive comparison. `startsWith` and `endsWith` allow to switch between comparison modes using a parameter. However, if you need a case-insensitive comparison for `like`, `equals` or `contains` queries, it is recommended to use [Native Queries](#) as SODA does not provide such an option.

Contains

Allows to retrieve objects constraining by included value (collection). If applied to a string will behave as "Like".

```

SodaExample.java: testContains
public static void testContains() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database();
    if (container != null) {
        try {
            ArrayList list = new ArrayList();
            Pilot pilot1 = new Pilot("Test 1", 1);
            list.add(pilot1);
            Pilot pilot2 = new Pilot("Test 2", 2);
            list.add(pilot2);
            Team team = new Team("Ferrari", list);
            container.store(team);

            Query query = container.query();
            query.constrain(Team.class);
            query.descend("pilots").constrain(pilot2).contains();
            ObjectSet result = query.execute();
            listResult(result);
        } catch (Db4oException ex) {
            System.out.println("Db4o Exception: " + ex.getMessage());
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}

```

Identity Comparison

db4o database identity can also be used as a constraint. In this case only objects with the same database instance will be retrieved:

```
SodaExample.java: testIdentity
public static void testIdentity() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database();
    if (container != null) {
        try {
            Pilot pilot = new Pilot("Test Pilot1", 100);
            Car car = new Car("BMW", pilot);
            container.store(car);
            // Change the name, the pilot instance stays the same
            pilot.setName("Test Pilot2");
            // create a new car
            car = new Car("Ferrari", pilot);
            container.store(car);

            // Simple Query:
            Query query1 = container.query();
            query1.constrain(Car.class);
            query1.descend("_pilot").constrain(pilot);
            ObjectSet result = query1.execute();
            listResult(result);

            // identity query:
            Query query2 = container.query();
            query2.constrain(Car.class);
            // All cars having pilot with the same database identity
            // will be retrieved. As we only created Pilot object once
            // it should mean all car objects
            query2.descend("_pilot").constrain(pilot).identity();
            result = query2.execute();
            listResult(result);
        } catch (Db4oException ex) {
            System.out.println("Db4o Exception: " + ex.getMessage());
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}
```

Sorting Results

It is also possible to have db4o sort the results.

```
QueryExample.java: retrieveSorted
```

```

private static void retrieveSorted(ObjectContainer container) {
    Query query = container.query();
    query.constrain(Pilot.class);
    query.descend("name").orderAscending();
    ObjectSet result = query.execute();
    listResult(result);
    query.descend("name").orderDescending();
    result = query.execute();
    listResult(result);
}

```

Joining Constraints

In some situations you need a way to arbitrarily join constraints, for example:

(a and b) or (c and d)

In this case you will just need to add new constraints to the root constraint and join them in advance using Constraint syntax:

```

QueryExample.java: retrieveByJoinedConstraints
private static void retrieveByJoinedConstraints(ObjectContainer container) {
    Query query = container.query();
    query.constrain(Pilot.class);
    Constraint constr1 = query.descend("name").constrain(
        "Michael Schumacher");
    Constraint constr2 = query.descend("points").constrain(
        new Integer(100)).and(constr1);

    Constraint constr3 = query.descend("name").constrain(
        "Rubens Barrichello");

    query.descend("points").constrain(
        new Integer(99)).and(constr3).or(constr2);

    ObjectSet result = query.execute();
    listResult(result);
}

```

All these techniques can be combined arbitrarily, of course. Please try it out. There still may be cases left where the predefined query API constraints may not be sufficient - don't worry, you can always let db4o run any arbitrary code that you provide in an Evaluation. Evaluations will be discussed in a [Evaluations chapter](#).

SODA Query Engine

Query Processor operates in two stages:

1. In a first stage it creates a tree of candidate object IDs using an index. The Best index is searched, which means a field index for field constraints (if available), which returns the smallest candidate result set. Field index allows to create often smaller candidate tree, already filtered on some criteria. If the best index is not found, class index is used to create a candidate tree of all instances of matching class or classes.
2. In a second stage, all candidates are run against the SODA processor to run all constraints against all objects, whether the field of a constraint is indexed or not.

The first stage of the query processor operates directly on BTrees. BTrees are used for class indexes (always), and field indexes (configurable).

BTree algebra may create unions and intersections (and, or, greater, smaller) between BTree ranges, working with pointers into BTrees without ever having to scan through all index entries. A BTree node points to the object's file positions through object ID.

Query processing starts from evaluating leaf nodes of the query graph and then going on to the top level filtering or joining the results.

For a simple query:

```
query.constrain(Pilot.class);
```

class index will be used to get ID's of all Pilot objects.

For a more complex query:

```
query.constrain(Pilot.class);

Constraint constr = query.descend("name").constrain("Michael Schumacher");

query.descend("points").constrain(new Integer(100)).and(constr);
```

In case there are indexes on "name" and "points" fields, Pilot candidates will be created from "points" indexes having value of 100 and "name" indexes with value "Michael Schumacher". With BTree indexes this search will be really fast. If there is no index all Pilot IDs will be used as candidates. After the candidates are collected all existing constraints will be tested against them to filter out results that do not match the criteria.

Inheritance

In the case of inherited classes of interfaces:

- querying against parent class or interface will include results for all sub-classes/implementations;
- field indexes can be used and should be defined against the parent class fields

Query Modes

What is the best way to process queries? How to get the optimum performance for your application needs?

Those of you, who have dealt with query time and memory constraints (situations, when query result is bigger than the memory available, or when query time is longer than the whole functional operation) should know how searching a suitable solution might affect the whole application design and implementation.

Luckily db4o takes most of the trouble for itself. There are 3 query modes allowing to fine tune the balance between speed, memory consumption and availability of the results:

- immediate
- lazy
- snapshot

The query mode can be set using:

Java:

```
configuration.queries().evaluationMode(QueryEvaluationMode)
```

Let's look at each of them in more detail.

More Reading:

- [Immediate Queries](#)
- [Lazy Queries](#)
- [Snapshot Queries](#)

Immediate Queries

This is the default query mode: the whole query result is evaluated upon query execution and object IDs list is produced as a result.

```
QueryModesExample.java: testImmediateQueries
private static void testImmediateQueries()  {
    System.out.println(
"Testing query performance on 10000 pilot objects in Immediate mode");
    fillUpDB(10000);
    Configuration configuration = Db4o.newConfiguration();
    configuration.queries().evaluationMode(QueryEvaluationMode.IMMEDIATE);
    ObjectContainer container = Db4o.openFile(configuration, DB4O_FILE_NAME);
    try  {
        QueryStats stats = new QueryStats();
```

```

stats.connect(container);
Query query = container.query();
query.constrain(Pilot.class);
query.descend("points").constrain(99).greater();
query.execute();
long executionTime = stats.executionTime();
System.out.println("Query execution time: "
    + executionTime);
} finally {
    container.close();
}
}

```

Obviously object evaluation takes some time and in a case of big resultsets 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 or even minutes.

This mode makes the whole objects result set available at once - ID 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 resultset 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:

```

QueryModesExample.java: testImmediateChanged
private static void testImmediateChanged()  {
    System.out
        .println("Testing immediate mode with field changes");
    fillUpDB(10);
    Configuration configuration = Db4o.newConfiguration();
    configuration.queries().evaluationMode(QueryEvaluationMode.IMMEDIATE);
    ObjectContainer container = Db4o.openFile(configuration, DB4O_FILE_NAME);
    try {
        Query query1 = container.query();
        query1.constrain(Pilot.class);
        query1.descend("points").constrain(5).smaller();
        ObjectSet result1 = query1.execute();

        // change field
        Query query2 = container.query();
        query2.constrain(Pilot.class);
        query2.descend("points").constrain(2);
        ObjectSet result2 = query2.execute();
        Pilot pilot2 = (Pilot) result2.queryByExample(0);
        pilot2.addPoints(22);
        container.store(pilot2);
        listResult(result1);
    } finally {

```

```
        container.close();
    }
}
```

Immediate Mode Pros And Cons

Pros:

- 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).

Cons:

- 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

In Lazy Querying mode objects are not evaluated at all, instead of this an iterator is created against the best index found. Further query processing (including all index processing) will happen when the user application iterates through the resulting `ObjectSet`. This allows you to get the first query results almost immediately.

```
QueryModesExample.java: testLazyQueries
private static void testLazyQueries() {
    System.out
        .println(
    "Testing query performance on 10000 pilot objects in Lazy mode");
    fillUpDB(10000);
    Configuration configuration = Db4o.newConfiguration();
    configuration.queries().evaluationMode(QueryEvaluationMode.LAZY);
    ObjectContainer container = Db4o.openFile(configuration, DB4O_FILE_NAME);
    try {
        QueryStats stats = new QueryStats();
        stats.connect(container);
        Query query = container.query();
        query.constrain(Pilot.class);
        query.descend("points").constrain(99).greater();
        query.execute();
        long executionTime = stats.executionTime();
        System.out.println("Query execution time: "
            + executionTime);
    } finally {
}
```

```
        container.close();
    }
}
```

In addition to the very fast execution this method also ensures very small memory consumption, as lazy queries do not need an intermediate representation as a set of IDs in memory. With this approach a lazy query ObjectSet does not have to cache a single object or ID. The memory consumption for a query is practically zero, no matter how large the resultset 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:

```
QueryModesExample.java: testLazyConcurrent
private static void testLazyConcurrent()  {
    System.out
        .println("Testing lazy mode with concurrent modifications");
    fillUpDB(10);
    Configuration configuration = Db4o.newConfiguration();
    configuration.queries().evaluationMode(QueryEvaluationMode.LAZY);
    ObjectContainer container = Db4o.openFile(configuration, DB4O_FILE_NAME);
    try  {
        Query query1 = container.query();
        query1.constrain(Pilot.class);
        query1.descend("points").constrain(5).smaller();
        ObjectSet result1 = query1.execute();

        Query query2 = container.query();
        query2.constrain(Pilot.class);
        query2.descend("points").constrain(1);
        ObjectSet result2 = query2.execute();
        Pilot pilotToDelete = (Pilot) result2.queryByExample(0);
        System.out.println("Pilot to be deleted: "
            + pilotToDelete);
        container.delete(pilotToDelete);
        Pilot pilot = new Pilot("Tester", 2);
        System.out.println("Pilot to be added: " + pilot);
        container.store(pilot);

        System.out
            .println("Query result after changing from the same transaction");
        listResult(result1);
    } finally  {
        container.close();
    }
}
```

Pros and Cons for Lazy Queries

Pros:

- 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.

Cons:

- Lazy queries check candidates when iterating through the resulting `ObjectSet`. 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 a way one would work 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` resultset of the query.

```
QueryModesExample.java: testSnapshotQueries
private static void testSnapshotQueries() {
    System.out.println(
        "Testing query performance on 10000 pilot objects in Snapshot mode");
    fillUpDB(10000);
    Configuration configuration = Db4o.newConfiguration();
    configuration.queries().evaluationMode(QueryEvaluationMode.SNAPSHOT);
    ObjectContainer container = Db4o.openFile(configuration, DB4O_FILE_NAME);
    try {

```

```

QueryStats stats = new QueryStats();
stats.connect(container);
Query query = container.query();
query.constrain(Pilot.class);
query.descend("points").constrain(99).greater();
query.execute();
long executionTime = stats.executionTime();
System.out.println("Query execution time: "
    + executionTime);
} finally {
    container.close();
}
}

```

Snapshot queries ensure better performance than Immediate queries, but the performance will depend on the size of the resultset.

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 (compare the results of this code snippet to the one from [Lazy Queries](#) topic):

```

QueryModesExample.java: testSnapshotConcurrent
private static void testSnapshotConcurrent() {
    System.out
        .println("Testing snapshot mode with concurrent modifications");
    fillUpDB(10);
    Configuration configuration = Db4o.newConfiguration();
    configuration.queries().evaluationMode(QueryEvaluationMode.SNAPSHOT);
    ObjectContainer container = Db4o.openFile(configuration, DB4O_FILE_NAME);
    try {
        Query query1 = container.query();
        query1.constrain(Pilot.class);
        query1.descend("points").constrain(5).smaller();
        ObjectSet result1 = query1.execute();

        Query query2 = container.query();
        query2.constrain(Pilot.class);
        query2.descend("points").constrain(1);
        ObjectSet result2 = query2.execute();
        Pilot pilotToDelete = (Pilot) result2.queryByExample(0);
        System.out.println("Pilot to be deleted: "
            + pilotToDelete);
        container.delete(pilotToDelete);
        Pilot pilot = new Pilot("Tester", 2);
        System.out.println("Pilot to be added: " + pilot);
        container.store(pilot);

        System.out
            .println("Query result after changing from the same transaction");
        listResult(result1);
    }
}

```

```
    } finally {
        container.close();
    }
}
```

Pros and Cons for Snapshot Queries

Pros:

- 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.

Cons:

- The entire candidate index will be loaded into memory. It will stay there until the query ObjectSet is garbage collected. In a C/S setup, the memory will be used on the server side

Client/Server applications with the risk of concurrent modifications should prefer Snapshot mode to avoid side effects from other transactions.

SODA Evaluations

Evaluations provide user-defined custom constraints as an instrument to run any arbitrary code in a SODA query. It gives you maximum power over the query execution, but certainly has its own drawbacks. Let's have a closer look.

More Reading:

- [Evaluation API](#)
- [Using Evaluations](#)
- [Drawbacks](#)

Evaluation API

The evaluation API consists of two interfaces, *Evaluation* and *Candidate*. Evaluation implementations are implemented by the user and injected into a query. During a query, they will be called from db4o with a candidate instance in order to decide whether to include it into the current (sub-)result.

The Evaluation interface contains a single method only:

Java:

```
public void evaluate(Candidate candidate)
```

This will be called by db4o to check whether the object encapsulated by this candidate should be included into the current candidate set.

The Candidate interface provides three methods:

Java:

```
public Object getObject()  
public void include(boolean flag)  
public ObjectContainer objectContainer()
```

An Evaluation implementation may call getObject() to retrieve the actual object instance to be evaluated, it may call include() to instruct db4o whether or not to include this object in the current candidate set, and finally it may access the current database directly by calling objectContainer().

Using Evaluations

For a simple example, let's take a Car class keeping a history of SensorReadout instances in a List member. Now imagine that we wanted to retrieve all cars that have assembled an even number of history entries. A quite contrived and seemingly trivial example, however, it gets us into trouble: Collections are transparent to the query API, it just 'looks through' them at their respective members.

So how can we get this done? Let's implement an Evaluation that expects the objects passed in to be instances of type Car and checks their history size.

```
EvenHistoryEvaluation.java  
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */  
package com.db4odoc.evaluations;  
  
import com.db4o.query.*;  
  
public class EvenHistoryEvaluation implements Evaluation {  
    public void evaluate(Candidate candidate) {  
        Car car=(Car)candidate.getObject();  
        candidate.include(car.getHistory().size() % 2 == 0);  
    }  
}
```

To test it, let's add two cars with history sizes of one and two respectively:

```
EvaluationExample.java: storeCars
```

```

private static void storeCars(ObjectContainer container) {
    Pilot pilot1 = new Pilot("Michael Schumacher", 100);
    Car car1 = new Car("Ferrari");
    car1.setPilot(pilot1);
    car1.snapshot();
    container.store(car1);
    Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
    Car car2 = new Car("BMW");
    car2.setPilot(pilot2);
    car2.snapshot();
    car2.snapshot();
    container.store(car2);
}

```

and run our evaluation against them:

```

EvaluationExample.java: queryWithEvaluation
private static void queryWithEvaluation(ObjectContainer container) {
    Query query = container.query();
    query.constrain(Car.class);
    query.constrain(new EvenHistoryEvaluation());
    ObjectSet result = query.execute();
    listResult(result);
}

```

Drawbacks

While evaluations offer you another degree of freedom for assembling queries, they come at a certain cost: As you may already have noticed from the example, evaluations work on the fully instantiated objects, while 'normal' queries peek into the database file directly. So there's a certain performance penalty for the object instantiation, which is wasted if the object is not included into the candidate set.

Another restriction is that, while 'normal' queries can bypass encapsulation and access candidates' private members directly, evaluations are bound to use their external API, just as in the language itself.

One last hint for Java users: Evaluations are expected to be serializable for client/server operation. So be careful when implementing them as (anonymous) inner classes and keep in mind that those will carry an implicit reference to their surrounding class and everything that belongs to it. Best practice is to always implement evaluations as normal top level or static inner classes.

Sorting Query Results

Often applications need to present query results in a sorted order. There are several ways to achieve this with db4o.

This Pilot class will be used in the following examples:

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

package com.db4odoc.sorting;

public class Pilot {
    private String name;

    private int points;

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

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

    public String getName() {
        return name;
    }

    public int getPoints() {
        return points;
    }

    public String toString() {
        if (points == 0) {
            return name;
        } else {
            return name + "/" + points;
        }
    }
}
```

The database will be filled with the following Pilot objects:

```
SortingExample.java: setObjects
public static void setObjects() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {

```

```

        for (int i = 0; i < 10; i++)  {
            for (int j = 0; j < 5; j++)  {
                Pilot pilot = new Pilot("Pilot #" + i, j + 1);
                container.store(pilot);
            }
        }
    } finally {
    container.close();
}
}

```

The following topics discuss some of the possible methods and point out their advantages and disadvantages.

More Reading:

- [SODA Sorting](#)
- [Native Query Sorting](#)
- [Evaluation Results Sorting](#)

SODA Sorting

SODA query API gives you a possibility to sort any field in ascending or descending order and combine sorting of different fields. For example, let's retrieve the objects of the Pilot class [saved before](#), sorting "points" field in descending order and "name" field in ascending.

```

SortingExample.java: getObjectsSODA
public static void getObjectsSODA()  {

    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try  {
        Query query = container.query();
        query.constrain(Pilot.class);
        query.descend("name").orderAscending();
        query.descend("points").orderDescending();
        long t1 = System.currentTimeMillis();
        ObjectSet result = query.execute();
        long t2 = System.currentTimeMillis();
        long diff = t2 - t1;
        System.out.println("Time to query and sort with SODA: "
            + diff + " ms.");
        listResult(result);
    } finally  {
        container.close();
    }
}

```

```

SortingExample.vb: GetObjectsSODA
Private Shared Sub GetObjectsSODA()
    Dim db As IObjectContainer = Db4oFactory.OpenFile(Db4oFileName)
    Try
        Dim query As IQuery = db.Query
        query.Constrain(GetType(Pilot))
        query.Descend("_name").OrderAscending()
        query.Descend("_points").OrderDescending()
        Dim dt1 As DateTime = DateTime.UtcNow
        Dim result As IObjectSet = query.Execute
        Dim dt2 As DateTime = DateTime.UtcNow
        Dim diff As TimeSpan = dt2 - dt1
        Console.WriteLine("Time to query and sort with SODA: " +
+ diff.TotalMilliseconds.ToString() + " ms.")
        ListResult(result)
    Finally
        db.Close()
    End Try
End Sub

```

Obvious disadvantages of this method:

- limitations of SODA queries (not type-safe and not compile-time checked);
- limitations if sorting mechanism (only alphabetical for strings, numerical for numbers and object id for objects)

The valuable advantage of this method is its high performance.

Native Query Sorting

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

Java:

```
<TargetType> ObjectSet<TargetType> query(Predicate<TargetType> predicate, QueryComparator<TargetType> comparator)
```

In order to get the same results as in [SODA Sorting](#) example we will write the following code:

```

SortingExample.java: getObjectsNQ
public static void getObjectsNQ() {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        long t1 = System.currentTimeMillis();
        ObjectSet result = container.query(
            new Predicate<Pilot>() {
                public boolean match(Pilot pilot) {
                    return true;
                }
            }
        );
    }
}
```

```

        }
    }, new QueryComparator<Pilot>() {
        public int compare(Pilot p1, Pilot p2) {
            int result = p1.getPoints()
                - p2.getPoints();
            if (result == 0) {
                return p1.getName().compareTo(
                    p2.getName());
            } else {
                return -result;
            }
        }
    });
long t2 = System.currentTimeMillis();
long diff = t2 - t1;
System.out
    .println("Time to execute with NQ and comparator: "
        + diff + " ms.");
listResult(result);
} finally {
    container.close();
}
}
}

```

Advantages of NQ sorting:

- type-safe queries and sorting, compile-time checking;
- ability to implement any custom sorting algorithm

The main disadvantage is decreased performance as at current stage [optimization](#) of sorted Native Queries is not possible.

Evaluation Results Sorting

In some cases you may find it necessary to use Evaluations. There is no standard sorting API for the Evaluation results. But you can sort the returned result set using standard Java collection API.

For example, let's retrieve the objects of the Pilot class [saved before](#), selecting only pilots with even points and sorting them according to their name:

```

SortingExample.java: getObjectsEval
public static void getObjectsEval() {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        long t1 = System.currentTimeMillis();
        Query query = container.query();
        query.constrain(Pilot.class);
        query.constrain(new Evaluation() {

```

```

public void evaluate(Candidate candidate) {
    Pilot pilot = (Pilot) candidate.getObject();
    candidate.include(pilot.getPoints() % 2 == 0);
}
});
List<Pilot> result = new ArrayList<Pilot>(query.execute());
Collections.sort(result, new Comparator<Pilot>() {
    public int compare(Pilot p1, Pilot p2) {
        return p1.getName().compareTo(p2.getName());
    }
});
long t2 = System.currentTimeMillis();
long diff = t2 - t1;
System.out
    .println("Time to execute with Evaluation query and collection sorting: "
        + diff + " ms.");
listResult(result);
} finally {
    container.close();
}
}
}

```

This sorting method can be used to sort query results when the sorting can not be added to the query (Evaluations, QBE).

Custom Query Comparator

Db4o allows using a custom comparator for query processing. This can be useful for replacing the standard way of selecting query results, for example, when a non-standard attribute type should be compared as string or integer

The usage is best shown on an example.

Let's create a custom class containing string information:

```

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

class MyString  {

    private String _string;

    public MyString(String string)  {
        _string = string;
    }

    public String toString()  {
        return _string;
    }
}

```

```
    }  
}
```

MyString class will be used for an attribute in the following class:

```
Record.java  
/** Copyright (C) 2007 Versant Inc. http://www.db4o.com */  
package com.db4odoc.comparing;  
  
public class Record {  
    private MyString _record;  
  
    public Record(String record) {  
        _record = new MyString(record);  
    }  
  
    public String toString() {  
        return _record.toString();  
    }  
}
```

Let's save some Record objects to the database:

```
CompareExample.java: storeRecords  
private static void storeRecords(Configuration configuration) {  
    new File(FILENAME).delete();  
    ObjectContainer container = Db4o.openFile(configuration, FILENAME);  
    try {  
        Record record = new Record("Michael Schumacher, points: 100");  
        container.store(record);  
        record = new Record("Rubens Barrichello, points: 98");  
        container.store(record);  
        record = new Record("Kimi Raikonen, points: 55");  
        container.store(record);  
    } finally {  
        container.close();  
    }  
}
```

Selecting the query results, we need to compare string values contained in MyString objects. This can be configured with the following setting:

```
CompareExample.java: configure
```

```

private static Configuration configure() {
    Configuration configuration = Db4o.newConfiguration();
    configuration.objectClass(MyString.class).compare(new ObjectAttribute() {
        public Object attribute(Object original) {
            if (original instanceof MyString) {
                return ((MyString) original).toString();
            }
            return original;
        }
    });
    return configuration;
}

```

This piece of code registers an attribute provider for special query behavior.

The query processor will compare the object returned by the attribute provider instead of the actual object, both for the constraint and the candidate persistent object.

The querying code will look like this:

```

CompareExample.java: checkRecords
private static void checkRecords(Configuration configuration) {
    ObjectContainer container = Db4o.openFile(configuration, FILENAME);
    try {
        Query q = container.query();
        q.constrain(new Record("Rubens"));
        q.descend("_record").constraints().contains();
        ObjectSet result = q.execute();
        listResult(result);
    } finally {
        container.close();
    }
}

```

Using query comparator feature we can change the standard way of selecting query results. This can be helpful for:

- querying user types using simple string or numeric representation
- encapsulating additional logic into querying algorithm.

Transparent Persistence

One of db4o goals is to make database transparent to the application logic. Would not it be nice after an initial registering of an object with a database with a single store() method to leave the database to manage all the future object modification? This idea went a long way from initial vague

thoughts to a final implementation in db4o version 7.1 and was named Transparent Persistence. Let's look at the implementation in a more detail.

More Reading:

- [Transparent Persistence Implementation](#)
- [TP Enhanced Example](#)
- [Detailed Example](#)
- [Collection Example](#)
- [Transparent Persistence For Java Collections](#)

Transparent Persistence Implementation

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

- classes available for Transparent Persistence should implement Activatable interface, which allows to bind an object in the reference cache to the current object container.
- persistent object 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 object 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, TP 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.

Note that Transparent Persistence is based on Transparent Activation, so it is strongly recommended to study [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.add(new TransparentPersistenceSupport());
```

2. Implement Activatable/IActivatable 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. 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 Enhanced Example

As it was mentioned [before](#) TP uses the same technology as TA. Due to that enhancement strategies and tools are the same for Transparent Persistence. Actually TP enhancement automatically includes TA enhancement, as TP relies on object activation before modification.

On Java side enhancement can be done at build-time by using a customized ant build script. For more information see [TP Enhancement On Java](#).

TP Enhancement On Java

TP 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;
}
```

Now we will store a linked list of 10 elements:

```

TPEexample.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:

```

TPEexample.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
-->
<target name="compile">
    <javac fork="true" destdir="${basedir}/bin">
        <classpath>
            <path element location="${basedir}/lib/db4o-7.1.26.8872-java5.jar" />
        </classpath>
        <src path="${basedir}/src" />
        <include name="**/*.java" />
    </javac>
</target>

</project>

```

The build script relies on several jars:

- ant.jar - [Ant](#) library
- bloat-1.0.jar - bloat bytecode instrumentation library
- db4o-7.12-instrumentation.jar - db4o instrumentation library on top of bloat
- db4o-7.12-java5.jar - db4o jar
- db4o-7.12-taj.jar - db4o transparent activation support
- db4o-7.12-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-7.12-java5.jar

java com.db4odoc.tpbuilddate.TPExample

```

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
}

```

Detailed Example

Let's look at a primitive example, demonstrating manual implementation of Activatable/IActivatable interface for TP. We will use a class similar to the [one](#) used in Transparent Activation chapters.

```

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

package com.db4odoc.tpxexample;

import com.db4o.activation.*;
import com.db4o.ta.*;

public class SensorPanel implements Activatable  {

    private Object _sensor;

    private SensorPanel _next;

    /**/*activator registered for this class*/
    transient Activator _activator;

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

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

    /**/*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;
    }
}

```

```

}

// end bind

/**/*Call the registered activator to activate the next level,
 * the activator remembers the objects that were already
 * activated and won't activate them twice.
 */
public void activate(ActivationPurpose purpose)  {
    if (_activator == null)
        return;
    _activator.activate(purpose);
}
// end activate

public SensorPanel getNext()  {
    /**/*activate direct members*/
    activate(ActivationPurpose.READ);
    return _next;
}
// end getNext

public Object getSensor()  {
    /**/*activate direct members*/
    activate(ActivationPurpose.READ);
    return _sensor;
}
// end getSensor

public void setSensor(Object sensor)  {
    /**/*activate for persistence*/
    activate(ActivationPurpose.WRITE);
    _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
}

```

Note, that the only place where we can modify SensorPanel members is `setSensor` method/`Sensor` property, and that is where `activate` method is added.

Now we will only need to add Transparent Activation support:

```

TPEexample.java: configureTA
private static Configuration configureTA() {
    Configuration configuration = Db4o.newConfiguration();
    // add TA support
    configuration.add(new TransparentActivationSupport());
    return configuration;
}

```

Initial storing of the objects is done as usual with a single `store` call:

```

TPEexample.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();
        }
    }
}

```

Now we can test how Transparent Persistence helped us to keep the code simple. Let's select all elements from the linked SensorPanel list, modify them and store. As you remember default update depth is one, so without TP, we would have to store each member of the linked list (SensorPanel) separately. With TP enabled there is absolutely nothing to do: commit call will find all activatable objects and store those that were modified.

```

        TPEexample.java: testTransparentPersistence
private static void testTransparentPersistence()  {
    storeSensorPanel();
    Configuration configuration = configureTA();

    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();
        }
    }
}
}

```

That's all. The benefits that we've got:

- clean and friendly to refactorings code;
- performance benefit: only modified objects are stored;
- hassle-free development.

Collection Example

In the [previous example](#) we reviewed how Transparent Persistence should be used with simple types. Let's now look how it is done with the collections.

In order to make collections TP Activatable you will need to use db4o-specific ArrayList4 collection. This collection implements .NET/Java collection interfaces, therefore it can be easily integrated with your code. The following class contains a collection of Pilot objects:

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

package com.db4odoc.tpxexample;

import java.util.*;

import com.db4o.activation.*;
import com.db4o.collections.*;
import com.db4o.ta.*;

public class Team implements Activatable {

    private List<Pilot> _pilots = new ArrayList4<Pilot>();

    String _name;

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

```

        _activator.activate(purpose);
    }

    public void addPilot(Pilot pilot)  {
        // activate before adding new pilots
        activate(ActivationPurpose.WRITE);
        _pilots.add(pilot);
    }

    public void listAllPilots()  {
        // activate before printing the collection members
        activate(ActivationPurpose.READ);

        for (Iterator<Pilot> iter = _pilots.iterator(); iter.hasNext();) {
            Pilot pilot = (Pilot) iter.next();
            System.out.println(pilot);
        }
    }

    List<Pilot> getPilots()  {
        activate(ActivationPurpose.READ);
        return _pilots;
    }
}

```

You can see that except for using `ArrayList`, the implementation follows the same rules as in the previous [simple example](#).

Its usage has no surprises as well:

```

TPCollectionExample.java: storeCollection
private static void storeCollection()  {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database(configureTP());
    if (container != null)  {
        try  {
            Team team = new Team();
            for (int i = 0; i < 10; i++)  {
                team.addPilot(new Pilot("Pilot #" + i));
            }
            container.store(team);
        } catch (Exception ex)  {
            ex.printStackTrace();
        } finally  {
            closeDatabase();
        }
    }
}

```

```

TPCollectionExample.java: testCollectionPersistence
private static void testCollectionPersistence()  {

```

```

storeCollection();
ObjectContainer container = database(configureTP());
if (container != null) {
    try {
        Team team = (Team) container.queryByExample(new Team()).next();
        // this method will activate all the members in the collection
        Iterator<Pilot> it = team.getPilots().iterator();
        while (it.hasNext()) {
            Pilot p = it.next();
            p.setName("Modified: " + p.getName());
        }
        team.addPilot(new Pilot("New pilot"));
        // explicitly commit to persist changes
        container.commit();
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally {
        // If TP changes were not committed explicitly,
        // they would be persisted with the #close call
        closeDatabase();
    }
}
// reopen the database and check the changes
container = database(configureTP());
if (container != null) {
    try {
        Team team = (Team) container.queryByExample(new Team()).next();
        team.listAllPilots();
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally {
        closeDatabase();
    }
}
}

```

So the only thing you should remember when using TP with collections is to use ArrayList4 instead of platform-specific collection.

Working With Structured Objects

In real world objects are referenced by each other creating deep reference structures.

This chapter will give you an overview of how db4o deals with structured objects.

For an example we will use a simple model, where Pilot class is referenced from Car class.

```

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

```

```
package com.db4odoc.structured;

public class Pilot {
    private String name;
    private int points;

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

    public int getPoints() {
        return points;
    }

    public void addPoints(int points) {
        this.points += points;
    }

    public String getName() {
        return name;
    }

    public String toString() {
        return name + "/" + points;
    }
}
```

```
Car.java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */
package com.db4odoc.structured;

public class Car {
    private String model;
    private Pilot pilot;

    public Car(String model) {
        this.model = model;
        this.pilot = null;
    }

    public Pilot getPilot() {
        return pilot;
    }

    public void setPilot(Pilot pilot) {
        this.pilot = pilot;
    }

    public String getModel() {
        return model;
    }
}
```

```

public String toString() {
    return model + "[" + pilot + "]";
}
}

```

More Reading:

- [Retrieving Structured Objects](#)
- [Updating Structured Objects](#)
- [Deleting Structured Objects](#)

Retrieving Structured Objects

QBE

To retrieve all cars a simple 'blank' prototype can be used.

```

StructuredExample.java: retrieveAllCarsQBE
private static void retrieveAllCarsQBE(ObjectContainer container) {
    Car proto = new Car(null);
    ObjectSet result = container.queryByExample(proto);
    listResult(result);
}

```

You can also query for all pilots, of course.

```

StructuredExample.java: retrieveAllPilotsQBE
private static void retrieveAllPilotsQBE(ObjectContainer container) {
    Pilot proto = new Pilot(null, 0);
    ObjectSet result = container.queryByExample(proto);
    listResult(result);
}

```

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

```

StructuredExample.java: retrieveCarByPilotQBE
private static void retrieveCarByPilotQBE(ObjectContainer container) {
    Pilot pilotproto = new Pilot("Rubens Barrichello", 0);
    Car carproto = new Car(null);
    carproto.setPilot(pilotproto);
    ObjectSet result = container.queryByExample(carproto);
}

```

```
        listResult(result);
    }
```

What about retrieving a pilot by car? You simply don't need that -if you already know the car, you can simply access the pilot field directly.

```
StructuredExample.java: retrieveCarByPilotQBE
private static void retrieveCarByPilotQBE(ObjectContainer container) {
    Pilot pilotproto = new Pilot("Rubens Barrichello", 0);
    Car carproto = new Car(null);
    carproto.setPilot(pilotproto);
    ObjectSet result = container.queryByExample(carproto);
    listResult(result);
}
```

Native Queries

Using native queries with constraints on deep structured objects is straightforward, you can do it just like you would in plain other code. Let's constrain our query to only those cars driven by a Pilot with a specific name:

```
StructuredExample.java: retrieveCarsByPilotNameNative
private static void retrieveCarsByPilotNameNative(
    ObjectContainer container) {
    final String pilotName = "Rubens Barrichello";
    ObjectSet results = container.query(new Predicate<Car>() {
        public boolean match(Car car) {
            return car.getPilot().getName().equals(pilotName);
        }
    });
    listResult(results);
}
```

SODA Query API

In order to use SODA for querying for a car given its pilot's name you have to descend two levels into our query.

```
StructuredExample.java: retrieveCarByPilotNameQuery
private static void retrieveCarByPilotNameQuery(
    ObjectContainer container) {
    Query query = container.query();
    query.constrain(Car.class);
```

```

query.descend("pilot").descend("name").constrain(
    "Rubens Barrichello");
ObjectSet result = query.execute();
listResult(result);
}

```

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

```

StructuredExample.java: retrieveCarByPilotProtoQuery
private static void retrieveCarByPilotProtoQuery(
    ObjectContainer container) {
    Query query = container.query();
    query.constrain(Car.class);
    Pilot proto = new Pilot("Rubens Barrichello", 0);
    query.descend("pilot").constrain(proto);
    ObjectSet result = query.execute();
    listResult(result);
}

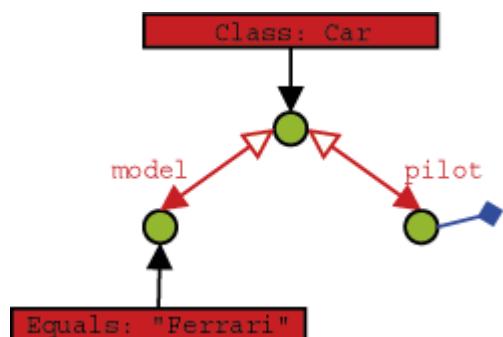
```

Descending into a query provides you with another query. Starting out from a query root you can descend in multiple directions. In practice this is the same as ascending from one child to a parent and descending to another child. The queries turn one-directional references in objects into true relations. Here is an example that queries for "a Pilot that is being referenced by a Car, where the Car model is 'Ferrari'" :

```

StructuredExample.java: retrievePilotByCarModelQuery
private static void retrievePilotByCarModelQuery(
    ObjectContainer container) {
    Query carquery = container.query();
    carquery.constrain(Car.class);
    carquery.descend("model").constrain("Ferrari");
    Query pilotquery = carquery.descend("pilot");
    ObjectSet result = pilotquery.execute();
    listResult(result);
}

```



Updating Structured Objects

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

```
StructuredExample.java: updateCar
private static void updateCar(ObjectContainer container) {
    ObjectSet result = container.query(new Predicate<Car>() {
        public boolean match(Car car) {
            return car.getModel().equals("Ferrari");
        }
    });
    Car found = (Car) result.next();
    found.setPilot(new Pilot("Somebody else", 0));
    container.store(found);
    result = container.query(new Predicate<Car>() {
        public boolean match(Car car) {
            return car.getModel().equals("Ferrari");
        }
    });
    listResult(result);
}
```

Let's modify the pilot, too.

```
StructuredExample.java: updatePilotSingleSession
private static void updatePilotSingleSession(
    ObjectContainer container) {
    ObjectSet result = container.query(new Predicate<Car>() {
        public boolean match(Car car) {
            return car.getModel().equals("Ferrari");
        }
    });
    Car found = (Car) result.next();
    found.getPilot().addPoints(1);
    container.store(found);
    result = container.query(new Predicate<Car>() {
        public boolean match(Car car) {
            return car.getModel().equals("Ferrari");
        }
    });
    listResult(result);
}
```

Nice and easy, isn't it? But there is something that is not obvious in this example. Let's see what happens if we split this task in two separate db4o sessions: In the first we modify our pilot and update his car:

```
StructuredExample.java: updatePilotSeparateSessionsPart1
```

```

private static void updatePilotSeparateSessionsPart1(
    ObjectContainer container) {
    ObjectSet result = container.query(new Predicate<Car>() {
        public boolean match(Car car) {
            return car.getModel().equals("Ferrari");
        }
    });
    Car found = (Car) result.next();
    found.getPilot().addPoints(1);
    container.store(found);
}

```

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

```

StructuredExample.java: updatePilotSeparateSessionsPart2
private static void updatePilotSeparateSessionsPart2(
    ObjectContainer container) {
    ObjectSet result = container.query(new Predicate<Car>() {
        public boolean match(Car car) {
            return car.getModel().equals("Ferrari");
        }
    });
    listResult(result);
}

```

If you will execute this code you will see that Pilot's points are not changed. What's happening here and what can we do to fix it?

Update Depth

Imagine a complex object with many members that have many members themselves. When updating this object, db4o would have to update all its children, grandchildren, etc. This poses a severe performance penalty and will not be necessary in most cases - sometimes, however, it will. So, in our previous update example, we were modifying the Pilot child of a Car object. When we saved the change, we told db4o to save our Car object and assumed that the modified Pilot would be updated. But we were modifying and saving in the same manner as we were in the first update sample, so why did it work before? The first time we made the modification, db4o never actually had to retrieve the modified Pilot; it returned the same one that was still in memory that we modified, but it never actually updated the database. Restarting the application would show that the value was unchanged. To be able to handle this dilemma as flexible as possible, db4o introduces the concept of update depth to control how deep an object's member tree will be traversed on update. The default update depth for all objects is 0, meaning that only primitive and String members will be updated, but changes in object members will not be reflected. db4o provides means to control

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

```
StructuredExample.java: updatePilotSeparateSessionsImprovedPart1
private static Configuration updatePilotSeparateSessionsImprovedPart1() {
    Configuration configuration = Db4o.newConfiguration();
    configuration.objectClass("com.db4o.f1.chapter2.Car")
        .cascadeOnUpdate(true);
    return configuration;
}
```

```
StructuredExample.java: updatePilotSeparateSessionsImprovedPart2
private static void updatePilotSeparateSessionsImprovedPart2(
    ObjectContainer container) {
    ObjectSet result = container.query(new Predicate<Car>() {
        public boolean match(Car car) {
            return car.getModel().equals("Ferrari");
        }
    });
    Car found = (Car) result.next();
    found.getPilot().addPoints(1);
    container.store(found);
}
```

```
StructuredExample.java: updatePilotSeparateSessionsImprovedPart3
private static void updatePilotSeparateSessionsImprovedPart3(
    ObjectContainer container) {
    ObjectSet result = container.query(new Predicate<Car>() {
        public boolean match(Car car) {
            return car.getModel().equals("Ferrari");
        }
    });
    listResult(result);
}
```

You can also achieve expected results using:

- > Java:
 - 1. ExtObjectContainer#set(object, depth) to update exact amount of referenced fields
 - 2. Use configuration.objectClass(clazz).updateDepth(depth) setting to define sufficient update depth for a specific object
- 4. Use global setting for all the persisted objects:

Java:

```
configuration.updateDepth(depth);
```

However global updateDepth is not flexible enough for real-world objects having different depth of reference structures.

ATTENTION: Setting global update depth to the maximum value will result in serious performance penalty. Please, use this setting ONLY for debug purposes.

Note, that container configuration must be set before the container is opened and/or passed to the `openFile/openClient/openServer` method.

Deleting Structured Objects

As we have already seen, we call `delete()` on objects to get rid of them.

```
StructuredExample.java: deleteFlat
private static void deleteFlat(ObjectContainer container) {
    ObjectSet result = container.query(new Predicate<Car>() {
        public boolean match(Car car) {
            return car.getModel().equals("Ferrari");
        }
    });
    Car found = (Car) result.next();
    container.delete(found);
    result = container.queryByExample(new Car(null));
    listResult(result);
}
```

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

```
StructuredExample.java: retrieveAllPilotsQBE
private static void retrieveAllPilotsQBE(ObjectContainer container) {
    Pilot proto = new Pilot(null, 0);
    ObjectSet result = container.queryByExample(proto);
    listResult(result);
}
```

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

Recursive Deletion

The problem of recursive deletion (and its solution, too) is quite similar to the [update](#) problem. Let's configure db4o to delete a car's pilot, too, when the car is deleted.

```
StructuredExample.java: deleteDeepPart1
private static Configuration deleteDeepPart1() {
    Configuration configuration = Db4o.newConfiguration();
    configuration.objectClass("com.db4o.f1.chapter2.Car")
        .cascadeonDelete(true);
    return configuration;
}
```

```
StructuredExample.java: deleteDeepPart2
private static void deleteDeepPart2(ObjectContainer container) {
    ObjectSet result = container.query(new Predicate<Car>() {
        public boolean match(Car car) {
            return car.getModel().equals("BMW");
        }
    });
    Car found = (Car) result.next();
    container.delete(found);
    result = container.query(new Predicate<Car>() {
        public boolean match(Car car) {
            return true;
        }
    });
    listResult(result);
}
```

Cascade on delete configuration only affects the direct children of the deleted object. For example, if Pilot class will have a field `idBook` of Class `IdBook`, `IdBook` instances won't be deleted if the configuration above will be used. In order to delete `IdBook` instances you will need to enable `cascadeonDelete` for `Pilot` class.

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

Another way to organize cascaded deletion is using [callbacks](#). The callbacks allow you to specify explicitly, which objects are to be deleted.

Recursive Deletion Revisited

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

```
StructuredExample.java: deleteDeepRevisited
```

```

private static void deleteDeepRevisited(ObjectContainer container) {
    ObjectSet result = container.query(new Predicate<Pilot>() {
        public boolean match(Pilot pilot) {
            return pilot.getName().equals("Michael Schumacher");
        }
    });
    Pilot pilot = (Pilot) result.next();
    Car car1 = new Car("Ferrari");
    Car car2 = new Car("BMW");
    car1.setPilot(pilot);
    car2.setPilot(pilot);
    container.store(car1);
    container.store(car2);
    container.delete(car2);
    result = container.query(new Predicate<Car>() {
        public boolean match(Car car) {
            return true;
        }
    });
    listResult(result);
}

```

```

StructuredExample.java: retrieveAllPilots
private static void retrieveAllPilots(ObjectContainer container) {
    ObjectSet result = container.queryByExample(Pilot.class);
    listResult(result);
}

```

Currently db4o does **not** check whether objects to be deleted are referenced anywhere else, so please be very careful when using this feature. However it is fairly easy to implement referential checking on deletion using `deleting()` callback. See [Callbacks chapter](#) for more information.

Activation

Activation is a db4o mechanism, which controls object's fields instantiation. Why is it necessary? Let's look at an example of a database, that has a Tree structure, i.e. there is one Root object, which has N nodes, each node in its turn has K subnodes etc. Let the whole structure has M levels. What happens when you run a query, retrieving the root object? All the sub-objects will have to be created in the memory. If N,K,M are large numbers, you will most probably end up with the Out-OfMemory exception.

Luckily db4o does not behave like this - when query retrieves objects, their fields are loaded into memory (activated in db4o terms) only to a certain depth - 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 (for classes) or to default values (for primitive types).

Activation occurs in the following cases:

1. ObjectSet#next() is called on an ObjectSet retrieved in a query;
2. Object is activated explicitly with ObjectContainer#activate(object, depth);
3. built-in collection element is accessed;
4. for the environment collections (.NET, Java) their members are activated automatically, when the collection is activated, using at least depth 1 for lists and depth 2 for maps.

More Reading:

- [Global Activation Settings](#)
- [Object-Specific Activation](#)
- [Transparent Activation Framework](#)

Global Activation Settings

Java:

```
configuration.activationDepth(activationDepth)
```

configures global activation depth, which will be used for all objects instead of the default value. This method should be called before opening a database file.

Java:

```
configuration.activationDepth(activationDepth)
```

has a similar effect, but the setting will be applied to the specific ObjectContainer and can be changed for the open database file.

```
ActivationExample.java: testActivationConfig
private static void testActivationConfig()  {
    storeSensorPanel();
    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().activationDepth(1);
    ObjectContainer container = Db4oEmbedded.openFile(configuration,
        DB4O_FILE_NAME);
    try  {
        System.out.println("Object container activation depth = 1");
        ObjectSet<SensorPanel> result = container
            .queryByExample(new SensorPanel(1));
        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 {
    container.close();
}
}

```

By configuring db4o you can have full control over activation behavior. The two extremes:

- using an activationDepth of Integer.MAX_VALUE lets you forget about manual activation, but does not give you the best performance and memory footprint;
- using an activationDepth of 0 and activating and deactivating all objects manually keeps memory consumption extremely low, but needs more coding and attention.

Object-Specific Activation

You can tune up activation settings for specific classes with the following methods:

Java:

```

configuration.common().objectClass("yourClass")
.minimumActivationDepth(minimumDepth)

configuration.common().objectClass("yourClass")
.maximumActivationDepth(maximumDepth)

```

Cascading the activation depth to member fields, the depth value is reduced by one for the field. If the depth exceeds the maximumDepth specified for the class of the object, it is reduced to the maximumDepth. If the depth value is lower than the minimumDepth it is raised to the minimumDepth.

```

ActivationExample.java: testMaxActivate
private static void testMaxActivate() {
    storeSensorPanel();
    // note that the maximum is applied to the retrieved root object and
    // limits activation
    // further down the hierarchy
    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().objectClass(SensorPanel.class)
        .maximumActivationDepth(2);

    ObjectContainer container = Db4oEmbedded.openFile(configuration,
        DB4O_FILE_NAME);

```

```

try  {
    System.out.println("Maximum activation depth = 2 (default = 5)");
    ObjectSet<SensorPanel> result = container
        .queryByExample(new SensorPanel(1));
    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  {
    container.close();
}
}

```

```

ActivationExample.java: testMinActivate
private static void testMinActivate()  {
    storeSensorPanel();
    // note that the minimum applies for *all* instances in the hierarchy
    // the system ensures that every instantiated List object will have it's
    // members set to a depth of 1
    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().objectClass(SensorPanel.class)
        .minimumActivationDepth(1);
    ObjectContainer container = Db4oEmbedded.openFile(configuration,
        DB4O_FILE_NAME);
    try  {
        System.out.println("Minimum activation depth = 1");
        ObjectSet<SensorPanel> result = container
            .queryByExample(new SensorPanel(1));
        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  {
        container.close();
    }
}

```

You can set up automatic activation for specific objects or fields:

```
Java: configuration.common().objectClass("yourClass").cascadeOnActivate (bool)  
con-  
figuration.common().objectClass("yourClass").objectField("field").cascadeOnActivate(bool)
```

Cascade activation will retrieve the whole object graph, starting from the specified object(field). This setting can lead to increased memory consumption.

```
ActivationExample.java: testCascadeActivate  
private static void testCascadeActivate() {  
    storeSensorPanel();  
    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();  
    configuration.common().objectClass(SensorPanel.class)  
        .cascadeOnActivate(true);  
    ObjectContainer container = Db4oEmbedded.openFile(configuration,  
        DB4O_FILE_NAME);  
    try {  
        System.out.println("Cascade activation");  
        ObjectSet<SensorPanel> result = container  
            .queryByExample(new SensorPanel(1));  
        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 {  
            container.close();  
        }  
    }
```

An alternative to cascade activation can be manual activation of objects:

```
Java: ObjectContainer#activate(object, activationDepth);
```

Manual deactivation may be used to save memory:

```
Java: ObjectContainer#deactivate(object, activationDepth);
```

These 2 methods give you an excellent control over object activation, but they obviously need more attention from the application side.

```
ActivationExample.java: testActivateDeactivate
private static void testActivateDeactivate() {
    storeSensorPanel();
    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().activationDepth(0);
    ObjectContainer container = Db4oEmbedded.openFile(configuration,
        DB4O_FILE_NAME);
    try {
        System.out.println("Object container activation depth = 0");
        ObjectSet<SensorPanel> result = container
            .queryByExample(new SensorPanel(1));
        System.out.println("Sensor1:");
        listResult(result);
        SensorPanel sensor1 = (SensorPanel) result.queryByExample(0);
        testActivated(sensor1);

        System.out.println("Sensor1 activated:");
        container.activate(sensor1, 4);
        testActivated(sensor1);

        System.out.println("Sensor5 activated:");
        result = container.queryByExample(new SensorPanel(5));
        SensorPanel sensor5 = (SensorPanel) result.queryByExample(0);
        container.activate(sensor5, 4);
        listResult(result);
        testActivated(sensor5);

        System.out.println("Sensor1 deactivated:");
        container.deactivate(sensor1, 5);
        testActivated(sensor1);

        // DANGER !!!!.
        // If you use deactivate with a higher value than 1
        // make sure that you know whereto members might branch
        // Deactivating list1 also deactivated list5
        System.out.println("Sensor 5 AFTER DEACTIVATE OF Sensor1.");
        testActivated(sensor5);
    } finally {
        container.close();
    }
}
```

Transparent Activation Framework

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) 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.

More Reading:

- [TA Implementation](#)
- [TA Enhanced Example](#)
- [Detailed Example](#)
- [Collection Example](#)
- [Object Types In TA](#)
- [TA Diagnostics](#)
- [TA For Public Fields](#)

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.
- To instruct db4o to operate in Transparent Activation mode, call: Db4o.configure().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 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](#)
- [Collection Example](#)

TA Enhanced Example

As it was mentioned [before](#) you can inject TA 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 `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-7.12-tools.jar
- db4o-7.12-taj.jar
- db4o-7.12-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

TA 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.

Let's look at an example.

We will use [SensorPanel](#) class from the [Activation](#) example.

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(
        TAInstrumentationExample.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();
        }
    }
}
```

```
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 the [previous topic](#) we discussed how TA can be enabled on classes while they are loaded. In this topic we will look at even more convenient and performant way of enhancing classes to support TA: during application build time.

For our example we will take the same classes as in the [previous example](#), with the exception of TAInstrumentationRunner class, which won't be needed for build-time enhancement. Basically, we will move all the enhancing functionality of TAInstrumentationRunner into the build script. For this example we will create an ant script, which should be run after the classes (or jar) is built.

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

<target name="buildall">

  <!-- 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-7.12-instrumentation.jar
 - db4o-7.12-java5.jar
 - db4o-7.12-taj.jar
 - db4o-7.12-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-7.12-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:

- Add NQ optimization in the same enhancer task
- Use ClassFilter to select classes for enhancement
- Use regex to select classes for enhancement
- 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](#).

SensorPanel

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

package com.db4odoc.tpbuilddate;

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
}

```

Detailed Example

Let's look at the manual Transparent Activation implementation. This example will help you to understand how TA is implemented under the hood.

We will take the example class from the [Activation](#) chapter and modify it to enable TA:

- implement `Activatable` interface (`bind` method)
- add `_activator` variable to keep the current activator;
- create `activate()` method;
- call `activate()` method each time field objects are required.

```

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

package com.db4odoc.taexamples;

import com.db4o.activation.*;
import com.db4o.ta.*;

public class SensorPanelTA /*must implement Activatable for TA*/implements Activatable {

    private Object _sensor;

    private SensorPanelTA _next;

    /**/*activator registered for this class*/
    transient Activator _activator;

    public SensorPanelTA()  {
        // default constructor for instantiation
    }

    public SensorPanelTA(int value)  {
        _sensor = new Integer(value);
    }
}

```

```

/**/*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;
}

/**/*Call the registered activator to activate the next level,
 * the activator remembers the objects that were already
 * activated and won't activate them twice.
 */
public void activate(ActivationPurpose purpose)  {
    if (_activator == null)
        return;
    _activator.activate(purpose);
}

public SensorPanelTA getNext()  {
    /**/*activate direct members*/
    activate(ActivationPurpose.READ);
    return _next;
}

public Object getSensor()  {
    /**/*activate direct members*/
    activate(ActivationPurpose.READ);
    return _sensor;
}

public SensorPanelTA createList(int length)  {
    return createList(length, 1);
}

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

protected SensorPanelTA newElement(int value)  {
    return new SensorPanelTA(value);
}

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

```

```
    }  
}
```

As you can see from the example class we can call `activate()` to activate the field objects. However, transparent activation is currently not available directly on field variables, you will have to wrap them into methods.

Let's create a configuration for transparent activation:

```
TAEexample.java: configureTA  
private static Configuration configureTA()  {  
    Configuration configuration = Db4o.newConfiguration();  
    // add TA support  
    configuration.add(new TransparentActivationSupport());  
    // activate TA diagnostics to reveal the classes that are not TA-enabled.  
    activateDiagnostics(configuration);  
    return configuration;  
}
```

We can test TA using the configuration above:

```
TAEexample.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  
            SensorPanelTA list = new SensorPanelTA().createList(10);  
            container.store(list);  
        } finally  {  
            closeDatabase();  
        }  
    }  
}
```

```
TAEexample.java: testActivation  
private static void testActivation()  {  
    storeSensorPanel();  
    Configuration configuration = configureTA();  
  
    ObjectContainer container = database(configuration);  
    if (container != null)  {  
        try  {  
            ObjectSet result = container.queryByExample(new SensorPanelTA(1));  
            listResult(result);  
            if (result.size() > 0)  {  
                SensorPanelTA sensor = (SensorPanelTA) result.queryByExample(0);  
            }  
        } catch (Db4oException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```

// the object is a linked list, so each call to next()
// will need to activate a new object
SensorPanelTA next = sensor.getNext();
while (next != null) {
    System.out.println(next);
    next = next.getNext();
}
}
} finally {
    closeDatabase();
}
}
}
}

```

Collection Example

Db4o provides proprietary TA-aware collection implementations:

- `ArrayList4` and `ArrayMap4` for Java (Recommended way to implement collections for Java 5 is using [Transparent Persistence For Java Collections](#))
- `ArrayList4` and `ArrayDictionary4` for .NET

implementations for Map and List interfaces. Both implementations, when instantiated as a result of a query, are transparently activated when internal members are required to perform an operation. Db4o implementations provide an important advantage over JDK collections when running in transparent activation mode, based on the ability to control their activation.

`ArrayList4` implements the generic list interface using an array to store elements. When an `ArrayList4` instances activated all the elements of the array are loaded into memory.

`ArrayMap4` and `ArrayDictionary4` classes implement the Map and `IDictionary` interface respectively using two arrays to store keys and values. The array values are transparently loaded into memory when `ArrayMap4/ArrayDictionary4` instance is activated.

For an example, we will use a `Team` class with a collection of `Pilot` objects:

```

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

package com.db4odoc.tpeexample;

import java.util.*;

import com.db4o.activation.*;
import com.db4o.collections.*;

```

```

import com.db4o.ta.*;

public class Team implements Activatable {
    private List<Pilot> _pilots = new ArrayList4<Pilot>();
    String _name;
    //TA Activator
    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 object fields
    public void activate(ActivationPurpose purpose) {
        if (_activator == null) return;
        _activator.activate(purpose);
    }

    public void addPilot(Pilot pilot) {
        // activate before adding new pilots
        activate(ActivationPurpose.WRITE);
        _pilots.add(pilot);
    }

    public void listAllPilots() {
        // activate before printing the collection members
        activate(ActivationPurpose.READ);

        for (Iterator<Pilot> iter = _pilots.iterator(); iter.hasNext();) {
            Pilot pilot = (Pilot) iter.next();
            System.out.println(pilot);
        }
    }

    List<Pilot> getPilots() {
        activate(ActivationPurpose.READ);
        return _pilots;
    }
}

```

```

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

```

```

package com.db4odoc.taexamples;

import com.db4o.activation.*;
import com.db4o.ta.Activatable;

public class Pilot implements Activatable {

    private String _name;

    transient Activator _activator;

    public Pilot(String name) {
        _name = name;
    }

    // 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 String getName() {
        // even simple String needs to be activated
        activate(ActivationPurpose.READ);
        return _name;
    }

    public String toString() {
        // use getName method, which already contains activation call
        return getName();
    }
}

```

Store and retrieve.

<pre> TAEexample.java: storeCollection private static void storeCollection() { </pre>

```

new File(DB4O_FILE_NAME).delete();
ObjectContainer container = database(configureTA());
if (container != null) {
    try {
        Team team = new Team();
        for (int i = 0; i < 10; i++) {
            team.addPilot(new Pilot("Pilot #" + i));
        }
        container.store(team);
        container.commit();
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally {
        closeDatabase();
    }
}
}

```

```

TAEexample.java: testCollectionActivation
private static void testCollectionActivation() {
    storeCollection();
    ObjectContainer container = database(configureTA());
    if (container != null) {
        try {
            Team team = (Team) container.queryByExample(new Team()).next();
            // this method will activate all the members in the collection
            team.listAllPilots();
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            closeDatabase();
        }
    }
}

```

Object Types In TA

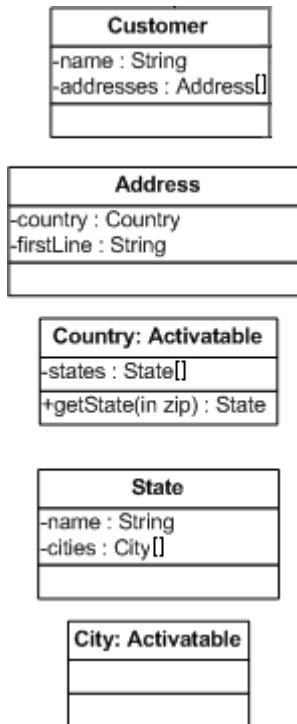
When working in TA enabled environment you must remember that db4o treats Activatable (TA Aware) and non Activatable (other) types differently.

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:



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]
.countries[N].name

c.addresses[0].country.states[N].city    .addresses[0]
.countries[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
4. 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 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 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.

Update Depth

Update depth term is used to determine a number of levels of member objects down the hierarchy, which will be updated automatically, when the top-level object is updated. Understanding of update depth and its performance impact is especially important for working with deep object graphs and collections.

The default update depth value is 0, which means that `objectContainer#set(object)` method will only update the object passed as a parameter and any changes to its member objects will be lost.

Update depth can be changed globally or for a specific class.

Global update depth settings:

Java:

```
configuration.updateDepth(depth)
```

Class-specific update settings:

Java:

```
configuration.objectClass(ListObject.class).cascadeOnUpdate(flag);  
configuration.objectClass(ListObject.class).updateDepth(depth);
```

Here depth parameter specifies the number of member object levels down the hierarchy, which will be updated when the top-level object is saved.

Flag parameter determines whether update operation should be cascaded to all the member objects.

Note, that the recommended setting for the update depth is 0, i.e. the default one. Setting update depth to a higher value inevitably leads to performance downgrade. Storing the required amount of levels can instead be achieved by using a special `store` method:

Java:

```
objectContainer.ext().store(object, updateDepth);
```

Further reading:

[Update Depth for Structured Objects](#)

[Collections Update Depth](#)

Collections Update Depth

When you work with collections you should pay a special attention to your update depth setting, as the performance impact of this setting increases with the number of objects in the collection.

For example let's consider a class like this:

Java:

```
class ListObject {  
    List <DataObject> data;  
}
```

Let's assume that ListObject has a data list of 1000 DataObjects.

Update depth = 1

data field object (List) will be updated if ListObject is saved.

Update depth = 2

data object (List) and all 1000 DataObjects in the list will be updated if ListObject is saved.

It is easy to see that after a certain update depth value all the list objects are getting updated, which produces a serious performance penalty. The following examples show how to avoid the performance drop and still get the expected results.

More Reading:

- [Insert And Remove](#)
- [Updating List Objects](#)

Insert And Remove

When an object is inserted or removed from the list, only the list object needs to be updated, the objects in the list are not going to be changed. That means that for the object from the [previous topic](#) we will need to set update depth = 1.

Let's fill up the database with 2 long lists of objects:

```

ListOperationsExample.java: fillUpDb
private static void fillUpDb()
{
    int listCount = 2;
    int dataCount = 50000;
    long elapsedTime = 0;
    new File(DBFILE).delete();
    ObjectContainer db = Db4o.openFile(DBFILE);
    try
    {
        long t1 = System.currentTimeMillis();

        for (int i = 0; i < listCount; i++)
        {
            ListObject lo = new ListObject();
            lo.setName("list" + String.format("%3d", i));
            for (int j = 0; j < dataCount; j++)
            {
                DataObject dataObject = new DataObject();
                dataObject.setName( "data" + String.format("%5d", j));
                dataObject.setData( System.currentTimeMillis()
+ " ---- Data Object " + String.format("%5d", j));
                lo.getData().add(dataObject);
            }
            db.store(lo);
        }
        long t2 = System.currentTimeMillis();
        elapsedTime = t2 - t1;
    }
    finally
    {
        db.close();
    }
    System.out.println("Completed " + listCount + " lists of "
+ dataCount + " objects each.");
    System.out.println("Elapsed time: " + elapsedTime + " ms.");
}

```

We will remove an object from one list and insert it into another:

```

ListOperationsExample.java: removeInsert
private static void removeInsert()
{
    ObjectContainer db = Db4o.openFile(DBFILE);
    long timeElapsed = 0;
    try
    {
        // set activation depth to 1 for the quickest execution
        db.ext().configure().updateDepth(1);
        List<ListObject> result = db.<ListObject>query(ListObject.class);
        if (result.size() == 2)

```

```

{
    // retrieve 2 ListObjects
    ListObject lo1 = result.queryByExample(0);
    ListObject lo2 = result.queryByExample(1);
    DataObject dataObject = lo1.getData().queryByExample(0);
    // move the first object from the first
    // ListObject to the second ListObject
    lo1.getData().remove(dataObject);
    lo2.getData().add(dataObject);

    System.out.println("Removed from the first list, count is "
+ lo1.getData().size());
    System.out.println("Added to the second list, count is "
+ lo2.getData().size());
    long t1 = System.currentTimeMillis();
    // save ListObjects. UpdateDepth = 1 will ensure that
    // the DataObject list is saved as well
    db.store(lo1);
    db.store(lo2);
    db.commit();
    long t2 = System.currentTimeMillis();
    timeElapsed = t2 - t1;
}
}
finally
{
    db.close();
}
System.out.println("Storing took: " + timeElapsed + " ms.");
}

```

Remember, if an object is removed from the list and is not going to be used any more, it should be deleted manually:

```
db.Delete(dataObject)
```

Let's check if the results are correct:

```

ListOperationsExample.java: checkResults
private static void checkResults()
{
    ObjectContainer db = Db4o.openFile(DBFILE);
    try
    {
        List<ListObject> result = db.<ListObject>query(ListObject.class);
        if (result.size() > 0)
        {
            // activation depth should be enough to activate
            // ListObject, DataObject and its list members
            int activationDepth = 3;
            db.ext().configure().activationDepth(activationDepth);
        }
    }
}
```

```

        System.out.println("Result count was " +
result.size() + " looping with activation depth" + activationDepth);
        for (int i = 0; i < result.size(); i++) {
            ListObject lo = (ListObject)result.queryByExample(i);
            System.out.println("ListObj " + lo.getName() +
" has " + ((lo.getData() == null) ? "<null>" : lo.getData().size()) +" objects");
            System.out.println((lo.getData() != null &&
lo.getData().size() > 0) ? lo.getData().queryByExample(0).toString() : "<null>" + " at index 0");
            System.out.println();
        }
    }
    finally
    {
        db.close();
    }
}

```

You will see that insert/remove operation takes much less time with the correct update depth setting.

Updating List Objects

As we discussed [before](#) updating list members using update depth is quite inefficient. An alternative approach can be retrieving and updating each object from the list separately:

```

ListOperationsExample.java: updateObject
private static void updateObject()
{
    long timeElapsed = 0;

    ObjectContainer db = Db4o.openFile(DBFILE);
    try
    {
        // we can set update depth to 0
        // as we update only the current object
        db.ext().configure().updateDepth(0);
        List<ListObject> result = db.<ListObject>query(ListObject.class);
        if (result.size() == 2)
        {
            ListObject lol = result.queryByExample(0);
            // Select a DataObject for update
            DataObject dataobject = lol.getData().queryByExample(0);
            dataobject.setName("Updated");
            dataobject.setData(System.currentTimeMillis()+
" ---- Updated Object ");

            System.out.println("Updated list " +
lol.getName() + " dataobject " + lol.getData().queryByExample(0));
            long t1 = System.currentTimeMillis();

```

```

        // save only the DataObject. List of DataObjects will
        // automatically include the new value
        db.store(dataobject);
        db.commit();
        long t2 = System.currentTimeMillis();
        timeElapsed = t2 -t1;
    }
}
finally
{
    db.close();
}
System.out.println("Storing took: " + timeElapsed +" ms.") ;
}

```

In this case only the object of interest is updated, which takes much less time than updating the whole list.

Delete Behavior

Db4o delete interface is very simple:

```
objectContainer#Delete(object)
```

Any object, stored to db4o, can be deleted in this way. However, in many cases you may want to delete not only the passed object, but also its dependent objects.

The following topics discuss how to make the deletion easy and effective in different situations:

[Deleting Structured Objects](#)

[Deleting Collection Members](#)

[Deleting Collections](#)

[Cascaded Behavior](#)

[Referential Integrity](#)

Filling The Database

```

ListDeletingExample.java: fillUpDb
private static void fillUpDb(int listCount)  {
    int dataCount = 50;
    long elapsedTime = 0;
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try  {
        long t1 = System.currentTimeMillis();

```

```

for (int i = 0; i < listCount; i++) {
    ListObject lo = new ListObject();
    lo.setName("list" + String.format("%3d", i));
    for (int j = 0; j < dataCount; j++) {
        DataObject dataObject = new DataObject();
        dataObject.setName("data" + String.format("%5d", j));
        dataObject.setData(System.currentTimeMillis()
            + " ---- Data Object " + String.format("%5d", j));
        lo.getData().add(dataObject);
    }
    container.store(lo);
}
long t2 = System.currentTimeMillis();
elapsedTime = t2 - t1;
} finally {
    container.close();
}
System.out.println("Completed " + listCount + " lists of " + dataCount
    + " objects each.");
System.out.println("Elapsed time: " + elapsedTime + " ms.");
}

```

```

ListDeletingExample.vb: FillUpDb
Private Shared Sub FillUpDb(ByVal listCount As Integer)
    Dim dataCount As Integer = 50
    Dim sw As Stopwatch = New Stopwatch
    File.Delete(Db4oFileName)
    Dim db As IObjectContainer = Db4oFactory.OpenFile(Db4oFileName)
    Try
        sw.Start()
        Dim i As Integer = 0
        While i < listCount
            Dim lo As ListObject = New ListObject()
            lo.Name = "list" + i.ToString("00")
            Dim j As Integer = 0
            While j < dataCount
                Dim dataObject As DataObject = New DataObject()
                dataObject.Name = "data" + j.ToString("00000")
                dataObject.Data = DateTime.Now.ToString +
" ---- Data Object " + j.ToString("00000")
                lo.Data.Add(dataObject)
                System.Math.Min(System.Threading.Interlocked.Increment(j), j - 1)
            End While
            db.Store(lo)
            System.Math.Min(System.Threading.Interlocked.Increment(i), i - 1)
        End While
        sw.Stop()
    Finally
        db.Close()
    End Try
    Console.WriteLine("Completed {0} lists of {1} objects each.", _

```

```

listCount, dataCount)
    Console.WriteLine("Elapsed time: {0}", sw.Elapsed.ToString)
End Sub

```

Deleting Collections

As it was discussed in [Deleting Structured Objects](#) chapter, deleting a top-level object does not mean deleting all of the member objects. The same rule applies for collections. The recommendation would be to use cascadeOnDelete setting for a collection, which should be deleted with all its members.

For the following example we will use [DataObject and ListObject](#) classes. The database will be filled up with the [FillUpDb](#) method.

```

ListDeletingExample.java: deleteTest
private static void deleteTest()  {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try  {
        // set cascadeonDelete in order to delete member objects
        container.ext().configure().objectClass(ListObject.class).cascadeonDelete(
            true);
        List<ListObject> result = container.<ListObject> query(ListObject.class);
        if (result.size() > 0)  {
            // retrieve a ListObject
            ListObject lol = result.queryByExample(0);
            // delete the ListObject with all the field objects
            container.delete(lol);
        }
    } finally  {
        container.close();
    }
    // check ListObjects and DataObjects in the database
    container = Db4o.openFile(DB4O_FILE_NAME);
    try  {
        List<ListObject> listObjects = container
            .<ListObject> query(ListObject.class);
        System.out.println("ListObjects in the database:  "
            + listObjects.size());
        List<DataObject> dataObjects = container
            .<DataObject> query(DataObject.class);
        System.out.println("DataObjects in the database:  "
            + dataObjects.size());
    } finally  {
        container.close();
    }
}

```

Please, remember that there is no referential integrity check on delete: deleted objects might be referenced from elsewhere in your code.

Deleting Collection Members

For the following examples we will use [DataObject and ListObject](#) classes. The database will be filled up with the [FillUpDb](#) method.

If you want to delete all members in a list you can use remove list function.

```
ListDeletingExample.java: removeTest
private static void removeTest()  {
    // set update depth to 1 as we only
    // modify List field
    Configuration configuration = Db4o.newConfiguration();
    configuration.objectClass(ListObject.class).updateDepth(1);
    ObjectContainer container = Db4o.openFile(configuration, DB4O_FILE_NAME);
    try  {
        List<ListObject> result = container.<ListObject> query(ListObject.class);
        if (result.size() > 0)  {
            // retrieve a ListObject
            ListObject lol = result.queryByExample(0);
            // remove all the objects from the list
            lol.getData().removeAll(lol.getData());
            container.store(lol);
        }
    } finally  {
        container.close();
    }
    // check DataObjects in the list
    // and DataObjects in the database
    container = Db4o.openFile(DB4O_FILE_NAME);
    try  {
        List<ListObject> result = container.<ListObject> query(ListObject.class);
        if (result.size() > 0)  {
            ListObject lol = result.queryByExample(0);
            System.out.println("DataObjects in the list:  "
                + lol.getData().size());
        }
        List<DataObject> removedObjects = container
            .<DataObject> query(DataObject.class);
        System.out.println("DataObjects in the database:  "
            + removedObjects.size());
    } finally  {
        container.close();
    }
}
```

However as you will see from the example above, removed objects are not deleted from the database. Here you should be very careful: if you want to delete DataObjects, which were removed from the list you must be sure that they are not referenced by existing objects. Check

[Referential Integrity](#) article for more information.

The following example shows how to delete DataObjects from the database as well as from the list:

```
ListDeletingExample.java: removeAndDeleteTest
private static void removeAndDeleteTest() {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        // set update depth to 1 as we only
        // modify List field
        container.ext().configure().objectClass(ListObject.class).updateDepth(1);
        List<ListObject> result = container.<ListObject> query(ListObject.class);
        if (result.size() > 0) {
            // retrieve a ListObject
            ListObject lol = result.queryByExample(0);
            // create a copy of the objects list
            // to memorize the objects to be deleted
            List tempList = new ArrayList(lol.getData());
            // remove all the objects from the list
            lol.getData().removeAll(tempList);
            // and delete them from the database
            Iterator<DataObject> it = tempList.iterator();
            while (it.hasNext()) {
                container.delete(it.next());
            }
            container.store(lol);
        }
    } finally {
        container.close();
    }
    // check DataObjects in the list
    // and DataObjects in the database
    container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        List<ListObject> result = container.<ListObject> query(ListObject.class);
        if (result.size() > 0) {
            ListObject lol = result.queryByExample(0);
            System.out.println("DataObjects in the list: " +
                + lol.getData().size());
        }
        List<DataObject> removedObjects = container
            .<DataObject> query(DataObject.class);
        System.out.println("DataObjects in the database: " +
            + removedObjects.size());
    } finally {
        container.close();
    }
}
```

Example Classes

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

package com.db4odoc.listdeleting;

public class DataObject {
    String _name;
    String _data;

    public DataObject()
    {
    }

    public String getName()
    {
        return _name;
    }

    public void setName(String name)
    {
        _name = name;
    }

    public String getData()
    {
        return _data;
    }

    public void setData(String data)
    {
        _data = data;
    }

    public String toString()
    {
        return String.format("%s/%s", _name, _data);
    }
}
```

Object Construction

Sometimes you may find that db4o refuses to store instances of certain classes, or appears to store them, but delivers incomplete instances on queries. To understand the problem and the alternative solutions at hand, we'll have to take a look at the way db4o "instantiates" objects when retrieving them from the database.

More Reading:

- [Creating objects](#)
- [Configuration](#)

- [Troubleshooting](#)
- [Examples](#)

Creating objects

Db4o currently knows three ways of creating and populating an object from the database. The approach to be used can be configured globally and on a per-class basis.

Using a constructor

The most obvious way is to call an appropriate constructor. Db4o does *not* require a public or no-args constructor. It can use any constructor that accepts default (null/0) values for all of its arguments without throwing an exception. Db4o will test all available constructors on the class (including private ones) until it finds a suitable one. What if no such constructor exists?

Bypassing the constructor

Db4o can also bypass the constructors declared for this class using platform-specific mechanisms. (For Java, this option is only available on JREs ≥ 1.4 .) This mode allows reinstantiating objects whose class doesn't provide a suitable constructor. However, it will (silently) break classes that rely on the constructor to be executed, for example in order to populate transient members. *If this option is available in the current runtime environment, it will be the default setting.*

Using a translator

If none of the two approaches above is suitable, db4o provides a way to specify in detail how instances of a class should be stored and reinstantiated by implementing the Translator interface and registering this implementation for the offending class. [Translators chapter](#) cover this topic in detail.

Configuration

The instantiation mode can be configured globally or on a per class basis.

```
Java: configuration.callConstructors(true)
```

This will configure db4o to use constructors to reinstantiate any object from the database. (The default is *false*).

```
Java: configuration.objectClass(Foo.class).callConstructor(true)
```

This will configure db4o to use constructor calls for this class and all its subclasses.

Troubleshooting

In the default configuration db4o will check that the objects can be stored correctly:

Java: `configuration.exceptionsOnNotStorable(true)`

If this setting triggers exceptions in your code, or if instances of a class seem to lose members during storage, check the involved classes (especially their constructors) for problems similar to the one shown in the following section.

Examples

```
C1.java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */

package com.db4odoc.constructors;

class C1 {
    private String s;

    private C1(String s) {
        this.s=s;
    }

    public String toString() {
        return s;
    }
}
```

```
C2.java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */

package com.db4odoc.constructors;

class C2 {
    private transient String x;
    private String s;

    private C2(String s) {
        this.s=s;
        this.x="x";
    }

    public String toString() {
        return s+x.length();
    }
}
```

The above C2 class needs to have callConstructors set to true. Otherwise, since transient members are not stored and the constructor code is not executed, `toString()` will potentially run into a `NullPointerException` on `x.length()`.

```
C3.java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */

package com.db4odoc.constructors;

class C3 {
    private String s;
    private int i;

    private C3(String s) {
        this.s=s;
        this.i=s.length();
    }

    public String toString() {
        return s+i;
    }
}
```

The above C3 class needs to have callConstructors set to false (the default), since the (only) constructor will throw a `NullPointerException` when called with a null value.

```
C4.java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */

package com.db4odoc.constructors;

class C4 {
    private String s;
    private transient int i;

    private C4(String s) {
        this.s=s;
        this.i=s.length();
    }

    public String toString() {
        return s+i;
    }
}
```

This class cannot be cleanly reinstated by db4o: both approaches will fail, so one has to resort to configuring a translator.

Implementation Strategies

This topic set explains the specifics of db4o implementation. Going through the topics will help you to reveal the real power of db4o and learn to apply it in your environment.

More Reading:

- [Maintenance](#)
- [Type Handling](#)
- [TypeHandlers](#)
- [Using Annotations](#)
- [Enhancement Tools](#)
- [Unique Constraints](#)
- [Object Callbacks](#)
- [Callbacks](#)
- [Storage](#)
- [Translators](#)
- [Db4o Reflection API](#)
- [Db4o meta-information](#)
- [Native Query Collection](#)
- [Refactoring and Schema Evolution](#)
- [Aliases](#)
- [Encryption](#)
- [IDs and UUIDs](#)
- [Freespace Management System](#)
- [String Encoding](#)
- [Reporting](#)
- [Exceptions](#)

Maintenance

db4o is designed to minimize maintenance tasks to the absolute minimum. The stored class schema adapts to the application automatically as it is being developed. db4o "understands" the addition and removal of fields which allows it to continue to work against modified classes without having to reorganize the database file. Internally db4o works with a superset of all class versions previously used.

However there are two recommended maintenance tasks, that can both be fully automated remotely with API calls. They will be reviewed in the following chapters.

More Reading:

- [Defragment](#)
- [Updating Db4o File Format](#)
- [System Info](#)
- [Backup](#)

Defragment

Db4o database file is structured as sets of free and occupied slots, very much like 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.

Defragment tool helps to fix this problem, creating a new database file and copying 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.

More Reading:

- [How To Use Defragmentation](#)
- [Defragmentation Configuration](#)
- [Tracking Defragmentation Errors](#)
- [Defragmentation Examples](#)

How To Use Defragmentation

The simplest way to defragment a db4o file would be:

Java:

```
Defragment.defrag("filename")
```

The file must not be opened by another process during defragmentation!

This will move the file *filename* to *filename.backup*, then create a defragmented version of this file in the original position. If the backup file already exists, this will throw an IOException and no action will be taken.

You can also specify the backup filename manually:

Java:

```
Defragment.defrag(filename, backupfile);
```

For more detailed configuration of the defragmentation process, you can use a [DefragmentConfig](#) instance within the following methods:

Java:

```
Defragment.defrag(configuration);  
Defragment.defrag(configuration, listener);
```

You can use listener parameter to track problems during defragmentation process. For more information see [Tracking Defragmentation Errors](#).

Defragmentation can throw IOException in the following situations:

- backup file exists
- database file not found
- database file is opened by another process

Defragmentation Configuration

DefragmentConfig class allows you fine-tune the defragmentation process. This topic discusses different settings available through DefragmentConfig.

Original File

The path to the file to be defragmented. Can be specified in the constructor:

```
configuration = new DefragmentConfig(origPath)
```

Backup File

```
configuration = new DefragmentConfig(origPath, backupPath)
```

The path to the backup of the original file. If this file exists before the defragmentation, an IOException will be thrown and no action taken.

If you want the backup file to be deleted automatically before the defragment run, specify:

Java:

```
configuration.forceBackupDelete(true)
```

Mapping

You can also specify the desired Mapping to be used internally:

```
configuration = new DefragmentConfig(origPath, backupPath, mapping)
```

`mapping` is an object of a class implementing ContextIDMapping interface. Mapping is used to keep track of objects moved during defragmentation. Db4o provides 2 mapping classes.

TreeIDMapping - default in-memory mapping. Will increase the memory usage, but is a faster alternative. Set up [objectCommitFrequency](#) to control memory usage.

BTreeIDMapping - mapping is done in a separate file using B-tree method. Reduces the memory usage, but is a much slower option.

Class Filter

Defragmentation process uses `StoredClassFilter accept` method to define which classes should be left in a database after the defragmentation. By default, all classes are left. However, you can use AvailableClassFilter to get rid of the deleted classes instances:

Java:

```
configuration.storedClassFilter(new AvailableClassFilter())
```

In this case only the classes known to the classloader will be left in the database, the rest will be deleted.

Database Configuration

For db4o configurations that influence low-level file layout details, it is important to provide the defragmentation process with the copy of db4o configuration:

Java:

```
configuration.db4oConfig(db4oConfiguration)
```

For more information about db4o configuration see [Configuration](#).

Commit Frequency

Commit frequency sets the number of processed objects that should trigger an immediate commit of the target file. By default, frequency = 0 and commit never happens.

Java:

```
configuration.objectCommitFrequency(frequency)
```

This method can be used to reduce memory usage during defragmentation.

Upgrading

You can upgrade your database file together with the defragmentation:

Java:

```
configuration.upgradeFile(tempFile)
```

This method can be used to reduce memory usage during defragmentation, however it will make it slower.

Tracking Defragmentation Errors

DefragmentListener/IDefragmentListener interface is provided to track system structure problems during a defragmentation process. DefragmentListener provides the following method, which will be called, when a problem is detected:

Java:

```
void notifyDefragmentInfo(DefragmentInfo info);
```

For an example of the listener implementation see [Defragmentation Examples](#).

Defragmentation Examples

The simplest defragmentation can look like this:

```
DefragmentExample.java: simplestDefragment
private static void simplestDefragment() {
    try {
        Defragment.defrag(DB_FILE);
    } catch (IOException ex) {
        System.out.println(ex.toString());
    }
}
```

The following example shows how to implement defragmentation listener:

```
DefragmentExample.java: defragmentWithListener
private static void defragmentWithListener() {
    DefragmentConfig config=new DefragmentConfig(DB_FILE);
    try {
        Defragment.defrag(config, new DefragmentListener() {
            public void notifyDefragmentInfo(DefragmentInfo info) {
                System.err.println(info);
            }
        });
    } catch (Exception ex) {
        System.out.println(ex.toString());
    }
}
```

The following example will run defragment using TreeIDMapping and commit frequency of 1 commit per 5000 objects. The backup file will be deleted if already exists, only available to the classloader classes will be left in the database (java version) and the file will be upgraded if necessary.

```
DefragmentExample.java: configuredDefragment
private static void configuredDefragment() {
    DefragmentConfig config=new
DefragmentConfig(DB_FILE, BACKUP_FILE, new TreeIDMapping());
    config.objectCommitFrequency(5000);
    config.db4oConfig(Db4o.cloneConfiguration());
    config.forceBackupDelete(true);
    config.storedClassFilter(new AvailableClassFilter());
    config.upgradeFile(DB_FILE + ".upg");
    try {
        Defragment.defrag(config);
    } catch (Exception ex) {
        System.out.println(ex.toString());
    }
}
```

System Info

SystemInfo is a utility class, providing system information about db4o database.

Currently SystemInfo includes the following methods:

Method	Functionality
freespaceSize	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](#).

freespaceEntryCount 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.

totalSize Returns the total size of the database on disk.

In order to get SystemInfo instance you must issue the following call:

Java:

```
SystemInfo info = container.ext().systemInfo();
```

SystemInfo can be used for different maintenance services. The example below presents a simple program, which can be scheduled to run automatically at certain intervals for a database check up.

```
SystemInfoExample.java: testSystemInfo
private static void testSystemInfo() {
    long dbSize = _container.ext().systemInfo().totalSize();
    long fsSize = _container.ext().systemInfo().freespaceSize();
    if (dbSize > MAX_DB_SIZE) {
        System.out.println("Attention! Database file size is \
over the limit. Maintenance required");
    }
    _logPS.println("Total database size: " + dbSize);
    if (fsSize > MAX_FS_SIZE) {
        System.out.println("Attention! Freespace size is \
over the limit. Maintenance required");
    }
    _logPS.println("Database freespace size: " + fsSize);
    _logPS.println("Database freespace entries: " +
    _container.ext().systemInfo().freespaceEntryCount());
}
```

Updating Db4o File Format

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 or ObjectManager, db4o does not upgrade databases by default. Database upgrading can be turned on with the following configuration switch:

Java:

```
Db4o.configure().allowVersionUpdates(true)
```

Please note that, once the database file version is updated, there is no way to get back to the older version of the database file. If 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 with code like the following:

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

package com.db4odoc.versionupdate;

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

public class UpdateExample {

    public static void main(String[] args) {
        Db4o.configure().allowVersionUpdates(true);
        ObjectContainer objectContainer = Db4o.openFile(args[0]);
        objectContainer.close();
        System.out.println("The database is ready for the version " + Db4o.version());
    }
}
```

Recommendations for upgrading:

- backup your database file to be able to switch back.
- check the content of database files before and after upgrading, using `com.db4o.tools.Statistics`.
- [Defragmenting](#) a database file with the new db4o version after upgrading can make the database more efficient.

Backup

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

The respective API calls for backups are:

Java:

```
ObjectContainer.ext().backup(String path) ObjetServer.ext().backup(String path)
```

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

It is recommended to backup the current development state of an application (ideally source code and bytecode) along with the database files since the old code can make it easier to work with the old data.

Type Handling

This topic set discusses special db4o classes.

More Reading:

- [Collections](#)
- [Blobs](#)
- [Static Fields And Enums](#)
- Delegates And Events
- [BigMath](#)
- [Final Fields](#)

BigMath

If you are dealing with very big numbers, you might be using BigDecimal or BigInteger java classes. These classes are specially designed to allow computations with of arbitrary precision. Internally the values are stored in byte arrays for both types. Now, thinking about it - it should not be a problem for db4o to store such values, as it is just a matter of storing a class with the actual value in a byte array field. However, a deeper consideration uncovers the following problems:

- BigInteger/BigDecimal representation is different in different java versions, which can cause problems re-instantiating the objects from a database created with a different Java version.
- BigDecimal relies on transient field setup in the constructor, which means that constructor use is compulsory
- db4o would store instances of these classes as full object graphs: A BigDecimal contains a BigInteger which contains a byte array, plus some other fields. This graph would faithfully be persisted into the database file and it would have to be read and reconstructed on access - activation depth applies.
- Querying and indexing will essentially be broken due to the above limitations.

To see the problem you can try the following simple test:

```
Probability.java
/**/* Copyright (C) 2004 - 2009 Versant Corporation http://www.versant.com */
package com.db4odoc.BigMath;
```

```

import java.math.*;

public class Probability {
    // Using BigDecimal to represent very small probabilities
    BigDecimal value;

    public Probability(String sValue) {
        this.value = new BigDecimal(sValue);
    }

    public String toString() {
        return value.toString();
    }
}

```

```

BigMathExample.java: testDefaultConfiguration
private static void testDefaultConfiguration() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer db = Db4oEmbedded.openFile(Db4oEmbedded
        .newConfiguration(), DB4O_FILE_NAME);
    Probability p1 = new Probability("2e-324");
    db.store(p1);
    p1 = new Probability("3e-324");
    db.store(p1);
    p1 = new Probability("4e-324");
    db.store(p1);
    db.close();

    db = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(),
        DB4O_FILE_NAME);
    Query q = db.query();
    q.constrain(Probability.class);
    q.descend("value").constrain(new BigDecimal("3e-324")).greater();
    List<Probability> result = q.execute();
    for (Probability p : result) {
        System.out.println(p);
    }
    db.close();
}

```

You will see that comparison is not done correctly. However some of the queries can work fine, like unoptimized native queries for example.

In order to solve the above mentioned problems db4o implements special typehandlers for BigInteger and BigDecimal, which allow to treat them as normal value types, i.e. in the same way as long and double, just without precision limitation. These typehandlers are implemented in db4o optional jar (required on the classpath) and should be added to the configuration before opening the file with the following method:

```

BigMathExample.java: configBigMathSupport
private static EmbeddedConfiguration configBigMathSupport() {
    EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();

```

```

        config.common().add(new BigMathSupport());
        return config;
    }
}

```

Note, that `BigMathSupport` is a part of common configuration interface and should be added through `common()` method.

Now the initial example should produce expected results:

```

BigMathExample.java: testBigMathConfiguration
private static void testBigMathConfiguration() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer db = Db4oEmbedded.openFile(configBigMathSupport(),
        DB4O_FILE_NAME);
    Probability p1 = new Probability("2e-324");
    db.store(p1);
    p1 = new Probability("3e-324");
    db.store(p1);
    p1 = new Probability("4e-324");
    db.store(p1);
    db.close();

    db = Db4oEmbedded.openFile(configBigMathSupport(), DB4O_FILE_NAME);
    Query q = db.query();
    q.constrain(Probability.class);
    q.descend("value").constrain(new BigDecimal("3e-324")).greater();
    List<Probability> result = q.execute();
    for (Probability p : result) {
        System.out.println(p);
    }
    db.close();
}

```

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 RAM;
- querying interface;
- objects' modification;
- information backup;
- client/server processing.

Db4o provides you with a flexibility of using 2 different solutions for this case:

1. Blob (Java package: `com.db4o.types.Blob`, .NET namespace: `Db4oTypes.IBlob`)
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. C/S communication runs asynchronous in separate thread
3. special code is necessary to store and load
4. no concerns about activation depth

byte[] array

1. data in the same file
2. C/S communication runs in the normal communication thread
3. transparent handling without special concerns
4. 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.

More Reading:

- [Db4o Blob Implementation](#)

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. Pros and Cons for the points, mentioned above:

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
2. C/S communication runs asynchronous in separate thread
 - + asynchronous storage allows the main application thread to continue its work, while blobs are being stored
3. special code is necessary to store and load
 - it is more difficult to move objects between db4o database files
4. no concerns about activation depth
 - + big objects won't be loaded into memory as part of the activation process

Let's look, how it works.

First, BLOB storage should be defined:

```
Java: Db4o.configure().setBlobPath(storageFolder);
```

where storageFolder is a String value representing local or server path to store BLOBs. If that value is not defined, db4o will use the default folder "blobs" in user directory.

We will use a modified Car class, which holds reference to the car photo:

```
Car.java
/** Copyright (C) 2004 - 2009 Versant Inc. http://www.db4o.com */
package com.db4odoc.blobs;

public class Car {
    private String model;
    private CarImage img;

    public Car(String model) {
        this.model=model;
        img=new CarImage();
        img.setFile(model+".jpg");
    }

    public CarImage getImage() {
        return img;
    }

    public String toString() {
        return model +"(" + img.getFile() + ")";
    }
}
```

CarImage is a wrapper to BLOB, representing the photo:

```
CarImage.java
/**/* Copyright (C) 2004 - 2009 Versant Inc. http://www.db4o.com */

package com.db4odoc.blobs;

import java.io.File;

import com.db4o.ext.Status;
import com.db4o.types.Blob;
```

```

public class CarImage {
    Blob blob;
    private String fileName = null;
    private String inFolder = "blobs\\in\\";
    private String outFolder = "blobs\\out\\";

    public CarImage() {

    }

    public void setFile(String fileName) {
        this.fileName = fileName;
    }

    public String getFile() {
        return fileName;
    }

    public boolean readFile() throws java.io.IOException {
        blob.readFrom(new File(inFolder + fileName));
        double status = blob.getStatus();
        while(status > Status.COMPLETED) {
            try {
                Thread.sleep(50);
                status = blob.getStatus();
            } catch (InterruptedException ex) {
                System.out.println(ex.getMessage());
            }
        }
        return (status == Status.COMPLETED);
    }

    public boolean writeFile() throws java.io.IOException {
        blob.writeTo(new File(outFolder + fileName));
        double status = blob.getStatus();
        while(status > Status.COMPLETED) {
            try {
                Thread.sleep(50);
                status = blob.getStatus();
            } catch (InterruptedException ex) {
                System.out.println(ex.getMessage());
            }
        }
        return (status == Status.COMPLETED);
    }
}

```

inFolder ("blobs\in\") is used as a location of existing files, which are to be stored into db4o, and outFolder ("blobs\out\") will be the place for images, retrieved from the database.

readFile method allows blob to be read from the specified location into db4o storage:

```
Java: Blob.readFrom( File )
```

As reading is done in a dedicated thread, you can use Blob#getStatus() in a loop to create a progress window.

The same applies to the write operation, which copies BLOB, stored with db4o, to the specified filesystem location.

Let's store some cars together with their images in our database:

```
BlobExample.java: storeCars
private static void storeCars() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4oEmbedded.openFile(Db4oEmbedded
        .newConfiguration(), DB4O_FILE_NAME);
    try {
        Car car1 = new Car("Ferrari");
        container.store(car1);
        storeImage(car1);
        Car car2 = new Car("BMW");
        container.store(car2);
        storeImage(car2);
    } finally {
        container.close();
    }
}
```

```
BlobExample.java: storeImage
private static void storeImage(Car car) {
    CarImage img = car.getImage();
    try {
        img.readFile();
    } catch (java.io.IOException ex) {
        System.out.println(ex.getMessage());
    }
}
```

CarImage is stored in the database just like normal object, no BLOB data is transferred before explicit call (Blob#readFrom in CarImage#readFile method), which copies the images to the storageFolder.

Please, note, that CarImage reference should be stored to the database before uploading actual data, because the Blob field is only instantiated once the object is stored (otherwise you will get a null reference exception). To get the images back to the filesystem we can run a usual query:

```

BlobExample.java: retrieveCars
private static void retrieveCars() {
    ObjectContainer container = Db4oEmbedded.openFile(DB4O_FILE_NAME);
    try {
        Query query = container.query();
        query.constrain(Car.class);
        List<Car> result = query.execute();
        getImages(result);
    } finally {
        container.close();
    }
}

```

and get BLOB data using retrieved Car references:

```

BlobExample.java: getImages
private static void getImages(List<Car> result) {
    for (Car car: result) {
        System.out.println(car);
        CarImage img = car.getImage();
        try {
            img.writeFile();
        } catch (java.io.IOException ex) {
            System.out.print(ex.getMessage());
        }
    }
}

```

Retrieved images are placed in CarImage.outFolder ("blobs\out").

So query interface operates on references - no BLOB data is loaded into memory until explicit call (Blob#writeTo). This also means, that activationDepth does not affect Blob objects and best querying performance is achieved without additional coding.

Collections

[Translators](#) chapter of the documentation explains why translators are necessary to store and retrieve some types of classes. Db4o uses translators internally to manage storing and retrieving of collections.

Java collections were first implemented in JDK 1.2. Before that Vector implementation was used to store growable arrays of objects. Under the hood, when collection object is stored to the database different actions are taken for different versions of java:

1. before JDK1.2: collection class is translated with TVector class;
2. after: collection is translated with TCollection, TMap, THashtable translators.

In fact, the functionality of those translators is pretty much the same. On storing collection is transferred to an array of objects (`Object[]`) and that array gets stored to the database file.

`Map` (Java), `HashTable` and `SortedList` for .NET objects are stored as an array of objects of special type:

```
public class Entry { public Object key; public Object value; . . . }
```

OnInstantiate method of collection translators creates a new instance of respective collection or map and restores its values from the saved object array.

Unfortunately this implementation is not very efficient for searches/updates of a certain value in a collection, as the whole collection should be instantiated to access any of its elements.

More Reading:

- [Built-in db4o collections](#)
- [Fast collections](#)
- [Collections Or Arrays](#)

Built-in db4o collections

Db4o provides its own built-in collection implementation to improve performance and decrease memory consumption:

Java:

```
ObjectContainer.ext().collections.newLinkedList();  
ObjectContainer.ext().collections.newHashMap();  
ObjectContainer.ext().collections.newIdentityHashMap();
```

The `LinkedList` implementation only holds the first and the last elements in RAM, all the other objects are loaded on demand. Apparently, this implementation provides good performance for sequential traversal, but it is still quite inefficient for random selection/update.

The `HashMap` implementation only holds an array of hash values in RAM and loads keys and objects on demand.

Db4o collections also provide the following functionality, which helps a programmer to produce expected results with as little work as possible:

- Newly added objects are automatically persisted.
- Collection elements are automatically activated, when they are needed. The activation depth is configurable with `Db4oCollection#activationDepth(int)`

- Removed objects can be deleted automatically, if the list is configured with `Db4oCollection#deleteRemoved(boolean)`

Weak Reference system ensures that objects are freed from RAM, as soon as there are no references to them.

These implementations can be faster for some cases and slower for others.

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 if you are looking into the future you might want to design your system in a way that can easily be switched to the next db4o planned feature: Fast Collections. For more details see [Fast collections](#). Please, feel free to use db4o Jira to track [the feature](#) progress and be notified on the latest updates.

Fast collections

Db4o's solution for the best collection performance and lowest memory consumption is to implement them directly on top of BTrees without an intermediate "stored-object-db4o" layer (P1Object, P1Collection, P2LinkedList).

This task is still under development, but already it makes sense to be ready to switch to the new fast collections seamlessly.

Current recommendation for collection usage with db4o is:

- Declare members of persistent classes as interface (`java.util.List / System.Collections.IList`).
- Create central factory method to implement concrete collection (can be switched to fast collection implementation easily).

Please, avoid the following realizations, which will make the switching more difficult:

- Declaring concrete implementations as fields in persistent classes
- Deriving from JDK collection classes
- Using third-party non-standard collections

Let's look at application design, which will allow you to upgrade your application to fast collections with the least effort.

In our example we will save a list of pilots as members of one team. To make it simple let's use the following factory class to get the proper list implementation:

```
CollectionFactory.java
```

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

package com.db4odoc.lists;

import java.util.ArrayList;
import java.util.List;

public class CollectionFactory {
    public static List newList() {
        return new ArrayList();
    }
}
```

The concrete class returned by the CollectionFactory can be changed to any other collection implementation (fast collection) with the minimum coding effort.

We will use the following class as a team of pilots:

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

package com.db4odoc.lists;

import java.util.List;

public class Team {
    private List pilots;
    private String name;

    public Team() {
        pilots = CollectionFactory.newList();
    }

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

    public String getName() {
        return name;
    }

    public void addPilot(Pilot pilot) {
        pilots.add(pilot);
    }

    public Pilot getPilot(int index) {
        return (Pilot)pilots.queryByExample(index);
    }
}
```

```

public void removePilot(int index) {
    pilots.remove(index);
}

public void updatePilot(int index, Pilot newPilot) {
    pilots.store(index, newPilot);
}
}

```

The idea of the new fast collection implementation is to allow select/update of collection elements without an intermediate "stored-object-db4o" layer. This will allow random activation and fast querying, thus providing a considerable performance improvement especially on big collections holding deep object graphs.

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 JDK 1.3-1.4 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 JDK 5 and JDK 6. 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 ObjectContainer to connect the translator to the TestFinal class:

```
Db4o.configure().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](#)
- [Usage of static fields](#)
- [Java enumerations](#)
- [.NET Enumerations](#)

Java enumerations

Enumerated types were brought into Java with the JDK 1.5 release. In fact they represent a class with static fields similar to the one reviewed in the [Static fields paragraph](#).

```

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

package com.db4odoc.enums;

public enum Qualification {

```

```

WINNER("WINNER"),
PROFESSIONAL("PROFESSIONAL"),
TRAINEE("TRAINEE");

private String qualification;

private Qualification(String qualification) {
    this.qualification = qualification;
}

public void testChange(String qualification) {
    this.qualification = qualification;
}

public String toString() {
    return qualification;
}
}

```

db4o takes care about storing enumeration objects automatically without any additional settings:

```

EnumExample.java: setPilots
private static void setPilots() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container=Db4o.openFile(DB4O_FILE_NAME);
    try {
        container.store(new Pilot("Michael Schumacher",Qualification.WINNER));
        container.store(new Pilot("Rubens Barrichello",Qualification.PROFESSIONAL));
    } finally {
        container.close();
    }
}

```

```

EnumExample.java: checkPilots
private static void checkPilots() {
    ObjectContainer container=Db4o.openFile(DB4O_FILE_NAME);
    try {
        ObjectSet result = container.query(Pilot.class);
        System.out.println("Saved pilots: " + result.size());
        for(int x = 0; x < result.size(); x++) {
            Pilot pilot = (Pilot )result.queryByExample(x);
            if (pilot.getQualification() == Qualification.WINNER) {
                System.out.println("Winner pilot: " + pilot);
            } else if (pilot.getQualification() == Qualification.PROFESSIONAL) {
                System.out.println("Professional pilot: " + pilot);
            } else {
                System.out.println("Uncategorized pilot: " + pilot);
            }
        }
    } finally {
        container.close();
    }
}

```

Another specific feature of enums in db4o: deletion is not possible:

```

EnumExample.java: deletePilots
private static void deletePilots() {
    System.out.println("Qualification enum before delete Pilots");
    printQualification();
    Configuration configuration = Db4o.newConfiguration();
    configuration.objectClass(Pilot.class)
    .objectField("qualification").cascadeonDelete(true);
    ObjectContainer container=Db4o.openFile(configuration, DB4O_FILE_NAME);

    try  {
        ObjectSet result = container.query(Pilot.class);
        for(int x = 0; x < result.size(); x++) {
            Pilot pilot = (Pilot )result.queryByExample(x);
            container.delete(pilot);
        }
    } finally {
        container.close();
    }
    System.out.println("Qualification enum after delete Pilots");
    printQualification();
}

```

```

EnumExample.java: checkPilots
private static void checkPilots() {
    ObjectContainer container=Db4o.openFile(DB4O_FILE_NAME);
    try  {
        ObjectSet result = container.query(Pilot.class);
        System.out.println("Saved pilots: " + result.size());
        for(int x = 0; x < result.size(); x++) {
            Pilot pilot = (Pilot )result.queryByExample(x);
            if (pilot.getQualification() == Qualification.WINNER) {
                System.out.println("Winner pilot: " + pilot);
            } else if (pilot.getQualification() == Qualification.PROFESSIONAL) {
                System.out.println("Professional pilot: " + pilot);
            } else {
                System.out.println("Uncategorized pilot: " + pilot);
            }
        }
    } finally {
        container.close();
    }
}

```

Deletion of references does not automatically delete the enum. Even explicit deletion does not work:

```

EnumExample.java: deleteQualification
private static void deleteQualification() {
    System.out.println("Explicit delete of Qualification enum");
    Configuration configuration = Db4o.newConfiguration();
    configuration.objectClass(Qualification.class).cascadeonDelete(true);
    ObjectContainer container=Db4o.openFile(configuration, DB4O_FILE_NAME);
    try  {
        ObjectSet result = container.query(Qualification.class);

```

```

        for(int x = 0; x < result.size(); x++) {
            Qualification pq = (Qualification)result.queryByExample(x);
            container.delete(pq);
        }
    } finally {
    container.close();
}
printQualification();
}

```

Enum update works in the same way as for normal static objects - updated enum should be explicitly saved to database (#set(enum)).

```

EnumExample.java: updateQualification
private static void updateQualification() {
    System.out.println("Updating WINNER qualification constant");
    ObjectContainer container=Db4o.openFile(DB4O_FILE_NAME);
    try {
        Query query = container.query();
        query.constrain(Qualification.class);
        query.descend("qualification").constrain("WINNER");
        ObjectSet result = query.execute();
        for(int x = 0; x < result.size(); x++) {
            Qualification qualification = (Qualification)result.queryByExample(x);
            qualification.testChange("WINNER2006");
            container.store(qualification);
        }
    } finally {
        container.close();
    }
    printQualification();
}

```

You can use either build-in Java enums or write your own. Db4o will take care of keeping object references unique and database file as small as possible.

Static fields API

By default db4o does not persist static fields. Normally this is not necessary as static values are set for a class, not for an object. However you can set up db4o to store static fields if you need to implement constant or enumeration:

Java:

```
Db4o.configure().objectClass(Foo.class).persistStaticFieldValues()
```

Do not use this option unnecessarily, as it will slow down the process of opening database files and the stored objects will occupy space in the database file.

This option does not have any effect on primitive types (int, boolean, etc). Use their object alternatives instead (Integer, Boolean, etc).

When this setting is on for a specific class, all non-primitive-typed static field values of this class are stored the first time an object of the class is stored, and restored, every time a database file is opened afterwards, after class meta information is loaded for this class (when the class objects are retrieved with a query, for example).

A good example of non-primitive constant type is type-safe enumeration implementation:

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

package com.db4odoc.staticfields;

public class PilotCategories {
    private String qualification = null;

    public final static PilotCategories WINNER = new PilotCategories(
        "WINNER");

    public final static PilotCategories TALENTED = new PilotCategories(
        "TALENTED");

    public final static PilotCategories AVERAGE = new PilotCategories(
        "AVERAGE");

    public final static PilotCategories DISQUALIFIED = new PilotCategories(
        "DISQUALIFIED");

    private PilotCategories(String qualification) {
        this.qualification = qualification;
    }

    public PilotCategories() {
    }

    public void testChange(String qualification) {
        this.qualification = qualification;
    }

    public String toString() {
        return qualification;
    }
}
```

Let's use it with

```

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

package com.db4odoc.staticfields;

public class Pilot {
    private String name;

    private PilotCategories category;

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

    public PilotCategories getCategory() {
        return category;
    }

    public String getName() {
        return name;
    }

    public String toString() {
        return name + "/" + category;
    }
}

```

```

StaticFieldExample.java: setPilots
private static void setPilots() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database();
    if (container != null) {
        try {
            container.store(new Pilot("Michael Schumacher",
                PilotCategories.WINNER));
            container.store(new Pilot("Rubens Barrichello",
                PilotCategories.TALENTED));
        } finally {
            closeDatabase();
        }
    }
}

```

We can try to save pilots with the default db4o settings:

```

StaticFieldExample.java: checkPilots
private static void checkPilots() {

```

```

ObjectContainer container = database();
if (container != null) {
    try {
        ObjectSet result = container.query(Pilot.class);
        for (int x = 0; x < result.size(); x++) {
            Pilot pilot = (Pilot) result.queryByExample(x);
            if (pilot.getCategory() == PilotCategories.WINNER) {
                System.out.println("Winner pilot: " + pilot);
            } else if (pilot.getCategory() == PilotCategories.TALENTED) {
                System.out.println("Talented pilot: " + pilot);
            } else {
                System.out.println("Uncategorized pilot: " + pilot);
            }
        }
    } finally {
        closeDatabase();
    }
}
}

```

That does not work however. We will have to explicitly point out, which class's static fields we want to save:

```

StaticFieldExample.java: configure
private static void configure() {
    System.out.println("Saving static fields can be turned on for individual classes.");
    _configuration = Db4o.newConfiguration();
    _configuration.objectClass(PilotCategories.class)
        .persistStaticFieldValues();
}

```

Try to save and check pilots again - you should see that with this configuration enumeration values are actually correctly bound to their runtime values.

As it was mentioned before, it is important to keep static values in one place and do not allow different objects to modify them. If we try to change static value from the referencing object:

```

StaticFieldExample.java: updatePilots
private static void updatePilots() {
    System.out
        .println("Updating PilotCategory in pilot reference:");
    ObjectContainer container = database();
    if (container != null) {
        try {
            ObjectSet result = container.query(Pilot.class);
            for (int x = 0; x < result.size(); x++) {

```

```
Pilot pilot = (Pilot) result.queryByExample(x);
if (pilot.getCategory() == PilotCategories.WINNER) {
    System.out.println("Winner pilot: " + pilot);
    PilotCategories pc = pilot.getCategory();
    pc.testChange("WINNER2006");
    container.store(pilot);
}
printCategories(container);
} finally {
    closeDatabase();
}
}
```

the value just does not change. You can check it with the `checkPilots` method above. In order to update static field we will have to do that explicitly:

```
StaticFieldExample.java: updatePilotCategories
private static void updatePilotCategories()  {
    System.out.println("Updating PilotCategories explicitly:");
    ObjectContainer container = database();
    if (container != null)  {
        try  {
            ObjectSet result = container.query(PilotCategories.class);
            for (int x = 0; x < result.size(); x++)  {
                PilotCategories pc = (PilotCategories) result.queryByExample(x);
                if (pc == PilotCategories.WINNER)  {
                    pc.testChange("WINNER2006");
                    container.store(pc);
                }
            }
        }
        printCategories(container);
    } finally  {
        closeDatabase();
    }
}
System.out.println("Change the value back:");
container = database();
if (container != null)  {
    try  {
        ObjectSet result = container.query(PilotCategories.class);
        for (int x = 0; x < result.size(); x++)  {
            PilotCategories pc = (PilotCategories) result.queryByExample(x);
            if (pc == PilotCategories.WINNER)  {
                pc.testChange("WINNER");
                container.store(pc);
            }
        }
    }
    printCategories(container);
} finally  {
```

```
        closeDatabase();  
    }  
}  
}
```

Please, check the result with the `checkPilots` method. You will see that the reference has changed correctly.

What about deletion? Similar to update we cannot delete static fields from the referenced object, but we can delete them directly from the database:

```
StaticFieldExample.java: addDeleteConfiguration
private static void addDeleteConfiguration() {
    if (_configuration != null) {
        _configuration.objectClass(Pilot.class).cascadeonDelete(true);
    }
}
```

```
StaticFieldExample.java: deleteTest
private static void deleteTest()  {
    // use delete configuration
    ObjectContainer container = database();
    if (container != null)  {
        try  {
            System.out.println("Deleting Pilots :");
            ObjectSet result = container.query(Pilot.class);
            for (int x = 0; x < result.size(); x++)  {
                Pilot pilot = (Pilot) result.queryByExample(x);
                container.delete(pilot);
            }
            printCategories(container);
            System.out.println("Deleting PilotCategories :");
            result = container.query(PilotCategories.class);
            for (int x = 0; x < result.size(); x++)  {
                container.delete(result.queryByExample(x));
            }
            printCategories(container);
        } finally  {
            closeDatabase();
        }
    }
}
```

Usage of static fields

As an object database db4o can take advantage of programming language specifics such as static modifier. Why and where should it be used?

Usually static fields are used to store enums and constants. Obviously these objects can be stored in application code only, keeping database file smaller and decreasing memory consumption at run-time. But it can be not the best option in the case when the constant (enum) value can be changed in application lifecycle: the references from all the database objects will have to be updated explicitly.

Db4o suggests another approach to keeping constant values. For a class

```
Car.java
1/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */
2
3package com.db4odoc.staticfields;
4
5import java.awt.*;
6
7public class Car {
8    Color color;
9}
```

the color field can be set to Color enumeration value like that:

```
StaticFieldExample.java: setCar
private static void setCar() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        Car car = new Car();
        car.color = Color.GREEN;
        container.store(car);
    } finally {
        container.close();
    }
}
```

Now, when ObjectContainer is reopened and the green car is retrieved from the database, the static instances in RAM will be associated with the previously persisted instances using #bind() under the hood. So that the following check is possible:

```
car.color == Color.GREEN
```

This also means that ,if Color.GREEN constant will get another internal value (RGB(0,255,10) instead of RGB(0,255,0) for instance), all the references from the database will be associated with the new value.

Static field values are associated with their persistent identities only once, when an ObjectContainer is opened. After that they are not stored, unless the developer does it deliberately. Objects instantiation from the database does not create any more instances of static values.

Since each static field exists only once in the VM, there are no versioning, locking or multiuser access problems.

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: configuration `readAs` method, [Translators](#), transient [fields in Java](#) and .NET and [classes](#), custom marshallers 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 simple way to write a specific persistence solution for any custom object, and now db4o provides this way through a pluggable TypeHandler4/ITypeHandler4 interface:

Java:

```
configuration.registerTypeHandler(TypeHandlerPredicate, TypeHandler4);
```

In the method above TypeHandler4 interface provides methods that define how an object is converted to a low-level byte-array and back and how it behaves in a query:

```
TypeHandler4.java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */

package com.db4o.internal;

import com.db4o.ext.*;
import com.db4o.internal.fieldhandlers.*;
import com.db4o.marshall.*;

/**
 * @exclude
 */
public interface TypeHandler4 extends Comparable4, FieldHandler {

    void delete(DeleteContext context) throws Db4oIOException;
```

```

    void defragment(DefragmentContext context);

    Object read(ReadContext context);

    void write(WriteContext context, Object obj);

}

```

```

Comparable4.java
/**/* Copyright (C) 2004 Versant Inc. http://www.db4o.com */

package com.db4o.internal;

import com.db4o.foundation.*;

/**
 * @exclude
 */
public interface Comparable4 {
    PreparedComparison prepareComparison(Object obj);
}

```

TypeHandlerPredicate provides a #match method, which returns true for objects that should be handled with the specified TypeHandler.

Type handler functionality is best explained on a [working example](#).

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

Custom Typehandler Example

For a custom typehandler example we will try to write a very simple typehandler for StringBuffer(java) and StringBuilder(.NET). These classes are basically just another representation of String, so we can look at the StringHandler implementation in db4o source.

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 #write method, which determines how the object is persisted:

```

StringBufferHandler.java: write
public void write(WriteContext context, Object obj)  {
    String str = ((StringBuffer)obj).toString();
    WriteBuffer buffer = context;
    buffer.writeInt(str.length());
    writeToBuffer(buffer, str);
}

```

```
}
```

```
StringBufferHandler.java: writeToBuffer
private static void writeToBuffer(WriteBuffer buffer, String str) {
    final int length = str.length();
    char[] chars = new char[length];
    str.getChars(0, length, chars, 0);
    for (int i = 0; i < length; i++) {
        buffer.writeByte((byte) (chars[i] & 0xff));
        buffer.writeByte((byte) (chars[i] >> 8));
    }
}
```

As you can see from the code above, there are 3 steps:

1. Get the buffer from WriteContext/I WriteContext
2. Write the length of the StringBuffer/StringBuilder
3. Transfer the object to char array and write them in Unicode

Next step is to read the same from the buffer. It is just opposite to the write method:

```
StringBufferHandler.java: read
public Object read(ReadContext context) {
    ReadBuffer buffer = context;
    String str = "";
    int length = buffer.readInt();
    if (length > 0) {
        str = readBuffer(buffer, length);
    }
    return new StringBuffer(str);
}
```

```
StringBufferHandler.java: readBuffer
private static String readBuffer(ReadBuffer buffer, int length) {
    char[] chars = new char[length];
    for(int ii = 0; ii < length; ii++) {
        chars[ii] = (char)((buffer.readByte() & 0xff) | ((buffer.readByte() & 0xff) << 8));
    }
    return new String(chars, 0, length);
}
```

Delete is simple - we just reposition the buffer offset to the end of the slot:

```
StringBufferHandler.java: delete
public void delete(DeleteContext context) {
    context.readSlot();
}
```

Try to experiment with the #delete method by implementing cascade on delete. Use First-ClassObjectHandler as an example.

We are done with the read/write operations. But as you remember, in order to read an object, we must find it through a query, and that's where we will need a #compare method (well, you do not need it if your query does not contain any comparison criteria, but this is normally not the case):

```
StringBufferHandler.java: prepareComparison
public PreparedComparison prepareComparison(Context ctx, final Object obj) {
    return new PreparedComparison() {
        public int compareTo(Object target) {
            return compare((StringBuffer)obj, (StringBuffer)target);
        }
    };
}
```

```
StringBufferHandler.java: compare
private final int compare(StringBuffer a_compare, StringBuffer a_with) {
    if (a_compare == null) {
        if (a_with == null) {
            return 0;
        }
        return -1;
    }
    if (a_with == null) {
        return 1;
    }
    char c_compare[] = new char[a_compare.length()];
    a_compare.getChars(0, a_compare.length() - 1, c_compare, 0);
    char c_with[] = new char[a_with.length()];
    a_with.getChars(0, a_with.length() - 1, c_with, 0);

    return compareChars(c_compare, c_with);
}
```

```
StringBufferHandler.java: compareChars
private static final int compareChars(char[] compare, char[] with) {
    int min = compare.length < with.length ? compare.length : with.length;
    for (int i = 0; i < min; i++) {
        if (compare[i] != with[i]) {
            return compare[i] - with[i];
        }
    }
    return compare.length - with.length;
}
```

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):

```
StringBufferHandler.java: defragment
```

```

public void defragment(DefragmentContext context) {
    // To stay compatible with the old marshaller family
    // In the marshaller family 0 number 8 represented
    // length required to store ID and object length information
    context.incrementOffset(8);
}

```

This Typehandler implementation can be tested with a class below. Please, pay special attention to `#configure` method, which adds `StringBufferHandler/StringBuilderHandler` to the database configuration:

```

TypehandlerExample.java
package com.db4odoc.typehandler;

import java.io.File;
import java.io.IOException;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.config.Configuration;
import com.db4o.defragment.Defragment;
import com.db4o.ext.DatabaseFileLockedException;
import com.db4o.query.Query;
import com.db4o.reflect.ReflectClass;
import com.db4o.reflect.generic.GenericReflector;
import com.db4o.reflect.jdk.JdkReflector;
import com.db4o.typehandlers.TypeHandlerPredicate;

public class TypehandlerExample {

    private final static String DB4O_FILE_NAME = "reference.db4o";
    private static ObjectContainer _container = null;

    public static void main(String[] args) throws IOException {
        testReadWriteDelete();
        //testDefrag();
        testCompare();
    }
    // end main

    private static Configuration configure() {
        Configuration configuration = Db4o.newConfiguration();
        // add a custom typehandler support

        TypeHandlerPredicate predicate = new TypeHandlerPredicate() {
            public boolean match(ReflectClass classReflector, int version) {
                GenericReflector reflector = new GenericReflector(
                    null, new JdkReflector(Thread.currentThread().getContextClassLoader()));

```

```

        ReflectClass claxx = reflector.forName(StringBuffer.class.getName());
        boolean res = claxx.equals(classReflector);
            return res;
        }
    };

    configuration.registerTypeHandler(predicate, new StringBufferHandler());
    return configuration;
}
// end configure

private static void testReadWriteDelete() {
    storeCar();
    // Does it still work after close?
    retrieveCar();
    // Does deletion work?
    deleteCar();
    retrieveCar();
}
// end testReadWriteDelete

private static void retrieveCar() {
    ObjectContainer container = database(configure());
    if (container != null) {
        try {
            ObjectSet result = container.query(Car.class);
            Car car = null;
            if (result.hasNext()) {
                car = (Car)result.next();
            }
            System.out.println("Retrieved: " + car);
        } finally {
            closeDatabase();
        }
    }
}
// end retrieveCar

private static void deleteCar() {
    ObjectContainer container = database(configure());
    if (container != null) {
        try {
            ObjectSet result = container.query(Car.class);
            Car car = null;
            if (result.hasNext()) {
                car = (Car)result.next();
            }
            container.delete(car);
            System.out.println("Deleted: " + car);
        } finally {
            closeDatabase();
        }
    }
}

```

```

}

// end deleteCar

private static void storeCar() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database(configure());
    if (container != null) {
        try {
            Car car = new Car("BMW");
            container.store(car);
            car = (Car) container.query(Car.class).next();
            System.out.println("Stored: " + car);

        } finally {
            closeDatabase();
        }
    }
}

// end storeCar

private static void testCompare() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database(configure());
    if (container != null) {
        try {
            Car car = new Car("BMW");
            container.store(car);
            car = new Car("Ferrari");
            container.store(car);
            car = new Car("Mercedes");
            container.store(car);
            Query query = container.query();
            query.constrain(Car.class);
            query.descend("model").orderAscending();
            ObjectSet result = query.execute();
            listResult(result);

        } finally {
            closeDatabase();
        }
    }
}

// end testCompare

public static void testDefrag() throws IOException {
    new File(DB4O_FILE_NAME + ".backup").delete();
    storeCar();
    Defragment.defrag(DB4O_FILE_NAME);
    retrieveCar();
}
}

// end testDefrag

private static ObjectContainer database(Configuration configuration) {
    if (_container == null) {

```

```

        try  {
            _container = Db4o.openFile(configuration, DB4O_FILE_NAME);
        } catch (DatabaseFileLockedException ex)  {
            System.out.println(ex.getMessage());
        }
    }
    return _container;
}
// end database

private static void closeDatabase()  {
    if (_container != null)  {
        _container.close();
        _container = null;
    }
}
// end closeDatabase

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

```

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 want to customize the way Strings are stored (special encodings).
- 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
- You want to implement cascade on delete through the typehandler

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

- Custom handling of platform-specific generic collections
- Fast collection handlers that operate on a lower level and scale for large collections
- 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.

TypeHandler Usages

This chapter provides some of the typehandler examples, which can be used as a replacement of translators.

Selective Field Persistence

In the first example we will look at the case, when we want to save only certain fields of a class. Suppose we have the following class:

```
NotStorable.java
/**Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */
package com.db4odoc.typehandler.translators;

public class NotStorable {
    private int id;

    private String name;

    private transient int length;

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

    public int getId() {
        return id;
    }

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

```

    }

    public int getLength()  {
        return length;
    }

    public String toString()  {
        return id + "/" + name + ": " + length;
    }
}

```

In this case we obviously do not want to store transient `length` field. To achieve this we can write a new typehandler, which will store a duplicate of the class without the `length` field. To make the work easier, we will extend the default `FirstClassObjectHandler`. First class means any persistent object that has its identity in the database.

```

NotStorableTypehandler.java
package com.db4odoc.typehandler.translators;

import com.db4o.internal.handlers.*;
import com.db4o.internal.marshall.*;
import com.db4o.marshall.*;
import com.db4o.typehandlers.*;

public class NotStorableTypehandler extends FirstClassObjectHandler
implements TypeHandler4, FirstClassHandler,
VariableLengthTypeHandler
{

    private class Storable
    {
        private int id;
        private String name;
    }

    public void write(WriteContext context, Object obj)  {
        NotStorable item = (NotStorable)obj;
        Storable storableItem = new Storable();
        storableItem.id = item.getId();
        storableItem.name = item.getName();

        context.writeObject(storableItem);
    }

    public Object read(ReadContext context)  {
        UnmarshallingContext _context = (UnmarshallingContext) context;
        NotStorable item = (NotStorable)_context.persistentObject();
        Storable storableItem = (Storable)context.readObject();
        item = new NotStorable(storableItem.id, storableItem.name);
        return item;
    }
}

```

```
    }  
}  
}
```

You can see that in this case, we are just replacing NotStorable class with a duplicate class Storable and save it instead.

Let's test how it works. First we should create configuration to include the typehandler:

And now test store and retrieve:

```
TranslatorExample.java: tryStoreAndRetrieve  
private static void tryStoreAndRetrieve(Configuration configuration) {  
    new File(DB4O_FILE_NAME).delete();  
    ObjectContainer container = Db4o.openFile(configuration, DB4O_FILE_NAME);  
    try {  
        NotStorable notStorable = new NotStorable(42, "Test");  
        System.out.println("ORIGINAL: " + notStorable);  
        container.store(notStorable);  
    } catch (Exception exc) {  
        System.out.println(exc.toString());  
        return;  
    } finally {  
        container.close();  
    }  
    container = Db4o.openFile(configuration, DB4O_FILE_NAME);  
    try {  
        ObjectSet result = container.queryByExample(NotStorable.class);  
        while (result.hasNext()) {  
            NotStorable notStorable = (NotStorable) result.next();  
            System.out.println("RETRIEVED: " + notStorable);  
        }  
    } finally {  
        container.close();  
    }  
}
```

This works. However, this example has certain limitations: the fields were in fact discarded and encapsulated into a Storable object. This means that we won't be able to query for fields or index on fields.

Ignore Fields Persistence

In the next example we will use one of the db4o internal typehandlers, which allows to skip all the field information for certain objects. This can be useful to save the database space when the object

information is not required for future retrieval. Let's use the following classes:

```
PersistentItem.java
package com.db4odoc.typehandler.translators;

public class PersistentItem {
    String id;
    TransientItem item;

    public PersistentItem (String id, TransientItem item) {
        this.id = id;
        this.item = item;
    }

    public String toString() {
        return id + ":" + item.toString();
    }
}
```

Let's assume that `TransientItem` field information is not required to be stored in our application. We can save the database space then, by using `IgnoreFieldsTypeHandler`. The configuration will look like this:

```
TranslatorExample.java: ignoreFieldsConfiguration
private static Configuration ignoreFieldsConfiguration() {
    Configuration configuration = Db4o.newConfiguration();
    // add a custom typehandler support
    TypeHandlerPredicate predicate = new TypeHandlerPredicate() {
        public boolean match(ReflectClass classReflector) {
            GenericReflector reflector = new GenericReflector(
                null, new JdkReflector
(Thread.currentThread().getContextClassLoader()));
            ReflectClass claxx = reflector.forName(TransientItem.class.getName());
            boolean res = claxx.equals(classReflector);
            return res;
        }
    };

    configuration.registerTypeHandler(predicate,
new IgnoreFieldsTypeHandler());
    return configuration;
}
```

We can check the result with the following code:

```
TranslatorExample.java: ignoreFieldsTypehandlerExample
private static void ignoreFieldsTypehandlerExample() {
```

```

new File(DB4O_FILE_NAME).delete();
Configuration configuration = ignoreFieldsConfiguration();
ObjectContainer container = Db4o.openFile(configuration, DB4O_FILE_NAME);
try {
    PersistentItem persistentItem =
new PersistentItem("1", new TransientItem(1));
    System.out.println("ORIGINAL: " + persistentItem);
    container.store(persistentItem);
} catch (Exception exc) {
    System.out.println(exc.toString());
    return;
} finally {
    container.close();
}
container = Db4o.openFile(configuration, DB4O_FILE_NAME);
try {
    Query query = container.query();
    query.constrain(PersistentItem.class);
    ObjectSet result = query.execute();
    while (result.hasNext()) {
        PersistentItem persistentItem = (PersistentItem) result.next();
        System.out.println("RETRIEVED: " + persistentItem);
    }
} finally {
    container.close();
}
}

```

Using Annotations

JDK1.5 platform introduced new metadata feature called [annotations](#). Annotations allow programmers to decorate Java code with their attributes, which can be used afterwards for automatic code generation, documentation, security checking etc. Annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program.

Annotations can make your code cleaner, protecting you from common errors (using deprecated API, typos in overriding methods) and taking part of you work.

You can use annotations to affect db4o behavior. At present we provide only one annotation:

`@Indexed`

This annotation can be applied to class fields

```

Car.java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */
package com.db4odoc.annotations;

```

```
import com.db4o.config.annotations.Indexed;

public class Car {
    @Indexed
    private String model;
    private int year;
}
```

and its functionality is equivalent to the db4o configurations setting:

```
Db4o.configure().objectClass(clazz).objectField("fieldName").indexed(true)
```

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 (BI). Bytecode instrumentation is a process of inserting of special, usually short, sequences of bytecode at designated points of Java or .NET class. BI is typically used for profiling or monitoring, however the range of use of bytecode instrumentation is not limited by this tasks: BI can be applied anywhere where a specific functionality should be plugged into the ready built classes.

db4o Enhamncement Tools currently have 2 usecases for bytecode instrumentation:

- [Native Query Optimization](#);
- [Transparent Activation](#)
- [Transparent Persistence](#)

In NQ optimization case bytecode instrumentation is used as a more performant alternative to a run-time optimization. When an NQ is optimized the user and compiler-friendly syntax of NQ predicate is replaced with a query-processor-friendly code (bytecode in the case of BI). 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.

In TA case, classes are required to implement Activatable interface to support transparent activation. In many cases you won't want to "pollute" your proprietary 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 BI comes handy: Activatable interface 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 BI afterwards.

Bytecode instrumentation in Java can be run at build time (also known as static instrumentation). In

this case a special (build) script calls BI on the classes before packaging them to jar, or on the jar itself (classes are extracted, instrumented and jarred again). This is the fastest solution as no time is spent on bytecode instrumentation at runtime.

Another method is to use BI 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.

The following topics discuss BI implementation for db4o needs in more detail and explain the tools and API that should be used for BI tasks.

- [Enhancement For Java](#)

Enhancement For Java

db4o enhancement framework relies on the following jars:

bloat-1.0	Third-party bytecode instrumentation library
db4o-7.12-instrumentation	Instrumentation library on top of bloat
db4o-7.12-tools	Enhancement and other utilities

In addition

- for TA /TP instrumentation enhancement db4o-7.12-taj.jar should be used (contains TA /TP instrumentation classes);
- for NQ optimization db4o-7.12-nqopt.jar is used (provides instrumentation functionality for NQ).

The basic steps required to enhance classes are:

1. Create 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 (see ClassFilter hierarchy for a list of all implementations).
2. Create BloatClassEdit array of classes capable of editing class bytecode. BloatClassEdit is an interface in db4oinstrumentation project. Among its implementations are TranslateNQToSODAEdit (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 db4oinstrumentation 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 db4oinstrumentation 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](#)
- [TA Enhancement at Build Time](#)
- [TP Enhancement at Build Time](#)
- [NQ Enhancement at Loading Time](#)
- [NQ Enhancement at Build Time](#)
- [Complex Example](#)

Complex Example

The following example shows some advanced enhancement features:

- Instrumentation of jarred items
- Annotation based class filter
- TA and NQ enhancement in one go
- Load-time TA enhancement for collections

More Reading:

- [Model Classes](#)
- [Load Time Enhancement](#)
- [Build Time Enhancement](#)

Model Classes

Two simple classes Pilot and Id are pre-packed in a Pilot.jar:

```
Pilot.java
/**/* Copyright (C) 2007 Versant Inc. http://www.db4o.com */
package enhancement.model;

public class Pilot {
    String _name;
    Id _id;

    public Pilot(String name, Id id) {
        _name = name;
        _id = id;
    }

    public String get_name() {
        return _name;
    }
    public void set_name(String _name) {
```

```

        this._name = _name;
    }
    public Id get_id()  {
        return _id;
    }
    public void set_id(Id _id)  {
        this._id = _id;
    }

    public String toString() {
        return _name + ":" + _id;
    }
}

```

```

Id.java
/**/* Copyright (C) 2007 Versant Inc. http://www.db4o.com */
package enhancement.model;

public class Id  {
    String _id;

    public Id(String id) {
        _id = id;
    }

    public String toString() {
        return _id;
    }
}

```

Linked collection of Car objects shows collection enhancement:

```

Car.java
package enhancement.model;

import tacustom.*;

@Db4oPersistent
public class Car  {

    private String _model = null;
    Pilot _pilot;

    public Car(String content, Pilot pilot)  {
        _model = content;
        _pilot = pilot;
    }

    public String content()  {
        return _model;
    }

    public void content(String content)  {
        _model = content;
    }
}

```

```

@Override
public String toString() {
    return _model + "/" + _pilot;
}

public String getModel() {
    return _model;
}

public Pilot getPilot() {
    return _pilot;
}
}

```

```

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

package enhancement.model;

import java.util.*;
import tacustom.*;

@Db4oPersistent

public class MaintenanceQueue<Item> {

    public MaintenanceQueue<Item> _next;

    private Item _value;

    public MaintenanceQueue(Item value) {
        _value = value;
    }

    public static MaintenanceQueue<Integer> newList(int depth) {
        if (depth == 0) {
            return null;
        }
        MaintenanceQueue<Integer> head = new MaintenanceQueue<Integer>(depth);
        head._next = newList(depth - 1);
        return head;
    }

    /**
     * Overrides this method to assert that <code>other</code> is only
     * activated with depth 1.
     */
    @SuppressWarnings("unchecked")
    public boolean equals(Object other) {
        return ((MaintenanceQueue<Item>) other)._next == null;
    }

    public boolean hasNext() {
        return _next != null;
    }
}

```

```

}

public MaintenanceQueue<Item> next()  {
    return _next;
}

public int size()  {
    if(_next == null)  {
        return 1;
    }
    return _next.size() + 1;
}

public Item get(int idx)  {
    if(idx == 0)  {
        return value();
    }
    return _next.queryByExample(idx - 1);
}

public Item value()  {
    return _value;
}

public void add(Item item)  {
    if(_next != null)  {
        _next.add(item);
    }
    else  {
        _next = new MaintenanceQueue<Item>(item);
    }
}

public Iterator<Item> iterator()  {
    return new LinkedListIterator<Item>(this);
}

public String toString()  {
    return "LinkedList: " + _value;
}

public static <Item> MaintenanceQueue<Item> add(MaintenanceQueue<Item> list, Item item)  {
    if(list == null)  {
        return new MaintenanceQueue<Item>(item);
    }
    list.add(item);
    return list;
}

private final static class LinkedListIterator<Item> implements Iterator<Item>  {
    private MaintenanceQueue<Item> _current;

    public LinkedListIterator(MaintenanceQueue<Item> list)  {
        _current = list;
    }
}

```

```

public boolean hasNext()  {
    return _current != null;
}

public Item next()  {
    Item item = _current.value();
    _current  = _current.next();
    return item;
}

public void remove()  {
    throw new UnsupportedOperationException();
}
}

public static <Item> Iterator<Item> iterator(MaintenanceQueue<Item> list)  {
    return (list == null ? new LinkedListIterator<Item>(null) : list.iterator());
}

}

```

Load Time Enhancement

The following code is used to store and retrieve MaintenanceQueue objects containing references to Car, Pilot and Id objects:

```

EnhancerMain.java: main
public static void main(String[] args)  {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer db = Db4o.openFile(configuration(), DB4O_FILE_NAME);
    MaintenanceQueue<Car> queue = null;
    for(int number = 0; number < DEPTH; number++)  {
        queue = MaintenanceQueue.add(queue, new Car("Car" + number,
            new Pilot("Pilot #" + number, new Id("110021" + number))));
    }
    db.store(queue);
    db.close();

    db = Db4o.openFile(configuration(), DB4O_FILE_NAME);
    EventRegistry registry = EventRegistryFactory.forObjectContainer(db);
    registry.activated().addListener(new EventListener4()  {
        public void onEvent(Event4 event, EventArgs args)  {
            ObjectEventArgs objArgs = (ObjectEventArgs) args;
            System.out.println("ACTIVATED: " + objArgs.object());
        }
    });
    ((ObjectContainerBase)db).getNativeQueryHandler().addListener(new Db4oQueryExecutionListener()
        public void notifyQueryExecuted(NQOptimizationInfo info)  {
            System.out.println(info);
        }
    );
}

List<MaintenanceQueue<Car>> result = db.query(new Predicate<MaintenanceQueue<Car>>()
    @Override
    public boolean matches(MaintenanceQueue<Car> item)  {
        return true;
    }
}

```

```

public boolean match(MaintenanceQueue<Car> queue) {
    return queue.value().getModel().equals("Car0");
}
});
System.out.println(result.size());
//for (Iterator<MaintenanceQueue<Car>> i = result.iterator(); i.hasNext();) {
    MaintenanceQueue<Car> carQueue = result.queryByExample(0);
    Car car = carQueue.value();
    System.out.println(car);

    Pilot pilot = car.getPilot();
    System.out.println(pilot);
    while (carQueue.hasNext()) {
        carQueue = carQueue.next();
        car = carQueue.value();
        System.out.println(car);

        pilot = car.getPilot();
        System.out.println(pilot);
    }
}
db.close();
new File(DB4O_FILE_NAME).delete();
}

```

```

EnhancerMain.java: configuration
private static Configuration configuration() {
    Configuration config = Db4o.newConfiguration();
    config.add(new TransparentActivationSupport());
    // NOTE: required for load time instrumentation!
    config.reflectWith(new JdkReflector(EnhancerMain.class.getClassLoader()));
    return config;
}

```

Please, run this method to see that in TA mode all the objects are fully activated immediately. Also NQ info reports that the queries run dynamically optimized.

In order to use TA advantages (lazy activation), we launch the application through an instrumenting classloader. The following configuration options are 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 optimization code unaware of TA. 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.

```
EnhancerStarter.java: main
public static void main(String[] args) throws Exception {
    ClassFilter filter = new ByNameClassFilter("enhancement.", true);
    BloatClassEdit[] edits = { new TranslateNQToSODAEdition() , new InjectTransparentActivationEdition(f
    URL[] urls = { new File("/work/workspaces/db4o/tatest/bin").toURI().toURL() };
    Db4oInstrumentationLauncher.launch(edits, urls, EnhancerMain.class.getName(), new String[] {});
}
```

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.

Build Time Enhancement

For build-time enhancement you will need to put the following jars into the lib folder of EnhancementExample project:

- bloat-1.0.jar
- db4o-7.12-classedit.jar
- db4o-7.12-java5.jar
- db4o-7.12-nqopt.jar
- db4o-7.12-taj.jar
- db4o-7.12-tools.jar
- pilot.jar
- tacustom.jar

pilot.jar supplies Pilot and Id classes presented [before](#)

tacustom.jar provides an annotation-based class filter. Using this jar we can mark the classes that should be enhanced with `@Db4oPersistent` annotation to filter them from the other classes.

You can use the following script to enhance your classes:

```
build.xml
<?xml version="1.0"?>

<!--
 TA and NQ build time instrumentation sample.
-->

<project name="enhance_db4o" default="run-enhanced">

<!-- The classpath needed for the enhancement process,
```

```

including the application classpath -->
<path id="db4o.enhance.path">
  <fileset dir="lib">
    <include name="**/*.jar"/>
  </fileset>
</path>

<!-- Define enhancement tasks (from resource in db4otools.jar). -->
<typedef
  resource="instrumentation-def.properties"
  classpathref="db4o.enhance.path"
  loaderref="db4o.enhance.loader" />

<!-- A custom filter that selects classes with the @Db4oPersistent annotation -->
<typedef
  name="annotation-filter"
  classname="tacustom.AnnotationClassFilter"
  classpathref="db4o.enhance.path"
  loaderref="db4o.enhance.loader" />

<!-- Example for a regexp pattern for selecting classes to be instrumented. -->
<regexp pattern="^enhancement\\.model\\." id="re.model.only" />

<target name="compile">

  <mkdir dir="${basedir}/bin" />
  <delete dir="${basedir}/bin" quiet = "true">
    <include name="**/*"/>
  </delete>

  <javac srcdir="${basedir}/src" destdir="${basedir}/bin" source="1.5" target="1.5">
    <classpath refid="db4o.enhance.path" />
  </javac>

</target>

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

  <!-- Prepare the target folders -->
  <mkdir dir="${basedir}/enhanced-bin" />
  <delete dir="${basedir}/enhanced-bin" quiet = "true">
    <include name="**/*"/>
  </delete>
  <mkdir dir="${basedir}/enhanced-lib" />
  <delete dir="${basedir}/enhanced-lib" quiet = "true">
    <include name="**/*"/>
  </delete>

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

  <classpath refid="db4o.enhance.path" />

```

```

<!-- Fileset for original class files to be instrumented -->
<sources dir="${basedir}/bin">
    <include name="enhancement/**/*.class" />
</sources>
<!-- Fileset for original jars to be instrumented -->
<jars dir="${basedir}/lib">
    <include name="pilot.jar" />
</jars>

<!-- Instrument Native Query predicates -->
<native-query-step />

        <!-- Instrument TA field access. -->
<transparent-activation-step>
    <!-- Instrument classes that are annotated as @Db4oPersistent. -->
    <annotation-filter />
    <!-- Instrument classes from the specified paths only. -->
    <regexp refid="re.model.only" />
        <regexp pattern="^enhancement\\.model\\." />
</transparent-activation-step>

</db4o-instrument>

</target>

<target name="run-unenhanced" depends="compile">

<java classname="enhancement.EnhancerMain" failonerror="true">
    <classpath>
        <pathelement location="${basedir}/bin" />
        <pathelement location="${basedir}/lib/pilot.jar" />
        <path refid="db4o.enhance.path" />
    </classpath>
</java>
</target>

<target name="run-enhanced" depends="enhance">

<java classname="enhancement.EnhancerMain" failonerror="true">
    <classpath>
        <pathelement location="${basedir}/enhanced-bin" />
        <pathelement location="${basedir}/enhanced-lib/pilot.jar" />
        <path refid="db4o.enhance.path" />
    </classpath>
</java>
</target>

</project>

```

The core part of this script is inside the *db4o-instrument* task, which is imported by the first *typedef* instruction. (The second *typedef* imports the custom annotation class filter.) The *classTargetDir* and *jarTargetDir* attributes specify the target folders where instrumented class files and

instrumented jar files should be created, respectively.

The nested *classpath* is just a normal Ant path type and should cover the full application classpath. In the example, we are using one single classpath for task definition and application for convenience - in a real project, these are better kept separate, of course. The nested *sources* FileSet specifies the location of the class files to be instrumented. Similarly, the *jars* FileSet specifies the location of jar files to be instrumented. Both are optional (providing neither *sources* nor *jars* doesn't make much sense, of course). If one is left out, the corresponding target folder attribute is not required, either.

The remaining nested arguments specify the instrumentation steps to be processed. For Native Query optimization, there is no further configuration - it will simply try to instrument all Predicate implementations. Transparent Activation instrumentation allows to specify more fine-grained filters to select the classes to be instrumented. This can be Ant regular expression types or arbitrary custom *ClassFilters*. These are OR-ed together and used to further constrain the implicit filter provided by the *sources/jars* FileSets. In the example, we are constraining TA instrumentation to classes that are either annotated with the *@Db4oPersistent* annotation, or whose fully qualified name matches the given regexes.

After running the *enhance* target, the instrumented model classes should appear in the enhanced-bin folder, and an instrumented version of the pilot.jar should have been created in the enhanced-lib folder.

For rather straightforward projects you can alternatively use the *db4o-enhance* task variant that provides a default setting for joint NQ/TA instrumentation (but doesn't allow fine-grained configuration for the single instrumentation steps in return). This is demonstrated by the following build script for the same sample project.

```
build-simple.xml
<?xml version="1.0"?>

<!--
  Simple TA and NQ build time instrumentation sample.

  This version uses db4o-enhance instead of db4o-instrument.
  db4o-enhance provides a default
    configuration for NQ/TA instrumentation,
  while db4o-instrument requires to configure
    (and optionally fine-tune) the single instrumentation
  steps. Other than that, the
    configuration options for the two are identical.
-->

<project name="enhance_db4o" default="run-enhanced">

<!-- The classpath needed for the enhancement process,
including the application classpath -->
<path id="db4o.enhance.path">
  <pathelement path="${basedir}" />
```

```

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

<!-- Define enhancement tasks (from resource in db4otools.jar). -->
<typedef
    resource="instrumentation-def.properties"
    classpathref="db4o.enhance.path" />

<target name="compile">

    <mkdir dir="${basedir}/bin" />
    <delete dir="${basedir}/bin" quiet = "true">
        <include name="**/*"/>
    </delete>

    <javac srcdir="${basedir}/src" destdir="${basedir}/bin">
        <classpath refid="db4o.enhance.path" />
    </javac>

</target>

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

    <!-- Prepare the target folders -->
    <mkdir dir="${basedir}/enhanced-bin" />
    <delete dir="${basedir}/enhanced-bin" quiet = "true">
        <include name="**/*"/>
    </delete>
    <mkdir dir="${basedir}/enhanced-lib" />
    <delete dir="${basedir}/enhanced-lib" quiet = "true">
        <include name="**/*"/>
    </delete>

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

        <classpath refid="db4o.enhance.path" />
        <!-- Fileset for original class files to be instrumented -->
        <sources dir="${basedir}/bin" />
        <!-- Fileset for original jars to be instrumented -->
        <jars dir="${basedir}/lib">
            <include name="pilot.jar" />
        </jars>

    </db4o-enhance>
</target>

<target name="run-unenhanced" depends="compile">

```

```

<java classname="enhancement.EnhancerMain" failonerror="true">
    <classpath>
        <pathelement location="${basedir}/bin" />
        <pathelement location="${basedir}/lib/pilot.jar" />
        <path refid="db4o.enhance.path" />
    </classpath>
</java>

</target>

<target name="run-enhanced" depends="enhance">

    <java classname="enhancement.EnhancerMain" failonerror="true">
        <classpath>
            <pathelement location="${basedir}/enhanced-bin" />
            <pathelement location="${basedir}/enhanced-lib/pilot.jar" />
            <path refid="db4o.enhance.path" />
        </classpath>
    </java>
</target>

</project>

```

Simplest Enhancement Script

If you are not interested in writing complicated ant build scripts for your project, but still need to use enhancement, we've created a simplest enhance script that can do the job for any java project with only minor changes or without any changes at all. Here is the script:

```

Build.Xml
<?xml version="1.0"?>
<project name="db4o enhance project" default="enhance" basedir=".">
    <path id="project.classpath">
        <pathelement path="${basedir}/bin" />
        <fileset dir="lib">
            <include name="**/*.jar" />
        </fileset>
    </path>
    <taskdef name="db4o-enhance"
        classname="com.db4o.enhance.Db4oEnhancerAntTask"
        classpathref="project.classpath" />
    <target name="enhance">
        <db4o-enhance classtargetdir="${basedir}/bin"
            jartargetdir="${basedir}/lib" nq="true" ta="true"
            collections="true">
            <classpath refid="project.classpath" />
            <sources dir="${basedir}/bin" />
            <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>
</project>
```

In order to use it - just put it in your project's folder. You will also need to put db4o-7.12-all-java5.jar (it contains all the required jars) into your lib folder. If your project uses non-standard Java layout (web application, maven application etc), you will have to correct the paths to bin and lib folders in the build.xml.

This build.xml file includes TA/TP enhancement (normal and collections) and NQ optimization. These are the default options. For a more detailed description of Ant instrumentation the following options can be used:

nq: switch for native query optimization enhancement

ta: switch for transparent activation/persistence enhancement

collections: switch for enhancement of platform collections for transparent activation/persistence (only effective if ta is switched on)

In order to run the enhancement you can just type "ant" in the command line in your project's directory.

You can also integrate this build in the build process in your IDE. If you are using Eclipse follow these steps:

- Right-click the project in the Package Explorer and select "Properties"
- Select "Builders" in the left pane
- Add new builder after the default Java builder, call it "Enhancement"
- Select the build.xml file on the "Main" tab of the new builder properties

Now enhancement will be run after the normal compilation (you may need to clean and rebuild the project to see the effect).

Unique Constraints

Unique Constraints feature was first introduced in db4o 6.2.

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.

Multiple constraints can be defined on the same class if required.

More Reading:

- [How To Use Unique Constraints](#)
- [Unique Constraints Example](#)

How To Use Unique Constraints

In order to work with the unique constraints you will need to remember the following 3 steps.

1. Add an index for a field you wish to be unique:

Java:

```
configuration.objectClass(Item.class).objectField("field").indexed(true);
```

Unique Constraints Example

Let's look at a simple example with 2 clients adding objects of the same class to the database. First of all, let's create an appropriate configuration:

```
UniqueConstraintExample.java: configure
private static Configuration configure() {
    Configuration configuration = Db4o.newConfiguration();
    configuration.objectClass(Pilot.class).objectField("name")
        .indexed(true);
    configuration.add(new UniqueFieldValueConstraint(Pilot.class,
        "name"));
    return configuration;
}
```

Configuration returned by `configure` method will be passed to the server.

```
UniqueConstraintExample.java: storeObjects
private static void storeObjects() {
    new File(FILENAME).delete();
    ObjectServer server = Db4o.openServer(configure(), FILENAME,
        0);
    Pilot pilot1 = null;
    Pilot pilot2 = null;
    try {
        ObjectContainer client1 = server.openClient();
        try {
            // creating and storing pilot1 to the database
            pilot1 = new Pilot("Rubens Barichello", 99);
            client1.store(pilot1);
```

```

ObjectContainer client2 = server.openClient();
try {
    // creating and storing pilot2 to the database
    pilot2 = new Pilot("Rubens Barichello", 100);
    client2.store(pilot2);
    // commit the changes
    client2.commit();
} catch (UniqueFieldValueConstraintViolationException ex) {
    System.out
        .println("Unique constraint violation in client2 saving: "
            + pilot2);
    client2.rollback();
} finally {
    client2.close();
}
// Pilot Rubens Barichello is already in the database,
// commit will fail
client1.commit();
} catch (UniqueFieldValueConstraintViolationException ex) {
    System.out
        .println("Unique constraint violation in client1 saving: "
            + pilot1);
    client1.rollback();
} finally {
    client1.close();
}
} finally {
    server.close();
}
}
}

```

Running this example you will get an exception trying to commit `client1` as the changes made by `client2` containing a `Pilot` object with the same name are already committed to the database.

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

Callbacks

External Callbacks enable you to add Listeners to an ObjectContainer for the following db4o events

- QueryStarted
- QueryFinished
- Creating (first time an object is about to be saved)
- Created (after the object is saved)
- Activating
- Activated
- Deactivating
- Deactivated
- Updating
- Updated
- Deleting (not available in networked client/server mode)
- Deleted (not available in networked client/server mode)
- Committing
- Committed

QueryStarted and QueryFinished events accept QueryEventArgs as a parameter and can be used to gather query statistics information.

Created, Activated, Deactivated, Updated and Deleted events accept ObjectEventArgs and can be used to gather statistics information or to initiate some special behavior after the action has been taken.

Creating, Activating, Deactivating, Updating and Deleting events accept CancellableObjectEventArgs. Their primary usage is to perform action validity check and to stop the execution if necessary.

Committing event can be used to check some application-specific conditions before commit. For example it can

be used to check unique constraints. Committing event accepts CommitEventArgs as a parameter.

Committed event is raised when the container has completely finished the commit operation. Event subscribers get the notification in a separate thread. Committed event accepts CommitEventArgs.

More Reading:

- [Event Registry API](#)
- [Possible Usecases](#)
- [Benefits](#)
- [Commit-Time Callbacks](#)

Event Registry API

External callbacks should be registered with db4o EventRegistry. Follow the steps below to start using your own event handlers:

1. Obtain an instance of EventRegistry object for your ObjectContainer

Java:

```
EventRegistry registry = EventRegistryFactory.forObjectContainer(container);
```

2. Register the required event. For "created" event the code is the following:

Java:

```
registry.created().addListener(EventListener4)
```

3. Create your own event handler:

Java:

```
EventListener4 createdEvent = new EventListener4(){ onEvent(Event4 event, EventArgs args) { // handling code } }
```

The action raised the event can be canceled in Creating, Activating, Deactivating, Updating and Deleting event handlers. These events accept CancellableEventArgs as a parameter. In order to cancel the action use:

Java:

```
cancellableEventArgs.cancel()
```

Here cancellableEventArgs is an event argument of CancellableEventArgs type.

In java cancellableEventArgs should be obtained by explicit casting:

```
EventListener4 listener = new EventListener4() {  
  
    public void onEvent(Event4 e, EventArgs args) {  
  
        CancellableEventArgs cancellableArgs = (CancellableEventArgs)args;  
  
        ....  
    }  
}
```

4. After the work is done you can unregister the events:

Java:

```
registry.created().removeListener(createdEvent);
```

EventRegistry features:

- You can register several event handlers for a single event.
- You can get different EventRegistry's for different ObjectContainer instances using EventRegistryFactory/IEventRegistryFactory.
- In Java, callbacks are implemented as Listeners, .NET uses Native events
- Callbacks only work run in local mode or on the server side in client/server mode.
- Each event applies to all the objects or queries(QueryStarted/QueryFinished events). In order to distinguish the specific case, to which the handler should be applied, use the event arguments.

For example:

```
CallbacksExample.java: testCreated  
private static void testCreated() {  
    new File(DB4O_FILE_NAME).delete();  
    ObjectContainer container = Db4oEmbedded.openFile(Db4oEmbedded  
        .newConfiguration(), DB4O_FILE_NAME);  
    try {  
        EventRegistry registry = EventRegistryFactory  
            .forObjectContainer(container);  
        // register an event handler, which will print all the car objects,  
        // that have been created  
        registry.created().addListener(new EventListener4() {  
            public void onEvent(Event4 e, EventArgs args) {  
                ObjectEventArgs queryArgs = ((ObjectEventArgs) args);  
            }  
        });  
    } catch (Db4oIOException e) {  
        e.printStackTrace();  
    }  
}
```

```

        Object obj = queryArgs.object();
        if (obj instanceof Pilot) {
            System.out.println(obj.toString());
        }
    });
}

Car car = new Car("BMW", new Pilot("Rubens Barrichello"));
container.store(car);
} finally {
    container.close();
}
}

```

Possible Usecases

There are many usecases for external callbacks, including:

- cascaded deletes, updates
- referential integrity checks
- gathering statistics
- autoassigned 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:

- [Cascaded Behavior](#)
- [Referential Integrity](#)
- [Query Statistics](#)
- [Autoincrement](#)

Cascaded Behavior

You can use different object events to initiate cascaded behavior on update, activate, delete. The following example shows how to ensure that all the referenced objects are deleted when the parent object is deleted:

```

CallbacksExample.java: testCascadedDelete
private static void testCascadedDelete()  {
    fillDB();
    final ObjectContainer container = Db4oEmbedded.openFile(DB4O_FILE_NAME);
    try  {

```

```

// check the contents of the database
List<Object> result = container.queryByExample(null);
listResult(result);

EventRegistry registry = EventRegistryFactory
    .forObjectContainer(container);
// register an event handler, which will delete the pilot when his
// car is deleted
registry.deleted().addListener(new EventListener4() {
    public void onEvent(Event4 e, EventArgs args) {
        ObjectEventArgs queryArgs = ((ObjectEventArgs) args);
        Object obj = queryArgs.object();
        if (obj instanceof Car) {
            container.delete(((Car) obj).getPilot());
        }
    }
});
// delete all the cars
List<Car> cars = container.query(Car.class);
for (Car c: cars) {
    container.delete(c);
}
// check if the database is empty
result = container.queryByExample(null);
listResult(result);
} finally {
    container.close();
}
}
}

```

Note, that in client/server mode deleted event is only raised on the server side, therefore the code above won't work.

Autoincrement

Db4o does not deliver a field autoincrement feature, which is common in RDBMS. If your application logic requires this feature you can implement it using External Callbacks. One of the possible solutions is presented below.

We will need an object to store the last generated ID and to return a new ID on request:

```

IncrementedId.java
/**/* Copyright (C) 2009 Versant Inc. http://www.db4o.com */
/**/*
 * Singleton class used to keep auotincrement information
 * and give the next available ID on request
 */
package com.db4odoc.autoincrement;

```

```

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

public class IncrementedId {
    private int no;
    private static IncrementedId ref;

    private IncrementedId() {
        this.no = 0;
    }

    // end IncrementedId

    public int getNextID(ObjectContainer db) {
        no++;
        db.store(this);
        return no;
    }

    // end increment

    public static IncrementedId getIdObject(ObjectContainer db) {
        // if ref is not assigned yet:
        if (ref == null) {
            // check if there is a stored instance from the previous
            // session in the database
            ObjectSet<IncrementedId> os = db.queryByExample(IncrementedId.class);
            if (os.size() > 0)
                ref = (IncrementedId) os.next();
        }

        if (ref == null) {
            // create new instance and store it
            System.out.println("Id object is created");
            ref = new IncrementedId();
            db.store(ref);
        }
        return ref;
    }
    // end getIdObject
}

```

You can test the results with the following code:

```

AutoIncExample.java: storeObjects
public static void storeObjects(ObjectContainer db) {
    TestObject test;
    test = new TestObject("FirstObject");
    db.store(test);
    test = new TestObject("SecondObject");
    db.store(test);
    test = new TestObject("ThirdObject");
}

```

```
    db.store(test);
}
```

Please, note that the suggested implementation **cannot be used in a multithreaded environment**. In such environment you will have to make sure that the IncrementedId class can only be saved to the database once, and that 2 threads cannot independently and simultaneously increment IncrementedId counter.

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:

```
CallbacksExample.java: testIntegrityCheck
private static void testIntegrityCheck()  {
    fillDB();
    final ObjectContainer container = Db4oEmbedded.openFile(DB4O_FILE_NAME);
    try  {
        EventRegistry registry = EventRegistryFactory
            .forObjectContainer(container);
        // register an event handler, which will stop deleting a pilot when
        // it is referenced from a car
        registry.deleting().addListener(new EventListener4()  {
            public void onEvent(Event4 e, EventArgs args)  {
                CancellableEventArgs cancellableArgs = ((CancellableEventArgs) args);
                Object obj = cancellableArgs.object();
                if (obj instanceof Pilot)  {
                    Query q = container.query();
                    q.constrain(Car.class);
                    q.descend("pilot").constrain(obj);
                    ObjectSet result = q.execute();
                    if (result.size() > 0)  {
                        System.out
                            .println("Object "
                                + (Pilot) obj
                                + " can't be deleted as object container has references to it");
                        cancellableArgs.cancel();
                    }
                }
            }
        });
        // check the contents of the database
        List result = container.queryByExample(null);
        listResult(result);
    }
}
```

```

// try to delete all the pilots
List<Pilot> pilots = container.query(Pilot.class);
for (Pilot p: pilots) {
    container.delete(p);
}
// check if any of the objects were deleted
result = container.queryByExample(null);
listResult(result);
} finally {
    container.close();
}
}

```

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.

Query Statistics

QueryStarted and QueryFinished events can be used to gather various statistics about query execution.

For example you can use queryStarted() and queryFinished() events to calculate query time, and activated() event to count objects activated in a query.

Java version provides an example of query events usage in `QueryStats` class from tools package.

In order to use `QueryStats`:

1. Connect a `QueryStats` object to the `ObjectContainer`

```

QueryStats stats = new QueryStats();
stats.connect(db);

```

2. Execute your query

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

3. Get the metrics from the `QueryStats` object

```

long executionTime = stats.executionTime();
int activationCount = stats.activationCount();

```

4. After the work is done you can disconnect `QueryStats` from the `ObjectContainer`

```
stats.disconnect();
```

Benefits

External callbacks help you to solve many different problems and customize db4o behavior. Among their benefits:

- With external callbacks you do not have to pollute your object model with persistence code. This is exceptionally valuable when the objects are inherited from external application or library.
- Multiple event handlers can be registered on particular events, keeping your code clean and easily readable.
- You can "plug-in" different modules to perform different tasks. An example can be a module responsible for assigning unique IDs to your objects.

Commit-Time Callbacks

Commit-time callbacks were introduced in db4o version 6.2.

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.

More Reading:

- [How To Use Commit-Time Callbacks](#)
- [Committing Event Example](#)
- [Committed Event Example](#)
- [Car](#)

How To Use Commit-Time Callbacks

The general usage of external callbacks is described in [Event Registry API](#). Commit-time event handlers can be defined:

Java:

```
registry.committing().addListener(new EventListener4() { ... });
registry.committed().addListener(new EventListener4() { ... });
```

Event arguments can be reached through `CommitEventArgs` parameter. `CommitEventArgs` class provides the following properties:

`added` - lists the objects added in the current transaction.

`deleted` - lists the objects deleted in the current transaction.

`updated` - lists the objects updated in the current transaction.

`transaction` - returns the current transaction.

Committed Event Example

Committed callbacks can be used in various scenarios:

- backup on commit;
- database replication on commit;
- client database synchronization.

In our example we will create an implementation for the last case.

When several clients are working on the same objects it is very possible that the data will be outdated on some of the clients. Before the commit-callbacks feature was introduced the solution was to call `refresh` regularly to get object updates from the server. With the commit-callback this process can be easily automated:

- objects are modified when the commit is done;
- the successful commit triggers committed event on the clients;
- committed event handler updates modified objects on the clients.

Let's open 2 clients, which will work with [Car](#) objects, and register committed event listeners for them.

```
PushedUpdatesExample.java: openClient
private ObjectContainer openClient() {
```

```

try {
    ObjectContainer client = Db4oClientServer.openClient(
        Db4oClientServer.newClientConfiguration(), "localhost",
        PORT, USER, PASSWORD);
    EventListener4 committedEventListener = createCommittedEventListener();
    EventRegistry eventRegistry = EventRegistryFactory
        .forObjectContainer(client);
    eventRegistry.committed().addListener(committedEventListener);
    // save the client-listener pair in a map, so that we can
    // remove the listener later
    clientListeners.put(client, committedEventListener);
    return client;
} catch (Exception ex) {
    ex.printStackTrace();
}
return null;
}

```

```

PushedUpdatesExample.java: createCommittedEventListener
private EventListener4 createCommittedEventListener() {
    return new EventListener4() {
        public void onEvent(Event4 e, EventArgs args) {
            CommitEventArgs cArgs = (CommitEventArgs) args;
            Transaction trans = (Transaction)cArgs.transaction();
            // get all the updated objects
            ObjectInfoCollection updated = cArgs.updated();
            Iterator4 infos = updated.iterator();
            while (infos.moveToNext()) {
                ObjectInfo info = (ObjectInfo) infos.current();
                Object obj = info.getObject();
                // refresh object on the client
                trans.objectContainer().ext().refresh(obj, 2);
            }
        }
    };
}

```

Run the following method to see how the 2 clients work concurrently on the same object:

```

PushedUpdatesExample.java: run
public void run() throws IOException, DatabaseFileLockedException {
    new File(DB4O_FILE_NAME).delete();
    ObjectServer server =
        Db4oClientServer.openServer(Db4oClientServer
            .newServerConfiguration(), DB4O_FILE_NAME, PORT);
    try {
        server.grantAccess(USER, PASSWORD);

        ObjectContainer client1 = openClient();
        ObjectContainer client2 = openClient();

        if (client1 != null && client2 != null) {

```

```

try  {
    // wait for the operations to finish
    waitForCompletion();

    // save pilot with client1
    Car client1Car = new Car("Ferrari", 2006, new Pilot(
        "Schumacher"));
    client1.store(client1Car);
    client1.commit();

    waitForCompletion();

    // retrieve the same pilot with client2
    Car client2Car = (Car) client2.query(Car.class).next();
    System.out.println(client2Car);

    // modify the pilot with client1
    client1Car.setModel(2007);
    client1Car.setPilot(new Pilot("Hakkinen"));
    client1.store(client1Car);
    client1.commit();

    waitForCompletion();

    // client2Car has been automatically updated in
    // the committed event handler because of the
    // modification and the commit by client1
    System.out.println(client2Car);

    waitForCompletion();
} catch (Exception ex)  {
    ex.printStackTrace();
} finally {
    closeClient(client1);
    closeClient(client2);
}
}

} catch (Exception ex)  {
ex.printStackTrace();
} finally {
server.close();
}
}

```

You should see that client2 picked up the changes committed from the client1 automatically due to the committed event handler.

Working with the committed event you should remember that the listener is called in a separate thread, which needs to be synchronized with the main application thread. This functionality is not implemented in the presented example, instead a simple thread Sleep(1000) method is used (Wait-

ForCompletion method), which is not reliable at all. For a reliable execution use events and notifications from the committed callbacks.

It is a good practice to remove the committed event handlers from the registry before shutting down the clients:

```
PushedUpdatesExample.java: closeClient
private void closeClient(ObjectContainer client)  {
    // remove listeners before shutting down
    if (clientListeners.queryByExample(client) != null)  {
        EventRegistry eventRegistry = EventRegistryFactory
            .forObjectContainer(client);
        eventRegistry.committed().removeListener(
            clientListeners.queryByExample(client));
        clientListeners.remove(client);
    }
    client.close();
}
```

The example presented above works very well for remote client-server mode when object identity recognition between clients and the server is managed by internal core logic. However, if you will try the same in embedded client-server you will find out that the committed event handler is actually useless. Why is it so? In embedded mode clients and server objects are kept in the same operating memory, but each client and the server own their own fragment of it where the objects are instantiated. This means that the same object (Car in our example) will be instantiated twice: for client1 and for client2. In order to recognize those instances as the same database object, we will need to add id comparison to the event handler:

```
PushedUpdatesExample.java: createCommittedEventListenerForEmbedded
private EventListener4 createCommittedEventListenerForEmbedded(
    final ObjectContainer objectContainer)  {
    return new EventListener4()  {
        public void onEvent(Event4 e, EventArgs args)  {
            Transaction trans =
((InternalObjectContainer)objectContainer).transaction();

            // get all the updated objects
            ObjectInfoCollection updated = ((CommitEventArgs) args)
                .updated();
            Iterator4 infos = updated.iterator();
            while (infos.moveNext())  {
                ObjectInfo info = (ObjectInfo) infos.current();
                // Obtain object reference in local cache by object id
                Object obj = trans.objectForIdFromCache((int)info.getInternalID());

                if (obj != null)  {
                    // refresh object on the client
                    objectContainer.ext().refresh(obj, 2);
                }
            }
        }
    };
}
```

```
        }
    }
}
};
```

The logic above will link the changed objects from one client to the objects in memory of another client and will update them if necessary.

Committing Event Example

Let's look at an example of committing event usage. In this example committing callback will be used to ensure that only unique objects can be saved to the database. Duplicate objects - objects, which have the same fields - will cause an exception at the commit time.

We will use a simple Item class for an example:

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

package com.db4odoc.commitcallbacks;

public class Item {
    private int _number;
    private String _word;

    public Item(int number, String word) {
        _number = number;
        _word = word;
    }

    public String getWord() {
        return _word;
    }

    public int getNumber() {
        return _number;
    }

    public String toString() {
        return _number + "/" + _word;
    }
}
```

The following methods will configure a commit-time callback method for uniqueness check:

```
CommitCallbackExample.java: configure
```

```

private static void configure() {
    EventRegistry registry = EventRegistryFactory
        .forObjectContainer(container());
    // register an event handler, which will check object uniqueness on
    // commit
    registry.committing().addListener(new EventListener4() {
        public void onEvent(Event4 e, EventArgs args) {
            CommitEventArgs commitArgs = ((CommitEventArgs) args);
            // uniqueness should be checked for both added and updated
            // objects
            checkUniqueness(commitArgs.added());
            checkUniqueness(commitArgs.updated());
        }
    });
}

```

CommitCallbackExample.java: checkUniqueness

```

private static void checkUniqueness(ObjectInfoCollection collection) {
    Iterator4 iterator = collection.iterator();
    while (iterator.moveNext()) {
        ObjectInfo info = (ObjectInfo) iterator.current();
        // only check for Item objects
        if (info.getObject() instanceof Item) {
            Item item = (Item) info.getObject();
            // search for objects with the same fields in the database
            List found = container().queryByExample(
                new Item(item.getNumber(), item.getWord()));
            if (found.size() > 1) {
                throw new Db4oException("Object is not unique: " + item);
            }
        }
    }
}

```

let's save one initial object `Item(1, "one")`

CommitCallbackExample.java: storeFirstObject

```

private static void storeFirstObject() {
    ObjectContainer container = container();
    try {
        // creating and storing item1 to the database
        Item item = new Item(1, "one");
        container.store(item);
        // no problems here
        container.commit();
    } catch (Db4oException ex) {
        System.out.println(ex.getMessage());
        container.rollback();
    }
}

```

Now we can check the functionality of the committing callback using the following code:

```
CommitCallbackExample.java: storeOtherObjects
private static void storeOtherObjects()  {
    ObjectContainer container = container();
    // creating and storing similar items to the database
    Item item = new Item(2, "one");
    container.store(item);
    item = new Item(1, "two");
    container.store(item);
    try  {
        // commit should work as there were no duplicate objects
        container.commit();
    } catch (Db4oException ex)  {
        System.out.println(ex.getMessage());
        container.rollback();
    }
    System.out.println("Commit successful");

    // trying to save a duplicate object to the database
    item = new Item(1, "one");
    container.store(item);
    try  {
        // Commit should fail as duplicates are not allowed
        container.commit();
    } catch (Db4oException ex)  {
        System.out.println(ex.getMessage());
        container.rollback();
    }
}
```

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);
}
}
```

Storage

Db4o allows a user to configure an IO storage mechanism to be used. Currently db4o provides the following mechanisms:

- [FileStorage](#)
- [CachingStorage](#)
- [MemoryStorage](#)
- [NonFlushingStorage](#)

It is also possible to create your own custom mechanism by [implementing Storage interface](#). Possible usecases for a custom IO mechanism:

- improved performance with a native library
- mirrored write to 2 files
- encryption
- read-on-write fail-safety control

FileStorage

FileStorage is the base storage mechanism, providing the functionality of file access. It is used as the underlying mechanism for other storage implementations provided by db4o. For example:

```
new NonFlushingStorage(new FileStorage());
```

Normally the above limits the usage of the FileStorage class. However, you can also use it as the main storage mechanism for your database, if performance is not important to you.

Java:

```
EmbeddedConfiguration#file().storage(new FileStorage());
```

The benefit of using FileStorage is in [decreased memory consumption](#).

Memory Storage

MemoryStorage allows to create and use db4o database fully in RAM. This strategy eliminates long disk access times making db4o much faster.

MemoryStorage can be used both in embedded and client-server modes:

Java:

```
//opening an embedded session
embeddedConfiguration = Db4oEmbedded.newConfiguration();

embeddedConfiguration.file().storage(new MemoryStorage());

Db4oEmbedded.openFile(embeddedConfiguration,
"myEmbeddedDb.db4o");

// opening a server for a client/server
sessionServerConfiguration serverConfiguration =
Db4oClientServer.newServerConfiguration();
serverConfiguration.file().storage(new MemoryStorage());

Db4oClientServer.openServer(serverConfiguration,
"myServerDb.db4o", PORT);
```

MemoryStorage can be created without any additional parameters passed to the constructor. In this case default configuration values will be used. At present the following configuration settings are available for MemoryStorage:

Growth Strategy

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.

Java:

```
MemoryStorage ms = new MemoryStorage(new ConstantGrowthStrategy(growthSize))
```

growthSize - is the size in bytes, which will be used to increase the reserved database storage size.

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 (see Reuse-

MemoryBin). MemoryBins are identified by their URI, i.e. when an objectContainer 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.

More Reading:

- [Simple Example](#)
- [Storing MemoryBin Data](#)
- [Reusing MemoryBins](#)

Simple Example

Let's have a look at a simple example. For reuse, it is convenient to create a separate method, which will produce the necessary configuration:

```
MemoryStorageExample.java: getInMemoryConfig
private static EmbeddedConfiguration getInMemoryConfig()  {
    MemoryStorage ms = new MemoryStorage();
    EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
    config.file().storage(ms);
    return config;
}
```

In this method we use create a default MemoryStorage with doubling growth strategy and a new bin.

From the following code you can see that further usage is the same as with the default disk storage:

```
MemoryStorageExample.java: createDatabase
private static void createDatabase(EmbeddedConfiguration config)  {
    // the following line is only necessary for the database on disk
    new File(DB4O_URI).delete();

    long startTime = System.currentTimeMillis();
    // DB4O_URI will be the name of the database file for the default
    // configuration and the name of a MemoryBin for in-memory database
    ObjectContainer container = Db4oEmbedded.openFile(config, DB4O_URI);
```

```

try {
    Library lib = new Library();
    for (int i = 0; i < OBJECT_COUNT; i++) {
        lib.addBook(new Book("title" + i, lib));
    }
    container.store(lib);
    Book book = (Book) container.queryByExample(
        new Book("title1", null)).queryByExample(0);
    System.out.println(book);
} finally {
    container.close();
}
System.out.println(String.format("Time to create a database: %d ms",
    System.currentTimeMillis() - startTime));
}

```

To demonstrate the performance improvement, please run the method above with the default configuration:

```

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

```

Storing MemoryBin Data

In the [previous example](#) we've learned how to use a MemoryStorage to create a database and run queries in it. However, in most of the cases you will still need a backup of the data in memory earlier or later. To get it, you will need to close the object container and store the contents of the MemoryBin to a normal file.

```

MemoryStorageExample.java: storeMemoryBin
public static void storeMemoryBin() throws IOException {
    MemoryStorage origStorage = new MemoryStorage();
    EmbeddedConfiguration origConfig = Db4oEmbedded.newConfiguration();
    origConfig.file().storage(origStorage);
    ObjectContainer origDb = Db4oEmbedded.openFile(origConfig, DB4O_URI);
    origDb.store(new Book("Alice in Wonderland"));
    origDb.close();

    MemoryBin origBin = origStorage.bin(DB4O_URI);
    byte[] data = origBin.data();
    OutputStream out = new FileOutputStream("memory.db4o");
    out.write(data, 0, data.length);
    out.close();
}

```

```

ObjectContainer newDb = Db4oEmbedded.openFile(Db4oEmbedded
    .newConfiguration(), "memory.db4o");
Book book = (Book) newDb
    .queryByExample(new Book("Alice in Wonderland")).queryByExample(0);
System.out.println(book);
newDb.close();
}

```

Note, that the result file can be used as a normal db4o database.

Reusing MemoryBins

In the [previous example](#) we saw how a MemoryBin can be stored to a normal disk file. You can also transfer it to the other parts of an application as a normal variable (byte array).

```

MemoryStorageExample.java: reuseMemoryBin
public static void reuseMemoryBin()  {
    // Create original storage
    MemoryStorage origStorage = new MemoryStorage();
    EmbeddedConfiguration origConfig = Db4oEmbedded.newConfiguration();
    origConfig.file().storage(origStorage);
    // A new MemoryBin identified by DB4O_URI will be created
    ObjectContainer origDb = Db4oEmbedded.openFile(origConfig, DB4O_URI);
    origDb.store(new Book("Alice in Wonderland"));
    origDb.close();

    // Retrieve original bin by its URI
    MemoryBin origBin = origStorage.bin(DB4O_URI);
    // This bin contains the database data
    byte[] data = origBin.data();

    // Create a new bin, using the data from the original bin
    MemoryBin newBin = new MemoryBin(data, new DoublingGrowthStrategy());
    MemoryStorage newStorage = new MemoryStorage();
    // Assign the new bin to a new MemoryStorage using the same DB4O_URI
    newStorage.bin(DB4O_URI, newBin);
    EmbeddedConfiguration newConfig = Db4oEmbedded.newConfiguration();
    newConfig.file().storage(origStorage);
    // Open object container using the new bin with the original data
    ObjectContainer newDb = Db4oEmbedded.openFile(newConfig, DB4O_URI);
    // test that the data is still there
    Book book = (Book) newDb
        .queryByExample(new Book("Alice in Wonderland")).queryByExample(0);
    System.out.println(book);
    newDb.close();
}

```

From this and [previous example](#), you can see that MemoryBin API gives you lots of flexibility: combining persistence and exchange strategies you can achieve more complex functionality, such as synchronization, archiving, versioning, disconnected remote mode etc.

CachingStorage

CachingStorage is db4o default storage.

Default implementation uses LRU/Q2 caching mechanism to reduce disk access times and improve performance. You can find the LRU/Q2 description on the internet or you can look for the concrete implementation in db4o source code: LRU2QCache, LRU2QXCache and LRUCache. By default, CachingStorage class makes use of LRU2QCache, however other implementations can be used as well, as it will be discussed later.

Cache is characterized by the amount of pages that can be utilized and the page size. The multiplication of these 2 parameters gives the maximum cache size that can be used.

Bigger page size means faster handling of the information as there is no need to switch between pages for longer. On the other hand bigger page size will mean higher memory consumption, as memory will be reserved for the whole page size, when the new page is needed. Modifying these values and running performance tests, you can achieve the best performance/memory consumption combination for your system. The default values are the following:

Page count = 64 Page size = 1024,

Which gives us a total of 64 KB of cache memory.

In order to modify page size and amount use the following configuration calls:

Java:

```
// config is an instance of either EmbeddedConfiguration or ServerConfiguration  
config.file().storage(new CachingStorage(new FileStorage(), PAGE_COUNT, PAGE_SIZE));
```

For more information on Page size and count please see [Cache Configuration Example](#).

By default db4o uses LRU2QCache, which is a simplified implementation of the 2 Queue algorithm described here:

<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.34.2641>

You can also make use of the full version of the algorithm or a simple LRU, by overriding newCache/NewCache method in CachingStorage class:

```
FullLRUCachingStorage.java
```

```

/* Copyright (C) 2004 - 2009 Versant Corporation http://www.versant.com */
package com.db4odoc.Storage;

import com.db4o.internal.caching.*;
import com.db4o.io.*;

public class FullLRUCachingStorage extends CachingStorage {

    private int _pageCount = 64;

    public FullLRUCachingStorage(Storage storage) {
        super(storage);
    }

    public FullLRUCachingStorage(Storage storage, int pageCount, int pageSize) {
        super(storage, pageCount, pageSize);
        _pageCount = pageCount;
    }

    @Override
    protected Cache4<Object, Object> newCache() {
        // for a simple LRU algorythm use CacheFactory.newLRUCache(_pageCount);
        return CacheFactory.new2QXCache(_pageCount);
    }

}

```

If you have a better idea of cache implementation you are welcome to create it by overriding Cache4<K,V>(Java) or ICache4<K,V<(.NET) and using the new implementation in the newCache/NewCache method in CachingStorage class as in the example above.

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 Item-Store 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. Commit 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:

```
NonFlushingExample.java: testNonFlushingAdapter
private static void testNonFlushingAdapter() {
    new File(DB4O_FILE_NAME).delete();
    EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
    config.file().storage(new NonFlushingStorage(new FileStorage()));
    ObjectContainer container = Db4oEmbedded.openFile(config, DB4O_FILE_NAME);
    try {
        Pilot pilot = new Pilot("Rubens Barrichello", 99);
        container.store(pilot);
        long startTime = System.currentTimeMillis();
        container.commit();
        System.out.println("Commit time with NonFlushingStorage: " +
                           (System.currentTimeMillis() - startTime) + " ms.");
    } finally {
        container.close();
    }
}

new File(DB4O_FILE_NAME).delete();
container = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), DB4O_FILE_NAME);
try {
    Pilot pilot = new Pilot("Rubens Barrichello", 99);
    container.store(pilot);
    long startTime = System.currentTimeMillis();
    container.commit();
    System.out.println("Commit time with default Storage: " +
```

```

        (System.currentTimeMillis() - startTime) + " ms.");
    } finally {
        container.close();
    }
}
/p>

```

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());
```

Translators

The [Object Construction](#) chapter covers the alternative configurations db4o offers for object reinstantiation. What's left to see is how we can store objects of a class that can't be cleanly stored with either of these approaches.

More Reading:

- [Java Example Class](#)
- [The Translator API](#)
- [Java Translator Implementation](#)
- [Built-In Translators](#)

Java Example Class

Let's use the following class as an example of a class which can not be stored clearly with db4o.

```

        NotStorable.java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */
package com.db4odoc.translators;

public class NotStorable {
    private int id;

    private String name;

    private transient int length;

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

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public int getLength() {
        return length;
    }

    public String toString() {
        return id + "/" + name + ":" + length;
    }
}

```

We'll be using this code to store and retrieve and instance of this class with different configuration settings:

```

        TranslatorExample.java: tryStoreAndRetrieve
private static void tryStoreAndRetrieve(Configuration configuration) {
    ObjectContainer container = Db4o.openFile(configuration, DB4O_FILE_NAME);
    try {
        NotStorable notStorable = new NotStorable(42, "Test");
        System.out.println("ORIGINAL: " + notStorable);
        container.store(notStorable);
    } catch (Exception exc) {
        System.out.println(exc.toString());
        return;
    } finally {
        container.close();
    }
    container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        ObjectSet result = container.queryByExample(NotStorable.class);
        while (result.hasNext()) {

```

```

        NotStorable notStorable = (NotStorable) result.next();
        System.out.println("RETRIEVED: " + notStorable);
        container.delete(notStorable);
    }
} finally {
    container.close();
}
}

```

Bypassing the constructor

```

TranslatorExample.java: tryStoreWithoutCallConstructors
private static void tryStoreWithoutCallConstructors() {
    Configuration configuration = Db4o.newConfiguration();
    configuration.exceptionsOnNotStorable(false);
    configuration.objectClass(NotStorable.class)
        .callConstructor(false);
    tryStoreAndRetrieve(configuration);
}

```

In this case our object seems to be nicely stored and retrieved, however, it has forgotten about its length, since db4o doesn't store transient members and the constructor code that sets it is not executed.

Using the constructor

```

TranslatorExample.java: tryStoreWithCallConstructors
private static void tryStoreWithCallConstructors() {
    Configuration configuration = Db4o.newConfiguration();
    configuration.exceptionsOnNotStorable(true);
    configuration.objectClass(NotStorable.class)
        .callConstructor(true);
    tryStoreAndRetrieve(configuration);
}

```

At storage time, db4o tests the only available constructor with null arguments and runs into a NullPointerException, so it refuses to accept our object.

(Note that this test only occurs when configured with exceptionsOnNotStorable (default setting is true) - otherwise db4o will silently fail when trying to reinstantiate the object.)

This still does not work for our case because the native pointer will definitely be invalid.

In order to solve the problem we will need to use [db4o Translators](#).

The Translator API

So how do we get our object into the database, now that everything seems to fail? Db4o provides a way to specify a custom way of storing and retrieving objects through the ObjectTranslator and ObjectConstructor interfaces.

ObjectTranslator

The ObjectTranslator API looks like this:

Java:

```
public Object onStore(ObjectContainer container, Object applicationObject);  
  
public void onActivate(ObjectContainer container, Object applicationObject, Object storedObject);  
  
public Class storedClass()
```

The usage is quite simple: When a translator is configured for a class, db4o will call its onStore method with a reference to the database and the instance to be stored as a parameter and will store the object returned. This object's type has to be primitive from a db4o point of view and it has to match the type specification returned by storedClass().

On retrieval, db4o will create a blank object of the target class (using the configured instantiation method) and then pass it on to onActivate() along with the stored object to be set up accordingly.

ObjectConstructor

However, this will only work if the application object's class provides sufficient setters to recreate its state from the information contained in the stored object, which is not the case for our example class.

For these cases db4o provides an extension to the ObjectTranslator interface, ObjectConstructor, which declares one additional method:

Java:

```
public Object onInstantiate(ObjectContainer container, Object storedObject);
```

If db4o detects a configured translator to be an ObjectConstructor implementation, it will pass the stored class instance to the onInstantiate() method and use the result as a blank application object to be processed by onActivate().

Note that, while in general configured translators are applied to subclasses, too, ObjectConstructor application object instantiation will not be used for subclasses (which wouldn't make much sense, anyway), so ObjectConstructors have to be configured for the concrete classes.

[Java Translator Implementation](#) provides a Translator usage example for Java platform.

.NET Translator Implementation provides a Translator usage example for .NET platform.

Java Translator Implementation

To translate NotStorable instances, we will pack their id and name values into an Object array to be stored and retrieve it from there again. Note that we don't have to do any work in onActivate(), since object reinstantiation is already fully completed in onInstantiate().

```
NotStorableTranslator.java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */
package com.db4odoc.translators;

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

public class NotStorableTranslator implements ObjectConstructor {
    public Object onStore(ObjectContainer container,
        Object applicationObject) {
        System.out.println("onStore for " + applicationObject);
        NotStorable notStorable = (NotStorable) applicationObject;
        return new Object[] { new Integer(notStorable.getId()),
            notStorable.getName() };
    }

    public Object onInstantiate(ObjectContainer container,
        Object storedObject) {
        System.out.println("onInstantiate for " + storedObject);
        Object[] raw = (Object[]) storedObject;
        int id = ((Integer) raw[0]).intValue();
        String name = (String) raw[1];
        return new NotStorable(id, name);
    }

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

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

Let's try it out:

```
TranslatorExample.java: storeWithTranslator
private static void storeWithTranslator() {
    Configuration configuration = Db4o.newConfiguration();
    configuration.objectClass(NotStorable.class).translate(
        new NotStorableTranslator());
    tryStoreAndRetrieve(configuration);
}
```

ObjectTranslators let you reconfigure the state of a 'blank' application object reinstantiated by db4o, ObjectConstructors also take care of instantiating the application object itself. ObjectTranslators and ObjectConstructors can be used for classes that cannot cleanly be stored and retrieved with db4o's standard object instantiation mechanisms.

Built-In Translators

Db4o supplies some build-in translators, which can be used in general cases. Most of them are used internally and are not a part of public API, however they can serve a good example of a translator implementation.

More Reading:

- TNull Translator
- Collection Translators
- TSerializable Translator
- TTransient Translator
- TCultureInfo Translator
- TType Translator
- TClass Translator

Db4o Reflection API

Reflection API gives your code access to internal information for classes loaded into the JVM. It allows you to work with classes defined in runtime and not in code.

Reflection works with metadata - data that describes other data. 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 in runtime, create new objects, call their methods and operate with the fields.

These features make reflection especially useful for creating libraries that work with objects in very general ways. For example, reflection is often used in frameworks that persist objects to databases, XML, or other external formats.

More Reading:

- [GenericReflector](#)
- [Using db4o reflection API](#)

- [Creating your own 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 possible instantiation of user objects using class name saved in the database file and class information from the JVM. However db4o reflection API can also work on generic objects when a class information is not available.

Db4o uses GenericReflector as a wrapper around specific reflector (delegate). GenericReflector is set when an ObjectContainer is opened. All subsequent reflector calls are routed through this interface.

GenericReflector keeps list of known classes in memory. When the GenericReflector 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 usecases:

- running a db4o server without deploying application classes;
- running db4o on Java dialects without reflection (J2ME CLDC, MIDP);
- easier access to stored objects where classes or fields are not available;
- running refactorings in the reflector;
- building interfaces to db4o from any programming language.

One of the live usecases is ObjectManager, which uses GenericReflector to read C# objects from Java.

Using db4o reflection API

Db4o reflector can be used in your application just like normal language reflector. Let's create a new database with a couple of cars in it:

```
ReflectorExample.java: setCars
private static void setCars()
{
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container=Db4o.openFile(DB4O_FILE_NAME);
    try {
        Car car1 = new Car("BMW");
        container.store(car1);
        Car car2 = new Car("Ferrari");
        container.store(car2);

        System.out.println("Saved:");
        Query query = container.query();
        query.constrain(Car.class);
    }
}
```

```

        ObjectSet results = query.execute();
        listResult(results);
    } finally {
        container.close();
    }
}

```

We can check, what information is available for db4o reflector:

```

                    ReflectorExample.java: getReflectorInfo
private static void getReflectorInfo()
{
    ObjectContainer container=Db4o.openFile(DB4O_FILE_NAME);
    try {
        System.out.println("Reflector in use: " + container.ext().reflector());
        System.out.println("Reflector delegate" +container.ext().reflector().getDelegate());
        ReflectClass[] knownClasses = container.ext().reflector().knownClasses();
        int count = knownClasses.length;
        System.out.println("Known classes: " + count);
        for (int i=0; i <knownClasses.length; i++) {
            System.out.println(knownClasses[i]);
        }
    } finally {
        container.close();
    }
}

```

All the information about Car class can also be retrieved through reflector:

```

                    ReflectorExample.java: getCarInfo
private static void getCarInfo()
{
    ObjectContainer container=Db4o.openFile(DB4O_FILE_NAME);
    try {
        GenericReflector reflector = new GenericReflector(null,container.ext().reflector());
        ReflectClass carClass = reflector.forName(Car.class.getName());
        System.out.println("Reflected class "+carClass);
        // public fields
        System.out.println("FIELDS:");
        ReflectField[] fields = carClass.getDeclaredFields();
        for (int i = 0; i < fields.length; i++)
            System.out.println(fields[i].getName());

        // public methods
        System.out.println("METHODS:");
        ReflectMethod method = carClass.getMethod("getPilot",null);
        System.out.println(method.getClass());
    }
}

```

```

    } finally {
        container.close();
    }
}

```

We can use classes retrieved using reflection to create queries:

```

ReflectorExample.java: getCars
private static void getCards()
{
    ObjectContainer container=Db4o.openFile(DB4O_FILE_NAME);
    try {
        GenericReflector reflector = new GenericReflector(null,container.ext().reflector());
        ReflectClass carClass = reflector.forName(Car.class.getName());
        System.out.println("Reflected class "+carClass);
        System.out.println("Retrieved with reflector:");
        Query query = container.query();
        query.constrain(carClass);
        ObjectSet results = query.execute();
        listResult(results);
    } finally {
        container.close();
    }
}

```

Creating your own reflector

By default db4o uses JdkReflector(Java) or NetReflector (.NET) as a GenericReflector delegate.

However, the programmer can instruct db4o to use a specially designed reflection implementation:

Java: `Db4o.configure().reflectWith(reflector)`

where reflector is one of the available reflectors or your own reflector implementation.

At present db4o comes with SelfReflector, which was designed for environments, which do not have built-in support for reflections (J2ME for example). In this implementation all the classes' information is stored in special registry. User classes should implement `self_get` and `self_set` methods to be registered individually and become "known" to SelfReflector.

Specific reflectors can be written for special usecases.

Let's look how to create a reflector. Remember that db4o relies on reflector to read the database, so errors in reflector may prevent your database from opening.

To keep things simple we will write a LoggingReflector, its only difference from standard reflector is that information about loaded classes is outputted to console. All reflectors used by db4o should implement com.db4o.reflect.Reflector/Db4oObjects.Db4o.Reflect.IReflector interface.

```
LoggingReflector.java
/** Copyright (C) 2004 Versant Inc. http://www.db4o.com */

package com.db4odoc.reflections;

import com.db4o.internal.Platform4;
import com.db4o.reflect.ReflectArray;
import com.db4o.reflect.ReflectClass;
import com.db4o.reflect.Reflector;
import com.db4o.reflect.ReflectorConfiguration;
import com.db4o.reflect.jdk.ClassLoaderJdkLoader;
import com.db4o.reflect.jdk.JavaReflectClass;
import com.db4o.reflect.jdk.JdkClass;
import com.db4o.reflect.jdk.JdkLoader;
import com.db4o.reflect.jdk.JdkReflector;

/**
 * db4o wrapper for JDK reflector functionality
 *
 * @see com.db4o.ext.ExtObjectContainer#reflector()
 * @see com.db4o.reflect.generic.GenericReflector
 *
 * @sharpen.ignore
 */
public class LoggingReflector implements Reflector {

    private final JdkLoader _classLoader;
    protected Reflector _parent;
    private ReflectArray _array;
    private ReflectorConfiguration _config;

    /**
     * Constructor
     *
     * @param classLoader
     *          class loader
     */
    public LoggingReflector(ClassLoader classLoader) {
        this(new ClassLoaderJdkLoader(classLoader));
    }

    /**
     * Constructor
     *
     * @param classLoader
     *          class loader
     */
    public LoggingReflector(JdkLoader classLoader) {
        this(classLoader, null);
    }
}
```

```
}

private LoggingReflector(JdkLoader classLoader,
    ReflectorConfiguration config)  {
    _classLoader = classLoader;
    _config = config;
}

/***
 * ReflectArray factory
 *
 * @return ReflectArray instance
 */
public ReflectArray array()  {
    if (_array == null)  {
        _array = new LoggingArray(parent());
    }
    return _array;
}

/***
 * Creates a copy of the object
 *
 * @param obj
 *          object to copy
 * @return object copy
 */
public Object deepClone(Object obj)  {
    return new LoggingReflector(_classLoader, _config);
}

/***
 * Returns ReflectClass for the specified class
 *
 * @param clazz
 *          class
 * @return ReflectClass for the specified class
 */
public ReflectClass forClass(Class clazz)  {
    ReflectClass rc = createClass(clazz);
    System.out.println("forClass: " + clazz + " -> "
        + (rc == null ? "" : rc.getName()));

    return rc;
}

/***
 * Returns ReflectClass for the specified class name
 *
 * @param className
 *          class name
 * @return ReflectClass for the specified class name
 */
public ReflectClass forName(String className)  {
```

```

Class clazz = _classLoader.loadClass(className);
ReflectClass rc = createClass(clazz);
System.out.println("forName: " + className + " -> "
    + (rc == null ? "" : rc.getName()));
return rc;

}

/***
 * creates a Class reflector when passed a class. This method is protected
 * to allow overriding in cusom reflectors that override JdkReflector.
 *
 * @param clazz
 *         the class
 * @return the class reflector
 */
protected JdkClass createClass(Class clazz) {
    if (clazz == null) {
        return null;
    }
    JdkReflector jdkReflector = new JdkReflector(this.getClass()
        .getClassLoader());
    jdkReflector.configuration(_config);

    JdkClass rc = new JdkClass(parent(), jdkReflector, clazz);
    return rc;
}

/***
 * Returns ReflectClass for the specified class object
 *
 * @param a_object
 *         class object
 * @return ReflectClass for the specified class object
 */
public ReflectClass forObject(Object a_object) {
    if (a_object == null) {
        return null;
    }
    ReflectClass rc = parent().forClass(a_object.getClass());
    System.out.println("forObject:" + a_object + " -> "
        + (rc == null ? "" : rc.getName()));
    return rc;
}

/***
 * Method stub. Returns false.
 */
public boolean isCollection(ReflectClass candidate) {
    return false;
}

/***
 * Method stub. Returns false.
 */

```

```

/*
public boolean methodCallsSupported()  {
    return true;
}

/**
 * Sets parent reflector
 *
 * @param reflector
 *         parent reflector
 */
public void setParent(Reflector reflector)  {
    _parent = reflector;
}

/**
 * Creates ReflectClass[] array from the Class[] array using the reflector
 * specified
 *
 * @param reflector
 *         reflector to use
 * @param clazz
 *         class
 * @return ReflectClass[] array
 */
public static ReflectClass[] toMeta(Reflector reflector, Class[] clazz)  {
    ReflectClass[] claxx = null;
    if (clazz != null)  {
        claxx = new ReflectClass[clazz.length];
        for (int i = 0; i < clazz.length; i++)  {
            if (clazz[i] != null)  {
                claxx[i] = reflector.forClass(clazz[i]);
            }
        }
    }
    return claxx;
}

/**
 * Creates Class[] array from the ReflectClass[] array
 *
 * @param claxx
 *         ReflectClass array
 * @return Class[] array
 */
static Class[] toNative(ReflectClass[] claxx)  {
    Class[] clazz = null;
    if (claxx != null)  {
        clazz = new Class[claxx.length];
        for (int i = 0; i < claxx.length; i++)  {
            clazz[i] = toNative(claxx[i]);
        }
    }
    return clazz;
}

```

```

}

/**
 * Translates a ReflectClass into a native Class
 *
 * @param claxx
 *          ReflectClass to translate
 * @return Class
 */
public static Class toNative(ReflectClass claxx) {
    if (claxx == null) {
        return null;
    }
    System.out.println("toNative: " + claxx.getName());
    if (claxx instanceof JavaReflectClass) {
        return ((JavaReflectClass) claxx).getJavaClass();
    }
    ReflectClass d = claxx.getDelegate();
    if (d == claxx) {
        return null;
    }
    return toNative(d);
}

public void configuration(ReflectorConfiguration config) {
    _config = config;
}

public ReflectorConfiguration configuration() {
    return _config;
}

Object nullValue(ReflectClass clazz) {
    return Platform4.nullValue(toNative(clazz));
}

private Reflector parent() {
    if (_parent == null) {
        return this;
    }
    return _parent;
}
}

```

It is easy to see that this reflector provides the same functionality as JdkReflector or NetReflector extended by console output. The following simple test will show how it works:

<pre>ReflectorExample.java: testReflector private static void testReflector()</pre>

```

{
    LoggingReflector logger = new LoggingReflector(Db4o.class.getClassLoader());
    Configuration configuration = Db4o.newConfiguration();
    configuration.reflectWith(logger);
    ObjectContainer container = Db4o.openFile(configuration, DB4O_FILE_NAME);
    try {
        ReflectClass rc = container.ext().reflector().forName(Car.class.getName());
        System.out.println("Reflected class: " + rc);
    } finally {
        container.close();
    }
}

```

The output can help you to track all the loaded classes.

Reflection is a powerful tool, which plays a fundamental role in db4o. Understanding reflection will help you to understand the whole db4o functionality in detail.

Db4o meta-information

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

More Reading:

- [Accessing db4o meta-information](#)
- [StoredClass and StoredField interfaces](#)

Accessing db4o meta-information

Db4o provides an access to the database meta-information through its extended object container interface (ExtObjectContainer(Java)/IExtObjectContainer(.NET)).

Within the object database meta-schema is represented by classes and their fields. To access their meta-information db4o provides special interfaces:

- [StoredClass\(Java\)/IStoredClass\(.NET\)](#)
- [StoredField\(Java\)/IStoredField\(.NET\)](#)

The following ExtObjectContainer methods give you access to the [StoredClass](#).

Java: ExtObjectContainer#storedClass(Foo.class)

returns [StoredClass](#) for the specified clazz, which can be specified as:

- a fully qualified classname;
- a Class/Type object;
- any object to be used as a template.

Java: ExtObjectContainer#storedClasses()

returns an array of all StoredClass meta-information objects.

```
MetaInfoExample.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();
    }
}
```

```
MetaInfoExample.java: getMetaObjects
private static void getMetaObjects()  {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        System.out
            .println("Retrieve meta information for class: ");
        StoredClass sc = container.ext().storedClass(
            Car.class.getName());
        System.out.println("Stored class: " + sc.toString());

        System.out
            .println("Retrieve meta information for all classes in database: ");
        StoredClass sclasses[] = container.ext().storedClasses();
        for (int i = 0; i < sclasses.length; i++)  {
            System.out.println(sclasses[i].getName());
        }
    } finally {
        container.close();
    }
}
```

StoredClass and StoredField interfaces

(IStoredClass and IStoredField in .NET)

Let's look closer at the class meta-information interfaces.

They look quite similar to reflection API, but unlike reflection there is no information about methods and constructors.

You can only use `StoredClass` to get the class's fields:

Java: `StoredClass#getStoredFields()`

returns all stored fields of this stored class.

Java: `StoredClass#storedField(name, type)`

returns an existing stored field of this stored class.

You can also use this interface to explore classes hierarchy.

Java: `StoredClass#getParentStoredClass`

returns the parent of the class.

`StoredField` interface gives you access to various meta-field information, such as field name, field type. It also provides some helpful methods for manipulating fields accepting their object as a variable (see db4o API for more information).

```
MetaInfoExample.java: GetMetaObjectsInfo
private static void getMetaObjects() {
    ObjectContainer container = Db4oEmbedded.openFile(DB4O_FILE_NAME);
    try {
        System.out
            .println("Retrieve meta information for class: ");
        StoredClass sc = container.ext().storedClass(
            Car.class.getName());
        System.out.println("Stored class: " + sc.toString());

        System.out
            .println("Retrieve meta information for all classes in database: ");
        StoredClass sclasses[] = container.ext().storedClasses();
        for (int i = 0; i < sclasses.length; i++) {
            System.out.println(sclasses[i].getName());
        }
    } finally {
        container.close();
    }
}
```

```
    }  
}
```

Native Query Collection

This collection of Native Queries examples is intended to show different implementations of the querying interface and help users with the practical use cases.

Persistent classes used in the examples are defined in [Persistent Classes](#).

More Reading:

- [Simple Selection](#)
- [Using Selection Criteria](#)
- [Selecting Ranges](#)
- [Sorting](#)
- [Result Representation](#)
- [Calculation Examples](#)
- [Combined Result Sets](#)
- [Parameterized NQ](#)
- [Store Pilots](#)
- [Store Persons](#)
- [Store Cars](#)
- [Persistent Classes](#)

Calculation Examples

This set of examples shows how to use Native Queries to perform different calculations on the objects in the database. [Store Pilots](#) function is used to fill in the database.

SumPilotPoints

Calculate the sum of the points of all the pilots in the database.

```
CalculationExamples.java: sumPilotPoints
```

```

private static void sumPilotPoints() {
    ObjectContainer container = database();

    if (container != null) {
        try {
            SumPredicate sumPredicate = new SumPredicate();
            List<Pilot> result = container.query(sumPredicate);
            listResult(result);
            System.out.println("Sum of pilots points: " + sumPredicate.sum);
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}

```

```

CalculationExamples.java: SumPredicate
private static class SumPredicate extends Predicate<Pilot> {
    private int sum = 0;

    public boolean match(Pilot pilot) {
        // return all pilots
        sum += pilot.getPoints();
        return true;
    }
}

```

SelectMinPointsPilot

Find a pilot having the minimum points.

```

CalculationExamples.java: selectMinPointsPilot
private static void selectMinPointsPilot() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            List<Pilot> result = container.query(new Predicate<Pilot>() {
                public boolean match(Pilot pilot) {
                    // return all pilots
                    return true;
                }
            }, new QueryComparator<Pilot>() {
                // sort by points then by name
                public int compare(Pilot p1, Pilot p2) {
                    return p1.getPoints() - p2.getPoints();
                }
            });
            if (result.size() > 0) {
                System.out.println("The min points result is: "
                    + result.queryByExample(0));
            }
        }
    }
}

```

```
        }
    } catch (Exception ex)  {
        System.out.println("System Exception: " + ex.getMessage());
    } finally {
        closeDatabase();
    }
}
```

AveragePilotPoints

Calculate what is the average amount of points for all the pilots in the database.

```
    CalculationExamples.java: averagePilotPoints
private static void averagePilotPoints()  {
    ObjectContainer container = database();

    if (container != null)  {
        try  {
            AveragePredicate averagePredicate = new AveragePredicate();
            List<Pilot> result = container.query(averagePredicate);
            if (averagePredicate.count > 0)  {
                System.out
                    .println("Average points for professional pilots: "
                        + averagePredicate.sum
                        / averagePredicate.count);
            } else  {
                System.out.println("No results");
            }
        } catch (Exception ex)  {
            System.out.println("System Exception: " + ex.getMessage());
        } finally  {
            closeDatabase();
        }
    }
}
```

```
CalculationExamples.java: AveragePredicate
private static class AveragePredicate extends Predicate<Pilot> {
    private int sum = 0;

    private int count = 0;

    public boolean match(Pilot pilot)  {
        // return professional pilots
        if (pilot.getName().startsWith("Professional"))  {
            sum += pilot.getPoints();
            count++;
            return true;
        }
        return false;
    }
}
```

```
    }  
}
```

CountSubGroups

Calculate how many pilots are in each group ("Test", "Professional").

```
CalculationExamples.java: countSubGroups  
private static void countSubGroups() {  
    ObjectContainer container = database();  
    if (container != null) {  
        try {  
            CountPredicate predicate = new CountPredicate();  
            List<Pilot> result = container.query(predicate);  
            listResult(result);  
            Iterator keyIterator = predicate.countMap.keySet().iterator();  
            while (keyIterator.hasNext()) {  
                String key = keyIterator.next().toString();  
                System.out  
                    .println(key + ": " + predicate.countMap.queryByExample(key));  
            }  
        } catch (Exception ex) {  
            System.out.println("System Exception: " + ex.getMessage());  
        } finally {  
            closeDatabase();  
        }  
    }  
}
```

```
CalculationExamples.java: CountPredicate  
private static class CountPredicate extends Predicate<Pilot> {  
  
    private HashMap countMap = new HashMap();  
  
    public boolean match(Pilot pilot) {  
        // return all Professional and Test pilots and count in  
        // each category  
        String[] keywords = { "Professional", "Test" };  
        for (int i = 0; i < keywords.length; i++) {  
            if (pilot.getName().startsWith(keywords[i])) {  
                if (countMap.containsKey(keywords[i])) {  
                    countMap.put(keywords[i], ((Integer) countMap  
                        .queryByExample(keywords[i])) + 1);  
                } else {  
                    countMap.put(keywords[i], 1);  
                }  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

```
    }  
}
```

Combined Result Sets

This set of examples shows how to use NQ to select objects from more than one result sets. [Store Cars](#) and [Store Persons](#) functions are used to fill in the database.

SelectPilotsAndTrainees

Selects all pilots and trainees using Person superclass.

```
MultiExamples.java: selectPilotsAndTrainees  
private static void selectPilotsAndTrainees()  {  
    ObjectContainer container = database();  
    if (container != null)  {  
        try  {  
            List<Person> result = container.query(new Predicate<Person>()  {  
                public boolean match(Person person)  {  
                    // all persons  
                    return true;  
                }  
            });  
            listResult(result);  
        } catch (Exception ex)  {  
            System.out.println("System Exception: " + ex.getMessage());  
        } finally  {  
            closeDatabase();  
        }  
    }  
}
```

SelectPilotsInRange

Selects all cars that have pilot field in the pre-selected Pilot array.

```
MultiExamples.java: selectPilotsInRange  
private static void selectPilotsInRange()  {  
    ObjectContainer container = database();  
    if (container != null)  {  
        try  {  
            List<Car> result = container.query(new Predicate<Car>()  {  
                private List<Pilot> pilots = null;  
  
                private List getPilotsList()  {  
                    if (pilots == null)  {  
                        pilots = database().query(new Predicate<Pilot>()  {  
                            ...  
                        });  
                    }  
                    return pilots;  
                }  
            });  
            listResult(result);  
        } catch (Exception ex)  {  
            System.out.println("System Exception: " + ex.getMessage());  
        } finally  {  
            closeDatabase();  
        }  
    }  
}
```

```
        public boolean match(Pilot pilot)  {
            return pilot.getName().startsWith("Test");
        }
    });
}
return pilots;
}

public boolean match(Car car)  {
    // all Cars that have pilot field in the
    // Pilots array
    return getPilotsList().contains(car.getPilot());
}
});
listResult(result);
} catch (Exception ex)  {
    System.out.println("System Exception: " + ex.getMessage());
} finally  {
    closeDatabase();
}
}
```

Parameterized NQ

The following examples show how to pass parameters to the Native Query predicate. [Store Pilots](#) function is used to fill in the database.

GetTestPilots

Using predicate constructor to specify the querying parameter.

```
ParameterizedExamples.java: getTestPilots
private static void getTestPilots()  {
    ObjectContainer container = database();
    if (container != null)  {
        try  {
            List<Pilot> result = container.query(new PilotNamePredicate(
                "Test"));
            listResult(result);
        } catch (Exception ex)  {
            System.out.println("System Exception: " + ex.getMessage());
        } finally  {
            closeDatabase();
        }
    }
}
```

```
ParameterizedExamples.java: PilotNamePredicate
private static class PilotNamePredicate extends Predicate<Pilot> {
    private String startsWith;
```

```

public PilotNamePredicate(String startsWith) {
    this.startsWith = startsWith;
}

public boolean match(Pilot pilot) {
    return pilot.getName().startsWith(startsWith);
}
}

```

GetProfessionalPilots

Using a function to process the NQ.

```

ParameterizedExamples.java: getProfessionalPilots
private static void getProfessionalPilots() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            List<Pilot> result = byNameBeginning("Professional");
            listResult(result);
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}

```

```

ParameterizedExamples.java: byNameBeginning
private static List<Pilot> byNameBeginning(final String startsWith) {
    return database().query(new Predicate<Pilot>() {
        public boolean match(Pilot pilot) {
            return pilot.getName().startsWith(startsWith);
        }
    });
}

```

Querying Class Hierarchy

The following example shows how to correctly implement parameterized NQ predicate for querying derived classes.

Let's use the class hierarchy as defined in [Persistent Classes](#).

PersonNamePredicate will be used for querying classes derived from Person:

```

ComplexParameterizedExample.java: PersonNamePredicate
private static class PersonNamePredicate<T extends Person> extends
    Predicate<T> {
    private String startsWith;

    public PersonNamePredicate(String startsWith) {
        this.startsWith = startsWith;
    }

    public PersonNamePredicate(Class<T> clazz, String startsWith) {
        super(clazz);
        this.startsWith = startsWith;
    }

    public boolean match(T candidate) {
        return candidate.getName().startsWith(startsWith);
    }
}

```

Let's save some Pilot and Trainee objects:

```

ComplexParameterizedExample.java: storePersons
private static void storePersons() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database();
    if (container != null) {
        try {
            Trainee trainee;
            // store OBJECT_COUNT pilots and trainees
            for (int i = 0; i < OBJECT_COUNT; i++) {
                trainee = new Trainee("Trainee #" + i, new Pilot(
                    "Professional Pilot #" + i, i));
                container.store(trainee);
            }
            // store a new trainee with a "Training" pilot
            trainee = new Trainee("Trainee #1", new Pilot(
                "Training Pilot #1", 20));
            container.store(trainee);
            container.commit();
        } catch (Db4oException ex) {
            System.out.println("Db4o Exception: " + ex.getMessage());
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}

```

Now we have both Pilot and Trainee objects starting from "Train". What should we do to retrieve only Trainee objects?

```

ComplexParameterizedExample.java: getTrainees
private static void getTrainees() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            // query for Trainee objects starting with "Train".
            // Wrongly created predicate mixes Trainee and Pilot
            // objects and creates a resultset based on only "Tr"
            // criteria (class of an object is not considered)

            testQuery(container, createPredicateWrong(Trainee.class,
                "Train"));
            // Correctly created result set returns only objects
            // of the requested class
            testQuery(container, createPredicateCorrect(Trainee.class,
                "Train"));
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}

```

```

ComplexParameterizedExample.java: testQuery
private static void testQuery(ObjectContainer container,
    Predicate<Trainee> predicate) {
    List<Trainee> result = container.query(predicate);
    System.out.println(result.size());
    try {
        for (Trainee trainee : result) {
            System.out.println(trainee);
        }
    } catch (Exception ex) {
        System.out.println(ex.toString());
    }
}

```

```

ComplexParameterizedExample.java: createPredicateWrong
private static <T extends Person> Predicate<T> createPredicateWrong(
    Class<T> clazz, String startsWith) {
    return new PersonNamePredicate<T>(startsWith);
}

```

```

ComplexParameterizedExample.java: createPredicateCorrect
private static <T extends Person> Predicate<T> createPredicateCorrect(
    Class<T> clazz, String startsWith) {
    return new PersonNamePredicate<T>(clazz, startsWith);
}

```

You can see that in order to get the correct results we need to supply our predicate class with the queried class, otherwise both Pilot and Trainee objects will be included in the result set.

Persistent Classes

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

package com.db4odoc.nqcollection;

public interface Person {
    public String getName();
}
```

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

package com.db4odoc.nqcollection;

public class Pilot implements Person {
    private String name;
    private int points;

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

    public String getName() {
        return name;
    }

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

    public int getPoints() {
        return points;
    }

    public boolean equals(Object obj) {
        if (obj instanceof Pilot) {
            return (((Pilot) obj).getName().equals(name) &&
                   ((Pilot) obj).getPoints() == points);
        }
        return false;
    }

    public String toString() {
        return name + "/" + points;
    }

    public int hashCode() {
```

```

        return name.hashCode() + points;
    }
}

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

package com.db4odoc.nqcollection;

public class Trainee implements Person {
    private String name;
    private Pilot instructor;

    public Trainee(String name, Pilot pilot) {
        this.name = name;
        this.instructor = pilot;
    }

    public String getName() {
        return name;
    }

    public Pilot getInstructor() {
        return instructor;
    }

    public String toString() {
        return name + "(" + instructor + ")";
    }
}

```

Result Representation

This example shows how to modify the query output without effecting the objects in the database. [Store Pilots](#) function is used to fill in the database.

SelectAndChangePilots

```

SimpleExamples.java: selectAndChangePilots
private static void selectAndChangePilots() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            List<Pilot> result = container.query(new Predicate<Pilot>() {
                public boolean match(Pilot pilot) {
                    // Add ranking to the pilots during the query.
                    // Note: pilot records in the database won't
                    // be changed!!!

```

```

        if (pilot.getPoints() <= 5)  {
            pilot.setName(pilot.getName() + ": weak");
        } else if (pilot.getPoints() > 5
                   && pilot.getPoints() <= 15)  {
            pilot.setName(pilot.getName() + ": average");
        } else if (pilot.getPoints() > 15)  {
            pilot.setName(pilot.getName() + ": strong");
        }
        return true;
    }
});
listResult(result);
} catch (Exception ex)  {
    System.out.println("System Exception: " + ex.getMessage());
} finally {
    closeDatabase();
}
}
}
}

```

Selecting Ranges

This group of examples shows how to select ranges of objects. [Store Pilots](#) function is used to fill in the database.

SelectTestPilots6PointsMore

Select "Test" pilots with the points range of more than 6.

```

SimpleExamples.java: selectTestPilots6PointsMore
private static void selectTestPilots6PointsMore()  {
    ObjectContainer container = database();
    if (container != null)  {
        try  {
            List<Pilot> result = container.query(new Predicate<Pilot>()  {
                public boolean match(Pilot pilot)  {
                    // all Pilots containing "Test" in the name
                    // and 6 point are included in the result
                    boolean b1 = pilot.getName().indexOf("Test") >= 0;
                    boolean b2 = pilot.getPoints() > 6;
                    return b1 && b2;
                }
            });
            listResult(result);
        } catch (Exception ex)  {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}

```

```
}
```

SelectPilots6To12Points

Select all pilots, who have points in [6,12] range.

```
SimpleExamples.java: selectPilots6To12Points
private static void selectPilots6To12Points() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            List<Pilot> result = container.query(new Predicate<Pilot>() {
                public boolean match(Pilot pilot) {
                    // all Pilots having 6 to 12 point are
                    // included in the result
                    return ((pilot.getPoints() >= 6) && (pilot.getPoints() <= 12));
                }
            });
            listResult(result);
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}
```

SelectPilotsRandom

Select pilots randomly: random array of point values is generated, those pilots who have points values within this array are included in the result set.

```
SimpleExamples.java: selectPilotsRandom
private static void selectPilotsRandom() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            List<Pilot> result = container.query(new Predicate<Pilot>() {
                private ArrayList randomArray = null;

                private List getRandomArray() {
                    if (randomArray == null) {
                        randomArray = new ArrayList();
                        for (int i = 0; i < 10; i++) {
                            randomArray.add((int) (Math.random() * 10));
                            System.out.println(randomArray.queryByExample(i));
                        }
                    }
                }
            });
            listResult(result);
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}
```

```
        }
        return randomArray;
    }

    public boolean match(Pilot pilot)  {
        // all Pilots having points in the values of
        // the randomArray
        return getRandomArray().contains(pilot.getPoints());
    }
});

listResult(result);
} catch (Exception ex)  {
    System.out.println("System Exception: " + ex.getMessage());
} finally {
    closeDatabase();
}
}
```

SelectPilotsEven

Select pilots with even points.

```
SimpleExamples.java: selectPilotsEven
private static void selectPilotsEven()  {
    ObjectContainer container = database();
    if (container != null)  {
        try  {
            List<Pilot> result = container.query(new Predicate<Pilot>() {
                public boolean match(Pilot pilot)  {
                    // all Pilots having even points
                    return pilot.getPoints() % 2 == 0;
                }
            });
            listResult(result);
        } catch (Exception ex)  {
            System.out.println("System Exception: " + ex.getMessage());
        } finally  {
            closeDatabase();
        }
    }
}
```

SelectAnyOnePilot

Select one pilot and quit.

SimpleExamples.java: selectAnyOnePilot

```

private static void selectAnyOnePilot() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            List<Pilot> result = container.query(new Predicate<Pilot>() {
                boolean selected = false;

                public boolean match(Pilot pilot) {
                    // return only first result (first result can
                    // be any value from the resultset)
                    if (!selected) {
                        selected = true;
                        return selected;
                    } else {
                        return !selected;
                    }
                }
            });
            listResult(result);
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}

```

SelectDistinctPilots

This example shows how to select only unique results from the non-unique contents in the database.

```

SimpleExamples.java: storeDuplicates
private static void storeDuplicates() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database();
    if (container != null) {
        try {
            Pilot pilot;
            for (int i = 0; i < OBJECT_COUNT; i++) {
                pilot = new Pilot("Test Pilot #" + i, i);
                container.store(pilot);
            }
            for (int i = 0; i < OBJECT_COUNT; i++) {
                pilot = new Pilot("Test Pilot #" + i, i);
                container.store(pilot);
            }
            container.commit();
        } catch (Db4oException ex) {
            System.out.println("Db4o Exception: " + ex.getMessage());
        } catch (Exception ex) {

```

```
        System.out.println("System Exception: " + ex.getMessage());
    } finally {
        closeDatabase();
    }
}
```

```
SimpleExamples.java: selectDistinctPilots
private static void selectDistinctPilots()  {
    ObjectContainer container = database();
    if (container != null)  {
        try  {
            DistinctPilotsPredicate predicate = new DistinctPilotsPredicate();
            List<Pilot> result = container.query(predicate);
            listResult(predicate.uniqueResult());
        } catch (Exception ex)  {
            System.out.println("System Exception: " + ex.getMessage());
        } finally  {
            closeDatabase();
        }
    }
}
```

```
SimpleExamples.java: DistinctPilotsPredicate  
1
```

Simple Selection

The following examples show how to use NQ to select all objects of the specified type from a database. [Store Pilots](#) function is used to fill in the database.

SelectAllPilots

For languages with generics support (Java5-6; .NET2.0-3.0):

```
SimpleExamples.java: selectAllPilots
private static void selectAllPilots()  {
    ObjectContainer container = database();
    if (container != null)  {
        try  {
            List<Pilot> result = container.query(new Predicate<Pilot>() {
                public boolean match(Pilot pilot)  {
                    // each Pilot is included in the result
                    return true;
                }
            });
            listResult(result);
        } catch (Exception ex)  {
            System.out.println("System Exception: " + ex.getMessage());
        } finally  {
```

```
        closeDatabase();
    }
}
```

SelectAllPilotsNonGeneric

For languages without generics support (Java1.1-1.4; .NET1.0):

```
SimpleExamples.java: selectAllPilotsNonGeneric
private static void selectAllPilotsNonGeneric() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            List result = container.query(new Predicate() {
                public boolean match(Object object) {
                    // each Pilot is included in the result
                    if (object instanceof Pilot) {
                        return true;
                    }
                    return false;
                }
            });
            listResult(result);
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}
```

Sorting

The following examples represent NQ sorting techniques. [Store Pilots](#) function is used to fill in the database.

GetSortedPilots

Select all the pilots from the database and sort descending by points.

```
SimpleExamples.java: getSortedPilots
public static void getSortedPilots() {
    ObjectContainer container = database();
    try {
        List result = container.query(new Predicate<Pilot>() {
            public boolean match(Pilot pilot) {

```

```

        return true;
    }
}, new QueryComparator<Pilot>() {
    // sort by points
    public int compare(Pilot p1, Pilot p2) {
        return p2.getPoints() - p1.getPoints();
    }
});
listResult(result);
} finally {
    closeDatabase();
}
}
}

```

GetPilotsSortByNameAndPoints

Select all pilots, sort descending by name and by points.

```

SimpleExamples.java: getPilotsSortByNameAndPoints
public static void getPilotsSortByNameAndPoints() {
    ObjectContainer container = database();
    try {
        List result = container.query(new Predicate<Pilot>() {
            public boolean match(Pilot pilot) {
                return true;
            }
        }, new QueryComparator<Pilot>() {
            // sort by name then by points: descending
            public int compare(Pilot p1, Pilot p2) {
                int result = p1.getName().compareTo(p2.getName());
                if (result == 0) {
                    return p1.getPoints() - p2.getPoints();
                } else {
                    return -result;
                }
            }
        });
        listResult(result);
    } finally {
        closeDatabase();
    }
}

```

GetPilotsSortWithComparator

Sort by points using pre-defined comparator.

SimpleExamples.java: getPilotsSortWithComparator
--

```

public static void getPilotsSortWithComparator()  {
    ObjectContainer container = database();
    try {
        List result = container.query(new Predicate<Pilot>()  {
            public boolean match(Pilot pilot)  {
                return true;
            }
        }, new PilotComparator());
        listResult(result);
    } finally {
        closeDatabase();
    }
}

```

```

SimpleExamples.java: PilotComparator
public static class PilotComparator implements Comparator<Pilot>  {
    public int compare(Pilot p1, Pilot p2)  {
        int result = p1.getName().compareTo(p2.getName());
        if (result == 0)  {
            return p1.getPoints() - p2.getPoints();
        } else  {
            return -result;
        }
    }
}

```

Store Cars

```

MultiExamples.java: storeCars
private static void storeCars()  {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database();
    if (container != null)  {
        try {
            Car car;
            for (int i = 0; i < OBJECT_COUNT; i++)  {
                car = new Car("BMW", new Pilot("Test Pilot #" + i, i));
                container.store(car);
            }
            for (int i = 0; i < OBJECT_COUNT; i++)  {
                car = new Car("Ferrari", new Pilot("Professional Pilot #"
                    + (i + 10), (i + 10)));
                container.store(car);
            }
            container.commit();
        } catch (Db4oException ex)  {
            System.out.println("Db4o Exception: " + ex.getMessage());
        } catch (Exception ex)  {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}

```

```
        }
    }
}
```

Store Persons

```
MultiExamples.java: storePilotsAndTrainees
private static void storePilotsAndTrainees() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database();
    if (container != null) {
        try {
            Pilot pilot;
            Trainee trainee;
            for (int i = 0; i < OBJECT_COUNT; i++) {
                pilot = new Pilot("Professional Pilot #" + i, i);
                trainee = new Trainee("Trainee #" + i, pilot);
                container.store(trainee);
            }
            container.commit();
        } catch (Db4oException ex) {
            System.out.println("Db4o Exception: " + ex.getMessage());
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}
```

Store Pilots

```
SimpleExamples.java: storePilots
private static void storePilots() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database();
    if (container != null) {
        try {
            Pilot pilot;
            for (int i = 0; i < OBJECT_COUNT; i++) {
                pilot = new Pilot("Test Pilot #" + i, i);
                container.store(pilot);
            }
            for (int i = 0; i < OBJECT_COUNT; i++) {
                pilot = new Pilot("Professional Pilot #" + (i + 10), i + 10);
                container.store(pilot);
            }
        }
```

```

        container.commit();
    } catch (Db4oException ex) {
        System.out.println("Db4o Exception: " + ex.getMessage());
    } catch (Exception ex) {
        System.out.println("System Exception: " + ex.getMessage());
    } finally {
        closeDatabase();
    }
}
}

```

Using Selection Criteria

The following examples show how to select objects matching a specific criteria. [Store Pilots](#) function is used to fill in the database.

SelectPilot5Points

Select only those pilots, which have 5 points.

```

SimpleExamples.java: selectPilot5Points
private static void selectPilot5Points() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            List<Pilot> result = container.query(new Predicate<Pilot>() {
                public boolean match(Pilot pilot) {
                    // pilots with 5 points are included in the
                    // result
                    return pilot.getPoints() == 5;
                }
            });
            listResult(result);
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}

```

SelectTestPilots

Select all pilots, whose name includes "Test".

```

SimpleExamples.java: selectTestPilots
private static void selectTestPilots() {

```

```

ObjectContainer container = database();
if (container != null) {
    try {
        List<Pilot> result = container.query(new Predicate<Pilot>() {
            public boolean match(Pilot pilot) {
                // all Pilots containing "Test" in the name
                // are included in the result
                return pilot.getName().indexOf("Test") >= 0;
            }
        });
        listResult(result);
    } catch (Exception ex) {
        System.out.println("System Exception: " + ex.getMessage());
    } finally {
        closeDatabase();
    }
}
}

```

SelectPilotsNumberX6

Select those pilots, whose name (number) ends with 6.

```

SimpleExamples.java: selectPilotsNumberX6
private static void selectPilotsNumberX6() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            List<Pilot> result = container.query(new Predicate<Pilot>() {
                public boolean match(Pilot pilot) {
                    // all Pilots with the name ending with 6 will
                    // be included
                    return pilot.getName().endsWith("6");
                }
            });
            listResult(result);
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}

```

Refactoring and Schema Evolution

Application design is a volatile thing: it changes from version to version, from one customer implementation to another. The database (if used) 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.

More Reading:

- [Automatic refactoring](#)
- [Refactoring API](#)
- [Field type change](#)
- [Refactoring Class Hierarchy](#)

Automatic refactoring

In simplest cases db4o handles schema changes automatically:

- If you **add** a new field, db4o automatically starts storing the new data. Older instances of your stored class (from before the field was added) are still loaded, but the new field is set to its default value, or null.
- If you **remove** a field, db4o ignores the stored value when activating instances of your class. The stored value is not removed from the database until the next Defragment, and is still accessible via the StoredClass/StoredField API.
- If you **add an interface** to be implemented by your stored classes, db4o automatically starts using it and you are able to retrieve your saved data using new interface.

Refactoring API

Db4o provides special API which can help you to move classes between packages (Java)/namespaces(.NET), rename classes or fields:

Java:

```
Db4o.configure()  
.objectClass("package.class").rename("newPackage.newClass");  
Db4o.configure().objectClass("package.class")  
.objectField("oldField").rename("newField");
```

The safe order of actions for rename calls is:

1. Backup your database and application
2. Close all open objectContainers if any
3. Rename classes or fields or copy classes to the new package/namespace in your application.
(Do not remove old classes yet).

4. Issue ObjectClass#rename and objectField#rename calls without having an ObjectContainer open.
5. Open database file and close it again without actually working with it.
6. Remove old classes (if applicable).

After that you will only see the new classes/fields in ObjectManager, the old ones will be gone.

Let's look how it works on an example. We will use initial class Pilot:

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

package com.db4odoc.refactoring;

public class Pilot {
    private String name;

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

    public String getName() {
        return name;
    }

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

and change it to the new class PilotNew renaming field and changing package/namespace at the same time:

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

package com.db4odoc.refactoring;

public class PilotNew {
    private String identity;

    private int points;

    public PilotNew(String name, int points) {
        this.identity = name;
        this.points = points;
    }

    public String getIdentity() {
```

```

        return identity;
    }

    public String toString() {
        return identity + "/" + points;
    }
}

```

First let's create a database and fill it with some values:

```

RefactoringExample.java: setObjects
private static void setObjects() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        Pilot pilot = new Pilot("Rubens Barrichello");
        container.store(pilot);
        pilot = new Pilot("Michael Schumacher");
        container.store(pilot);
    } finally {
        container.close();
    }
}

```

```

RefactoringExample.java: checkDB
private static void checkDB() {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        ObjectSet result=container.queryByExample(new Object());
        listResult(result);
    } finally {
        container.close();
    }
}

```

We already have PilotNew class so we can go on with renaming:

```

RefactoringExample.java: changeClass
private static void changeClass() {
    Configuration configuration = Db4o.newConfiguration();
    configuration.objectClass(Pilot.class).rename("com.db4odoc.f1.refactoring.PilotNew");
    configuration.objectClass(PilotNew.class).objectField("name").rename("identity");
    ObjectContainer container = Db4o.openFile(configuration, DB4O_FILE_NAME);
    container.close();
}

```

Now the data for the old Pilot class should be transferred to the new PilotNew class, and "name" field data should be stored in "identity" field.

To make our check more complicated let's add some data for our new class:

```
RefactoringExample.java: setNewObjects
private static void setNewObjects() {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        PilotNew pilot = new PilotNew("Rubens Barrichello", 99);
        container.store(pilot);
        pilot = new PilotNew("Michael Schumacher", 100);
        container.store(pilot);
    } finally {
        container.close();
    }
}
```

We can check what is stored in the database now:

```
RefactoringExample.java: retrievePilotNew
private static void retrievePilotNew() {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        ObjectSet result = container.query(PilotNew.class);
        listResult(result);
    } finally {
        container.close();
    }
}
```

There is one thing to remember. The rename feature is intended to rename a class from one name to the other. Internally this will rename the meta-information. If you will try to rename class to the name that is already stored in the database, the renaming will fail, because the name is reserved. In our example it will happen if setNewObjects method will be called before changeClass.

Field type change

The reviewed refactoring types are fairly easy. It gets more difficult when you need to change a field's type.

If you modify a field's type, db4o internally creates a new field of the same name, but with the new type. The values of the old typed field are still present, but hidden. If you will change the type back to the old type the old values will still be there.

You can access the values of the previous field data using StoredField API.

Java:

```
StoredClass#storedField(name, type)
```

gives you access to the field, which type was changed.

Java:

```
StoredField#get(Object)
```

allows you to get the old field value for the specified object.

To see how it works on example, let's change Pilot's field name from type string to type Identity:

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

package com.db4odoc.refactoring.newclasses;

public class Identity {
    private String name;

    private String id;

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

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

    public void setId(String id) {
        this.id = id;
    }

    public String toString() {
        return name + "[" + id + "]";
    }
}
```

Now to access old "name" values and transfer them to the new "name" we can use the following procedure:

```

RefactoringExample.java: transferValues
private static void transferValues() {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        StoredClass sc = container.ext().storedClass(
"com.db4odoc.f1.refactoring.oldclasses.Pilot");
        System.out.println("Stored class: " + sc.toString());
        StoredField sfOld = sc.storedField("name", String.class);
        System.out.println("Old field: " + sfOld.toString() + ";" +
sfOld.getStoredType());
        Query q = container.query();
        q.constrain(Pilot.class);
        ObjectSet result = q.execute();
        for (int i = 0; i < result.size(); i++) {
            Pilot pilot = (Pilot)result.queryByExample(i);
            System.out.println("Pilot=" + pilot);
            pilot.setName(new Identity(sfOld.queryByExample(pilot).toString(), ""));
            System.out.println("Pilot=" + pilot);
            container.store(pilot);
        }
    } finally {
        container.close();
    }
}

```

```

RefactoringExample.vb: TransferValues
Private Shared Sub TransferValues()
    Dim container As IObjectContainer = Db4oFactory.OpenFile(Db4oFileName)
    Try
        Dim sc As IStoredClass = container.Ext().StoredClass(GetType(Pilot))
        System.Console.WriteLine("Stored class: " + sc.GetName())
        Dim sfOld As IStoredField = sc.StoredField("_name", GetType(String))
        System.Console.WriteLine("Old field: " + _
sfOld.GetName() + ";" + sfOld.GetStoredType().GetName())
        Dim q As IQuery = container.Query()
        q.Constrain(GetType(Pilot))
        Dim result As IObjectSet = q.Execute()
        Dim obj As Object
        For Each obj In result
            Dim pilot As Pilot = CType(obj, Pilot)
            pilot.Name = New Identity(sfOld.QueryByExample(pilot).ToString(), "")
            System.Console.WriteLine("Pilot=" + pilot.ToString())
            container.Store(pilot)
        Next
    Finally
        container.Close()
    End Try
End Sub

```

These are the basic refactoring types, which can help with any changes you will need to make.

Refactoring Class Hierarchy

db4o does not directly support the following two refactorings:

- Inserting classes into an inheritance hierarchy.
- Removing class from inheritance hierarchies.

For example:

```
class A  
  
class B : A  
  
class C : B
```

1. A new Class D can not be introduced above C.
2. Classes A and B can not be removed.

The only current possible solution for the above refactorings is a workaround:

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 links from existing objects to the new classes.

More Reading:

- [Removing Class From A Hierarchy](#)
- [Inserting Class Into A Hierarchy](#)
- [A](#)
- [B](#)
- [C](#)

A

```
A.java  
/** Copyright (C) 2007 Versant Inc. http://www.db4o.com */  
package com.db4odoc.refactoring.initial;  
  
public class A {  
  
    public String name;  
  
    public String toString() {  
        return name;  
    }  
}
```

B

```
B.java
/** Copyright (C) 2007 Versant Inc. http://www.db4o.com */
package com.db4odoc.refactoring.initial;

public class B extends A {
    public int number;

    public String toString() {
        return name + "/" + number;
    }
}
```

C

```
C.java
/** Copyright (C) 2007 Versant Inc. http://www.db4o.com */
package com.db4odoc.refactoring.initial;

public class C extends B {
}
```

Removing Class From A Hierarchy

Let's use [A](#),[B](#) and [C](#) classes and remove B class, copying its values to the updated C class.

First of all let's store some class objects to the database:

```
refactoringExample.java: storeData
public static void storeData() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        A a = new A();
        a.name = "A class";
        container.store(a);

        B b = new B();
        b.name = "B class";
        b.number = 1;
        container.store(b);

        C c = new C();
        c.name = "C class";
    }
```

```

        c.number = 2;
        container.store(c);
    } finally {
        container.close();
    }
}

```

```

RefactoringExample.java: readData
public static void readData() {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        ObjectSet result = container.queryByExample(new D());
        System.out.println();
        System.out.println("D class: ");
        listResult(result);
    } finally {
        container.close();
    }
}

```

If we will remove B class and update C class to inherit from A, we won't be able to read C data from the database anymore (exception). In order to preserve C data we will need to transfer it to another class:

```

D.java
/**/* Copyright (C) 2007 Versant Inc. http://www.db4o.com */
package com.db4odoc.refactoring.refactored;

import com.db4odoc.refactoring.initial.A;

public class D extends A {
    public int number;

    public String toString() {
        return name + "/" + number;
    }
}

```

We can also transfer B data into this class.

Once D class is created we can run the data transfer:

```

refactoringUtil.java: moveValues
public static void moveValues() {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        // querying for B will bring back B and C values
    }
}

```

```

ObjectSet result = container.queryByExample(new B());
while (result.hasNext()) {
    B b = (B)result.next();
    D d = new D();
    d.name = b.name;
    d.number = b.number;
    container.delete(b);
    container.store(d);
}

} finally {
    container.close();
    System.out.println("Done");
}
}

```

Now B and C classes can be safely removed from the project and all the references to them updated to D. We can check that all the values are in place:

```

RefactoringExample.java: readData
public static void readData() {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        ObjectSet result = container.queryByExample(new D());
        System.out.println();
        System.out.println("D class: ");
        listResult(result);
    } finally {
        container.close();
    }
}

```

When performing refactoring on your working application do not forget to make a copy of the code and data before making any changes!

Inserting Class Into A Hierarchy

We will use the same [A](#), [B](#) and [C](#) classes as in the [previous example](#).

The goal is to insert a new class with additional members between B and C class in the hierarchy.

Let's store some class objects first:

```

refactoringExample.java: storeData
public static void storeData() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
}

```

```

try  {
    A a = new A();
    a.name = "A class";
    container.store(a);

    B b = new B();
    b.name = "B class";
    b.number = 1;
    container.store(b);

    C c = new C();
    c.name = "C class";
    c.number = 2;
    container.store(c);
} finally {
    container.close();
}
}

```

```

RefactoringExample.java: readData
public static void readData() {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        ObjectSet result = container.queryByExample(new D());
        System.out.println();
        System.out.println("D class: ");
        listResult(result);

        result = container.queryByExample(new E());
        System.out.println();
        System.out.println("E class: ");
        listResult(result);
    } finally {
        container.close();
    }
}

```

The following class will be inserted:

```

D.java
/**/* Copyright (C) 2007 Versant Inc. http://www.db4o.com */
package com.db4odoc.refactoring.refactored;

import java.util.Date;

import com.db4odoc.refactoring.initial.B;

public class D extends B  {
    public Date storedDate;

    public String toString() {

```

```

        return name + "/" + number + ": " + storedDate;
    }
}

```

Now C class must inherit from D. We can't change C class itself, because its data will be lost. Therefore we will create a new E class to hold C data:

```

E.java
1/**/* Copyright (C) 2007 Versant Inc. http://www.db4o.com */
2package com.db4odoc.refactoring.refactored;
3
4public class E extends D {
5
6}

```

When all the necessary classes are created we can copy C data into E class:

```

refactoringUtil.java: moveValues
public static void moveValues() {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        ObjectSet result = container.queryByExample(new C());
        while (result.hasNext()) {
            C c = (C)result.next();
            E e = new E();
            e.name = c.name;
            e.number = c.number;
            container.delete(c);
            container.store(e);
        }
    } finally {
        container.close();
        System.out.println("Done");
    }
}

```

Now C classes can be safely removed from the project and all the references to it updated to E(or D). We can check that all the values are in place:

```

RefactoringExample.java: readData
public static void readData() {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {

```

```

ObjectSet result = container.queryByExample(new D());
System.out.println();
System.out.println("D class: ");
listResult(result);

result = container.queryByExample(new E());
System.out.println();
System.out.println("E class: ");
listResult(result);
} finally {
    container.close();
}
}

```

When performing refactoring on your working application do not forget to make a copy of the code and data before making any changes!

Aliases

Db4o Alias/IAlias interface gives you an opportunity to have different names for your persistent classes in the runtime and in the database. The functionality is pretty simple: before class is saved to/retrieved from the database an alias collection is checked for the presence of an alias for the specified class name. If the alias exists, it is used for saving or retrieving instead of the original name.

Db4o provides 2 types of aliases:

- *TypeAlias* is used to alias specific class name to another class name
- *WildcardAlias* allows to alias a package, namespace or multiple similar named classes.

Aliases should be added or removed before a database file is opened.

Java:

```
CommonConfiguration.addAlias(alias)
```

More Reading:

- [TypeAlias](#)
- [WildcardAlias](#)
- [Cross-Platform Aliasing](#)
- [Cross-Platform Aliasing From .NET To Java](#)
- [Cross-Platform Client-Server](#)

TypeAlias

TypeAlias constructor accepts 2 parameters:

- stored class name - storedType
- runtime class name - runtimeType

Note that the runtimeType class should exist in your application when you configure the alias.

The alias matches are found by comparing full names of the stored and runtime classes

Let's declare a new TypeAlias

Java:

```
private static TypeAlias tAlias;
```

The following method will initialize tAlias and configure db4o to use it:

```
AliasExample.java: configureClassAlias
private static EmbeddedConfiguration configureClassAlias()  {
    // create a new alias
    tAlias = new TypeAlias("com.db4odoc.aliases.Pilot",
        "com.db4odoc.aliases.Driver");
    // add the alias to the db4o configuration
    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().addAlias(tAlias);
    // check how does the alias resolve
    System.out.println("Stored name for com.db4odoc.aliases.Driver: "
        + tAlias.resolveRuntimeName("com.db4odoc.aliases.Driver"));
    System.out.println("Runtime name for com.db4odoc.aliases.Pilot: "
        + tAlias.resolveStoredName("com.db4odoc.aliases.Pilot"));
    return configuration;
}
```

We can always check the results of adding an alias using resolveRuntimeName and resolveStoredName as you see in the example. Basically the same methods are used internally by db4o to resolve aliased names.

```
AliasExample.java: saveDrivers
private static void saveDrivers(EmbeddedConfiguration configuration)  {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4oEmbedded.openFile(configuration,
        DB4O_FILE_NAME);
    try  {
        Driver driver = new Driver("David Barrichello", 99);
```

```

        container.store(driver);
        driver = new Driver("Kimi Raikkonen", 100);
        container.store(driver);
    } finally {
        container.close();
    }
}

```

Due to the alias configured the database will have Pilot classes saved. You can check it using ObjectManager

WildcardAlias

WildcardAlias allows creating aliases for packages, namespaces or multiple similar classes. WildcardAlias constructor accepts 2 parameters:

- storedPattern
- runtimePattern

* symbol is used to specify the place where multiple matches are allowed (you can use only one * per pattern).

Let's look how to alias all classes within one package/namespace.

```

AliasExample.java: savePilots
private static void savePilots() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4oEmbedded.openFile(Db4oEmbedded
        .newConfiguration(), DB4O_FILE_NAME);
    try {
        Pilot pilot = new Pilot("David Barrichello", 99);
        container.store(pilot);
        pilot = new Pilot("Kimi Raikkonen", 100);
        container.store(pilot);
    } finally {
        container.close();
    }
}

```

You can check the matches for the concrete classes using resolveRuntimeName and resolveStoredName.

In order to add your own aliasing logic implement Alias interface (for example you may want to use more sophisticated pattern logic). Your own resolving logic implementation should reside in resolveRuntimeName and resolveStoredName methods.

Cross-Platform Aliasing

One of the most valuable aliases usescases is working with persistent Java classes from a .NET application and vice versa. You can use both TypeAlias and WildcardAlias depending on how many classes you need to access.

Below you will get an example of a system where classes were saved to the database from a Java application and read and modified later from a .NET application. A vice versa example is reviewed in [Cross-Platform Aliasing From .NET To Java](#).

For example, Pilot objects are saved to a database from a Java application:

```
InterLanguageExample.java: saveObjects
private static void saveObjects() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4oEmbedded.openFile(Db4oEmbedded
        .newConfiguration(), DB4O_FILE_NAME);
    try {
        Pilot pilot = new Pilot("David Barrichello", 99);
        container.store(pilot);
        pilot = new Pilot("Michael Schumacher", 100);
        container.store(pilot);
    } finally {
        container.close();
    }
}
```

In order to read the saved objects successfully from a .NET application we will need to define an alias for persistent classes and add JavaSupport to configuration. JavaSupport is a special configuration item, which defines aliases for all necessary internal db4o classes that might be referenced in the database file. We will use a [WildcardAlias](#) for all the persistent classes. This alias must reference the corresponding persistent class definitions on Java and .NET (remember that only matching field names will be accessible in aliased objects):

```
InterLanguageExample.cs: ConfigureAlias
private static IEmbeddedConfiguration ConfigureAlias()
{
    IEmbeddedConfiguration configuration =
Db4oEmbedded.NewConfiguration();
    configuration.Common.AddAlias(new WildcardAlias(
"com.db4odoc.aliases.*", "Db4odoc.Aliases.", Db4odoc"));
    configuration.Common.Add(new JavaSupport());
    return configuration;
}
```

Now the objects are accessible from the .NET application:

```
InterLanguageExample.cs: GetObjects
private static void GetObjects(IEmbeddedConfiguration configuration)
{
    IObjectContainer db = Db4oEmbedded.OpenFile(configuration, Db4oFileName);
    try
```

```

    {
        IList<Pilot> result = db.Query<Pilot>(delegate(Pilot pilot)
{ return pilot.Points%2 == 0; });
        for (int i = 0; i < result.Count; i++)
        {
            Pilot pilot = result[i];
            pilot.Name = "Modified " + pilot.Name;
            db.Store(pilot);
        }
        ListResult(result);
    }
    finally
    {
        db.Close();
    }
}

```

We can read the database from the initial Java application again. Note, that no alias is required as the class definitions were created from Java:

```

InterLanguageExample.java: readObjects
private static void readObjects() {
    ObjectContainer container = Db4oEmbedded.openFile(Db4oEmbedded
        .newConfiguration(), DB4O_FILE_NAME);
    try {
        ObjectSet<Pilot> result = container.queryByExample(new Pilot(null, 0));
        listResult(result);
    } finally {
        container.close();
    }
}

```

Cross-Platform Aliasing From .NET To Java

The following example shows a cross-platform system where classes were saved to the db4o database from a .NET application and read and modified later from a .Java application. A vice versa example is reviewed in [Cross-Platform Aliasing](#).

Pilot objects are saved to a database from a .NET application:

```

InterLanguageExample2.cs: SaveObjects
private static void SaveObjects()
{
    File.Delete(Db4oFileName);
    IObjectContainer container = Db4oEmbedded.OpenFile(
Db4oEmbedded.NewConfiguration(), Db4oFileName);
    try
    {
        Pilot pilot = new Pilot("David Barrichello", 99);
        container.Store(pilot);
        pilot = new Pilot("Michael Schumacher", 100);
        container.Store(pilot);
    }
}

```

```

        finally
        {
            container.Close();
        }
    }
}

```

In order to read the saved objects successfully from a java application we will need to define an alias for persistent classes and add DotnetSupport to configuration. DotnetSupport is a special configuration item, which includes aliases for all internal db4o classes, which might be referenced in the database file We will use a [WildcardAlias](#) to match the definitions for Java and .NET persistent classes. It is important to remember that persistent classes in package/namespace specified in WildcardAlias must have the same names and same field names to be matched:

```

InterLanguageExample2.java: configureAlias
private static EmbeddedConfiguration configureAlias()  {
    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().addAlias(new WildcardAlias(
        "Db4odocAliases.*", "Db4odoc",
        "com.db4odoc.aliases.*"));
    configuration.common().add(new DotnetSupport(false));
    return configuration;
}

```

Now the objects are accessible from the Java application:

```

InterLanguageExample2.java: getObjects
private static void getObjects(EmbeddedConfiguration configuration)  {
    ObjectContainer db = Db4oEmbedded.openFile(configuration, DB4O_FILE_NAME);
    try  {
        List<Pilot> result = db.query(new Predicate<Pilot>()  {
            public boolean match(Pilot pilot)  {
                return true;
            }
        });
        for (int i = 0; i < result.size(); i++)  {
            Pilot pilot = result.queryByExample(i);
            pilot.setName("Modified " + pilot.getName());
            db.store(pilot);
        }
        listResult(result);
    } finally  {
        db.close();
    }
}

```

We can read the database from the initial .NET application again. Note, that no alias is required as the class definitions were created in this application :

```

InterLanguageExample2.cs: ReadObjects
private static void ReadObjects()
{
    IObjectContainer container = Db4oEmbedded.OpenFile(
Db4oEmbedded.NewConfiguration(), Db4oFileName);
    try
    {

```

```

        IList<Pilot> result = container.Query<Pilot>();
        ListResult(result);
    }
    finally
    {
        container.Close();
    }
}

```

Cross-Platform Client-Server

Aliases in combination with JavaSupport and DotnetSupport allow to achieve another amazing functionality: using .NET server from Java application and Java server from .NET application. The basic steps are the same as discussed [before](#), however there are some additional specifics:

1. Persistent classes should be available on the server side and an appropriate alias used
2. Db4o version should be the same on the client and on the server

Let's look at it on an example.

More Reading:

- [Java Server-.NET Client](#)
- [Java Client-.NET Server](#)

Java Client-.NET Server

In the [previous example](#) we saw how to access Java server from .NET clients. The opposite is also possible.

Let's start a typical .NET db4o server (c# or VB):

c#:

```

StartServer.cs
/**/* Copyright (C) 2004 - 2009 Versant Corporation http://www.versant.com */
using System;
using System.Threading;
using Db4objects.Db4o;
using Db4objects.Db4o.Config;
using Db4objects.Db4o.Messaging;

namespace JavaDotnetClientServer
{
    class StartServer : ServerInfo, IMessageRecipient
    {
        private bool stop = false;

        /**/// <summary>
        /// starts a db4o server using the configuration from

```

```

///> ServerInfo.
///> </summary>
public static void Main(string[] arguments)
{
    new StartServer().RunServer();
}
// end Main

/**>>> <summary>
///> opens the IObjectServer, and waits forever until Close() is called
///> or a StopServer message is being received.
///> </summary>
public void RunServer()
{
    lock (this)
    {
        // Using the messaging functionality to redirect all
        // messages to this.processMessage
        IConfiguration configuration =
Db4oFactory.NewConfiguration();
        configuration.ClientServer().SetMessageRecipient(this);

        IObjectServer db4oServer = Db4oFactory.
OpenServer(configuration, FileName, Port);
        db4oServer.GrantAccess(User, Password);

        try
        {
            if (!stop)
            {
                // wait forever until Close will change stop variable
                Monitor.Wait(this);
            }
        }
        catch (Exception e)
        {
            Console.WriteLine(e.ToString());
        }
        db4oServer.Close();
    }
}
// end RunServer

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

```

```

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

```

```

ServerInfo.cs
/**/* Copyright (C) 2004 - 2009 Versant Corporation http://www.versant.com */
namespace JavaDotnetClientServer
{
    /**<summary>
    /// Configuration used for StartServer and StopServer.
    </summary>
    public class ServerInfo
    {
        /**<summary>
        /// the host to be used.
        /// If you want to run the client server examples on two computers,
        /// enter the computer name of the one that you want to use as server.
        </summary>
        public const string Host = "localhost";

        /**<summary>
        /// the database file to be used by the server.
        </summary>
        public const string FileName = "reference.db4o";

        /**<summary>
        /// the port to be used by the server.
        </summary>
        public const int Port = 0xdb40;

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

        /**<summary>
        /// the password for access control.
        </summary>
        public const string Password = "db4o";
    }
}

```

.NET:

```

StartServer.vb
' Copyright (C) 2004 - 2009 Versant Corporation http://www.versant.com

Imports System
Imports System.Threading
Imports Db4objects.Db4o
Imports Db4objects.Db4o.Config
Imports Db4objects.Db4o.Messaging

Class StartServerClass StartServer
    Inherits ServerInfo
    Implements IMessageRecipient
    Private [stop] As Boolean = False

    /**/''' <summary>
    ''' starts a db4o server using the configuration from
    ''' ServerInfo.
    ''' </summary>
    Public Shared Sub Main() Sub Main(ByVal arguments As String())
        Dim server As StartServer = New StartServer()
        server.RunServer()
    End Sub
    ' end Main

    /**/''' <summary>
    ''' opens the IObjectServer, and waits forever until Close() is called
    ''' or a StopServer message is being received.
    ''' </summary>
    Public Sub RunServer() Sub RunServer()
        SyncLock Me
            ' Using the messaging functionality to redirect all
            ' messages to this.processMessage
            Dim configuration As IConfiguration = Db4oFactory.NewConfiguration()
            configuration.ClientServer().SetMessageRecipient(Me)

            Dim db4oServer As IObjectServer = Db4oFactory. _
OpenServer(configuration, FileName, Port)
            db4oServer.GrantAccess(User, Password)

            Try
                If Not [stop] Then
                    ' wait forever until Close will change stop variable
                    Monitor.Wait(Me)
                End If
            Catch e As Exception
                Console.WriteLine(e.ToString())
            End Try
            db4oServer.Close()
        End SyncLock
    End Sub
    ' end RunServer

    /**/''' <summary>
    ''' messaging callback

```

```

''' see com.db4o.messaging.MessageRecipient#ProcessMessage()
''' </summary>
Public Sub ProcessMessage() Sub ProcessMessage(ByVal context As _
IMessageContext, ByVal message As Object) _
Implements IMessageRecipient.ProcessMessage
    If TypeOf message Is StopServer Then
        Close()
    End If
End Sub
' end ProcessMessage

/**/''' <summary>
''' closes this server.
''' </summary>
Public Sub Close() Sub Close()
    SyncLock Me
        [stop] = True
        Monitor.PulseAll(Me)
    End SyncLock
End Sub
' end Close

End Class

```

```

ServerInfo.vb
' Copyright (C) 2004 - 2009 Versant Corporation http://www.versant.com

/**/''' <summary>
''' Configuration used for StartServer and StopServer.
''' </summary>
Public Class ServerInfoClass ServerInfo
    /**/''' <summary>
    ''' the host to be used.
    ''' If you want to run the client server examples on two computers,
    ''' enter the computer name of the one that you want to use as server.
    ''' </summary>
    Public Const Host As String = "localhost"

    /**/''' <summary>
    ''' the database file to be used by the server.
    ''' </summary>
    Public Const FileName As String = "reference.db4o"

    /**/''' <summary>
    ''' the port to be used by the server.
    ''' </summary>
    Public Const Port As Integer = &HDB40

    /**/''' <summary>
    ''' the user name for access control.
    ''' </summary>
    Public Const User As String = "db4o"

    /**/''' <summary>

```

```

''' the password for access control.
''' </summary>
Public Const Password As String = "db4o"
End Class

```

Remember, that persistent class definitions should be present both on the client and the server side. For simplicity we will use the classes from the [previous example](#).

Java client testing code:

```

Client.java: main
public static void main(String[] args)  {
    ObjectContainer db = Db4oClientServer.openClient(getConfig(), host,
        port, user, password);
    db.store(new Car("Ferrari", new Pilot("Michael Schumacher")));
    db.store(new Car("BMW", new Pilot("Rubens Barrichello")));
    db.close();

    db = Db4oClientServer.openClient(getConfig(), host, port, user,
        password);
    List<Car> cars = db.queryByExample(Car.class);
    for (Car car : cars)  {
        System.out.println(car);
    }
    db.close();
}

```

And the most important part - configuration:

```

Client.java: getConfig
private static ClientConfiguration getConfig()  {
    ClientConfiguration config = Db4oClientServer.newClientConfiguration();
    config.common().add(new DotnetSupport(true));
    config.common().addAlias(
        new WildcardAlias(
            "JavaDotnetClientServer.*", JavaDotnetClientServer",
            "com.db4odoc.JavaDotNetClientServer.*"));
    return config;
}

```

Note, that we use DotnetSupport constructor with a boolean parameter set to true. This parameter defines if Client/Server support should be included.

Persistent class aliases are required for accessing object information on the server (both Wildcard and class-specific aliases can be used, as soon as they cover all the classes which you want to access).

Java Server-.NET Client

Java server code is a typical server code:

```

Server.java
/**/* Copyright (C) 2004 - 2009 Versant Corporation http://www.versant.com */

```

```

package com.db4odoc.JavaDotNetClientServer;

import com.db4o.*;
import com.db4o.cs.*;
import com.db4o.cs.config.*;
import com.db4o.messaging.*;

public class Server implements ServerInfo, MessageRecipient {

    /** */
    * setting the value to true denotes that the server should be closed
    */
    private boolean stop = false;

    /** */
    * starts a db4o server using the configuration from
    * {@link ServerInfo}.
    */
    public static void main(String[] arguments) {
        new Server().runServer();
    }
    // end main

    /** */
    * opens the ObjectServer, and waits forever until close() is called
    * or a StopServer message is being received.
    */
    public void runServer() {
        synchronized(this) {
            ServerConfiguration c = Db4oClientServer.newServerConfiguration();

            ObjectServer db4oServer = Db4oClientServer.openServer(c, FILE, PORT);
            db4oServer.grantAccess(USER, PASS);

            // Using the messaging functionality to redirect all
            // messages to this.processMessage
            db4oServer.ext().configure().clientServer().setMessageRecipient(this);

            // to identify the thread in a debugger
            Thread.currentThread().setName(this.getClass().getName());

            // We only need low priority since the db4o server has
            // it's own thread.
            Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
            try {
                if(! stop) {
                    // wait forever for notify() from close()
                    this.wait(Long.MAX_VALUE);
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
            db4oServer.close();
        }
    }
}

```

```

// end runServer

/** */
 * messaging callback
 * @see com.db4o.messaging.MessageRecipient#processMessage(ObjectContainer, Object)
 */
public void processMessage(MessageContext context, Object message) {
    if(message instanceof StopServer) {
        close();
    }
}
// end processMessage

/** */
 * closes this server.
 */
public void close() {
    synchronized(this) {
        stop = true;
        this.notify();
    }
}
// end close

}

```

```

ServerInfo.java
/**/* Copyright (C) 2004 - 2009 Versant Corporation http://www.versant.com */
package com.db4odoc.JavaDotNetClientServer;

/** */
* Configuration used for {@link StartServer} and {@link StopServer}.
*
* @sharpen.ignore
*/
public interface ServerInfo {

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

/** */
* the database file to be used by the server.
*/
public String FILE = "reference.db4o";

/** */
* the port to be used by the server.
*/
public int PORT = 0xdb40;

/** */

```

```

    * the user name for access control.
 */
public String USER = "db4o";

/** */
    * the password for access control.
 */
public String PASS = "db4o";
}

```

We'll save and retrieve simple Pilot and Car classes:

c#:

```

Pilot.cs
/**/* Copyright (C) 2004 - 2009 Versant Corporation http://www.versant.com */
using System;

namespace JavaDotnetClientServer
{
    class Pilot
    {
        private String name;

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

        public String getName()
        {
            return name;
        }

        public void setName(String pilotName)
        {
            this.name = pilotName;
        }

        public override String ToString()
        {
            return name;
        }
    }
}

```

```

Car.cs
/**/* Copyright (C) 2004 - 2009 Versant Corporation http://www.versant.com */
using System;

namespace JavaDotnetClientServer
{
    class Car
    {
        private String model;
    }
}

```

```

private Pilot pilot;

public Car(String carModel, Pilot carPilot)
{
    this.model = carModel;
    this.pilot = carPilot;
}

public Pilot Pilot
{
    get { return pilot; }
}

public string Model
{
    get { return model; }
}

public override String ToString()
{
    return Model + "[" + Pilot + "]";
}
}
}

```

VB.NET:

```

Pilot.vb
' Copyright (C) 2004 - 2009 Versant Corporation http://www.versant.com

Imports System

Class PilotClass Pilot
    Private name As [String]

    Public Sub New() Sub New(ByVal name As [String])
        Me.name = name
    End Sub

    Public Function getName() Function getName() As [String]
        Return name
    End Function

    Public Sub setName() Sub setName(ByVal pilotName As [String])
        Me.name = pilotName
    End Sub

    Public Overloads Overrides Function ToString() Function ToString() As [String]
        Return name
    End Function
End Class

```

```

Car.vb
' Copyright (C) 2004 - 2009 Versant Corporation http://www.versant.com

```

```

Imports System

Class CarClass Car
    Private model As String
    Private pilot As Pilot

    Public Sub New() Sub New(ByVal carModel As String, ByVal carPilot As Pilot)
        Me.model = carModel
        Me.pilot = carPilot
    End Sub

    Public Overloads Overrides Function ToString() Function ToString() As String
        Return String.Format("{0} [{1}]", model, pilot)
    End Function
End Class

```

Now, remember requirement #1: the same classes should be available on the server. So we create Pilot and Car version in Java (don't forget that member names should match all along:

```

Pilot.java
/**/* Copyright (C) 2004 - 2009 Versant Corporation http://www.versant.com */

package com.db4odoc.JavaDotNetClientServer;

public class Pilot {
    private String name;

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

    public String getName() {
        return name;
    }

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

    public String toString() {
        return name;
    }
}

```

```

Car.java
/**/* Copyright (C) 2004 - 2009 Versant Corporation http://www.versant.com */

package com.db4odoc.JavaDotNetClientServer;

public class Car {

```

```

private String model;
private Pilot pilot;

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

public Pilot getPilot()  {
    return pilot;
}

public String getModel()  {
    return model;
}

public String toString()  {
    return model+"["+pilot+"]";
}
}

```

Here is our testing code:

c#:

```

Client.cs: Main
public static void Main(string[] args)
{
    IObjectContainer db = Db4oClientServer.OpenClient(GetConfig(), host,
        port, user, password);
    db.Store(new Car("Ferrari", new Pilot("Michael Schumacher")));
    db.Store(new Car("BMW", new Pilot("Rubens Barrichello")));
    db.Close();

    db = Db4oClientServer.OpenClient(GetConfig(), host, port, user,
        password);
    IList<Car> cars = db.Query<Car>();
    foreach (Car car in cars)
    {
        System.Console.WriteLine(car);
    }
    db.Close();
}

```

VB.NET:

```

Client.vb: Main
Public Shared Sub Main() Sub Main(ByVal args As String())
    Dim db As IObjectContainer = Db4oClientServer. _
OpenClient(GetConfig(), host, port, user, password)
    db.Store(New Car("Ferrari", New Pilot("Michael Schumacher")))
    db.Store(New Car("BMW", New Pilot("Rubens Barrichello")))
    db.Close()

    db = Db4oClientServer.OpenClient(GetConfig(), host, port, user, password)

```

```

Dim cars As IList(Of Car) = db.Query(Of Car)()
For Each car As Car In cars
    System.Console.WriteLine(car)
Next
db.Close()
End Sub

```

Nothing special above - just storing and retrieving objects. The main interest is in the call to GetConfig() method:

c#:

```

Client.cs: GetConfig
private static IClientConfiguration GetConfig()
{
    IClientConfiguration config = Db4oClientServer.NewClientConfiguration();
    config.Common.AddAlias(new WildcardAlias("com.db4odoc.JavaDotNetClientServer.*",
        "JavaDotnetClientServer.*", JavaDotnetClientServer"));
    config.Common.Add(new JavaSupport());
    return config;
}

```

VB.NET:

```

Client.vb: GetConfig
Private Shared Function GetConfig() Function GetConfig() As IClientConfiguration
    Dim config As IClientConfiguration = _
Db4oClientServer.NewClientConfiguration()
    config.Common.AddAlias(New WildcardAlias( _
"com.db4odoc.JavaDotNetClientServer.*", "JavaDotnetClientServer.*", JavaDotnetClientServer"))
    config.Common.Add(New JavaSupport())
    Return config
End Function

```

From the above we can see that there are 2 steps required to get access to the Java server:

1. Add JavaSupport
2. Add alias for the persistent classes (In the example we use WildcardAlias, however separate aliases for Pilot and Car classes will work the same).

Encryption

db4o provides simple built-in encryption functionality. This feature is easy to turn on or off, and must be configured before opening a database file.

In addition, db4o provides the ability for you to plug in your own encrypting IO Adapters.

More Reading:

- [Custom Encryption Adapters](#)

Custom Encryption Adapters

db4o still provides a solution for high-security encryption by allowing any user to choose his own encryption mechanism that he thinks he needs. The db4o file IO mechanism is pluggable and any fixed-length encryption mechanism can be added. All that needs to be done is to write an IoAdapter plugin for db4o file IO.

This is a lot easier than it sounds. Simply:

- take the sources of com.db4o.io.RandomAccessFileAdapter as an example
- write your own IoAdapter implementation that delegates raw file access to another adapter using the GoF decorator pattern.
- Implement the #read() and #write() methods to encrypt and decrypt when bytes are being exchanged with the file
- plug your adapter into db4o with the following method:

```
Java: Db4o.configure().io(new MyEncryptionAdapter());
```

However, you'll have to keep in mind that db4o will write partial updates. For example, it may write a full object and then only modify one field entry later on. Therefore it is not sufficient to en-/decrypt each access in isolation. You'll rather have to make up a tiling structure that defines the data chunks that have to be en-/decrypted together.

A community project containing an XTEA encryption IoAdapter implementation can be found here:

<http://developer.db4o.com/ProjectSpaces/view.aspx/XTEA>

Another method to inject encryption capabilities into db4o for instances of specific classes only is to implement and configure an en-/decrypting translator.

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. The reference system also makes sure that every persistent object is instantiated only once, when a graph of objects is retrieved from the database, no matter which access path is chosen. If an object is accessed by multiple queries or by multiple navigation access paths, db4o will always return the one single object, helping you to put your object graph together exactly the same way as it was when it was stored, without having to use IDs.

The use of IDs does make sense when object and database are disconnected, for instance in stateless applications.

More Reading:

- [Internal IDs](#)
- [Unique Universal IDs](#)

Internal IDs

The internal db4o ID is a physical pointer into the database with only one indirection in the file to the actual object so it is the fastest external access to an object db4o provides. The internal ID of an object is available with

Java: `ObjectContainer.ext().getID(object)`

To get an object for an internal ID use

Java: `ObjectContainer.ext().getByID(id)`

Note that `#getByID()` does not activate objects. If you want to work with objects that you get with `#getByID()`, your code would have to make sure the object is [activated](#) by calling

Java: `ObjectContainer.activate(object, depth)`

db4o assigns internal IDs to any stored first class object. These internal IDs are guaranteed to be unique within one ObjectContainer/ObjectServer and they will stay the same for every object when an ObjectContainer/ObjectServer is closed and reopened. Internal IDs **will change** when an object is moved from one ObjectContainer to another, as it happens during [Defragment](#).

Unique Universal IDs

For long-term external references and to identify an object even after it has been copied or moved to another ObjectContainer, db4o supplies Unique Universal IDs (UUIDs).

Every newly created db4o database generates a signature object. It is stored as an instance of `Db4oDatabase` in your database file.

This signature is linked to all newly created objects (if UUID generation is enabled) as the "signature part" of the UUID.

Further to that db4o creates a timestamp for every new object and uses an internal counter to make sure that timestamps are unique. This is called the "long part" of the UUID.

The long part is indexed to make the search fast. If two objects with an identical long parts are found, the signature parts are compared also.

The long part of the UUID can also be used to find out when an object was created. You can use

Java:

```
com.db4o.foundation.TimeStampIdGenerator#idToMilliseconds()
```

to get object creation time in milliseconds.

UUIDs are guaranteed to be unique, if the signature of your db4o database is unique.

Normally any database has a unique signature unless its file is copied. The original and copied database files are identical, so they have the same signatures. If such files are used in replication, the process will end up with exceptions. What is the solution then?

Signature of a database file can be changed using

Java:

```
LocalObjectContainer#generateNewIdentity
```

method.

```
UUIDExample.java: testChangeIdentity
private static void testChangeIdentity()  {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    Db4oDatabase db;
    byte[] oldSignature;
    byte[] newSignature;
    try  {
        db = container.ext().identity();
        oldSignature = db.getSignature();
        System.out.println("oldSignature: "
            + printSignature(oldSignature));
        ((LocalObjectContainer) container).generateNewIdentity();
    } finally  {
        container.close();
    }
    container = Db4o.openFile(DB4O_FILE_NAME);
    try  {
        db = container.ext().identity();
        newSignature = db.getSignature();
        System.out.println("newSignature: "
            + printSignature(newSignature));
    } finally  {
        container.close();
    }
}
```

```

boolean same = true;

for (int i = 0; i < oldSignature.length; i++) {
    if (oldSignature[i] != newSignature[i]) {
        same = false;
    }
}

if (same) {
    System.out.println("Database signatures are identical");
} else {
    System.out.println("Database signatures are different");
}
}

```

UUIDs are not generated by default, since they occupy extra space in the database file and produce performance overhead for maintaining their index. UUIDs can be turned on globally or for individual classes:

Java: configuration.generateUUIDs(ConfigScope.GLOBALLY)

- turns on UUID generation for all classes in a database.

Java: configuration.objectClass(Foo.class).generateUUIDs(true)

- turns on UUID generation for a specific class.

You can get the UUID value for an object using the following methods:

Java:

ExtObjectContainer#getObjectInfo(Object) ObjectInfo#getUUID()

To get the object from the database, knowing its UUID, use:

Java:

ExtObjectContainer#getByUUID(Db4oUUID)

The following example shows the usage of UUID:

```

UUIDExample.java: setObjects
private static void setObjects()  {
    new File(DB4O_FILE_NAME).delete();
    Configuration configuration = Db4o.newConfiguration();
    configuration.objectClass(Pilot.class).generateUUIDs(true);
    ObjectContainer container = Db4o.openFile(configuration, DB4O_FILE_NAME);
    try  {
        Car car = new Car("BMW", new Pilot("Rubens Barrichello"));
        container.store(car);
    } finally  {
        container.close();
    }
}

```

```

UUIDExample.java: testGenerateUUID
private static void testGenerateUUID()  {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try  {
        Query query = container.query();
        query.constrain(Car.class);
        ObjectSet result = query.execute();
        Car car = (Car) result.queryByExample(0);
        ObjectInfo carInfo = container.ext().getObjectInfo(car);
        Db4oUUID carUUID = carInfo.getUUID();
        System.out
            .println("UUID for Car class are not generated:");
        System.out.println("Car UUID: " + carUUID);

        Pilot pilot = car.getPilot();
        ObjectInfo pilotInfo = container.ext().getObjectInfo(
            pilot);
        Db4oUUID pilotUUID = pilotInfo.getUUID();
        System.out.println("UUID for Pilot:");
        System.out.println("Pilot UUID: " + pilotUUID);
        System.out.println("long part: "
            + pilotUUID.getLongPart() + "; signature: "
            + printSignature(pilotUUID.getSignaturePart()));
        long ms = TimeStampIdGenerator.idToMilliseconds(pilotUUID
            .getLongPart());
        System.out.println("Pilot object was created: "
            + (new Date(ms)).toString());
        Pilot pilotReturned = (Pilot) container.ext().getByUUID(
            pilotUUID);
        System.out.println("Pilot from UUID: " + pilotReturned);
    } finally  {
        container.close();
    }
}

```

Sometimes you can find out that you need UUIDs only when the database is already created and has some data in it. What can you do in that case?

Fortunately enabling replication for existing data files is a very simple process:

Java:

```
configuration.objectClass(Task.class).enableReplication(true)
```

After that you will just need to use the old defragment tool from tools package supplied with the distribution before version 6.0 (source code only) to enable replication.

You can use UUID for replication and as a reference to a specific object instance from an external application or data store.

Freespace Management System

Db4o Freespace manager is a special system that is responsible for allocating, discarding, merging of space for db4o usage and keeping database size at minimum

Db4o organizes its files into variable-sized slots with one degree of indirection (pointer slots). The physical address of a pointer slot corresponds to the internal ID of the respective object. On the lowest level the database file is seen as a uniform pool of bytes, from which slots of variable size can be allocated.

Freespace manager keeps track of the portions of the database file that haven't been allocated for use and gets notified when slots are 'freed', i.e. returned back to the pool.

That is how it works:

- All objects are written to the database file immediately, when they are stored or updated.
- Updated objects get a new or previously freed place in the database file.
- New or modified pointers to new or modified objects are stored in RAM.
- Old and deleted objects are marked as 'freespace'.

In general case each persisted object occupies one slot. This slot contains metadata, values of direct primitive members and references to the slots of non-primitive members. What actually goes directly into a slot and what is external is not fixed and differs from version to version. However, there is one level of indirection: a reference to a non-primitive member will not refer to the member's slot directly, but rather to the 'pointer', which is an 8 byte slot containing the address in the file and the data length. The address of this pointer slot is the object's internal ID that is used for indexing, etc.

For a simple example, assume we have

```
class Car { String manufacturer; }

class Driver { String name; Car car; }
```

and an object graph like this

Driver: name='Barrichello', car={Car: manufacturer='BMW'}

This will translate into four slots

1234: [4711, length(Driver)] 4711: [Driver,'Barrichello',0815] 0815: [4321, length(Car)] 4321: [Car,'BMW']

with 1234 being the db4o ID of the driver and 0815 the ID of the car. Whenever the driver or the car object is updated, its actual slot may be stored somewhere else, but the pointer slot (the ID) will remain the same and keep track of the slot's address. (Please, note that this is a simplified example: actual implementation uses more slots and more sophisticated processing).

More Reading:

- [Two Freespace Systems](#)
- [How To Use FreespaceManager](#)
- [Defragmentation Role](#)

Two Freespace Systems

db4o comes with three freespace systems:

- RAM-based: the information about freespace is held in RAM;
- b-Tree-based: the information about freespace is written to disk, b-Trees are used to manage this information;
- index-based: similar to b-Trees, but existing index functionality to store freespace information.

You can configure db4o to use either of these by calling

Java: configuration.freespace().useRamSystem()

or

Java: configuration.freespace().useBTreeSystem()

or

Java: configuration.freespace().useIndexSystem()

This call should be made before you open the database for the first time

By default db4o uses **RAM freespace management system**. The information about free slots is loaded into memory on opening a database file and discarded on closing it. This system is quite fast, but it has its downside:

1. Higher RAM usage during operation.
2. Loss of freespace upon abnormal termination. That is done for security reasons and freespace can be reclaimed using defragmentation.

RAM-based freespace management is a good performance solution, but it can be insufficient for the systems with limited RAM resources and high probability of abnormal system termination (power failure on mobile devices).

In order to meet the requirements of such environments you can use new **b-Tree-based freespace management system**. It solves the problems of RAM-based system:

1. RAM usage is kept at the minimum.
2. No freespace is lost on abnormal system termination (database file won't grow unnecessarily).

How it works?:

- The system uses b-Trees to keep information about available freespace
- b-Trees operate against the file, and only uses memory for caching
- For every new write to the database file the system tries to find a freed slot, which is at least the size needed or greater, traversing freespace index
- When an object is updated or deleted, its 'old' slot is added to the freespace b-Tree entry
- This b-Tree system is ACID (no information is lost upon abnormal system termination)

b-Tree-based freespace system can show poorer performance compared to RAM-based system, as it needs to access the file to write updated freespace information.

However, b-Tree-based freespace system is fast enough, especially for mobile devices, where file access is not much slower than RAM-access, and ACID transactions together with low memory consumption are most valuable factors.

Index-based freespace system has similar to b-Tree characteristics, but poorer performance and is used for legacy reasons.

How To Use FreespaceManager

There are several configuration options that can help you to tune up your freespacemanager to achieve the best performance and reliability of your system. All methods should be called before opening database files.

Public interface FreespaceConfiguration provides methods to select freespace system (*useIndexSystem()*, *useRamSystem()*) as described before. See API documentation for more information.

Another FreespaceConfiguration method

Java:

```
void discardSmallerThan(int byteCount)
```

configures the minimum size of free space slots in the database file that are to be reused. FreespaceManager keeps 2 lists of all 'freed' space that can be reused (sorted by address and by size). In some cases (numerous updates, deletes) these lists can grow large, causing extra RAM consumption and performance loss for maintenance. With this method you can specify an upper bound for the byte slot size to discard from Freespace manager list. It is not recommended to specify a value of byteCount > 100 as freespace re-usage will become less efficient and the database file will grow faster. However, if defragment can be run frequently, it will also reclaim lost space and decrease the database file to the minimum size. Therefore byteCount may be set to bigger value.

By default byteCount = 0, which means that all 'freed' space is reused.

Another configuration setting that can be used with frequently defragmented systems

Java:

```
configuration.automaticShutDown(false);
```

Detailed description of this method can be found in [Tuning](#) part of the Reference documentation.

Defragmentation Role

So we have a reliable freespace manager, which will keep the database file size to the minimum. But do we still need to bother about [defragmentation](#)?

Yes, we do for several reasons:

- When the object is deleted, its space in the database is marked as 'freed'. But 8 bytes of its internal ID stay behind. Defragment cleans all this up by writing all objects to a completely new database file. The resulting file will be smaller and faster.
- 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, while their 'old' space is marked as free. Adjacent free slots will be merged on the fly, but free and used slots won't be clustered together unless defragmentation is run.

String Encoding

When db4o stores and loads strings it has to convert them to and from byte arrays. By default db4o provides 2 types of string encoding, which do this job: Unicode 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 ISO 8859-1, which only required 2 bytes per character. However ISO 8859-1 only supports latin alphabet.

In order to make string encoding more flexible and configurable pluggable string encoding was introduced in db4o version 7.7. The API looks like this:

```
Configuration#stringEncoding(stringEncoding)
```

Where stringEncoding can be either one of the default ones:

Java:

```
Configuration config = Db4o.newConfiguration();
config.stringEncoding(StringEncodings.utf8());
Db4o.openFile(config, "myDB.db4o");
```

or a newly implemented one:

Java:

```
Configuration config = Db4o.newConfiguration();
config.stringEncoding(new StringEncoding() {
    public byte[] encode(String str) {
        // TODO: implement
        return null;
    }
    public String decode(byte[] bytes, int start, int length) {
        // TODO: implement
        return null;
    }
});
Db4o.openFile(config, "myDB.db4o");
```

It is important to remember that the string encoding is selected once for the lifetime of a database and cannot be changed afterwards. With the default encoding it is enough to supply the encoding configuration when the database is created, it will be picked up automatically afterwards. With the custom string encoding's, you must make sure that the same encoding is submitted with every database open call. As db4o uses string encoding to store metadata information like field and class-names, opening a database with a wrong string encoding will fail with exceptions.

StringEncoding Example

Lets look at an example string encoding implementation. We need to provide 2 methods: encode, which takes a string and produces byte array and decode, taking a byte array and producing a string. To make it interesting, we will "encrypt" the bytes stored in the database, by shifting them 1 byte ahead with the following class:

```
StringEncoderExample.java: Encryption
private static class Encryption  {

    public static String encrypt(String s)
    {
        char[] array = s.toCharArray();
        for (int i = 0; i < array.length; i++)
        {
            array[i]++;
        }
        return new String(array);
    }

    public static String decrypt(String s)  {
        char[] array = s.toCharArray();
        for (int i = 0; i < array.length; i++)
        {
            array[i]--;
        }
        return new String(array);
    }
}
```

The following code creates a StringEncoding class and adds it to configuration:

```
StringEncoderExample.java: configure
protected static Configuration configure()  {
    Configuration config = Db4o.newConfiguration();
    config.stringEncoding(new StringEncoding()  {
        public byte[] encode(String str)  {
            str = Encryption.encrypt(str);
            int length = str.length();
            char[] chars = new char[length];
            str.getChars(0, length, chars, 0);
        }
    });
}
```

```

byte[] bytes = new byte[length * 2];
int count = 0;
for (int i = 0; i < length; i++) {
    bytes[count++] = (byte) (chars[i] & 0xff);
    bytes[count++] = (byte) (chars[i] >> 8);
}
return bytes;
}

public String decode(byte[] bytes, int start, int length) {
    int stringLength = length / 2;
    char[] chars = new char[stringLength];
    int j = start;
    for(int ii = 0; ii < stringLength; ii++) {
        chars[ii] = (char) ((bytes[j++]& 0xff) | ((bytes[j++]& 0xff) << 8));
        //j+=2;
    }
    return Encryption.decrypt(new String(chars,0,stringLength));
}

});
return config;
}

```

You can see that any string code can be easily integrated into the encoding procedure.

In order to test that the code works it is actually enough to just open the database 2 times: as it was mentioned before, internal db4o metadata is stored as strings and therefore opening the database second time will fail if the StringEncoding is not correct.

Reporting

db4o has been tested with several enterprise reporting systems:

Java:

- [Actuate BIRT](#)
- [Elixir Report](#)
- [Jaspersoft JasperReports](#)
- [Jinfonet JReport](#)

More Reading:

- [Reporting With JasperReports](#)
- [Reporting With BIRT](#)

Reporting With JasperReports

[JasperReports](#) is an open-source java-based reporting system. JasperReports can be easily integrated with any Java desktop application and has the ability to deliver rich content onto the screen, to the printer or into PDF, HTML, XLS, CSV and XML files.

More information about JasperReports including tutorial, reference and examples, can be obtained from its [official web-site](#).

This chapter will help you to start using JasperReports with your db4o database.

Before you proceed, you will need to [download](#) JasperReports. Check the full list of system requirements [here](#). For the examples discussed in this chapter you will only need to ensure that:

- you have JRE5 or higher installed;
- you have the following apache jars: commons-digester.jar, commons-collections.jar, commons-logging.jar, commons-beanutils.jar;
- you have iText library version 1.3 or higher.

It is recommended to use [iReport](#) designer with JasperReports for an easy visual report design, however it is not required for this tutorial.

More Reading:

- [Report Structure](#)
- [Data Source](#)
- [Data Preparation](#)
- [Report Design](#)
- [Report Generation](#)
- [ObjectDataSource](#)

Report Structure

Report design is defined in an xml file with a conventional extension *.jrxml. The xml structure is defined in a DTD file and can be downloaded from the [sourceforge](#).

A simplest table report definition can look like this:

```
simple_report.xml
<?xml version="1.0"?>
<!DOCTYPE jasperReport
    PUBLIC "-//JasperReports//DTD Report Design//EN"
    "http://jasperreports.sourceforge.net/dtds/jasperreport.dtd">

<jasperReport name="Simple_Report">
```

```

<field name="Name" class="java.lang.String"/>
<detail>
  <band height="20">
    <textField bookmarkLevel="2">
      <reportElement x="0" y="0" width="100" height="15"/>
      <textFieldExpression class="java.lang.String">$F{Name}
    </textFieldExpression>
  </textField>
  </band>
</detail>
</jasperReport>

```

For a simple example, we will use the following Jasper xml elements:

parameter - represents the definition of a report parameter.

In jrxml file:

```
<parameter name="Title" class="java.lang.String"> </parameter>
```

In *.java file:

```
Map parameters = newHashMap();
parameters.put("Title", "The Pilot Report");
```

parameters map is further passed to the [report filling function](#).

field - represents the definition of a data field that will store values retrieved from the data source of the report.

In *.jrxml file:

```
<field name="Name" class="java.lang.String"/>
```

In *.java file fields are handled by classes implementing [JRDataSource](#) interface.

For the full description of JasperReports xml elements please refer to the [Reference](#).

Data Source

JasperReports can be built on any data having visual representation. In order to make the data "understandable" to the report object, it should be supplied through a JRDataSource interface. The JasperReports package supplies several implementations, which can be used with a RDBMS, xml, csv and object data sources.

In order to represent db4o objects in the most convenient way, we will build a special JRDataSource implementation - ObjectDataSource - using reflection to obtain object field values.

ObjectDataSource will accept data as a list of objects, because this is the way it is returned from a db4o query:

```
List <Pilot> pilots = objectContainer.query(pilotPredicate);
```

```

ObjectDataSource.java: ObjectDataSource
/** */
* ObjectDataSource class is used to extract object field values for the report.
* <br><br>
* usage:<br>
* List pilots = <br>
* ObjectDataSource dataSource = new ObjectDataSource(pilots);<br>
* In the report (*.jrxml) you will need to define fields. For example: <br>
*   <field name="Name" class="java.lang.String"/><br>
* where field name should correspond to your getter method:<br>
* "Name" - for getName()<br>
* "Id" - for getId()<br>
*
*/
public class ObjectDataSource implements JRDataSource {
    private Iterator iterator;
    private Object currentValue;
    public ObjectDataSource(List list) {
        this.iterator = list.iterator();
    }
}

```

ObjectDataSource must implement 2 methods:

```
public boolean next()
```

and

```
public Object getFieldValue(JRField field)
```

The `next()` implementation is very simple: it just moves the current pointer to the next object in the list:

```

ObjectDataSource.java: next
public boolean next() throws JRException {
    currentValue = iterator.hasNext() ? iterator.next() : null;
    return (currentValue != null);
}

```

`getFieldValue` method should return the value for the specified field. The field is defined in [*.jrxml](#) file and is passed to the `JRDataSource` as a `JRField`. In the case of an object list datasource the objective is to correspond field names to the object field values. One of the ways to do this is to correspond the name of the field in the report to the name of the getter method in the object class. For example:

```

<field name="Name" class="java.lang.String"/>

class Pilot
{
    ...

```

```

public String getName() {
    return name;
}
}

```

The method name is "get" + JRField#getName() or "get" + "Name". Knowing the method name, we can invoke it using reflection and obtain the value of the object field:

```

ObjectDataSource.java: getFieldValue
public Object getFieldValue(JRField field) throws JRException {
    Object value = null;
    try {
        // getter method signature is assembled from "get" + field name
        // as specified in the report
        Method fieldAccessor = currentValue.getClass().getMethod("get" + field.getName(), null);
        value = fieldAccessor.invoke(currentValue, null);
    } catch (IllegalAccessException iae) {
        iae.printStackTrace();
    } catch (InvocationTargetException ite) {
        ite.printStackTrace();
    } catch (NoSuchMethodException nsme) {
        nsme.printStackTrace();
    }
    return value;
}

```

The full code of the class can be downloaded from [ObjectDataSource](#).

Data Preparation

Before the report can be generated, we will need to store some data to our database. We will use the following Pilot class:

```

Pilot.java
public class Pilot {
    private String name;

    private int points;

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

    public String getName() {
        return name;
    }

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

    public int getPoints() {

```

```

        return points;
    }

    public boolean equals(Object obj)  {
        if (obj instanceof Pilot)  {
            return (((Pilot) obj).getName().equals(name)  &&
                ((Pilot) obj).getPoints() == points);
        }
        return false;
    }

    public String toString()  {
        return name + "/" + points;
    }

    public int hashCode()  {
        return name.hashCode() + points;
    }
}

```

Pilot class has `name` and `points` fields, which can be obtained through `getName()` and `getPoints()` methods.

Let's store some pilots to the database:

```

JasperReportsExample.java: storePilots
private static void storePilots()  {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database();
    if (container != null)  {
        try  {
            Pilot pilot;
            for (int i = 0; i < OBJECT_COUNT; i++)  {
                pilot = new Pilot("Test Pilot #" + i, i);
                container.store(pilot);
            }
            for (int i = 0; i < OBJECT_COUNT; i++)  {
                pilot = new Pilot("Professional Pilot #" + (i + 10), i + 10);
                container.store(pilot);
            }
            container.commit();
        } catch (Db4oException ex)  {
            System.out.println("Db4o Exception: " + ex.getMessage());
        } catch (Exception ex)  {
            System.out.println("System Exception: " + ex.getMessage());
        } finally  {
            closeDatabase();
        }
    }
}

```

Report Design

Let's create a report representing pilots in a table. The table will have 2 fields: "Name" and "Points". The fields should be defined like this:

```
<field name="Name" class="java.lang.String"/>  
<field name="Points" class="java.lang.Integer"/>
```

The field values - `$F{Name}` and `$F{Points}` - can be used in the table part of the report:

```
<detail>  
  <band height="15">  
    <textField bookmarkLevel="2">  
      <reportElement x="150" y="0" width="175" height="15"/>  
      <box leftBorder="Thin" bottomBorder="Thin" leftPadding="10" rightPadding="10"/>  
      <textElement textAlignment="Left"/>  
      <textFieldExpression class="java.lang.String">$F{Name}</textFieldExpression>  
      <anchorNameExpression>$F{Name} + " (" + $F{Points} + ")"</anchorNameExpression>  
    </textField>  
    <textField isStretchWithOverflow="true">  
      <reportElement positionType="Float" x="325" y="0" width="50" height="15"/>  
      <box leftBorder="Thin" bottomBorder="Thin" rightBorder="Thin" leftPadding="10" rightPadding="10"/>  
      <textElement textAlignment="Right"/>  
      <textFieldExpression class="java.lang.Integer">$F{Points}</textFieldExpression>  
    </textField>  
  </band>  
</detail>
```

The full report design can be downloaded from [the-pilot-report.jrxml](#).

Report Generation

Now, when the [objects](#), [data source class](#) and the [report design](#) are ready, it is very easy to generate the report and save it to pdf, html or display on the screen:

```
JasperReportsExample.java: pilotsReport  
public static void pilotsReport() {  
  
    // obtain a list of objects for the report  
    List<Pilot> pilots = database().query(new Predicate<Pilot>() {
```

```

public boolean match(Pilot pilot)  {
    // each Pilot is included in the result
    return true;
}
});

// pass parameters to the report
Map parameters = new HashMap();
parameters.put("Title", "The Pilot Report");

try {
    // compile report design
    JasperReport jasperReport = JasperCompileManager
        .compileReport("reports/the-pilot-report.jrxml");

    // create an object datasource from the pilots list
    ObjectDataSource dataSource = new ObjectDataSource(pilots);

    // fill the report
    JasperPrint jasperPrint = JasperFillManager.fillReport(
        jasperReport, parameters, dataSource);

    // export result to the *.pdf
    JasperExportManager.exportReportToPdfFile(jasperPrint,
        "reports/the-pilot-report.pdf");

    // or export to *.html
    JasperExportManager.exportReportToHtmlFile(jasperPrint,
        "reports/the-pilot-report.htm");

    // or view it immediately in the Jasper Viewer
    JasperViewer.viewReport(jasperPrint);
} catch (JRException e)  {
    e.printStackTrace();
}
}
}

```

ObjectDataSource

```

ObjectDataSource.java
/**/* Copyright (C) 2007 Versant Inc. http://www.db4o.com */
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Iterator;
import java.util.List;

import net.sf.jasperreports.engine.JRDataSource;
import net.sf.jasperreports.engine.JRException;
import net.sf.jasperreports.engine.JRField;

/** */
* ObjectDataSource class is used to extract object field values for the report.
* <br><br>
* usage:<br>
* List pilots = <br>

```

```

* ObjectDataSource dataSource = new ObjectDataSource(pilots);<br>
* In the report (*.jrxml) you will need to define fields. For example: <br>
*   <field name="Name" class="java.lang.String"/><br>
*   where field name should correspond to your getter method:<br>
*   "Name" - for getName()<br>
*   "Id" - for getId()<br>
*
*/
public class ObjectDataSource implements JRDataSource  {

    private Iterator iterator;

    private Object currentValue;

    public ObjectDataSource(List list)  {
        this.iterator = list.iterator();
    }
    // end ObjectDataSource

    public Object getFieldValue(JRField field) throws JRException  {
        Object value = null;
        try  {
            // getter method signature is assembled from "get" + field name
            // as specified in the report
            Method fieldAccessor = currentValue.getClass().getMethod("get" + field.getName(), null);
            value = fieldAccessor.invoke(currentValue, null);
        } catch (IllegalAccessException iae)  {
            iae.printStackTrace();
        } catch (InvocationTargetException ite)  {
            ite.printStackTrace();
        } catch (NoSuchMethodException nsme)  {
            nsme.printStackTrace();
        }
        return value;
    }
    // end getFieldValue

    public boolean next() throws JRException  {
        currentValue = iterator.hasNext() ? iterator.next() : null;
        return (currentValue != null);
    }
    // end next
}

/p>
```

Reporting With BIRT

BIRT is an Eclipse-based open source reporting system. It can be used for web and desktop applications. BIRT can be integrated with db4o by using script DataSource.

The following topics present the most basic example of db4o-BIRT integration.

Before you proceed you will need to install [BIRT engine](#) and [Eclipse BIRT designer plugin](#).

For more information on the BIRT functionality and java integration API refer to the [BIRT official site](#) and [BIRT Integration Reference](#).

More Reading:

- [Data Preparation](#)
- [Report Design](#)
- [Scripted Data Source](#)
- [Application-Report Integration](#)

Full source code: [ReportsBIRT.zip](#)

Data Preparation

First of all we will need to prepare some data for the report. Let's use a simple Pilot class:

```
Pilot.java
public class Pilot {
    private String name;
    private int points;

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

    public String getName() {
        return name;
    }

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

    public int getPoints() {
        return points;
    }

    public boolean equals(Object obj) {
        if (obj instanceof Pilot) {
            return (((Pilot) obj).getName().equals(name) &&
                   ((Pilot) obj).getPoints() == points);
        }
        return false;
    }

    public String toString() {
        return name + "/" + points;
    }
}
```

```

public int hashCode() {
    return name.hashCode() + points;
}
}

```

The following method should be called to fill up the database with some sample data:

```

Db4oModule.java: storeData
public void storeData() {
    new File(DB_FILE).delete();
    ObjectContainer container = Db4o.openFile(DB_FILE);
    try {
        Pilot pilot = new Pilot("Michael Schumacher", 100);
        container.store(pilot);
        pilot = new Pilot("Rubens Barrichello", 99);
        container.store(pilot);
        pilot = new Pilot("Kimi Raikkonen", 100);
        container.store(pilot);
    } finally {
        container.close();
    }
}

```

For the table representation we will need a list of values:

```

Db4oModule.java: readData
public List readData() {
    ObjectContainer container = Db4o.openFile(DB_FILE);
    List result = new ArrayList();
    try {
        ObjectSet pilots = container.query(Pilot.class);
        while (pilots.hasNext()) {
            Pilot pilot = (Pilot) pilots.next();
            result.add(new String[] { pilot.getName(),
                String.format("%3d", pilot.getPoints()) });
        }
    } finally {
        container.close();
    }
    return (result);
}

```

Report Design

Open Report Design perspective (Window/Open Perspective/Report Design). Create a new report (select the project in the Navigator window, right-click, select New/Report). In the New Report window change the default filename to Pilots.rptdesign and press "Finish".

Report designer layout window opens.

Open Palette view and drag "table" icon onto the report layout. In the "Insert Table" window change number of columns to 2 and click "OK".

On the next step we will need to specify a DataSource for the report. Open Data Explorer view, right-click "Data Sources", select "New Data Source". In order to deal with the object data we will need to use a "Scripted Data Source". This data source does not accept any other parameters, so you can just click "Finish".

Now we can define a data set for our table. Select "Data Sets"/"New Data Set" from the drop-down menu. Change the data set name to "Pilot". In the next window define 2 columns: "Name" and "Points", and press "Finish".

Now we have data that can be used for the layout. Open report layout window again and drag Pilot.Name and Pilot.Points columns from the Data Explorer/Data Sets onto "Detail Row" on the table.

Scripted Data Source

At this point we have created the report layout and we've filled in the data. Now we need to "teach" the report to understand our data source. We can do this using a script.

Open Pilots.rptdesign view and select "Script" tag.

In the Data Explorer window select "Pilot" dataset. Select "open" script from the drop-down menu in the script window. Add the following script:

```
dataClass = new Packages.Db4oModule();

pilots = dataClass.readData();

totalrows = pilots.size();

currentrow = 0;
```

This script will prepare the data when the report is opened. Save the changes.

Select "fetch" script from the drop-down menu. Add the following script:

```
row["Pilot"] = currentrow;

row["Points"] = totalrows;

if (currentrow >= totalrows) {

    return(false);

}

var pilotsRow = pilots.get(currentrow);

var Pilot = pilotsRow[0];

var Points = pilotsRow[1];

row["Pilot"] = Pilot;

row["Points"] = Points;

currentrow = currentrow + 1;
```

```
return(true);
```

This script fetches the data for the rows in the table.

Application-Report Integration

The last step is to integrate the existing report into our java application. This is done with the help of [Report Engine API](#).

Add the libraries from BIRT_HOME/lib to the project. Create the following method in your main application class:

```
ReportMain.java: main
public static void main(String[] args) throws BirtException {
    // create engine configuration
    EngineConfig config = new EngineConfig();
    // specify the path to the BIRT installation
    config.setEngineHome(BIRT_HOME);
    // set the output directory for the logging and set the logging level
    config.setLogConfig(OUTPUT_DIR, Level.FINE);

    // start the Platform to load the plugins
    Platform.startup(config);
    // create a factory, which will supply ReportEngine instance
    IReportEngineFactory factory = (IReportEngineFactory) Platform
        .createFactoryObject(IReportEngineFactory.EXTENSION_REPORT_ENGINE_FACTORY);
    // there is a significant cost associated with creating a new engine
    // instance, therefore the application should use only one engine instance
    // for its reports
    engine = factory.createReportEngine(config);
    engine.changeLogLevel(Level.WARNING);

    IReportRunnable design;
    // Open the report design
    design = engine.openReportDesign(REPORT_DESIGN);
    // Create task to run and render the report
    IRunAndRenderTask task = engine.createRunAndRenderTask(design);

    // Set rendering options - such as file or stream output,
    // output format, whether it is embeddable, etc
    HTMLRenderOption options = new HTMLRenderOption();
    options.setOutputFileName(OUTPUT_DIR + "//" + REPORT_OUTPUT);
    // output format can be either html or pdf
    options.setOutputFormat("html");
    task.setRenderOption(options);

    // run the report
    task.run();
    task.close();

    // shutdown the engine and the platform after use
    // to clean up and unload the extensions
    engine.shutdown();
```

```
    Platform.shutdown();  
}
```

Now you can run the report and check the results in the OUTPUT_DIR folder.

Exceptions

Db4o versions prior to 6.2 were designed to silently "swallow" most of the exceptions. This mode proved to be efficient in the situations, when db4o is supposed to work without interruptions under any circumstances.

However there are many situations when it is vitally important for the user to know what went wrong and to be able to react accordingly. Some of the examples:

- callbacks;
- opening database file or client connection;
- unsupported schema changes;
- invalid ID format;
- corrupted database file, etc

The new approach introduced in the db4o 6.2 allows exceptions to "bubble up" to the user level.

More Reading:

- [Exception Types](#)
- [How To Work With Db4o Exceptions](#)

Exception Types

Using db4o you will have to deal with db4o-specific exceptions and system exceptions thrown directly out of db4o (like OutOfMemory error in Java or System.Exception in .NET).

Db4o-specific exceptions are Unchecked exceptions, which all inherit from a single root class Db4oException.

In Java Unchecked exceptions are inherited from RuntimeExceptions class, while in .NET all exceptions are unchecked.

Db4o exceptions are chained; you can get the cause of the exception using:

Java:

```
db4oException.getCause();
```

In order to see all db4o-specific exceptions you can examine the hierarchy of Db4oException class. Currently the following exceptions are available:

Db4oException - db4o exception wrapper: exceptions occurring during internal processing will be proliferated to the client calling code encapsulated in an exception of this type.

BackupInProgressException - an exception to be thrown when another process is already busy with the backup.

ConstraintViolationException - base class for all constraint exceptions.

UniqueFieldValueConstraintViolationException - an exception to be used to determine constraint violation on commit.

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.

DatabaseReadOnlyException - an exception to be thrown when a write operation was attempted on a database in read-only mode.

GlobalOnlyConfigException - this exception is thrown when a global-only configuration setting is attempted for the local configuration.

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 the password provided to access an encrypted database is not correct.

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.

1. Opening a database file can throw a DatabaseFileLockedException:

```
ExceptionExample.java: openDatabase
private static ObjectContainer openDatabase() {
    ObjectContainer container = null;
    try {
        container = Db4o.openFile(DB4O_FILE_NAME);
    } catch(DatabaseFileLockedException ex) {
        // System.out.println(ex.getMessage());
        // ask the user for a new filename, print
        // or log the exception message
        // and close the application,
        // find and fix the reason
        // and try again
    }
    return container;
}
```

2. Opening a client connection can throw IOException:

```
ExceptionExample.java: openClient
private static ObjectContainer openClient() {
    ObjectContainer container = null;
    try {
        container = Db4o.openClient("host", 0xdb40, "user", "password");
    } catch(Db4oIOException ex) {
        //System.out.println(ex.getMessage());
        // ask the user for new connection details, print
        // or log the exception message
        // and close the application,
        // find and fix the reason
        // and try again
    } catch(OldFormatException ex) {
        // see above
    } catch(InvalidPasswordException ex) {
        // see above
    }
    return container;
}
```

3. Working with db4o and committing a transaction can throw various exceptions; the best practice is to surround your db4o interaction with try-catch block.

```
ExceptionExample.java: work
private static void work() {
    ObjectContainer container = openDatabase();
    try {
        // do some work with db4o
        container.commit();
    } catch (Db4oException ex) {
        // handle exception .
    } catch (RuntimeException ex) {
        // handle exception .
    } finally {
        container.close();
    }
}
```

```
ExceptionExample.vb: Work
Private Shared Sub Work()
    Dim db As IObjectContainer = OpenDatabase()
    Try
        ' do some work with db4o
        db.Commit()
    Catch ex As Db4oException
        ' handle exception .
    Catch ex As Exception
        ' handle exception .
    Finally
        db.Close()
    End Try
End Sub
```

Download example code:

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 1.1
- Java JDK 1.2
- Java JDK 1.4
- Java JDK 5
- .NET 2.0
- .NET 2.0 CompactFramework
- .NET 3.5
- .NET 3.5 CompactFramework
- Mono

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:

- Symbian (PersonalJava)
- Savaje(J2ME CDC)
- Zaurus(Personal Java or Java Personal Profile)
- 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](#)
- [Deployment Instructions](#)

- [Cross-Platform Applications](#)
- [Symbian OS](#)
- [Servlets](#)
- [Xml Import-Export In Java](#)
- [Classloader issues](#)
- [Database For OSGi](#)
- [Android](#)

db4o on Java Platforms

All Java

- root package is com.db4o

JDK1.1

The major limitations of db4o for JDK1.1:

- no support for storing private fields (reflection in JDK 1.1 can only work on public fields)
- no support for JDK collections (java.util.list since JDK1.2)
- no support for weak references

db4o for Java 1.1 also goes without support for Native Query Optimization. NQ optimization uses bytecode optimizer library Bloat (<http://www.cs.purdue.edu/s3/projects/bloat/>), which is not JDK1.1 compatible.

JDK1.2 - JDK 1.3

Java JDKs after version 1.1 are free of JDK1.1 limitations mentioned above. You can also use Native Query Optimization since JDK1.2.

The main limitation of JDK1.2 - 1.3 is the lack of file locking functionality. As it is necessary to lock the database file in use, db4o simulates locking files by using a timer thread that writes access time to the file. This can be quite slow.

JDK 1.4

File locking functionality is available since JDK1.4. For the versions prior to JDK1.4 an expensive database locking simulation is used, which can considerably affect the performance. The file locking simulation can be switched off to improve the performance on versions prior to JDK1.4, though this setting can potentially result in a corrupted database:

```
Db4o.configure().lockDatabaseFile(false)
```

Db4o can bypass the constructors declared for the class using platform-specific mechanisms. (For Java, this option is only available on JREs ≥ 1.4 .) This mode allows reinstantiating objects even when their class doesn't provide a suitable constructor. For more information see [Constructors chapter](#)

If this option is available in the current runtime environment, it will be the default setting.

JDK 1.5

- 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.

Mobile Java editions

Currently db4o runs on J2ME dialects that support reflection, such as Personal Java, J2ME CDC and J2ME PersonalProfile. PersonalJava is closely equivalent to Java 1.1.8 regarding the libraries and features it contains (see JDK1.1 for the list of limitations). Use db4o-7.12-java1.1.jar with PersonalJava.

J2ME CDC and J2ME PersonalProfile are based on subsets of JDK1.3 or JDK1.4 depending on the version used.

J2ME CLDC and MIDP are not yet supported. Their support requires:

- replacing reflection (not available in CLDC) with a build-time preprocessor
- providing RMS RecordStore based I/O

Which db4o Java version to use?

Db4o comes with several jars supporting different java versions. Together with the advanced features of higher Java versions db4o provides valuable improvements to its functionality (see the comparison above).

To get the best functionality you must use the highest db4o java version that your virtual machine supports:

Java versions	Recommended db4o jar
JRE1.1	db4o-7.12-java1.1.jar
JRE1.2, JRE1.3, JRE1.4	db4o-7.12-java1.2.jar
JRE5, JRE6	db4o-7.12-java5.jar

When you use JRE1.4 you should use db4o-x-x-java1.2.jar with database locking simulation disabled:

```
Db4o.configure().lockDatabaseFile(false)
```

This will improve the performance dramatically, because JRE1.4 provides a built-in functionality to lock database files and the costly db4o database locking simulation is not needed.

Security Requirements On Java Platform

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
```

Deployment Instructions

This topic will help you to select correct db4o libraries for deployment with your db4o-based application.

Java deployment

For the basic db4o functionality you only need to deploy db4o jar itself:

Application	Required libraries
Standalone or client/server db4o application for Java 1.1	db4o-7.12-java1.1.jar
Standalone or client/server db4o application for Java 1.2	db4o-7.12-java1.2.jar
Standalone or client/server db4o application for Java 5	db4o-7.12-java5.jar

The following table shows, which features can be added to the basic functionality and the libraries that should be deployed in addition to the basic library to support these features:

Application features	Required libraries
Native Queries optimization (for Java > 1.1).	bloat-1.0.jar
Note, that the application can be NQ optimized in compile-time , db4o-7.12-nqopt.jar in which case NQ optimization libraries are not required to be deployed.	db4o-7.12-taj.jar
Transparent Activation support	bloat.jar
	db4o- 7.12 - instrumentation.jar
OSGI	db4o-7.12-osgi.jar

Cross-Platform Applications

db4o is a native java and .NET database. It reads Java and .NET objects and stores them in a platform independent format. In the runtime the database data is reconstructed into class objects using reflection. However, as the database storage is actually platform-independent, it does not matter if the data will be reconstructed into a java or .NET class object as soon as the class definition matches. Thus the same db4o database can be used both with Java and .NET application.

Further Reading:

- [Configuration](#)
- [Using Aliasing for Cross-Platform Development](#)
- [Limitations Of Db4o Cross-Platform Usage](#)

Configuration

In order to use the same database in Java and .NET application you will need to configure [Aliases](#). This is necessary due to the difference in Java and .NET class name format (it can be also helpful if you do not want to give the same names to the classes in different applications).

First of all you will need to alias the database itself:

Java:

```
Configuration configuration = Db4o.newConfiguration();

configuration.addAlias(
new                  TypeAlias("Db4objects.Db4o.Ext.Db4oDatabase,
Db4objects.Db4o","com.db4o.ext.Db4oDatabase"));
```

Then you will need to alias the persisted classes. If your class names match in Java and .NET, you can use a [WildcardAlias](#), which allows to alias all the classes in a package/namespace at once:

Java:

```
configuration.addAlias(
new                  WildcardAlias("Db4objects.Db4odoc.Aliases.*,
Db4objects.Db4odoc","com.db4odoc.aliases.*"));
```

If you want to alias only specific classes you can use [TypeAlias](#).

Java:

```
configuration.addAlias(
new                  TypeAlias("Db4objects.Db4odoc.Aliases.Pilot,
Db4objects.Db4odoc","com.db4odoc.aliases.Pilot"));
```

Remember that the configuration should be created and supplied to the object container on opening.

Limitations Of Db4o Cross-Platform Usage

db4o cross-platform functionality is work in progress. Currently, it provides the basic features that enable you to use Java database on .NET and vice versa. However, it is recommended to familiarize yourself with the current limitations:

1. Some objects are treated differently in Java and .NET and could not be translated cleanly. This includes:

- Enumerations. In Java enumerations are similar to classes whereas in .NET enumeration is just a primitive type. For more information on how db4o treats both types see [Static Fields And Enums](#)
- Final Fields in Java behave differently in different platforms, which does not correspond to .NET readonly or const members. See [Final Fields](#)
- Collections. Please, keep an eye on [COR-766](#) Jira issue to see when collections cross-platform handling will be fixed.

2. Cross-platform client/server usage (Java server, .NET client or vice versa) is currently out of order ([COR-765](#)).

In general using db4o in cross-platform environment, you must try to keep your persistent class definitions simple and unambiguously interpreted on both platforms. Avoid constructs that exist only on Java or only on .NET platform.

Please, keep an eye on db4o [Jira](#) and [news](#) to stay informed about the latest progress on db4o cross-platform functionality.

Symbian OS

Symbian OS is the global industry standard operating system for smartphones. You can find more information about it at <http://www.symbian.com/>.

UIQ (formerly known as User Interface Quartz) is a software platform based upon Symbian OS. UIQ-based devices support Java thus enabling you to use db4o.

Development Environment

Db4o was tested for compatibility with UIQ 2.1 SDK. You can download the SDK from:

http://developer.sonyericsson.com/site/global/docstools/symbian/p_symbian.jsp The UIQ Symbian SDK allows you to write applications for Symbian OS using your Windows PC and a suitable JDK (JDK 1.1.8). The SDK comes with UIQ emulator, which can be run on a Windows-based computer allowing you to test and debug your application before deployment.

The SDK also installs some third party software, including JRE 1.3, which under some conditions can break Java installations already present ('java.dll not found' error message). In this case you can uninstalling the JRE that comes with Symbian to solve the problem.

The Emulator has its own file system (you can get more information about how it is designed from the SDK documentation).

To be able to run Java applications and use db4o in the emulator, you will have to map _epoc_drive_j to \epoc32\java. Consult your MS Windows documentation on how to set environment variables.

Environment variables can be set locally at the command prompt using the syntax

```
set _epoc_drive_j=\epoc32\java\
```

You can also launch your application on the Emulator from the Windows command prompt:

```
pjava -cd j:\demo DemoApp
```

You have to ensure that the correct version of the emulator VM executable (pjava.exe) is used - the correct path is /runtime/epoc32/release/wins/urel.

Command-line launch also allows you to pass arguments to a class's main(). Please, note that path names given to pjava are paths within the Emulator's drivespace only; they are not Windows paths.

Some platforms will require additional tuning to run the Emulator successfully. The following advices should help, if you are experiencing problems running the emulator:

- use a -cd (change directory) argument for pJava as well as a full -cp, both expressed in terms of the virtual file system.
- add the path to the JDK runtime classes (j:/lib/classes.zip, equivalent to /runtime/epoc32/java/lib/classes.zip) to the -cp argument.

The last thing you need to do is to copy the proper version of db4o jar (JDK1.1) to the emulator file system directory (/runtime/epoc32/java) and add its location to the classpath.

To make the startup process easier we recommend to create a batch file to run your application, which can look like this:

```
REM deploy all db4o files to C:\Symbian\UIQ_21\runtime\epoc32\java  
SET SYMB_HOME=C:\Symbian\UIQ_21  
SET SYMB_EPOC32=%SYMB_HOME%\runtime\epoc32  
SET SYMB_BIN=%SYMB_EPOC32%\release\wins\urel  
SET _epoc_drive_j=%SYMB_EPOC32%\java\  
%SYMB_BIN%\pjava -cd J:\ -cp .;J:\;J:\classes\;J:\db4o-5.0-jav1.1.jar;J:\DemoApp.jar Demo-Class
```

Programming specifics

Tested version of Symbian JDK has problems with IO:

- seek() cannot move beyond the current file length;
- under certain (rare) conditions, calls to RandomAccessFile.length() seems to garble up the following reads.

To workaround these problems and make db4o file operations stable special SymbianIoAdapter is provided for Symbian OS:

```
Db4o.configure().io(new com.db4o.io.SymbianIoAdapter())
```

You can read more about using IOAdapters with db4o in IOAdapter chapter

The following example shows how SymbianIoAdapter can be used:

```
SymbianTest.java
/**/* Copyright (C) 2004 - 2006 Versant Inc. http://www.db4o.com */

package com.db4odoc.f1.symbian;

import java.io.File;
import java.io.IOException;

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

public class SymbianTest {

    public static final String YAPFILENAME = "formula1.yap";

    public static void main(String[] args) throws IOException {
        setObjects();
        setObjectsSymbian();
        getObjects();
        getObjectsSymbian();
    }
    // end main

    public static void setObjects() {
        System.out.println("\nSetting objects using RandomAccessFileAdapter");
        new File(YAPFILENAME).delete();
        Db4o.configure().io(new com.db4o.io.RandomAccessFileAdapter());
        try {
            ObjectContainer db = Db4o.openFile(YAPFILENAME);
            try {
                db.store(new SymbianTest());
            }
        }
    }
}
```

```

        } finally {
            db.close();
        }
    } catch (Exception ex) {
        System.out.println("Exception accessing file: " + ex.getMessage());
    }
}
// end setObjects

public static void setObjectsSymbian() {
    System.out.println("\nSetting objects using SymbianIoAdapter");
    new File(YAPFILENAME).delete();
    Db4o.configure().io(new com.db4o.io.SymbianIoAdapter());
    try {
        ObjectContainer db = Db4o.openFile(YAPFILENAME);
        try {
            db.store(new SymbianTest());
        } finally {
            db.close();
        }
    } catch (Exception ex) {
        System.out.println("Exception accessing file: " + ex.getMessage());
    }
}
// end setObjectsSymbian

public static void getObjects() {
    System.out.println("\nRetrieving objects using RandomAccessFileAdapter");
    Db4o.configure().io(new com.db4o.io.RandomAccessFileAdapter());
    try {
        ObjectContainer db = Db4o.openFile(YAPFILENAME);
        try {
            ObjectSet result=db.queryByExample(new Object());
            System.out.println("Objects in the database: " + result.size());
        } finally {
            db.close();
        }
    } catch (Exception ex) {
        System.out.println("Exception accessing file: " + ex.getMessage());
    }
}
// end getObjects

public static void getObjectsSymbian() {
    System.out.println("\nRetrieving objects using SymbianIoAdapter");
    Db4o.configure().io(new com.db4o.io.SymbianIoAdapter());
    try {
        ObjectContainer db = Db4o.openFile(YAPFILENAME);
        try {
            ObjectSet result=db.queryByExample(new Object());
            System.out.println("Objects in the database: " + result.size());
        } finally {
            db.close();
        }
    }
}

```

```

        } catch (Exception ex) {
            System.out.println("Exception accessing file: " + ex.getMessage());
        }
    }
    // end getObjectsSymbian
}

```

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 a db4o point of view. One is the more complex classloader environment - db4o needs to know itself (of course) 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 server correctly. This can be done at the servlet API layer or within the web application framework you are using.

On the servlet API layer, you could bind db4o server handling to the servlet context via an appropriate listener. A very basic sketch might look like this:

```

Db4oServletContextListener.java
/**/* Copyright (C) 2004 - 2006 Versant Inc. http://www.db4o.com */

package com.db4odoc.servlets;

import com.db4o.Db4o;
import com.db4o.ObjectServer;
import javax.servlet.*;

public class Db4oServletContextListener implements ServletContextListener {
    public static final String KEY_DB4O_FILE_NAME = "db4oFileName";

    public static final String KEY_DB4O_SERVER = "db4oServer";

    private ObjectServer server = null;

    public void contextInitialized(ServletContextEvent event) {
        close();
        ServletContext context = event.getServletContext();
        String filePath = context.getRealPath("WEB-INF/db/"
            + context.getInitParameter(KEY_DB4O_FILE_NAME));
        server = Db4o.openServer(filePath, 0);
        context.setAttribute(KEY_DB4O_SERVER, server);
        context.log("db4o startup on " + filePath);
    }

    public void contextDestroyed(ServletContextEvent event) {
        ServletContext context = event.getServletContext();
        context.removeAttribute(KEY_DB4O_SERVER);
    }
}

```

```

        close();
        context.log("db4o shutdown");
    }

    private void close() {
        if (server != null) {
            server.close();
        }
        server = null;
    }
}

```

This listener just has to be registered in the web.xml.

```

web.xml
<context-param>
    <param-name>db4oFileName</param-name>
    <param-value>db4oweb.yap</param-value>
</context-param>
<listener>
    <listener-class>
        com.db4odoc.servlets.Db4oServletContextListener
    </listener-class>
</listener>

```

Now db4o should be available to your application classes.

```
ObjectServer server=(ObjectServer)context.getAttribute("db4oServer")
```

A more complex and 'old school' example without using context listeners comes with the samples section of the db4o3 distribution that's still available from our web site.

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.)

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(XMLFILE_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

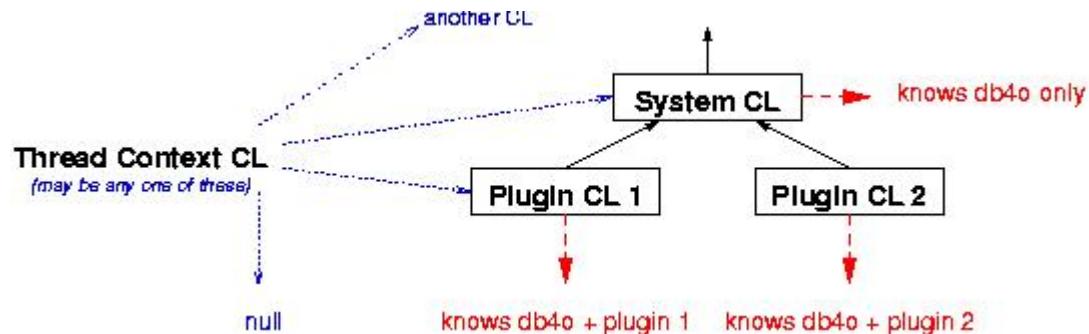
Db4o needs to know its own classes, of course, and it needs to know the class definitions of the objects it stores. (In Client/Server mode, both the server and the clients need access to the class definitions.) While this usually is a non-issue with self-contained standalone applications, it can become 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: `Db4o.configure().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.getClass().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 environment. Its usage is the usage of an OSGI service, which is well documented in the Internet. Short

, 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 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.jar can be found in /lib folder of the Java distribution with detailed API documentation in /doc/osgi folder.

If you are comfortable with OSGI and only need a short introduction to db4o osgi service you can read [Db4o-Osgi Usage](#).

For a more detailed explanation and example of db4o-service usage in an Eclipse plug-in see [Eclipse Plug-In With Db4o Service](#). This example will be also helpful for those who are new to OSGI.

Db4o-Osgi Usage

db4o-osgi service can be accessed like any other OSGI 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 and plug-in development.

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-6.3 for java ([download](#))
- eclipse IDE v.3.3. ([download](#)). Please, note that db4o-osgi uses Bundle-ActivationPolicy:lazy header to define the lazy bundle loading policy; this setting is only available since Eclipse v 3.3.

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 Db4o-Provider to access and store the data.

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

```
Db4oProvider.java
package memopugin;

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 memopugin.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;
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);
}
```

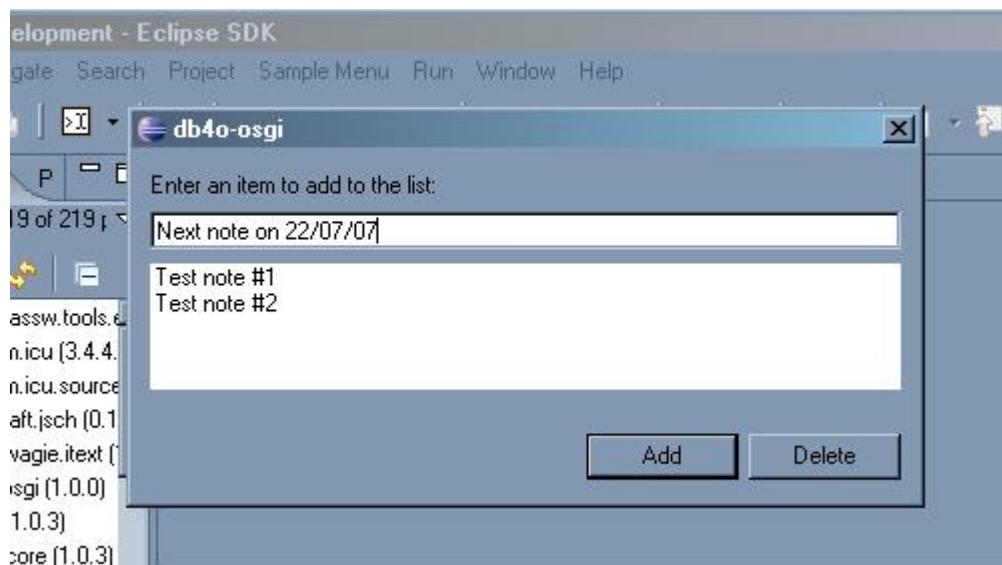
```
Activator.java: stop
/**
 * (non-Javadoc)
 * @see org.eclipse.ui.plugin.AbstractUIPlugin#stop(BundleContext)
 * Unregisters the db4o_osgi service from Db4oProvider
 */
public void stop(BundleContext context) throws Exception {
    Db4oProvider.UnInitialize();
    plugin = null;
    super.stop(context);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

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.

Android

[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, however some effort was added to create some convenience methods:

```
public      int      delete(String      table,      String      whereClause,
String[] whereArgs)

public      long     insert(String      table,      String      nullColumnHack,
ContentValues values)

public Cursor query(boolean distinct, String table, String[] columns, String selection,
String[] selectionArgs,
String groupBy, String having, String orderBy)

public      long     replace(String      table,      String      nullColumnHack,
ContentValues initialValues)
```

This may look better than SQL, but if you look closer you can see that all column names and selection criteria are specified as strings, so we still stay with a problem of run-time checking instead of compile-time.

Luckily even for this very early Android release we already have an alternative - db4o. Yes, db4o runs on Android out of the box and produces very competitive results as well. The following application compares db4o and SQLite usage for basic operations. It also contains some operation duration calculations that can be used to compare db4o vs SQLite performance. Please, note that these results are for an overview purpose only and there are many configuration settings that can change performance of both databases. Also running in emulator does not guarantee the same results as in real device..

More Reading:

- [General Info](#)
- [Application Structure](#)
- [Opening A Database](#)
- [Storing Data](#)
- [Retrieving Data](#)
- [Changing Data](#)
- [Deleting Data](#)

- [Backup](#)
- [Closing A Database](#)
- [Schema Evolution](#)
- [Car](#)
- [Pilot](#)

Step by Step installation

The steps to install db4o on Android are quite simple thanks to Eclipse.

1. Point your browser to <http://developer.db4o.com/files/default.aspx>. Here you have to choose stable, production or development release (I recommend the current production version) and then download the Java version of db4o.
2. Copy ***db4o-xxx-java5.jar*** available in the lib folder of your db4o installation to a lib folder in your Android project root directory.
3. Refresh the Eclipse project folders, click on lib, right-click on ***db4o-xxx-java5.jar*** and select "Add to build path".
4. You're done! You can now use db4o in your Android application and it will be deployed automatically when running the Android emulator.

Things to remember when using db4o under Andoid

1. Use the Java 5 version of the db4o library available on the lib folder of your db4o installation (***db4o-xxx-java5.jar***). The other db4o Java versions can be less performant.
2. Open the database relative to your parent activity (context) data directory:
`Db4o.openFile(dbConfig(), db4oDBFullPath(context));`

```
private String db4oDBFullPath(Context ctx) {
    return ctx.getDataDir() + "/" + "dbfile.yap";
}
```

Otherwise Android security will prevent the creation of the database file.

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.

Application Structure

The presented application consists of simple screen with several buttons, which trigger database operations. The operations are performed on db4o and SQLite database serially and the results with the calculated time of execution are written on the screen. All the database access logic is encapsulated in Db4oExample and SqlExample classes accordingly.

The data stored to the databases is represented by [Car](#) and [Pilot](#) classes.

It is recommended to install the application and test described functionality while reading along.

Opening A Database

The database is opened with "Open DB" button.

SQLite:

```
SqlExample.java: database
public static SQLiteDatabase database() {
    long startTime = 0;
    try {
        _db = _context.openDatabase(DATABASE_NAME, null);
    } catch (FileNotFoundException e) {
        try {
            _db =
                _context.createDatabase(DATABASE_NAME, DATABASE_VERSION, 0,
                                       null);
            _db.execSQL("create table " + 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 " + 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 CAR_PILOT ON " + DB_TABLE_CAR + " (pilot);");
        } catch (FileNotFoundException e1) {
            _db = null;
        }
    }
    logToConsole(startTime, "Database opened: ", false);
    return _db;
}
```

db4o:

```
Db4oExample.java: database
public static ObjectContainer database() {
    long startTime = 0;
    try {
        if(_container == null) {
            startTime = System.currentTimeMillis();
            _container = Db4o.openFile(configure(), db4oDBFullPath());
        }
    } catch (Exception e) {
        Log.e(Db4oExample.class.getName(), e.toString());
        return null;
    }
    logToConsole(startTime, "Database opened: ", false);
    return _container;
}
```

```

Db4oExample.java: configure
private static Configuration configure() {
    Configuration configuration = Db4o.newConfiguration();
    configuration.objectClass(Car.class).objectField("pilot").indexed(true);
    configuration.objectClass(Pilot.class).objectField("points").indexed(true);
    configuration.lockDatabaseFile(false);

    return configuration;
}

```

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

"Store" button creates and stores 100 of car objects (each including a reference to Pilot object) to each database.

SQLite:

```

SqlExample.java: fillUpDB
public static void fillUpDB() throws Exception {
    close();
    _context.deleteDatabase(DATABASE_NAME);
    SQLiteDatabase db = database();
    if (db != null) {
        long startTime = System.currentTimeMillis();
        for (int i=0; i<100;i++) {
            addCar(db,i);
        }
        logToConsole(startTime, "Stored 100 objects: ", false);
        startTime = System.currentTimeMillis();
    }
}

```

```

SqlExample.java: addCar
private static void addCar(SQLiteDatabase db, int number)
{
    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);
}

```

db4o:

```
Db4oExample.java: fillUpDB
public static void fillUpDB() throws Exception {
    close();
    new File(db4oDBFullPath()).delete();
    ObjectContainer container=database();
    if (container != null) {
        long startTime = System.currentTimeMillis();
        for (int i=0; i<100;i++) {
            addCar(container,i);
        }
        logToConsole(startTime, "Stored 100 objects: ", false);
        startTime = System.currentTimeMillis();
        container.commit();
        logToConsole(startTime, "Committed: ", true);
    }
}
```

```
Db4oExample.java: addCar
private static void addCar(ObjectContainer container, int points)
{
    Car car = new Car("BMW");
    car.setPilot(new Pilot("Tester", points));
    container.store(car);
}
```

You can see that db4o handles adding objects to the database in a much more elegant way - #set(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 9 points:

SQLite:

```
SqlExample.java: selectCar
public static void selectCar() {
    SQLiteDatabase db = database();
    if (db != null) {
        long startTime = System.currentTimeMillis();
        Cursor c =
```

```

        db.query("select c.model, p.name, " +
"p.points from car c, pilot p where c.pilot = p.id and p.points = 9;", null);
        if (c.count() == 0) {
            logToConsole(0,
"Car not found, refill the database to continue.", false);
            return;
        }
        c.first();
        Pilot pilot = new Pilot();
        pilot.setName(c.getString(1));
        pilot.setPoints(c.getInt(2));

        Car car = new Car();
        car.setModel(c.getString(0));
        car.setPilot(pilot);
        logToConsole(startTime, "Selected Car (" + car + "): ", false);
    }
}

```

db4o:

(Using SODA query)

```

Db4oExample.java: selectCar
public static void selectCar() {
    ObjectContainer container = database();
    if (container != null) {
        Query query = container.query();
        query.constrain(Car.class);
        query.descend("pilot").descend("points").constrain(new Integer(9));

        long startTime = System.currentTimeMillis();
        ObjectSet result = query.execute();
        if (result.size() == 0) {
            logToConsole(0, "Car not found, refill the database to continue.", false);
        } else {
            logToConsole(startTime, "Selected Car (" + result.next() + "): ", false);
        }
    }
}

```

Of course SODA query is not the best db4o querying mechanism: the preferred mechanism - Native Queries - will be reviewed in the following chapters. However, SODA is the closest to SQL and can serve a good comparison. 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:

```

SqlExample.java: updateCar
public static void updateCar() {
    SQLiteDatabase db = database();
    if (db != null) {
        long startTime = System.currentTimeMillis();
        // insert a new pilot
        ContentValues updateValues = new ContentValues();

        updateValues.put("id", 101);
        updateValues.put("name", "Tester1");
        updateValues.put("points", 25);
        db.insert(DB_TABLE_PILOT, null, updateValues);

        updateValues = new ContentValues();

        // update pilot in the car
        updateValues.put("pilot", 101);
        int count = db.update(DB_TABLE_CAR, updateValues,
"pilot in (select id from pilot where points = 15)", null);
        if (count == 0) {
            logToConsole(0, "Car not found, refill the database to continue.", false);
        } else {
            logToConsole(startTime, "Updated selected object: ", false);
        }
    }
}

```

db4o:

(Select Car using Native Query)

```

Db4oExample.java: updateCar
public static void updateCar() {
    ObjectContainer container=database();
    if (container != null) {
        try {
            long startTime = System.currentTimeMillis();
            ObjectSet result = container.query(new Predicate() {
                public boolean match(Object object) {
                    if (object instanceof Car) {
                        return ((Car)object).getPilot().getPoints() == 15;
                    }
                    return false;
                }
            });
            Car car = (Car)result.next();
            car.setPilot(new Pilot("Tester1", 25));
            container.store(car);
            logToConsole(startTime, "Updated selected object: ", false);
        } catch (Exception e) {
            logToConsole(0, "Car not found, refill the database to continue.", false);
        }
    }
}

```

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:

```
SqlExample.java: deleteCar
public static void deleteCar() {
    SQLiteDatabase db = database();
    if (db != null) {
        long startTime = System.currentTimeMillis();
        int count = db.delete(DB_TABLE_CAR,
"pilot in (select id from pilot where points = 5)", null);
        if (count == 0) {
            logToConsole(0,
"Car not found, refill the database to continue.", false);
        } else {
            logToConsole(startTime, "Deleted selected object: ", false);
        }
    }
}
```

db4o:

(Select Car using Native Query)

```
Db4oExample.java: deleteCar
public static void deleteCar() {
```

```

ObjectContainer container=database();
if (container != null) {
    try {
        long startTime = System.currentTimeMillis();
        ObjectSet result = container.query(new Predicate() {
            public boolean match(Object object) {
                if (object instanceof Car) {
                    return ((Car) object).getPilot().getPoints() == 5;
                }
                return false;
            }
        });
        Car car = (Car)result.next();
        container.delete(car);
        logToConsole(startTime, "Deleted selected object: ", false);
    } catch (Exception e) {
        logToConsole(0, "Car not found, refill the database to continue.", false);
    }
}
}
}

```

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

"Backup" button shows database backup ability.

On db4o a ExtObjectContainer#backup call is used to backup a database in use.

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.

Closing A Database

The following methods will close SQLite and db4o database accordingly:

SQLite:

```
SqlExample.java: close
/***
 * Close database connection
 */
public static void close()  {
    if(_db != null) {
        long startTime = System.currentTimeMillis();
        _db.close();
        logToConsole(startTime, "Database committed and closed: ", false);
        _db = null;
    }
}
```

db4o:

```
Db4oExample.java: close
/**
 * Close database connection
 */
public static void close()  {
    if(_container != null) {
        long startTime = System.currentTimeMillis();
        _container.close();
        logToConsole(startTime, "Database committed and closed: ", false);
        _container = null;
    }
}
```

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:

```
RegistrationRecord.java
package com.db4odoc.android.compare.refactored;

import java.util.Date;

public class RegistrationRecord {
    private String number;
    private Date year;

    public RegistrationRecord(String number, Date year) {
        this.number = number;
        this.year = year;
    }

    public String getNumber() {
        return number;
    }

    public void setNumber(String number) {
        this.number = number;
    }

    public Date getYear() {
        return year;
    }

    public void setYear(Date year) {
        this.year = year;
    }
}
```

Now we will need to modify Car class to attach the record:

```
Car.java
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */
package com.db4odoc.android.compare.refactored;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;
```

```

public class Car {
    private String model;
    private Pilot pilot;
    private RegistrationRecord registration;

    public RegistrationRecord getRegistration() {
        return registration;
    }

    public void setRegistration(RegistrationRecord registration) {
        this.registration = registration;
    }

    public Car() {

    }

    public Car(String model) {
        this.model=model;
        this.pilot=null;
    }

    public Pilot getPilot() {
        return pilot;
    }

    public void setPilot(Pilot pilot) {
        this.pilot = pilot;
    }

    public String getModel() {
        return model;
    }

    public String toString() {
        if (registration == null) {
            return model + "[+pilot+]";
        } else {
            DateFormat df = new SimpleDateFormat("d/M/yyyy");
            return model + ": " + df.format(registration.getYear());
        }
    }

    public void setModel(String model) {
        this.model = model;
    }
}

```

Ok, the application is changed to cater for 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:

```

Db4oExample.java: selectCarAndUpdate
public static void selectCarAndUpdate() {
    ObjectContainer container = database();
    if (container != null) {
        Query query = container.query();
        query.constrain(Car.class);
        query.descend("pilot").descend("points").constrain(new Integer(15));

        long startTime = System.currentTimeMillis();
        ObjectSet<Car> result = query.execute();
        result.reset();
        if (!result.hasNext()) {
            logToConsole(0, "Car not found, refill the database to continue.", false);
        } else {
            Car car = (Car) result.next();
            logToConsole(startTime, "Selected Car (" + car + "): ", false);
            startTime = System.currentTimeMillis();
            car.setRegistration(new RegistrationRecord("A1",
new Date(System.currentTimeMillis())));
            logToConsole(startTime, "Updated Car (" + car + "): ", true);
        }
    }
}

```

Schema Evolution in SQLite

For SQLite database model should be synchronized with the object model:

```

SqlExample.java: upgradeDatabase
public static void upgradeDatabase(SQLiteDatabase db) {
    db.execSQL("create table REG_RECORDS (
        + "id text primary key," + "year date);");
    db.execSQL("CREATE INDEX IDX_REG_RECORDS ON REG_RECORDS (id);");
    db.execSQL("alter table " + DB_TABLE_CAR + " add reg_record text;");

}

```

Now we can try to retrieve and update records:

```

SqlExample.java: selectCarAndUpdate
public static void selectCarAndUpdate() {
    SQLiteDatabase db = database();
    if (db != null) {
        long startTime = System.currentTimeMillis();

        db.execSQL("insert into REG_RECORDS (id,year) values ('A1', DATETIME('NOW'))");
        ContentValues updateValues = new ContentValues();

        // update car
        updateValues.put("reg_record", "A1");
        int count = db.update(DB_TABLE_CAR, updateValues, "pilot in (select id from pilot where p
        if (count == 0) {
            logToConsole(0, "Car not found, refill the database to continue.", false);
        } else {
    }
}

```

```

        Cursor c =
            db.query("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.count() == 0) {
            logToConsole(0, "Car not found, refill the database to continue.", false);
            return;
        }
        c.moveToFirst();
        String date = c.getString(2);
        SimpleDateFormat sf = new SimpleDateFormat("yyyy-MM-dd H:mm:ss");
        try {
            Date dt = sf.parse(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);
        } catch (ParseException e) {
            Log.e(Db4oExample.class.getName(), e.toString());
        }
    }
}

```

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](#).

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);
}

```

```
}
```

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);
}
}
```

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](#)
- [Configuration](#)
- [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 io-adapter:

Java:

```
configuration.io(new MemoryIoAdapter());
```

The following test can be performed to compare performance of a hard drive and a RAM drive:

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

    configureRamDrive();

    initForHardDriveTest();
    clean();
    System.out.println("Storing " + _count + " objects of depth "
+ _depth + " on a hard drive:");
    open();
    store();
    close();

    initForRamDriveTest();
    clean();
    System.out.println("Storing " + _count + " objects of depth "
+ _depth + " on a RAM disk:");
    open();
    store();
    close();

}
```

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

```
InsertPerformanceBenchmark.java: initForHardDriveTest
private void initForHardDriveTest() {
    _count = 30000;
    _depth = 3;
    _filePath = "performance.db4o";
    _isClientServer = false;

}
```

```
InsertPerformanceBenchmark.java: initForRamDriveTest
private void initForRamDriveTest() {
    _count = 30000;
    _depth = 3;
    _filePath = "r:\\performance.db4o";
    _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 RAM driver was downloaded [here](#) and installed on R:\\ drive.

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

Java:

Storing 100000 objects of depth 3 on a hard drive:

Store 300000 objects: 11912ms

Storing 100000 objects of depth 3 on a RAM disk:

Store 300000 objects: 9351ms

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 runClientServer() {

    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 = 10000;
    _depth = 3;
    _isClientServer = false;

}
```

```
InsertPerformanceBenchmark.java: initForClientServer
private void initForClientServer() {
    _count = 10000;
    _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:

Java:

```
embeddedConfiguration.file().storage(new MemoryStorage());
```

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

Java:

```
configuration.lockDatabaseFile(false);
```

[weakReferences](#)- switching weak references off during insert operation releases extra resources and removed the cleanup thread.

Java:

```
configuration.weakReferences(false);
```

[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.

Java:

```
config.file().storage(new NonFlushingStorage(new FileStorage()));
```

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 = 30000;
    _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 + "] via network:");
    deleteAny(objectsToDelete);
    close();
}

```

```
+ " 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 = 10000;
    _depth = 1;
    _isClientServer = false;

}
```

```
DeletePerformanceBenchmark.java: init
private void init() {
    _count = 10000;
    _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 = i;
            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 f
    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";
    _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 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();
}

// end runDifferentQueriesTest

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();
}

// 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]
    + ", " + 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();

}

// 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), 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);

    //
    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 NQ: 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").constrain("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());
    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/l: 464ms

Storing 100000 of objects of depth 3

Store 300000 objects: 21338ms

Select 1 object SODA: level1/l: 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
```

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 (nano-second) 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.

Configuration

db4o provides a wide range of configuration methods to request special behaviour.

For a complete list of all available methods see the following interfaces:

- [Common Configuration](#)
- [Embedded Configuration](#)
- [Client Configuration](#)
- [Server Configuration](#)
- [ObjectClass](#)
- [ObjectField](#)
- [Cache Configuration](#)
- [File Configuration](#)
- [Configuration Item](#)

The following paragraphs contain some useful hints around using configuration calls.

Working With Configuration

Configuration can be obtained for embedded and client-server modes:

Java:

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
ServerConfiguration serverConfiguration = Db4oClientServer.newServerConfiguration();
ClientConfiguration clientConfiguration = Db4oClientServer.newClientConfiguration();
```

To apply the configuration to an ObjectContainer/Client/Server use one of the following methods:

Java:

```
Db4oEmbedded.openFile(configuration, databaseFileName)
Db4oClientServer.openServer(serverConfiguration, databaseFileName, port)
Db4oClientServer.openClient(clientConfiguration, hostName, port, user, password)
```

Note, that the same configuration object can't be re-used in more than one object container. If you need to open several object containers with the same set of configuration settings, it is recommended to create a method, which will return a new configuration instance on request:

Java:

```
private Configuration configure(){
    EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
    config.callbacks(false);
    ...
    return config;
}
```

Configuration settings are **not** stored in db4o database files. Accordingly you will need to pass the same configuration set **every time** you open an ObjectContainer/Client/Server. 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.

Client/Server specific configuration calls reside in ClientConfiguration and ServerConfiguration interfaces (.NET conventional name IClientConfiguration and IServerConfiguration).

For db4o versions before 6.0 the above-mentioned method of working with the configuration was not available. Instead, you could use:

- global db4o configuration context;
- object container instance configuration.

These methods are still available, however their usage is not advised due to their limitations. Using new configuration methods with the old ones can be harmful.

Global Configuration

Global configuration can be set using:

Java:

```
Db4o.configure()
```

Starting from version 6.0 you can obtain a clone of the global configuration:

Java:

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

When an ObjectContainer/ObjectServer is opened, the global configuration context is cloned and copied into the newly opened ObjectContainer/ObjectServer. Subsequent calls against the global context with Db4o.configure() have no effect on open ObjectContainers/ObjectServers.

Global configuration has a number of disadvantages:

- the settings apply to all the object containers in the current VM/runtime
- the settings can't be reset.

Object Container Configuration

Object Container instance configuration can be used for an open ObjectContainer/Client/Server:

Java:

```
objectContainer.ext().configure() objectServer.ext().configure()
```

Closing the container will erase all the settings.

The obvious disadvantages are:

- configuration can not be created before the container is opened;
- some configuration methods effect the way the system works on opening an object container therefore have no influence on the open object container;
- some configuration methods only effect the creation of a database;
- configuration settings will need to be repeated as many times as the new container gets opened.

Further Reading

Some configuration switches are discussed in more detail in the following chapters:

- [Performance Hints](#)
- [Indexing](#)
- [Encryption](#)

Embedded Configuration

EmbeddedConfiguration interface contains configuration methods specific to db4o used in Embedded (standalone) mode. EmbeddedConfiguration implementation provides access to all common configuration methods through common() method (Java) or Common property (.NET).

Java:

```
public CacheConfiguration cache();
```

Returns cache configuration.

Java:

```
public FileConfiguration file();
```

Returns file configuration, giving access to file-related configuration methods.

Server Configuration

Server configuration interface provides access to the server configuration. This interface allows access to all common configuration methods through common() method (Java) or Common property (.NET). In addition to common configuration the following methods are available:

Java:

```
public CacheConfiguration cache();
```

Returns cache configuration.

Java:

```
public FileConfiguration file();
```

Returns file configuration, giving access to file-related configuration methods.

Client Configuration

Client configuration methods are defined in ClientConfiguration/IClientConfiguration interface. ClientConfiguration extends CommonConfiguration/ICommonConfiguration interface, which means that all [Common configuration](#) methods are available.

Java:

```
public MessageSender messageSender();
```

returns the message sender for this configuration.

Java:

```
void prefetchIDCount(int prefetchIDCount);
```

Sets the number of IDs to be prefetched from the server for new objects created on the client. For more information see [Prefetching IDs For New Objects](#)

Java:

```
void prefetchObjectCount(int prefetchObjectCount);
```

Sets the number of objects to be prefetched for an ObjectSet in client/server mode. For more information see [Prefetching Objects For Query Results](#).

Common Configuration

Db4o configuration resides in com.db4o.config package/Db4object.Db4o.Config namespace.

Below is a short description of all common configuration methods residing in CommonConfiguration interface. For embedded configuration specific methods see [Embedded Configuration](#), for client/server specific settings see [Client Configuration](#) and [Server Configuration](#) interfaces. For class and field specific configuration see [ObjectClass Configuration](#) and [ObjectField Configuration](#) interface.

Java:

```
public void activationDepth(int depth);  
public int activationDepth();
```

sets global activation depth to the specified value or gets its current value.

Java:

```
public void add(ConfigurationItem item);
```

adds a configuration item to be applied when an object container is opened. Examples of configuration items: TransparentActivationSupport, BigMathSupport etc.

Java:

```
public void addAlias(Alias alias);  
public void removeAlias(Alias alias);
```

these methods, as the names suggest, add or remove aliases. Aliases should be configured before opening a database file or connecting to a server. For more information on aliases please check [Aliases](#).

Java:

```
public void allowVersionUpdates(boolean flag);
```

Turns automatic database file format version updates on. For more information see [Updating Db4o File Format](#).

Java:

```
public void automaticShutDown(boolean flag);
```

turns automatic shutdown of the engine on and off.

Depending on the JDK, db4o uses one of the following two methods to shut down, if no more references to the ObjectContainer are being held or the JVM terminates:

- JDK 1.3 and above:

```
Runtime.addShutdownHook()
```

- JDK 1.2 and below:

```
System.runFinalizersOnExit(true) and code in the finalizer
```

Some JVMs have severe problems with both methods. For these rare cases the autoShutDown feature may be turned off.

The default and recommended setting is `true`.

Java:

```
public void bTreeNodeSize(int size);
```

configures the size of BTree nodes in indexes.

Java:

```
public void callbacks(boolean flag);
```

turns [object callbacks](#) on and off. [No callbacks](#) article explains how this setting can be useful for tuning.

Java:

```
public void callConstructors(boolean flag);
```

advises db4o to try instantiating objects with/without calling constructors.

Not all JDKs / .NET-environments support this feature. db4o will attempt, to follow the setting as good as the environment supports. In doing so, it may call implementation-specific features like

`sun.reflect.ReflectionFactory#newConstructorForSerialization` on the Sun Java 1.4.x/5 VM (not available on other VMs) and `FormatterServices.GetUninitializedObject()` on the .NET framework (not available on CompactFramework).

This setting may also be overridden for individual classes in `ObjectClass#callConstructor(boolean)`.

The default setting depends on the features supported by your current environment.

For more information see [Object Construction](#).

Java:

```
public void detectSchemaChanges(boolean flag);
```

tuning feature: configures whether db4o checks all persistent classes upon system startup for added or removed fields. For more information see [No schema changes](#).

Java:

```
public DiagnosticConfiguration diagnostic();
```

returns the configuration interface for diagnostics. See [Diagnostics](#) for more information.

Java:

```
public void exceptionsOnNotStorable(boolean flag);
```

configures whether Exceptions are to be thrown, if objects can not be stored. For more information see [ExceptionsOnNotStorable](#). The default for this setting is true, which means that an exception will be thrown if an object can't be stored fully to the database. In order to prevent the exception and make sure that the object is correctly stored to the database, a custom [Typehandler](#) can be written, which will make sure that only relevant information is persisted.

Java:

```
public void internStrings(boolean doIntern);
```

Configures db4o to call `intern()` on strings upon retrieval. For more information see Java/.NET documentation for string class.

Java:

```
public void messageLevel(int level);
```

Sets the detail level of db4o messages (from 1 to 3). For more information see [Debugging db4o](#)

Java:

```
public ObjectClass objectClass(Object clazz);
```

returns an ObjectClass/IObjectClass to configure the specified class.

The clazz parameter can be any of the following:

- a fully qualified classname as a string.
- a Class/Type object.
- any other object to be used as a template.

Java:

```
public void optimizeNativeQueries(boolean optimizeNQ);  
public boolean optimizeNativeQueries();
```

If set to true, db4o will try to optimize native queries dynamically at query execution time, otherwise it will run native queries in unoptimized mode as SODA evaluations. For more information see [Native Query Optimization](#).

Java:

```
public boolean outStream(PrintStream outStream);
```

assigns a stream, where db4o is to print its debug messages. For more information see [Customizing The Debug Message Output](#)

Java:

```
public QueryConfiguration queries();
```

returns the Query configuration interface. For more information see [Query Modes](#).

Java:

```
public void reflectWith(Reflector reflector);
```

Configures the use of a specially designed reflection implementation.

On platforms that do not support reflection, customized reflection implementations may be written to enable db4o. For more information see [Db4o Reflection API](#).

Java:

```
public void registerTypeHandler(TypeHandlerPredicate predicate, TypeHandler4 typehandler);
```

Registers a special typehandler for custom marshalling and comparison. Predicate parameter is used to filter out classes, to which the typehandler is applicable.

Java:

```
public void stringEncoding(StringEncoding encoding);
```

Registers a special string encoding for this database. The setting should be used before the first opening and it should not be changed on existing database. Encodings provided by db4o (utf8 and Unicode) will be remembered when the database is opened next time. Custom encodings (implemented by user) should be supplied into the configuration each time the database is opened.

Java:

```
public void testConstructors(boolean flag);
```

tuning feature: configures whether db4o should try to instantiate one instance of each persistent class on system startup. For more information see [No test instances](#)

Java:

```
public void updateDepth(int depth);
```

specifies the global updateDepth. For more information see [Update Depth](#)

Java:

```
public void weakReferences(boolean flag);
```

turns weak reference management on or off. For more information see [Turning Off Weak References](#).

Java:

```
public void weakReferenceCollectionInterval(int milliseconds);
```

configures the timer for WeakReference collection.

The default setting is 1000 milliseconds.

Configure this setting to zero to turn WeakReference collection off. For more information see [Weak References](#)

ObjectClass Configuration

ObjectClass provides an interface for class configuration. ObjectClass access is implemented in common configuration, thus it is accessible in embedded and client/server mode, both on the client and on the server.

Java:

```
ObjectClass oc = configuration.objectClass(clazz);
```

clazz parameter here can be one of the following:

- a fully qualified classname as a String.
- a Class object.
- any other object to be used as a template.

Java:

```
public void callConstructor(boolean flag);
```

advises db4o to try instantiating objects of this class with/without calling constructors. For more information see [Object Construction](#).

Java:

```
public void cascadeOnActivate(boolean flag);
```

sets cascaded activation behavior for this class. For more information see [Activation](#).

Java:

```
public void cascadeonDelete(boolean flag);
```

sets cascaded delete behavior for this class. For more information see [Delete Behavior](#).

Java:

```
public void cascadeOnUpdate(boolean flag);
```

sets cascaded update behavior for this class. For more information see [Update Depth](#).

Java:

```
public void compare(ObjectAttribute attributeProvider);
```

registers an attribute provider for special query behavior.

The query processor will compare the object returned by the attribute provider instead of the actual object, both for the constraint and the candidate persistent object. For more information see [Custom Query Comparator](#).

Java:

```
public void enableReplication(boolean setting);
```

enable replication of the specified class. Must be called before databases are created or opened so that db4o will control versions and generate UUIDs for objects of this class, which is required for using replication. For more information see [db4o Replication System \(dRS\)](#)

Java:

```
public void generateUUIDs(boolean setting);
```

generate UUIDs for stored objects of this class. For more information see [Unique Universal IDs](#).

Java:

```
public void generateVersionNumbers(boolean setting);
```

generate version numbers for stored objects of this class. For more information see [db4o Replication System \(dRS\)](#)

Java:

```
public void indexed(boolean flag);
```

turns the class index on or off.

db4o maintains an index for each class to be able to deliver all instances of a class in a query. If the class index is never needed, it can be turned off with this method to improve the performance on creation and deletion of the class objects.

Common cases where a class index is not needed:

- 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.

For more information see [No Class Index](#).

Java:

```
public void maximumActivationDepth (int depth);
```

sets the maximum activation depth for this class to the desired value. For more information see [Activation](#).

Java:

```
public void minimumActivationDepth (int depth);
```

sets the minimum activation depth to the desired value. For more information see [Activation](#).

Java:

```
public ObjectField objectField (String fieldName);
```

returns an ObjectField object to configure the specified field. For more information see [ObjectField Configuration](#).

Java:

```
public void persistStaticFieldValues();
```

turns on storing static field values for this class. For more information see [Static Fields And Enums](#).

Java:

```
public void readAs(Object clazz);
```

creates a temporary mapping of a persistent class to a different class.

If the meta-information for this ObjectClass has been stored to the database file, it will be read from the database file as if it was representing the class specified by the clazz parameter passed to this method.

The clazz parameter can be any of the following:

- a fully qualified classname as a String.
- a Class object.
- any other object to be used as a template.

This method will be ignored if the database file already contains meta-information for clazz.

This method is deprecated, use [Aliases](#) instead.

Java:

```
public void rename (String newName);
```

renames a stored class. For more information see [Refactoring and Schema Evolution](#).

Java:

```
public void storeTransientFields (boolean flag);
```

allows to specify if transient fields are to be stored. The default for every class is `false`.

For more information see [Storing Transient Fields](#).

Java:

```
public void translate (ObjectTranslator translator);
```

registers a translator for this class. For more information see [Translators](#).

Java:

```
public void updateDepth (int depth);
```

specifies the updateDepth for this class. For more information see [Update Depth](#).

ObjectField Configuration

ObjectField is an interface providing configuration methods for class fields. ObjectField instance can be obtained from [ObjectClass/IObjectClass](#) instance:

Java:

```
ObjectField of = configuration.objectClass(clazz).objectField("fieldName");
```

Java:

```
public void cascadeOnActivate(boolean flag);
```

sets cascaded activation behavior. For more information see [Activation](#).

Java:

```
public void cascadeonDelete(boolean flag);
```

sets cascaded delete behavior. For more information see [Delete Behavior](#).

Java:

```
public void cascadeOnUpdate(boolean flag);
```

sets cascaded update behavior. For more information see [Update Depth](#).

Java:

```
public void indexed(boolean flag);
```

turns indexing on or off. For more information see [Indexing](#).

Java:

```
public void rename(String newName);
```

renames a field of a stored class. For more information see [Refactoring and Schema Evolution](#).

Cache Configuration

CacheConfiguration/ICacheConfiguration interface currently provides only one method:

Java:

```
public void slotCacheSize(int size);
```

Sets the cache maximum size in slots. If this value is set to 0, no cache will be used

Freespace Configuration

When objects are updated or deleted, the space previously occupied in the database file is marked as "free". Freespace management system takes care of this space by maintaining two lists in RAM, sorted by address and by size. The following configuration methods allow to tune the freespace management system to help you optimize your system:

Java:

```
public void discardSmallerThan(int 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

Java:

```
public void freespaceFiller(FreespaceFiller freespaceFiller);
```

This method allows to specify a way to overwrite the freed space in the database. This may be desirable when the information needs to be reliably erased. A [custom FreespaceFiller](#) implementation can be created. For an example of db4o implementation of FreespaceFiller you can have a look at XByteFreespaceFiller in the db4o source code (IoAdaptedObjectContainer class).

XByteFreespaceFiller overwrites the free space with "XXXX" symbols and is switched on by rebuilding db4o core with debug = true in Deploy class.

Note, that using freespace filler will slow down IO operations on the database.

The following settings allow to select how the freespace information will be kept:

Java:

```
public void useBTreeSystem();  
public void useRamSystem();
```

.NET:

```
public void UseBTreeSystem();  
public void UseRamSystem();
```

When BTree system is used to keep freespace information, the performance is worse as the information is written to disk at each commit, but the memory consumption is lower. With the RAM system, all freespace information is kept in RAM, which allows the best performance, however this information is lost on abnormal system termination.

FreespaceFiller Example

First of all let's create a database and have a look at the resultant file:

```
FreespaceFillerExample.java: Item  
private static class Item  
{  
    String name;  
    String description;  
  
    public Item(String name, String description) {  
        this.name = name;  
        this.description = description;  
    }  
  
    @Override  
    public String toString() {  
        return String.format("%s, %s", name, description);  
    }  
}
```

```
FreespaceFillerExample.java: createDatabase  
private static void createDatabase(EmbeddedConfiguration config) {  
    new File(DB4O_FILE_NAME).delete();  
  
    ObjectContainer container = Db4oEmbedded.openFile(config, DB4O_FILE_NAME);
```

```

try {
    Item item;
    for (int i = 0; i < OBJECT_COUNT; i++) {
        item = new Item("Title" + i, "Just a description");
        container.store(item);
    }
} finally {
    container.close();
}
}

```

So, we've stored lots of Item objects, each of them has 2 string fields. Let's open reference.db4o file with any text editor and check if we can see some useful information:

Field	Name	Value
name	title1	Just a description
name	title2	Just a description
name	title3	Just a description
name	title4	Just a description
name	title5	Just a description
name	title6	Just a description
name	title7	Just a description
name	title8	Just a description
name	title9	Just a description
name	title10	Just a description

From the image above you can see that all the string fields are actually readable without any special software.

Now, lets create a default configuration - like Db4oEmbedded.newConfiguration() for Java or Db4oEmbedded.NewConfiguration() for .NET - and delete all the objects:

```
FreespaceFillerExample.java: deleteObjects
private static void deleteObjects(EmbeddedConfiguration config) {
    ObjectContainer container = Db4oEmbedded.openFile(config, DB4O_FILE_NAME);
    try {
        List<Item> result = container.queryByExample(Item.class);
        for (Item item : result) {
            container.delete(item);
        }
    } finally {
        container.close();
    }
}
```

If you will open the database file in a text editor again - you won't see much change - the database file is still the same size and all the string information from the Item objects is still there.

We can make the deletion more effective by creating a custom configuration with a freespace filler implementation:

```
FreespaceFillerExample.java: getConfig
private static EmbeddedConfiguration getConfig() {
    EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
    config.file().freespace().freespaceFiller(new FreespaceFiller() {

        @Override
        public void fill(BlockAwareBinWindow io) throws IOException {
            Random r = new Random();
            byte[] data = new byte[io.length()];
            r.nextBytes(data);
            io.write(0, data);
        }
    });
    return config;
}
```

You can see that the implementation is very simple - you only need to implement one method: fill. The parameter gives you a BlockAwareBinWindow and the implementation must create a byte array of the requested size to fill it in. In our implementation we use random bytes.

Now, let's create the database again and run delete objects with the new configuration and look at the database file again.

Configuration Item

ConfigurationItem/IConfigurationItem interface serves 2 purposes:

- it can be used to encapsulate a batch of configuration settings
- it can be used for settings that should be applied to an open object container.

Configuration items should be added to [Common Configuration](#) instance:

Java:

```
configuration.add(configurationItem);
```

Below is the description of currently available implementations.

BigMathSupport

Registers special typehandlers for BigInteger and BigDecimal support. Using BigMathSupport provides better performance and query support for the values of these types. However, using BigMathSupport only makes sense if you are planning to use BigInteger and/or BigDecimal types.

JavaSupport

Adds basic configuration settings that allow to access Java generated object container. Normally, additional [Alias](#) configuration will be required to query for specific classes.

PagedListSupport

Configures support for paged collections.

TransparentActivationSupport

Adds transparent activation support for this session. For more information see [Transparent Activation Framework](#).

TransparentPersistenceSupport

Adds transparent persistence support for this session. For more information see [Transparent Persistence](#).

UniqueFieldValueConstraint

Allows to set a class field to be unique in this database. For more information see [Unique Constraints](#).

File Configuration

FileConfiguration interface contains file-related configuration methods. These methods should be called for Embedded db4o object container or db4o object server.

Java:

```
public void blobPath(String path) throws IOException;
```

configures the path to be used to store and read Blob data. For more information see [Blobs](#)

Java:

```
public void blockSize(int bytes)
```

sets the storage data blocksize for new object containers. The standard setting is 1 allowing for a maximum database file size of 2GB. This value can be increased to allow larger database files, although some space will be lost to padding because the size of some stored objects will not be an exact multiple of the block size. A recommended setting for large database files is 8, since internal pointers have this length. This setting is only effective when the database is first created. The size can be any integer in range from 1 to 127.

Java:

```
public void databaseGrowthSize(int bytes)
```

configures the size database files should grow in bytes, when no free slot is found within. Tuning setting. Whenever no free slot of sufficient length can be found within the current database file, the database file's length is extended. This configuration 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.

Java:

```
public void disableCommitRecovery();
```

turns commit recovery off. db4o uses a two-phase commit algorithm. 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.

Java:

```
public FreespaceConfiguration freespace;
```

Returns freespace configuration for this file configuration.

Java:

```
public void generateUUIDs(ConfigScope)
```

configures db4o to generate UUIDs for stored objects. For more information see [Unique Universal IDs](#).

Java:

```
public void generateVersionNumbers(ConfigScope)
```

configures db4o to generate version numbers for stored objects. For more information see [Unique Universal IDs](#).

Java:

```
public void lockDatabaseFile(boolean flag)
```

can be used to turn the database file locking thread off.

Java:

```
public void readOnly(boolean flag)
```

turns readOnly mode on and off. Readonly mode allows to open an unlimited number of reading processes on one database file. It is also convenient for deploying db4o database files on CD-ROM.

In client-server environment this setting should be used on client side.

Java:

```
public void recoveryMode(boolean flag)
```

turns recovery mode on and off. Recovery mode can be used to try to retrieve as much as possible out of an already 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.

Java:

```
public void reserveStorageSpace(long byteCount)
```

tuning feature only: reserves a number of bytes in database files. The global setting is used for the creation of new 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. Less read/write head movement can result in improved performance. 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.

Java:

```
public void storage(Storage storage);  
public Storage storage();
```

Allows to configure a custom IO byte storage mechanism. One of the db4o implementations can be used, i.e. MemoryStorage, NonFlushingStorage etc. or a user-implemented storage based on Storage/IStorage interface. Possible usecases: improved performance with a native library, encryption, mirrored storage to 2 files etc.

Query Configuration

QueryConfiguration interface allows to configure query settings to be used by query processor. Query configuration can be accessed by all implementations of CommonConfiguration/ICommonConfiguration interface:

Java:

```
QueryConfiguration qc = configuration.queries();
```

Currently query configuration provide one method:

Java:

```
public void evaluationMode(QueryEvaluationMode mode);
```

This method allows to configure the way query results will be evaluated. `QueryEvaluationMode` class defines 3 constants, representing different modes:

- Immediate
- Lazy
- Snapshot

In Immediate mode, the query is evaluated immediately and the full set of object IDs is returned.

In Snapshot mode, all indexes constraints are evaluated immediately and a snapshot of indexed results is returned. All non-indexes constraints and all evaluations will be triggered when user iterates through the snapshot result.

In Lazy mode an iterator is created against the best found index. All the constraints evaluations will be done as user iterates through the result.

For more detailed information see [Query Modes](#).

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;
    }
}

```

Storing Transient Fields

Java:

```
Db4o.configure().objectClass(clazz).storeTransientFields(true)
```

This setting can be used to turn on storing of transient fields to db4o. It can be sometimes useful for debug purposes.

In order to test how it works add the following method to the example in [Transient Fields In Java](#)/[Transient Fields In .NET](#):

```
MarkTransientExample.java: configureSaveTransient
private static Configuration configureSaveTransient() {
    Configuration configuration = Db4o.newConfiguration();
    configuration.objectClass(Test.class).storeTransientFields(true);
    return configuration;
}
```

Indexing

db4o allows to index fields to provide maximum querying performance. To request an index to be created, you would issue the following API method call in your global [db4o configuration method](#) before you open an ObjectContainer/ObjectServer:

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

Java:
configuration.objectClass(Foo.class).objectField("bar").indexed(true);
```

If the configuration is set in this way, an index on the Foo#bar field will be created (if not present already) the next time you open an ObjectContainer/ObjectServer and you use the Foo class the first time in your application.

Contrary to all other [configuration calls](#) indexes - once created - will remain in a database even if the index configuration call is not issued before opening an ObjectContainer/ObjectServer.

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

Java:

```
configuration.objectClass(Foo.class).objectField("bar").indexed(false);
```

Actually dropping the index will take place the next time the respective class is used. db4o will tell you when it creates and drops indexes, if you choose a message level of 1 or higher:

Java:

```
configuration.messageLevel(1);
```

For creating and dropping indexes on large amounts of objects there are two possible strategies:

1. Import all objects with indexing off, configure the index and reopen the Object-Container/ObjectServer.
2. Import all objects with indexing turned on and commit regularly for a fixed amount of objects (~10,000).
 1. will be faster.
 2. will keep memory consumption lower.

For more information see [Enable Field Indexes](#) chapter.

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:

- [Enable Field Indexes](#)
- [Discarding Free Space](#)

- [Calling constructors](#)
- [Defragment](#)
- [Turning Off Weak References](#)
- [No Shutdown Thread](#)
- [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](#)
- [Unicode](#)
- [Prefetching IDs For New Objects](#)
- [Prefetching Objects For Query Results](#)
- [No Class Index](#)
- [No Field Evaluation](#)
- [RandomAccessFileAdapter](#)
- [Commit Strategies](#)
- [Database Size](#)
- [Optimizing Native Queries](#)
- [Using RandomAccessFileAdapter](#)

Enable Field Indexes

For class Car with field "pilot":

Java:

```
Db4o.configure().objectClass(Car.class).objectField("pilot").indexed(true)
```

Advantage

The fastest way to improve the performance of your queries is to enable indexing on some of your class's key fields. You can read how to do it in [Indexing](#) chapter of this documentation.

Further step of index tuning is to optimize indexes for Class.Field1.Field2 access. What will give us the best performance:

- index on Field1;
- index on Field2;
- index on both fields?

To find the answer let's consider classes Car and Pilot from the previous chapters. In order to see indexing influence we will put 10000 new cars in our storage (note that for db4o version > 5.6 the amount of objects should be much more to see the differences in execution time due to BTree based index optimized for big amounts of data):

```
IndexedExample.java: fillUpDB
private static void fillUpDB() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container=Db4o.openFile(DB4O_FILE_NAME);
    try {
        for (int i=0; i<10000;i++) {
            AddCar(container,i);
        }
    } finally {
        container.close();
    }
}
```

```
IndexedExample.java: addCar
private static void AddCar(ObjectContainer container, int points)
{
    Car car = new Car("BMW");
    car.setPilot(new Pilot("Tester", points));
    container.store(car);
}
```

Now we have lots of similar cars differing only in the amount of pilots' points - that will be our constraint for the query.

```
IndexedExample.java: noIndex
private static void noIndex()  {
    ObjectContainer container=Db4o.openFile(DB4O_FILE_NAME);
    try  {
        Query query = container.query();
        query.constrain(Car.class);
        query.descend("pilot").descend("points").constrain(new Integer(99));

        long t1 = System.currentTimeMillis();
        ObjectSet  result = query.execute();
        long t2 = System.currentTimeMillis();
        long  diff = t2 - t1;
        System.out.println("Test 1: no indexes");
        System.out.println("Execution time="+diff + " ms");
        listResult(result);
    }
    finally  {
        container.close();
    }
}
```

You can check execution time on your workstation using interactive version of this tutorial.

Let's create index for pilots and their points and test the same query again:

```
IndexedExample.java: fullIndex
private static void fullIndex()  {
    Configuration configuration = Db4o.newConfiguration();
    configuration.objectClass(Car.class).objectField("pilot").indexed(true);
    configuration.objectClass(Pilot.class).objectField("points").indexed(true);
    ObjectContainer container=Db4o.openFile(configuration, DB4O_FILE_NAME);
    try  {
        Query query = container.query();
        query.constrain(Car.class);
        query.descend("pilot").descend("points").constrain(new Integer(99));

        long t1 = System.currentTimeMillis();
        ObjectSet  result = query.execute();
        long t2 = System.currentTimeMillis();
        long  diff = t2 - t1;
        System.out.println("Test 2: index on pilot and points");
        System.out.println("Execution time="+diff + " ms");
        listResult(result);
    }
    finally  {
        container.close();
    }
}
```

```
}
```

That result is considerably better and proves the power of indexing.

But do we really need 2 indexes? Will single pilot or points index suffice? Let's test this as well:

```
IndexedExample.java: pilotIndex
private static void pilotIndex()  {
    Configuration configuration = Db4o.newConfiguration();
    configuration.objectClass(Car.class).objectField("pilot").indexed(true);
    configuration.objectClass(Pilot.class).objectField("points").indexed(false);
    ObjectContainer container=Db4o.openFile(configuration, DB4O_FILE_NAME);
    try  {
        Query query = container.query();
        query.constrain(Car.class);
        query.descend("pilot").descend("points").constrain(new Integer(99));

        long t1 = System.currentTimeMillis();
        ObjectSet  result = query.execute();
        long t2 = System.currentTimeMillis();
        long  diff = t2 - t1;
        System.out.println("Test 3: index on pilot");
        System.out.println("Execution time="+diff + " ms");
        listResult(result);
    }
    finally  {
        container.close();
    }
}
```

```
IndexedExample.java: pointsIndex
private static void pointsIndex()  {
    Configuration configuration = Db4o.newConfiguration();
    configuration.objectClass(Car.class).objectField("pilot").indexed(false);
    configuration.objectClass(Pilot.class).objectField("points").indexed(true);
    ObjectContainer container=Db4o.openFile(configuration, DB4O_FILE_NAME);
    try  {
        Query query = container.query();
        query.constrain(Car.class);
        query.descend("pilot").descend("points").constrain(new Integer(99));

        long t1 = System.currentTimeMillis();
        ObjectSet  result = query.execute();
        long t2 = System.currentTimeMillis();
        long  diff = t2 - t1;
        System.out.println("Test 4: index on points");
        System.out.println("Execution time="+diff + " ms");
        listResult(result);
    }
}
```

```
        }
    finally {
        container.close();
    }
}
```

Single index does not increase query performance on second level fields.

To maximize retrieval performance on encapsulated fields of different levels of enclosure

Class.Field1.Field2.Field3(.FieldN)

indexes for each field level should be created:

Class.Field1.Indexed(true)

Field1Class.Field2.Indexed(true)

Field2Class.Field3.Indexed(true)

...

Field(N-1)Class.FieldN.Indexed(true)

Alternate Strategies

Field indexes dramatically improve query performance but they may considerably reduce storage and update performance. The best way to decide where to put the indexes is to test them on completed application with typical data load.

Discarding Free Space

Java: `Db4o.configure().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.

Calling constructors

Java:

```
Db4o.configure().callConstructors(true)
```

Advantage

will configure db4o to use constructors to instantiate objects.

Effect

On VMs where this is supported (Sun Java VM > 1.4, .NET, Mono) db4o tries to create instances of objects without calling a constructor. On Java VMs db4o is using reflection for this feature so this may be considerably slower than using a constructor. For the best performance on Java it is recommended to add a public zero-parameter constructor to every persistent class and to turn constructors on. Benchmarks on .NET have shown that the default setting (`#callConstructors(false)`) is faster.

Alternate Strategies

Constructors can also be turned on for individual classes only with

Java: `Db4o.configure().objectClass(Foo.class).callConstructor(true)`

There are some classes (e.g. `java.util.Calendar`) that require a constructor to be called to work. Further details can be found in the [Constructors](#) chapter

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.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);           con-
figuration.objectClass("yourClass").
maximumActivationDepth(maximumDepth) ;                                     ;
configuration.objectClass("yourClass").
cascadeOnActivate;
(bool)                                configuration.objectClass("yourClass") .
objectField("field").cascadeOnActivate(bool);
```

For more information on activation strategies see [Activation chapter](#).

Turning Off Weak References

Java:

```
Db4o.configure().weakReferences(false)
```

Advantage

will configure db4o to use hard direct references instead of weak references to control instantiated and stored objects.

Effect

A db4o database keeps a reference to all persistent objects that are currently held in RAM, whether they were stored to the database in this session or instantiated from the database in this session. This is how db4o can "know" than an object is to be updated: Any "known" object must be an update, any "unknown" object will be stored as "new". (Note that the reference system will only be in place as long as an ObjectContainer is open. Closing and reopening an ObjectContainer will clean the references system of the ObjectContainer and all objects in RAM will be treated as

"new" afterwards.) In the default configuration db4o uses weak references and a dedicated thread to clean them up after objects have been garbage collected by the VM. Weak references need extra resources and the cleanup thread will have a considerable impact on performance since it has to be synchronized with the normal operations within the ObjectContainer. Turning off weak references will improve speed.

The downside: To prevent memory consumption from growing consistently, the application has to take care of removing unused objects from the db4o reference system by itself. This can be done by calling

Java:

```
ExtObjectContainer.purge(object)
```

Alternate Strategies

Java:

```
ExtObjectContainer.purge(object)
```

can also be called in normal weak reference operation mode to remove an object from the reference cache. This will help to keep the reference tree as small as possible. After calling #purge(object) an object will be unknown to the ObjectContainer so this feature is also suitable for batch inserts.

No Shutdown Thread

Java:

```
Db4o.configure().automaticShutDown(false)
```

Advantage

can prevent the creation of a shutdown thread on some platforms.

Effect

On some platforms db4o uses a ShutDownHook to cleanly close all database files upon system termination. If a system is terminated without calling ObjectContainer#close() for all open ObjectContainers, these ObjectContainers will still be usable but they will not be able to write back their freespace management system back to the database file. Accordingly database files will be observed to grow.

Alternate Strategies

Database files can be reduced to their minimal size with [Defragment](#)

No Class Index

Java:

```
configuration.objectClass("package.classname").indexed(false);
```

Turns class index off.

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.

Automatic Shutdown

```
configuration.automaticShutDown(flag)
```

Advantage

Automatic shutdown ensures that all db4o processes are terminated correctly.

Effect

Depending on the JDK, db4o uses one of the following two methods to shut down, if no more references to the ObjectContainer are being held or the JVM terminates:

- JDK 1.3 and above: `Runtime.addShutdownHook()`
- JDK 1.2 and below: `System.runFinalizersOnExit(true)`

and code in the finalizer.

`AutomaticShutDown` setting is true by default.

Alternate Strategies

Some JVMs have severe problems with both methods. For these rare cases the automaticShutdown feature may be turned off.

No callbacks

Java:

```
Db4o.configure().callbacks(false)
```

Advantage

will prevent db4o from looking for callback methods in all persistent classes on system startup.

Effect

Upon system startup, db4o will scan all persistent classes for methods with the same signature as the methods defined in com.db4o.ext.ObjectCallbacks, even if the interface is not implemented. db4o uses reflection to do so and on constrained environments this can consume quite a bit of time. If callback methods are not used by the application, callbacks can be turned off safely.

Alternate Strategies

Class configuration features are a good alternative to callbacks. The most recommended mechanism to cascade updates is:

Java:

```
Db4o.configure().objectClass("yourPackage.yourClass").cascadeOnUpdate(true)
```

B-Tree tuning

Db4o uses special B-tree indexes for increased query performance and reduced memory consumption (the feature was introduced since version 5.4 for class indexes and since 5.7 for field indexes).

Advantage

B-trees are optimized for scenarios when part or all of a data set is on secondary storage such as a hard disk, since disk accesses are extremely expensive operations. B-trees minimize the number of disk accesses required to find data by traversing a sorted tree structure and only need a single disk access per level of the tree.

In order to use B-tree capabilities for field indexes you will simply need to define indexed fields in your classes:

Java:

```
configuration.objectClass(Foo.class).objectField("field").indexed(true)
```

Effect

The caching behaviour of the B-trees can be configured with the following two switches:

Java:

```
configuration.bTreeCacheHeight(height)
```

configures the size of BTree nodes in indexes.

Java:

```
configuration.bTreeNodeSize(size)
```

configures caching of B-tree nodes. Clean B-tree nodes will be unloaded on #commit and #rollback unless they are configured as cached here.

Higher values for the cache height will get you better performance at more RAM consumption.

With the node size you can fine-tune exactly how many reads the B-tree will need to get to leaf nodes. Lower values will allow a lower memory footprint and more efficient reading and writing of small slots. Higher values will reduce the overall number of read and write operations and allow better performance at the cost of more RAM use.

If you raise the number of elements per node and/or the cache depth, you will use more RAM but achieve higher performance. In principle, if you set the node size to a very high value and cache the first node, you should get exactly the same behavior as with the old class indexes.

For now the default settings are 1 for the height of the cache and 100 for the size of the nodes.

When testing B-tree you should remember that B-trees only really start to give you performance advantages with larger numbers of objects. With object counts of 1,000 or 10,000 the old flat index is highly efficient because everything is kept in memory. Using tests with more than 100,000 objects you will really see things degrade with:

- performance, because of a full purge called each time on commit
- memory consumption, because the index will be reloaded completely immediately when the next object is added.

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 ObjectContainer 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"

See also [ACID Properties For Db4o](#).

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 loosing 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 trans-

action 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 ObjectContainer 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:

```
configuration.blockSize(newBlockSize); Defragment.defrag("sample.db4o")
```

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 that db4o should internally use, 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 Objectcontainers.

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.

File Header

Every database file starts with a fixed file header.

The information stored in the header is:

(byte) 'd'

(byte) 'b'

(byte) '4'
(byte) 'o'
(byte) headerVersion
(int) headerLock
(long) openTime
(long) accessTime
(int) Transaction pointer 1
(int) Transaction pointer 2
(int) blockSize
(int) classCollectionID
(int) freespaceID
(int) variablePartID

The format and length of the fixed header is constant.

It points to a second block of data in the database file the "variable part", with the variablePartID.

The information stored in the variable part is:

(int) converter version
(byte) freespace system used
(int) freespace address
(int) identity ID
(long) versionGenerator
(int) uuid index ID

- Implementation
 - com.db4o.internal.fileheader.FileHeader1
 - com.db4o.internal.fileheader.FileHeaderVariablePart1

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 typehandler 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.

No Field Evaluation

Java:

```
configuration.ObjectClass(clazz).ObjectField("field").queryEvaluation(false)
```

Advantage

Improves performance by reducing query evaluation volume

Effect

All fields are evaluated by default. Use this method to turn query evaluation off for specific fields, which are never used in queries. This setting can help you to increase the querying performance by reducing the evaluation task.

Alternate Strategies

Consider marking fields as [transient](#) if their persistence is not necessary.

No lock file thread

Java:

```
configuration.lockDatabaseFile(false)
```

Advantage

will prevent the creation of a lock file thread on Java platforms without NIO (< JDK 1.4.1).

Effect

If file locking is not available on the system, db4o will regularly write a timestamp lock information to the database file, to prevent other VM sessions from accessing the database file at the same time. Uncontrolled concurrent access would inevitably lead to corruption of the database file. If the application ensures that it can not be started multiple times against the database file, db4o file locking may not be necessary.

Alternate Strategies

Database files can safely be opened from multiple sessions in readonly mode. Use:

Java:

```
configuration.readOnly(true)
```

No schema changes

Java:

```
Db4o.configure().detectSchemaChanges(false)
```

Advantage

will prevent db4o from analysing the class structure upon opening a database file.

Effect

Upon system startup, db4o will use reflection to scan the structure of all persistent classes. This process can take some time, if a large number of classes are present in the database file. For the best possible startup performance on "warm" database files (all classes already analyzed in a previous startup), this feature can be turned off.

No test instances

Java:

```
configuration.testConstructors(false)
```

Advantage

will prevent db4o from creating a test instance of persistent classes upon opening a database file.

Effect

Upon system startup, db4o attempts to create a test instance of all persistent classes, to ensure that a public zero-parameter constructor is present. This process can take some time, if a large number of classes are present in the database file. For the best possible startup performance this feature can be turned off.

Alternate Strategies

In any case it's always good practice to create a zero-parameter constructor. If this is not possible because a class from a third party is used, it may be a good idea to write a [translator](#) that translates the third party class to one's own class.

Optimizing Native Queries

Advantage

Optimized Native Queries allow to achieve considerable performance improvements.

Effect

[Native Queries](#) allow to express a database query in a native object-oriented language. This solution is elegant and straightforward as no mixture of concepts (object and relational) occurs. However, the challenge is to make this solution performant.

If the NQ code is run as is, it requires instantiation of all the members of a class. This is very slow in most cases. In order to improve the performance a special optimizer is used by db4o. The idea of the optimization is to analyze the code in a Native Query and provide an alternative in a database query language. This can be done in runtime or build time.

For detailed information about optimization strategies, please, see [Native Query Optimization](#).

Alternate Strategies

Obviously, optimization is not possible in cases, when a native query does not have a database query alternative. To reveal those cases [db4o Diagnostic](#) system should be used.

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.

Alternate strategies

In some cases you may want to persist class meta-information without the actual object data. The example can be database singleton for [remote code execution](#)

Prefetching IDs For New Objects

Java:

```
void ClientServerConfiguration.prefetchIDCount(int prefetchIDCount);
```

Sets the number of IDs to be pre-allocated in the database for new objects created on the client

Advantage

Prefetching several IDs for the new objects created on the client allows to improve the performance by reducing client-server communication.

Effect

When a new object is created on a client, the client should contact the server to get the next available object ID. PrefetchIDCount allows to specify how many IDs should be pre-allocated on the server and prefetched by the client. This method helps to reduce client-server communication.

PrefetchIDCount can be tuned to approximately match the usual amount of objects created in one operation to improve the performance.

If PrefetchIDCount =1 a client will have to connect to the server for each new objects created

If PrefetchIDCount is bigger than the amount of new objects to be created the database will keep unnecessary preallocated space.

The default PrefetchIDCount is 10.

Alternate Strategies

A possible alternative can be using [Messaging](#)for bulk inserts on the server.

Prefetching Objects For Query Results

Java:

```
void ClientServerConfiguration.prefetchObjectCount(int prefetchObjectCount);
```

Sets the number of objects to be prefetched for an ObjectSet in C/S mode.

Advantage

PrefetchObjectCount setting allows to tune the way object results are delivered from the server.

Effect

Long query results can take significant time to be transferred from the server to the client. This can lead to blocking communication channel. In order to prevent this, query results are fetched from the server in portions. The default portion size is 10 objects.

PrefetchObjectCount setting allows you to change the default prefetched amount.

`prefetchObjectCount = 1` will enforce the client to send a request for each next object in the result set. This can be used to improve the performance for large objects.

With the default setting(10) the client receives the first 10 objects, the next 10 will be delivered on the first request.

Bigger values of `prefetchObjectCount` can improve the performance for small objects when the whole result set is required for immediate processing on the client.

`PrefetchObjectCount` setting also influences the processing of queries in lazy and snapshot modes.

FileStorage

Java:

```
configuration.file().storage(new FileStorage())
```

Advantage

Decreases memory consumption by using an IO adapter without caching.

Effect

Since db4o version 6.2 [CachingStorage](#) is used by default. This IO adapter has some valuable advantages, however the disadvantage is increased memory consumption. Using FileStorage can help to keep the memory consumption to the minimum.

Alternate Strategies

An alternative way to control the memory consumption is to configure it in the CachingStorage:

Java:

```
new CachedStorage(delegateAdapter, page_size, page_count);
```

`page_size * page_count` - will define the maximum amount of memory used for caching.

Unicode

Java:

```
configuration.unicode(false)
```

Advantage

Turning Unicode support off reduces the file storage space for strings by factor 2 and improves performance.

Effect

Enables/disables Unicode string storage format. Unicode allows you to store string data in any language to db4o.

This method needs to be called **before** a database file is created. db4o database files keep their string format after creation.

The default setting is `true`

Alternate Strategies

You can create your own string marshallers, using TypeHandler4 interface.

Using RandomAccessFileAdapter

Java:

```
Configuration config = Db4o.newConfiguration();
Config.io(new RandomAccessFileAdapter);
```

Advantage

Uses less memory than default CachedIoAdapter.

Effect

By default db4o uses CachedIoAdapter, which allows to improve bulk write performance, by creating cached pages in memory and thus reducing the amount of physical disc writes. This of course can increase the memory consumption, which can be a critical resource in some cases. To free up some additional memory it might be reasonable to switch to RandomAccessFileAdapter.

Alternate Strategies

The default CachedIoAdapter and configurable MemoryIoAdapter allow to achieve better performance by utilizing RAM instead of hard-drive.

Runtime Statistics

Db4o Runtime Statistics is a monitoring feature allowing to collect various important runtime db4o 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.

Runtime Statistics is a work-in-progress and the new statistics will be added as the need appears. The version 7.12 includes the following statistics:

- [Queries](#)

- number of class scans / sec

A class scan means that db4o will need to read each object of a given class in order to figure out whether the object should be included in the result set of a query. In this case indexing the parent referencing field would avoid this costly operation and improve query times.

- number of queries / sec

Number of SODA queries executed per second.

- Average query execution time

The time to retrieve the list of ids for the matching objects (this doesn't take the time to activate the objects into account)

- number of unoptimized native queries / sec

Number of native queries (per second) that were not optimized. Developers should strive to run only optimized native queries in order to get the best possible performance.

- number of native queries / sec

Total number of native queries (either optimized or not) executed per second.

- [FreeSpaceManager](#)

- average slot size

The average size of free storage slots available. Smaller slots are less likely to be reused.

- number of free slots

A high number of free slots indicates a high level of fragmentation.

- number of reused slots / sec

Number of reused free slots per second.

- number of bytes in free space management

A high value (with a high value of free slots) is a good indication that defragmenting the database may bring performance improvements.

- Reference System

- number of objects in reference system

Total number of objects in the db4o reference system.

- IO

- number of bytes read / sec
- number of bytes written / sec
- number of reads / sec
- number of writes / sec
- number of syncs / sec

- Networking

- Bytes Sent/Received

Bytes sent/received from/to clients.

- Connected Clients

Number of currently connected clients.

- Messages sent

Number of messages sent from/to db4o clients and servers. Developers can use this statistic as an indicator whenever their object models are driving db4o to exchange too many messages between clients/server. This statistic is also useful to help developers to tweak prefetchDepth/prefetchObjectCount configuration in order to minimize messages exchange between client / servers.

- .Net Specific

Example

Statistics Example

On the Java platform, statistics are published through Java Management Extensions (JMX). A variety of JMX clients is available. Many of them are linked from here.

In this tutorial we will use a standard tool from JDK 1.6 – jconsole. For general information on using jconsole, please , refer to [Java documentation](#).

All that you need to do to enable statistics collection in your application is to add a respective statistic to the Common Configuration provider, i.e:

```
Common Configuration#add(new IOMonitoringSupport()); Common Configuration#add(new QueryMonitoringSupport()); Common Configuration#add(new NativeQueryMonitoringSupport()); Common Configuration#add(new ReferenceSystemMonitoringSupport()); Common Configuration#add(new FreespaceMonitoringSupport()); Common Configuration#add(new com.db4o.-monitoring.cs.NetworkingMonitoringSupport());
```

Statistic classes are located in db4o-optional jar, and networking specific statistics are located in db4o-cs jar.

Note that since gathering these statistics does introduce some performance overhead (for most of the statistics this overhead is so small that it is immeasurable) each set of runtime statistics (Queries, FreeSpaceManager, ReferenceSystem, etc.) must be configured independently (so you can decide which ones you want to track and pay the performance hit only for those ones).

Statistics can be collected for both networked and standalone mode and are enabled per object container (database file) so you can monitor as many open databases as you want.

We will look at the example of networking mode to cover both.

The following code will be used to configure the client and the server:

```
MonitoringDemo.java: configure
private <T extends CommonConfigurationProvider T> configure(T config) {
    config.common().objectClass(Item.class).objectField("name").indexed(
        true);
    config.common().add(new IOMonitoringSupport());
    config.common().add(new QueryMonitoringSupport());
    config.common().add(new NativeQueryMonitoringSupport());
    config.common().add(new ReferenceSystemMonitoringSupport());
    config.common().add(new FreespaceMonitoringSupport());
    config.common().add(new NetworkingMonitoringSupport());
    config.common().add(new ObjectLifecycleMonitoringSupport());
    return config;
}
```

The client will store some objects to the database and perform some queries:

```
MonitoringDemo.j:ava ExecuteSodaQuery
private void executeSodaQuery(ObjectContainer objectContainer) {
    Query query = objectContainer.query();
    query.constrain(Item.class);
    query.descend("name").constrain("1");
    query.execute();
}
```

```
MonitoringDemo.java: executeOptimizedNativeQuery
private void executeOptimizedNativeQuery(ObjectContainer objectContainer) {
    objectContainer.query(new PredicateItem() {
        @Override
```

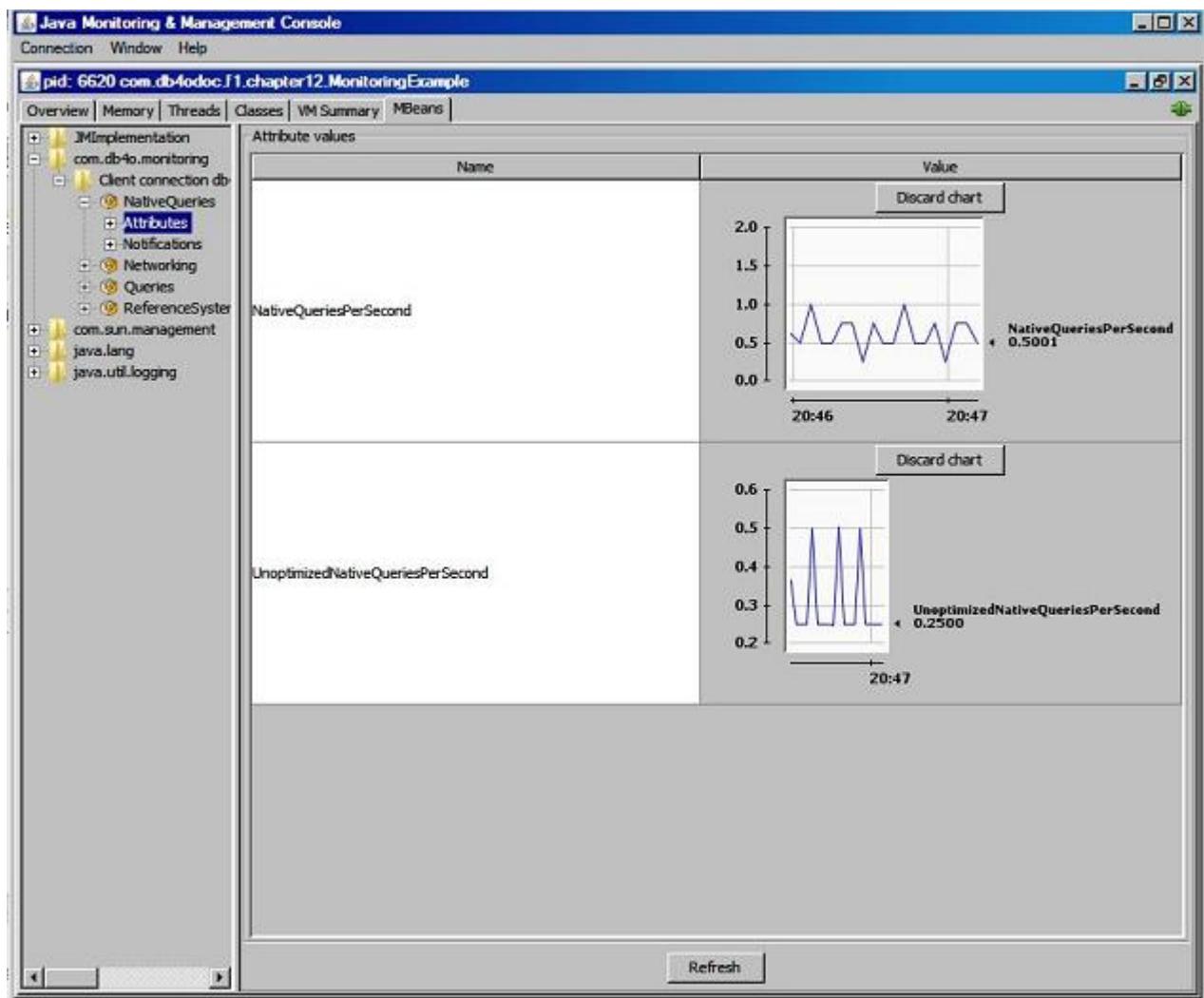
```
        public boolean match(<Item> candidate) {
            return candidate.name.equals("name1");
        }
    });
}
```

```
MonitoringDemo.java: executeUnOptimizedNativeQuery
private void executeUnOptimizedNativeQuery(ObjectContainer objectContainer) {
    objectContainer.query(new Predicate<Item>() {
        @Override
        public boolean match(Item candidate) {
            return candidate.name.charAt(0) == 'q';
        }
    });
}
```

Now start jconsole (it is location in the /bin folder of your java home). First you need to connect to the MonitoringDemo process. 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 doubleclicking on the number value a graph will be displayed. Here is a screenshot how results for Native Queries could look like:



For some "exceptional" events, we also provide the option to subscribe to 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.

Debugging db4o

Debugging is, in general, a cumbersome and tiring task. And it tends to be harder when various subsystems are tightly coupled, like db4o library and your application. How can db4o help you with the process?

More Reading:

- [Debug Messaging System](#)
- [Customizing The Debug Message Output](#)
- [ExceptionsOnNotStorable](#)
- [Using DTrace](#)
- [Reading Db4o File](#)

Debug Messaging System

Db4o messaging system is a special tool, which makes db4o functionality more transparent to the user. It can be used:

- in debugging session - to find out where the problem can reside;
- for learning - to watch, what does db4o actually do with the objects.

In order to activate messaging before opening a database file use:

Java:

```
configuration.messageLevel(level)
```

where *level* can be:

level = 0: no messages;

level > 0: normal messages;

level > 1: state messages (new object, object update, delete);

level > 2: activation messages (object activated, deactivated).

In order to set up a convenient output stream for the messages, call:

Java:

```
configuration.setOut(outStream)
```

By default the output is sent to System.out.

For more information on #setOut call see [Customizing The Debug Message Output](#).

#messageLevel(level) also can be set after a database has been opened:

```
Java: ObjectContainer#ext().configure().messageLevel(level)
```

The same applies for #setOut().

Let's use the simplest example to see all types of debug messages:

```
DebugExample.java: setCars
private static void setCars()
{
    // Set the debug message level to the maximum
    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().messageLevel(3);

    // Do some db4o operations
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container=Db4oEmbedded.openFile(configuration, DB4O_FILE_NAME);
    try {
        Car car1 = new Car("BMW");
        container.store(car1);
        Car car2 = new Car("Ferrari");
        container.store(car2);
        container.deactivate(car1,2);
        Query query = container.query();
        query.constrain(Car.class);
        List<Car> results = query.execute();
        listResult(results);
    } finally {
        container.close();
    }
}
```

Output looks quite messy, but allows you to follow the whole process. For debugging purposes messaging system provides a timestamp and internal ID information for each object (first number in state and activate messages).

Customizing The Debug Message Output

[Debug Messaging System](#) topic explains how to activate the debug messages in your application. However the default console output has very limited possibilities and can only be used effectively in console applications. To use debug messaging system in any environment db4o gives you an API to redirect debug output to another stream:

Java:

```
Configuration.setOut(java.io.PrintStream)
```

An example below shows how to create and use a log file for debug messages:

```
DebugExample.java: setCarsWithFileOutput
private static void setCarsWithFileOutput() throws FileNotFoundException
{
    // Create StreamWriter for a file
    FileOutputStream fos = new FileOutputStream("Debug.txt");
    PrintStream debugWriter = new PrintStream(fos);

    // Redirect debug output to the specified writer
    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().outStream(debugWriter);

    // Set the debug message level to the maximum
    configuration.common().messageLevel(3);

    // Do some db4o operations
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container=Db4oEmbedded.openFile(configuration, DB4O_FILE_NAME);
    try {
        Car car1 = new Car("BMW");
        container.store(car1);
        Car car2 = new Car("Ferrari");
        container.store(car2);
        container.deactivate(car1,2);
        Query query = container.query();
        query.constrain(Car.class);
        List<Car> results = query.execute();
        listResult(results);
    } finally {
        container.close();
        debugWriter.close();
    }
}
```

Using a log file for debug messages has several advantages:

- debug information is available after the application has terminated;
- console output is not polluted with debug messages;
- debug information from the clients can be available on the server.

You can always switch back to the default setting using:

Java:

```
Configuration.setOut(System.out)
```

ExceptionsOnNotStorable

There is another setting that can be of great value in debug process:

Java:

```
configuration.exceptionsOnNotStorable (true|false)
```

In some environments (especially those that provide plugin mechanics or perform some kind of class reloading) you may encounter strange problems due to classloader issues. These environments include servlet containers, Eclipse plugins, reloading JUnit test runners, etc.

In the default configuration ExceptionsOnNotStorable is set to true, which means that an exception will be thrown when db4o is not able to store db4o internal classes or set db4o internal translators, etc.

Db4o uses the context classloader by default. This is an appropriate choice in most situations, but it's not really reliable, since the concrete context classloader depends on the environment you're running in. Therefore you can explicitly specify the classloader to be used for db4o operation by calling

Java:

```
configuration.reflectWith(new JdkReflector(classLoader))
```

Basically you just have to find out the appropriate classloader and configure db4o accordingly. The right choice depends on the specific classloader hierarchy of your application context. Two examples:

- In servlet containers, there usually should be no problem, since most containers automatically set the context classloader to the webapp classloader. So it shouldn't matter, whether the db4o.jar resides in your webapp's WEB_INF/lib (where it will be loaded by the appropriate classloader itself, anyway) or in one of the shared lib folders (where the classloader responsible for loading db4o will not be able to see webapp specific classes).
- In Eclipse, the context classloader is the system classloader, which is agnostic of plugin-specific classes. You'll have to configure db4o to use your plugin's classloader, e.g. MyPlugin.class.getClassLoader(). (If the db4o.jar resides in your plugin, you'll get the same effect by just using Db4o.class.getClassLoader()).

The approach to solving classloader problems (not only for db4o, but generally) is:

- identify the classes/libs db4o needs to know
- identify the classloader hierarchy of your application context
- use the most generic classloader that knows all needed classes, either directly or indirectly via delegation

See also:[Classloader issues](#)

#exceptionsOnNotStorable(true) will also help you to identify classes that db4o cannot persist.

db4o needs a constructor that it can use to create user objects. Ideally this is a zero-parameter constructor (declared public for Java JDK versions prior to JDK 1.2). If db4o does not find a zero-parameter constructor, it iterates through all other constructors and internally attempts to create an instance of an object by passing appropriate null parameters. If this is successful with any of the present constructors, this constructor is used.

There are classes that do not have usable constructors. `java.net.URL` is an example from the Java JDK. In this case you have the following options:

- add a zero-parameter constructor specifically for db4o;
- derive from the class and add a zero-parameter constructor;
- add a custom translator.

If you need to quickly implement a solution for one of the JDK classes, and querying members is not an issue, you may choose to use the built-in serializable translator. Here is an example, how this is done for `java.net.URL`:

Java:

```
configuration.objectClass("java.net.URL").translate(new TSerializable());
```

The above code needs to be executed every time before the db4o engine is started. See also: [Constructors](#), [Translators](#).

Another db4o system, which can give you a valuable feedback about db4o functioning in your application is [Diagnostics](#).

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(long)` 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

Db4o engine provides user with a special mechanism showing runtime diagnostics information. This functionality can become your guide to excellent performance and low memory consumption. Diagnostics can be switched on in the configuration before opening the database file:

Java:

```
configuration.diagnostic().addListener(new DiagnosticListener())
```

where DiagnosticListener is a callback interface tracking diagnostic messages from different parts of the system:

```
public interface DiagnosticListener { public void onDiagnostic(Diagnostic d); }
```

Db4o provides 2 different listeners:

- DiagnosticToConsole (Java, prints diagnostic messages to the console);
- DiagnosticToTrace (.NET, prints diagnostic messages to the debug output window).

Every diagnostic message is represented by it's own type, all possible types can be found in the com.db4o.diagnostic package/namespace.

At the present moment the following diagnostic classes are implemented:

- ClassHasNoFields
- LoadedFromClassIndex
- NativeQueryNotOptimized
- UpdateDepthGreaterOne
- DescendIntoTranslator

More Reading:

- [ClassHasNoFields](#)
- [LoadedFromClassIndex](#)
- [NativeQueryNotOptimized](#)
- [UpdateDepthGreaterOne](#)
- [Diagnostic Messages Filter](#)
- [DescendIntoTranslator](#)

ClassHasNoFields

This diagnostic type provides information about classes in your persistent class hierarchy that have no persistent fields. The diagnostic message appears when the class is saved to the database. It is

recommended to remove such classes from the database to avoid the overhead for the maintenance of class indexes.

Let's look at the following example:

```
Empty.java
/* Copyright (C) 2004 - 2009 Versant Inc. http://www.db4o.com */
package com.db4odoc.diagnostics;

import java.util.Calendar;
import java.text.DateFormat;

public class Empty  {

    public Empty()  {}

    public String CurrentTime()
    {
        Calendar cl = Calendar.getInstance();
        DateFormat df = DateFormat.getDateInstance();
        String time = df.format(cl.getTime());
        return time;
    }

    public String ToString()
    {
        return CurrentTime();
    }
}
```

```
DiagnosticExample.java: setEmptyObject
private static void setEmptyObject(ObjectContainer container) {
    Empty empty = new Empty();
    container.store(empty);
}
```

```
DiagnosticExample.java: testEmpty
private static void testEmpty()  {
    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().diagnostic().addListener(new DiagnosticToConsole());
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container=Db4oEmbedded.openFile(configuration, DB4O_FILE_NAME);
    try  {
        setEmptyObject(container);
    }
    finally  {
        container.close();
    }
}
```

Diagnostic message is produced when the execution point reaches

```
db.set(empty)
```

Empty class does not keep any information and can be left in the application code; there is no need to put it in the database.

LoadedFromClassIndex

This diagnostic class is provided to keep track of all the queried fields in your application that have no indexes. The output produced can give a comprehensive picture of index tuning required.

NativeQueryNotOptimized

This diagnostics object informs you that a Native Query cannot be optimized. It means that it will be run by instantiating all objects of the candidate class. Try to simplify your query expression. For an example let's look at a predicate using 2 different unrelated clauses.

```
ArbitraryQuery.java
/**/* Copyright (C) 2004 - 2009 Versant Inc. http://www.db4o.com */
package com.db4odoc.diagnostics;

import com.db4o.query.Predicate;

public class ArbitraryQuery extends Predicate<Pilot> {
    public int[] points;

    public ArbitraryQuery(int[] points) {
        this.points=points;
    }

    public boolean match(Pilot pilot) {
        for (int i = 0; i < points.length; i++) {
            if (((Pilot)pilot).getPoints() == points[i])
            {
                return true;
            }
        }
        return ((Pilot)pilot).getName().startsWith("Rubens");
    }
}
```

```
DiagnosticExample.java: queryPilot
private static void queryPilot(ObjectContainer container) {
    int[] i = new int[] {19,100};
    List result = container.query(new ArbitraryQuery(i));
    listResult(result);
}
```

```

DiagnosticExample.java: testArbitrary
private static void testArbitrary() {
    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().diagnostic().addListener(new DiagnosticToConsole());
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container=Db4oEmbedded.openFile(configuration, DB4O_FILE_NAME);
    try {
        Pilot pilot = new Pilot("Rubens Barrichello",99);
        container.store(pilot);
        queryPilot(container);
    }
    finally {
        container.close();
    }
}

```

UpdateDepthGreaterOne

UpdateDepth configuration setting allows you to specify the level of objects' enclosure where update command will still be valid:

Java:

```
configuration.updateDepth(depth)
```

This setting has a considerable impact on performance and can make the application very slow. It is recommended to keep the default configuration setting (UpdateDepth(0)) and specify UpdateDepth for selected classes, where cascaded update will be really useful:

Java:

```
configuration.objectClass(Car.class).updateDepth(3)
```

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 that giving information about queries that are running on un-indexed fields. 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:

```
IndexDiagListener.java
/**/* Copyright (C) 2004 - 2009 Versant Inc. http://www.db4o.com */
package com.db4odoc.diagnostics;

import com.db4o.diagnostic.*;

public class IndexDiagListener implements DiagnosticListener
{
    public void onDiagnostic(Diagnostic d) {
        if (d.getClass().equals(LoadedFromClassIndex.class)) {
            System.out.println(d.toString());
        }
    }
}
```

The following command will install the new listener:

Java:

```
configuration.diagnostic().addListener(new IndexDiagListener())
```

We can check the efficacy of IndexDiagListener using queries from the previous paragraphs:

```
DiagnosticExample.java: testIndexDiagnostics
private static void testIndexDiagnostics() {
    new File(DB4O_FILE_NAME).delete();

    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().diagnostic().addListener(new IndexDiagListener());
    configuration.common().updateDepth(3);

    ObjectContainer container=Db4oEmbedded.openFile(configuration, DB4O_FILE_NAME);
    try {
        Pilot pilot1 = new Pilot("Rubens Barrichello",99);
        container.store(pilot1);
        Pilot pilot2 = new Pilot("Michael Schumacher",100);
        container.store(pilot2);
        queryPilot(container);
        setEmptyObject(container);
        Query query = container.query();
        query.constrain(Pilot.class);
        query.descend("points").constrain(new Integer(99));
        List result = query.execute();
        listResult(result);
    }
    finally {
        container.close();
    }
}
```

```
}
```

Potentially this piece of code triggers all the diagnostic objects, but we are getting only index warning messages due to IndexDiagListener.

DescendIntoTranslator

Translator API provides a special way of storing and retrieving objects. In fact the actual class is not stored in the database. Instead the information from that class is stored in a primitive object (object array) and the class is recreated during instantiation or activation.

Let's look how queries handle translated classes. Diagnostics system will help us to see, what is going on.

In our example class Car is configured to be saved and retrieved with CarTranslator class. CarTranslator saves only car model information appending it with the production date.

```
CarTranslator.java
/**Copyright (C) 2004 - 2009 Versant Inc. http://www.db4o.com */
package com.db4odoc.diagnostics;

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

public class CarTranslator
    implements ObjectConstructor {
    public Object onStore(ObjectContainer container,
        Object applicationObject) {
        Car car = (Car) applicationObject;

        String fullModel;
        if (hasYear(car.getModel())) {
            fullModel = car.getModel();
        } else {
            fullModel = car.getModel() + getYear(car.getModel());
        }
        return new Object[] {fullModel};
    }

    private String getYear(String carModel) {
        if (carModel.equals("BMW")) {
            return " 2002";
        } else {
            return " 1999";
        }
    }

    private boolean hasYear(String carModel) {
        return false;
    }
}
```

```

public Object onInstantiate(ObjectContainer container, Object storedObject) {
    Object[] raw=(Object[])storedObject;
    String model=(String)raw[0];
    return new Car(model);
}

public void onActivate(ObjectContainer container,
    Object applicationObject, Object storedObject) {
}

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

```

Let's clean our database and store 2 cars:

```

DiagnosticExample.java: storeTranslatedCars
private static void storeTranslatedCars() {
    new File(DB4O_FILE_NAME).delete();

    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().objectClass(Car.class).translate(new CarTranslator());
    configuration.common().objectClass(Car.class).callConstructor(true);

    ObjectContainer container = Db4oEmbedded.openFile(configuration, DB4O_FILE_NAME);
    try {
        Car car1 = new Car("BMW");
        System.out.println("ORIGINAL: " + car1);
        container.store(car1);
        Car car2 = new Car("Ferrari");
        System.out.println("ORIGINAL: " + car2);
        container.store(car2);
    } catch (Exception exc) {
        System.out.println(exc.toString());
        return;
    } finally {
        container.close();
    }
}

```

We can check the contents of our database with the following method:

```

DiagnosticExample.java: retrieveTranslatedCars
private static void retrieveTranslatedCars() {
    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().diagnostic().addListener(new TranslatorDiagListener());
}

```

```

        configuration.common().objectClass(Car.class).translate(new CarTranslator());
        configuration.common().objectClass(Car.class).callConstructor(true);
        ObjectContainer container = Db4oEmbedded.openFile(configuration, DB4O_FILE_NAME);
    try {
        List result = container.query(Car.class);
        listResult(result);
    } finally {
        container.close();
    }
}

```

TranslatorDiagListener is implemented to help us filter only those diagnostic messages, that concern translated classes (filtering diagnostics messages is explained in [Diagnostic Messages Filter](#) chapter).

We did not get any diagnostic messages here and the result shows the stored cars with extended model values.

To test Native Queries we will use the predicate, which retrieves only cars, produced in year 2002:

```

NewCarModel.java
/** Copyright (C) 2004 - 2009 Versant Inc. http://www.db4o.com */
package com.db4odoc.diagnostics;

import com.db4o.query.Predicate;

public class NewCarModel extends Predicate<Car> {
    public boolean match(Car car) {
        return ((Car)car).getModel().endsWith("2002");
    }
}

```

```

DiagnosticExample.java: retrieveTranslatedCarsNQ
private static void retrieveTranslatedCarsNQ() {
    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().diagnostic().addListener(new TranslatorDiagListener());
    configuration.common().objectClass(Car.class).translate(new CarTranslator());
    configuration.common().objectClass(Car.class).callConstructor(true);
    ObjectContainer container = Db4oEmbedded.openFile(configuration, DB4O_FILE_NAME);
    try {
        List result = container.query(new NewCarModel());
        listResult(result);
    } finally {
        container.close();
    }
}

```

A diagnostic message should appear pointing out, that the query is not correct in our case. Let's try to correct it using unoptimized NQ and evaluations.

```
DiagnosticExample.java: retrieveTranslatedCarsNQUnopt
private static void retrieveTranslatedCarsNQUnopt() {
    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().optimizeNativeQueries(false);
    configuration.common().diagnostic().addListener(new TranslatorDiagListener());
    configuration.common().objectClass(Car.class).translate(new CarTranslator());
    configuration.common().objectClass(Car.class).callConstructor(true);
    ObjectContainer container = Db4oEmbedded.openFile(configuration, DB4O_FILE_NAME);
    try {
        List result = container.query(new NewCarModel());
        listResult(result);
    } finally {
        container.close();
    }
}
```

We will use simple evaluation to check our cars:

```
CarEvaluation.java
/** Copyright (C) 2004 - 2009 Versant Inc. http://www.db4o.com */
package com.db4odoc.diagnostics;

import com.db4o.query.*;

public class CarEvaluation implements Evaluation {
    public void evaluate(Candidate candidate)
    {
        Car car=(Car)candidate.getObject();
        candidate.include(car.getModel().endsWith("2002"));
    }
}
```

```
DiagnosticExample.java: retrieveTranslatedCarsSODAEv
private static void retrieveTranslatedCarsSODAEv() {
    EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
    configuration.common().diagnostic().addListener(new TranslatorDiagListener());
    configuration.common().objectClass(Car.class).translate(new CarTranslator());
    configuration.common().objectClass(Car.class).callConstructor(true);
    ObjectContainer container = Db4oEmbedded.openFile(configuration, DB4O_FILE_NAME);
    try {
        Query query = container.query();
        query.constrain(Car.class);
        query.constrain(new CarEvaluation());
        List result = query.execute();
```

```
        listResult(result);
    } finally {
    container.close();
}
}
```

In both cases we the results are correct. Native Query optimization cannot be used with the translated classes, because the actual values of the translated fields are only known after instantiation and activation. That also means that translated classes can have a considerable impact on database performance and should be used with care.

Native Query Optimization

Native Queries will run out of the box in any environment. If optimization is turned on (default) Native Queries will be converted to [SODA](#) where this is possible, allowing 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 Java bytecode and .NET IL code are 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:

Java:

```
List<Pilot> pilots = container.query(new Predicate<Pilot>() {
    public boolean match(Pilot pilot) {
        return pilot.getName().equals("Michael Schumacher")
        && pilot.getPoints() == 100;
    }
});
```

First of all the following code will be extracted:

```
query#constrain(Pilot)
```

Then a more complex analysis will be run to convert the contents of the #match method into a SODA-understandable syntax. On a simple example it is easy to see what will happen:

Java:

```
return pilot.getName().equals("Michael Schumacher") && pilot.getPoints() == 100;
```

easily converts into:

```
CANDIDATE.name == "Michael Schumacher"  
CANDIDATE.points == 100
```

NQ Optimization For Java

NQ optimisation on Java requires db4onqopt.jar and bloat.jar to be present in the CLASSPATH. The Native Query optimizer is still under development to eventually "understand" all Java constructs. Current optimization supports the following constructs well:

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

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

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

db4o for Java supplies three different possibilities to run optimized native queries, optimization at

1. [query execution time](#)
2. [deployment time](#)
3. [class loading time](#)

For more information on NQ optimization see [Monitoring Optimization](#).

Optimization at Query Execution Time

Note: This will not work with JDK1.1.

To enable code analysis and optimization of native query expressions at query execution time, you just have to add db4o-7.12-nqopt.jar and bloat-1.x.jar to your CLASSPATH. Optimization can be turned on and off with the following configuration setting:

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

NQ Optimization At Load Time

Note: This will not work with JDK1.1.

Native Query predicates can be optimized when they are loaded into JVM. In order to do that you should make use of db4o [Enhancement Tools](#).

The idea is very simple:

- you create your application without any worries about NQ optimization
- when the application is ready, you use a special starter class, which calls a special class-loader to instrument your predicates and start the application.

Let's look how this is done on an example. We will use a well-known [Pilot](#) class, store it and use NQ to retrieve it:

```
NQExample.java: main
public static void main(String[] args)  {
    storePilots();
    selectPilot5Points();
}
```

```
NQExample.java: storePilots
private static void storePilots()  {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = database(configureNQ());
    if (container != null)  {
        try  {
            Pilot pilot;
            for (int i = 0; i < OBJECT_COUNT; i++)  {
                pilot = new Pilot("Test Pilot #" + i, i);
                container.store(pilot);
            }
            for (int i = 0; i < OBJECT_COUNT; i++)  {
                pilot = new Pilot("Professional Pilot #" + (i + 10), i + 10);
                container.store(pilot);
            }
            container.commit();
        } catch (Db4oException ex)  {
            System.out.println("Db4o Exception: " + ex.getMessage());
        } catch (Exception ex)  {
            System.out.println("System Exception: " + ex.getMessage());
        } finally  {
            closeDatabase();
        }
    }
}
```

```
}
```

```
NQExample.java: selectPilot5Points
private static void selectPilot5Points()  {
    ObjectContainer container = database(configureNQ());
    if (container != null)  {
        try  {
            List<Pilot> result = container.query(new Predicate<Pilot>()  {
                public boolean match(Pilot pilot)  {
                    // pilots with 5 points are included in the
                    // result
                    return pilot.getPoints() == 5;
                }
            });
            listResult(result);
        } catch (Exception ex)  {
            System.out.println("System Exception: " + ex.getMessage());
        } finally  {
            closeDatabase();
        }
    }
}
```

We will need to create a starter class, which will call the main method of the NQExample:

```
NQEnhancedStarter.java: main
public static void main(String[] args) throws Exception  {
    // Create class filter to point to the predicates to be optimized
    ClassFilter filter = new ByNameClassFilter("com.db4odoc.nqoptimize.", true);
    // Create NQ optimization class edit
    BloatClassEdit[] edits = { new TranslateNQToSODAEedit() };
    URL[] urls = { new File("/work/worksaces/db4o/nqtest/bin").
    toURI().toURL() };
    // launch the application using the class edit and the filter
    Db4oInstrumentationLauncher.launch(edits, urls,
    NQExample.class.getName(), new String[] {});
}
```

That's all. Now you can run your application using NQEnhancedStarter and all the predicated will be optimized while they are loaded. This will also save time on optimization at runtime.

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);
}
}

```

NQ Optimization At Build Time

Note: Instrumented optimized classes will work with JDK1.1, but the optimization process itself requires at least JDK 1.3.

In the [previous topic](#) we discussed how NQ optimization can be enabled on classes while they are loaded. In this topic we will look at even more convenient and performant way of enhancing classes to optimize NQ: during application build time.

For our example we will take the same classes as in the [previous example](#), with the exception of NQEnhancedStarter class, which won't be needed for build-time enhancement. Its functionality will be fulfilled by the build script. For this example we will create an ant script, which should be run after the classes (or jar) is built.

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 NQAntClassEditFactory to create TranslateNQToSODAEdit (can be based on class filter)
- Call Db4oFileEnhancerAntTask#execute, which will call Db4oClassInstrumenter#enhance passing the previously created TranslateNQToSODAEdit to optimize NQ in the supplied classes.

This can be done with the following script:

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

<!--
  NQ optimization build time enhancement sample.
-->

<project name="nqenhance" 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="native-query"
  classname="com.db4o.nativequery.main.NQAntClassEditFactory"
```

```

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 targetdir="${basedir}/enhanced-bin">

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

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

    </db4o-enhance>

</target>

</project>

```

In order to test this script:

- Create a new project, consisting of NQExample and Pilot classes from the [previous example](#)
- Add lib folder to the project root and copy the following jars from db4o distribution:
 - bloat-1.0.jar
 - db4o-7.12-classedit.jar
 - db4o-7.12-java5.jar
 - db4o-7.12-nqopt.jar
 - db4o-7.12-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-7.12-java5.jar

java com.db4odoc.nqoptimize.NQExample

```

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 TA instrumentation 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](#).

Monitoring Optimization

This feature is not complete and can only be used for experiments

Currently you can only attach a listener to the ObjectContainer:

```
NQExample.java: nqListener
public static void nqListener()  {
    ObjectContainer db = Db4o.openFile(DBFILENAME);
    ((InternalObjectContainer) db).getNativeQueryHandler().addListener(
        new Db4oQueryExecutionListener()  {
            public void notifyQueryExecuted(NQOptimizationInfo info)  {
                System.err.println(info);
            }
        });
}
```

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

ObjectContainerBase.UNOPTIMIZED ("UNOPTIMIZED")

if the predicate could not be optimized and is run in unoptimized mode

ObjectContainerBase.PREOPTIMIZED ("PREOPTIMIZED")

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

ObjectContainerBase.DYNOPTIMIZED ("DYNOPTIMIZED")

if the predicate was optimized at query execution time

Utility Methods

In this chapter we will review utility methods provided by extended ObjectContainer 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:

Java: ObjectContainer.set(peekPersistedObject)

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;
        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 it's 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

This topic set contains a collection of the most-common db4o usage pitfalls.

More Reading:

- [Anonymous Classes](#)
- [Equality Comparison](#)
- [Reference Cache In Client-Server Mode](#)
-
- [Transparent Activation](#)
- [Transparent Persistence](#)
- [Activation Depth](#)
- [Classloader And Generic Classes](#)
- [Client-Server Timeouts](#)
- [Dangerous Practices](#)
- [Native Queries](#)
- [Persistent Hashtables](#)
- [Update Depth](#)
- [Working With Large Amounts Of Data](#)

Anonymous Classes

Anonymous classes can be 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 marshalled to "lightweight" transport db4o containers together with the graph of referenced objects. The catch here is that in the bytecode anonymous classes contain a reference to their parent class, meaning that the parent class will be marshalled as well. Normally this is not an issue, but it can become a problem if the parent class can't be instantiated without a constructor or when callConstructors configuration setting is set to true.

For example, for the following class:

```
ExceptionExample.java
package com.db4odoc.exceptionsonnotstorable;
```

```

import java.util.*;

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

public class ExceptionExample {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }

    private final ObjectContainer _db;
    // more (non-persistable) state

    public ExceptionExample(String hostName, int port, String userName, String password) {
        Configuration config = Db4o.newConfiguration();
        _db = Db4o.openClient(config, hostName, port, userName, password);
    }

    public List<Item> retrieveItems() {
        ObjectSet<Item> result = _db.query(new Predicate<Item>() {
            public boolean match(Item item) {
                return true;
            }
        }, new Comparator<Item>() {
            public int compare(Item one, Item two) {
                return one.getName().compareTo(two.getName());
            }
        });
        return new ArrayList<Item>(result);
    }

}

```

The bytecode for the Predicate and the Comparator will produce 2 separate classes, holding a reference to ExceptionExample class:

```

class com.db4odoc.exceptionsonnotstorabile.ExceptionExample$1 extends com.db4o.query.-Predicate {
    final synthetic com.db4odoc.exceptionsonnotstorabile.ExceptionExample this$0;
    ExceptionExample$1(com.db4odoc.exceptionsonnotstorabile.ExceptionExample arg0);
    ...
}

class com.db4odoc.exceptionsonnotstorabile.ExceptionExample$2 implements java.util.C-
omparator {
    final synthetic com.db4odoc.exceptionsonnotstorabile.ExceptionExample this$0;
    ExceptionExample$2(com.db4odoc.exceptionsonnotstorabile.ExceptionExample arg0);
}

```

```
...
```

```
}
```

The code above will fail if both exceptionsOnNotStorable and callConstructors are set to true (in the default configuration callConstructors is false). Note, that if ExceptionExample class were Storable - there would be no issue either.

Equality Comparison

Db4o uses reference cache for quick access to persistent objects. Each persistent object is guaranteed to have only one instance in the object reference cache independently of whether it was saved or retrieved. You can retrieve the same object several times with different querying methods, but you will still get references to the same object, so that ref(1) == ref(2) == ... == ref(n).

In the same time it means that 2 objects, for example, one created in the runtime and another retrieved from the database, with the same data (field values) won't be equal for db4o.

There are 2 ways to compare db4o objects by data:

- using [QBE](#);
- implementing a suitable `equals` method.

Let's save an object to the database and try the above mentioned methods in practice.

```
EqualityExample.java: storePilot
private static void storePilot() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            Pilot pilot = new Pilot("Kimi Raikkonen", 100);
            container.store(pilot);
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}
```

More Reading:

- [QBE](#)
- [Using Equals](#)

QBE

If we have a prototype and want to find out if there is an object in the database with the same field values, we can simply use QBE:

```
EqualityExample.java: retrieveEqual
private static void retrieveEqual() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            ObjectSet result = container.queryByExample(
new Pilot("Kimi Raikkonen", 100));
            if (result.size() > 0) {
                System.out.println("Found equal object: " +
result.next().toString());
            } else {
                System.out.println(
"No equal object exist in the database");
            }
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}
```

This method allows to combine retrieval and comparing in one operation.

Using Equals

In some cases you can't use QBE as a retrieval method. In these cases you must override the object's `equals` method to allow you to compare objects data. For example:

```
Pilot.java: equals
public boolean equals(Pilot p) {
    return name.equals(p.getName()) && points == p.getPoints();
}
```

Note, that you must implement `hashCode/GetHashCode` method, when you implement `equals`:

```
Pilot.java: hashCode
public int hashCode() {
    return name.hashCode() ^ points;
}
```

Now we can use the `equals` method to compare an object from the database to an object prototype:

```
EqualityExample.java: testEquality
private static void testEquality() {
    ObjectContainer container = database();
    if (container != null) {
        try {
            ObjectSet<Pilot> result = container.query(new Predicate<Pilot>() {
                public boolean match(Pilot pilot) {
                    return pilot.getName().equals("Kimi Raikkonen") &&
                        pilot.getPoints() == 100;
                }
            });
            Pilot obj = (Pilot)result.next();
            Pilot pilot = new Pilot("Kimi Raikkonen", 100);
            String equality = obj.equals(pilot) ? "equal" : "not equal";
            System.out.println("Pilots are " + equality);
        } catch (Exception ex) {
            System.out.println("System Exception: " + ex.getMessage());
        } finally {
            closeDatabase();
        }
    }
}
```

Reference Cache In Client-Server Mode

Db4o uses 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.

The following example demonstrates this behaviour.

```
InconsistentGraphExample.java: main
public static void main(String[] args) throws IOException {
    new InconsistentGraphExample().run();
}
```

```
InconsistentGraphExample.java: run
public void run() throws IOException, DatabaseFileLockedException {
    new File(DB4O_FILE_NAME).delete();
    ObjectServer server = Db4o.openServer(DB4O_FILE_NAME, PORT);
    try {
        server.grantAccess(USER, PASSWORD);

        ObjectContainer client1 = server.openClient();
        ObjectContainer client2 = server.openClient();
```

```

if (client1 != null && client2 != null)  {
    try  {
        // wait for the operations to finish
        waitForCompletion();

        // save pilot with client1
        Car client1Car = new Car("Ferrari", 2006, new Pilot(
            "Schumacher"));
        client1.store(client1Car);
        client1.commit();
        System.out.println("Client1 version initially: " + client1Car);
        waitForCompletion();

        // retrieve the same pilot with client2
        Car client2Car = (Car) client2.query(Car.class).next();
        System.out.println("Client2 version initially: " + client2Car);

        // delete the pilot with client1
        Pilot client1Pilot = (Pilot)client1.query(Pilot.class).next();
        client1.delete(client1Pilot);
        // modify the car, add and link a new pilot with client1
        client1Car.setModel(2007);
        client1Car.setPilot(new Pilot("Hakkinen"));
        client1.store(client1Car);
        client1.commit();

        waitForCompletion();
        client1Car = (Car) client1.query(Car.class).next();
        System.out.println("Client1 version after update: " + client1Car);

        System.out.println();
        System.out.println(
"client2Car still holds the old object graph in its reference cache");
        client2Car = (Car) client2.query(Car.class).next();
        System.out.println("Client2 version after update: " + client2Car);
        ObjectSet result = client2.query(Pilot.class);
        System.out.println(
"Though the new Pilot is retrieved by a new query: ");
        listResult(result);

        waitForCompletion();
    } catch (Exception ex)  {
        ex.printStackTrace();
    } finally  {
        closeClient(client1);
        closeClient(client2);
    }
}
} catch (Exception ex)  {
    ex.printStackTrace();
} finally  {
    server.close();
}

```

```
    }  
}
```

In order to make the objects consistent on each client you must refresh them from the server when they get updated. This can be done by using [Committed Callbacks](#).

Download example code:

[Java](#)

Transparent Activation

[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 Activatable Objects

Problem

When TA is enabled Activatable objects are activated transparently on request, whereas not Activatable objects are fully activated. This is done to keep consistency in persistent objects behaviour. However, there is a potential danger of activating too many unnecessary objects and wasting system resources. You will experience lower performance and in the worst case you can end up with OutOfMemory error.

Solution

Make all persistent objects Activatable. This can be done through using load time (Java only) or build time Transparent Activation Enhancement. For more information see [Transparent Activation Framework](#) documentation.

Collections Activation

Problem

Current implementation of TA Framework does not support lazy activation of collection members, i.e. the whole collection will be activated as one object on the first request. However, this only applies to collection object itself, i.e. Activatable members of the collection will be activated in a transparent way.

Solution

Use db4o proprietary collections: com.db4o.collections package in Java and Db4o-objects.Db4o.Collections in .NET

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 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

Problem

Debugging instrumented classes may not work 100% correct both on Java and .NET. Some of the observed problems:

- In Visual Studio setting a breakpoint in the area effected by instrumentation seems to be not possible
- In Eclipse instrumented bytecode might be not displayed correctly in the debug mode

Solution

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):

```
TAEexample.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();
}
}

```

Transparent Persistence

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 `Integer.MAX_VALUE` 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 TP 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();
            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 TP.

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;
}

```

Activation Depth

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 2 main pitfalls that you must be aware about:

1. An object retrieved from the database is null.

This happens if the activation level is lower than needed. For example:

class Pilot has field Car:

```

Pilot {
    Car car;
}

```

and is saved to a database. Then `pilot` object us retrieved from the database with the activation depth is set to 0. In this case `pilot.car` will be equal to null and can be incorrectly interpreted.

2. Activation depth is set [globally](#) to a high value or is set to [cascadeOnActivate](#) for a heavily used object with a deep structure. This will result in a huge performance penalty and should

be avoided.

The automatic solution of the Activation issues is provided by [Transparent Activation Framework](#). However, understanding of Activation is still important.

For more information on activation see:

- [Global Activation Settings](#)
- [Object-Specific Activation](#)
- [Transparent Activation Framework](#)
- [Activation strategies](#)

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, use

```
Configuration#reflectWith(new JdkReflector(classLoader))  
where classLoader is the classloader of your application classes.
```

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](#).

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 initial db4o CS implementation used one-thread clients and connection was terminated when there was a timeout and a post-timeout check-up message could not get a response from the other side. However this approach failed for .NET implementation, as .NET sockets upon timeout continue to send and receive messages, but timeout settings are not respected anymore, which in fact leads to a very high CPU usage.

The next approach was to create a separate housekeeping thread on the server, which will send messages to the client regularly, thus checking if the client is still alive and in the same time giving the client information about the server. Unfortunately, the problem described above can occur to the housekeeping thread too.

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 are two small downsides:

1. Clients are not single-threaded anymore
2. If an action on the server takes longer than the socket timeout, the connection will be closed.

In the latter case you can adjust the behaviour of db4o with the following configuration settings:

Java:

```
configuration.clientServer().timeoutServerSocket(int milliseconds)
```

```
configuration.clientServer().timeoutClientSocket(int milliseconds)
```

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.

Dangerous Practices

Db4o databases are well protected against corruption. However some specific configurations can make your database file vulnerable.

1. [Configuration#lockDatabaseFile\(false\)](#)

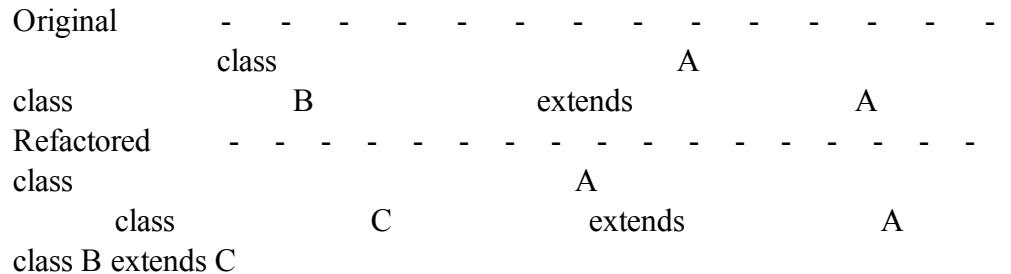
Java platforms before JDK1.4 do not prevent concurrent access to a file from different JVM. If database file locking is turned off on these platforms, concurrent write access to the same database file from different JVM sessions will corrupt the database file immediately. Do not use this setting unless your application logic guarantees that only one VM session can access your database file at a time. For more information see [No lock file thread](#).

3. [NonFlushingStorage](#)

In order to ensure ACID transaction db4o uses a [special strategy](#), which relies on the order of writes to the storage medium. On operating systems that use in-memory file caching, the OS cache may revert the order of writes to optimize file performance. db4o can enforce the correct order by flushing file buffers after every step of transaction commit. Turning this setting off puts you in potential danger of data corruption if a system or hardware failure occurs during commit.

4. The following refactorings are incompatible with db4o:

1. Adding classes within a class hierarchy or above a class hierarchy. Example:



2. Removing a class from the top or within a class hierarchy. Example:

Original -----
class A
class B extends A

```

class C extends B
Refactored -----
class A
class C extends A
3. Changing the type of a field to be an array or back. Example:
Original -----
class Foo { String bar; }
Refactored -----
class Foo { String [] bar; }

```

If you apply such a refactoring, you will not be able to read existing objects correctly.

More information on refactorings see [Refactoring and Schema Evolution](#)

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 2 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 NQ 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 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.

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 - hash code - based on a specific algorithm which can be 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. Otherwise you may want to invent your own collision resolution algorithm.

Update Depth

db4o update behavior is regulated by [Update Depth](#). Understanding Update Depth will help you to improve performance and avoid unnecessary memory usage.

When 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.

The following settings should be used with a special care and only with a good reason to do so:

1. Global deep update depth setting:

Java:

```
configuration.updateDepth(Integer.MaxValue);
```

This setting causes ALL objects in the database to be updated to the lowest possible level on each update.

2. Cascade on update:

Java:

```
configuration.objectClass(Item.class).cascadeOnUpdateDepth(true);
```

This setting will cause an update to the lowest level for Item class only. This can be unnecessary and therefore undesired in order to save system resources.

For more detailed information, please, refer to [Update Depth](#).

Working With Large Amounts Of Data

db4o is designed to manage 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](#):

Java:

```
configuration.blockSize(blockSize)
```

As a parameter you can specify any value between 1 and 127. The resulting maximum database file size will be a multiple of 2GB. A recommended setting for large database files is 8, since internal pointers have this length. Using `blockSize` the maximum database file size will be 16GB. The above method has to be called before an Object Container is opened the first time. During the lifetime of an Object Container the setting will have to stay the same. Since Defragment copies all objects to a new Object Container, it can be used to change the blockSize of an existing database:

Java:

```
Defragment.defrag("filename.db4o")
```

Performance

Navigation access times to objects and the performance of access by internal IDs remains constant, no matter how large database files are. Query performance on unindexed objects drops linearly with an increasing number of objects per class. Query performance on a large number of objects can be dramatically improved by [using indexes](#):

Java:

```
configuration.objectClass(Foo.class).objectField("bar").indexed(true);
```

.NET:

```
configuration.ObjectClass(typeof(Foo)).ObjectField("bar").Indexed(true);
```

db4o storage performance is very good. It is recommended to run your own benchmarks with large amounts of data to check the overall performance on your particular class hierarchy.

Client-Server

Client/Server mode is de-facto standard for any modern database. However there is a big difference between relational and object databases functionality in client-server mode.

With RDBMS everything is pretty straightforward: data is kept on a server and SQL commands generated on a client are used to operate them.

In db4o world SQL is an alien and querying syntax is based on class definitions. Therefore class libraries synchronization between client and server becomes essential.

More Reading:

- [Embedded](#)
- [Networked](#)
- [Native Queries In Client-Server Mode](#)
- [Server Without Persistent Classes Deployed](#)
- [Semaphores](#)
- [Pluggable Sockets](#)
- [Remote Code Execution](#)
- [Concurrency Control](#)
- [Messaging](#)
- [Batch Mode](#)

Embedded

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

```
ClientServerExample.java: accessLocalServer
private static void accessLocalServer()  {
    ObjectServer server = Db4oClientServer.openServer(Db4oClientServer
        .newServerConfiguration(), DB4O_FILE_NAME, 0);
    try  {
        ObjectContainer client = server.openClient();
        // Do something with this client, or open more clients
    }
```

```
        client.close();
    } finally {
        server.close();
    }
}
```

Again, we will delegate opening and closing the server to our environment to focus on client interactions.

```
ClientServerExample.java: queryLocalServer
private static void queryLocalServer(ObjectServer server)  {
    ObjectContainer client = server.openClient();
    listResult(client.queryByExample(new Car(null)));
    client.close();
}
```

The transaction level in db4o is *read committed*. However, each client container maintains its own weak reference cache of already known objects. To make all changes committed by other clients immediately, we have to explicitly refresh known objects from the server. We will delegate this task to a specialized version of our listResult() method.

```
ClientServerExample.java: listRefreshedResult
private static<T> void listRefreshedResult(ObjectContainer container,
    List<T> result, int depth)  {
    System.out.println(result.size());
    for (T t : result)  {
        container.ext().refresh(t, depth);
        System.out.println(t);
    }
}
```

```
ClientServerExample.java: demonstrateLocalReadCommitted
private static void demonstrateLocalReadCommitted(ObjectServer server)  {
    ObjectContainer client1 = server.openClient();
    ObjectContainer client2 = server.openClient();
    Pilot pilot = new Pilot("David Coulthard", 98);
    List<Car> result = client1.queryByExample(new Car("BMW"));
    Car car = result.queryByExample(0);
    car.setPilot(pilot);
    client1.store(car);
    listResult(client1.queryByExample(new Car(null)));
    listResult(client2.queryByExample(new Car(null)));
    client1.commit();
    listResult(client1.queryByExample(Car.class));
```

```

        listRefreshedResult(client2, client2.queryByExample(Car.class), 2);
        client1.close();
        client2.close();
    }
}

```

Simple rollbacks just work as you might expect now.

```

ClientServerExample.java: demonstrateLocalRollback
private static void demonstrateLocalRollback(ObjectServer server) {
    ObjectContainer client1 = server.openClient();
    ObjectContainer client2 = server.openClient();
    List<Car> result = client1.queryByExample(new Car("BMW"));
    Car car = result.queryByExample(0);
    car.setPilot(new Pilot("Someone else", 0));
    client1.store(car);
    listResult(client1.queryByExample(new Car(null)));
    listResult(client2.queryByExample(new Car(null)));
    client1.rollback();
    client1.ext().refresh(car, 2);
    listResult(client1.queryByExample(new Car(null)));
    listResult(client2.queryByExample(new Car(null)));
    client1.close();
    client2.close();
}
}

```

Networked

It's only a small step from an [embedded server](#) towards operating db4o over a TCP/IP network.

More Reading:

- [Network Server](#)
- [Out-of-band Signaling](#)
- [Simple db4o Server](#)

Network Server

In order to make the [embedded server](#) operate over a TCP/IP network, we just need to specify a port number greater than zero and set up one or more accounts for our client(s).

```

ClientServerExample.java: accessRemoteServer
private static void accessRemoteServer() throws IOException {
    ObjectServer server = Db4oClientServer.openServer(Db4oClientServer
        .newServerConfiguration(), DB4O_FILE_NAME, PORT);
}
}

```

```

server.grantAccess(USER, PASSWORD);
try {
    ObjectContainer client = Db4oClientServer.openClient(
        Db4oClientServer.newClientConfiguration(), "localhost",
        PORT, USER, PASSWORD);
    // Do something with this client, or open more clients
    client.close();
} finally {
    server.close();
}
}

```

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

```

ClientServerExample.java: queryRemoteServer
private static void queryRemoteServer(int port, String user, String password)
    throws IOException {
    ObjectContainer client = Db4oClientServer.openClient(Db4oClientServer
        .newClientConfiguration(), "localhost", port, user, password);
    listResult(client.queryByExample(new Car(null)));
    client.close();
}

```

Everything else is absolutely identical to the [local server examples](#).

```

ClientServerExample.java: demonstrateRemoteReadCommitted
private static void demonstrateRemoteReadCommitted(int port, String user,
    String password) throws IOException {
    ObjectContainer client1 = Db4oClientServer.openClient(Db4oClientServer
        .newClientConfiguration(), "localhost", port, user, password);
    ObjectContainer client2 = Db4oClientServer.openClient(Db4oClientServer
        .newClientConfiguration(), "localhost", port, user, password);
    Pilot pilot = new Pilot("Jenson Button", 97);
    List<Car> result = client1.queryByExample(new Car(null));
    Car car = result.queryByExample(0);
    car.setPilot(pilot);
    client1.store(car);
    listResult(client1.queryByExample(new Car(null)));
    listResult(client2.queryByExample(new Car(null)));
    client1.commit();
    listResult(client1.queryByExample(new Car(null)));
    listRefreshedResult(client2, client2.queryByExample(Car.class), 2);
    client1.close();
    client2.close();
}

```

```

ClientServerExample.java: demonstrateRemoteRollback
private static void demonstrateRemoteRollback(int port, String user,
    String password) throws IOException {
    ObjectContainer client1 = Db4oClientServer.openClient(Db4oClientServer
        .newClientConfiguration(), "localhost", port, user, password);
    ObjectContainer client2 = Db4oClientServer.openClient(Db4oClientServer
        .newClientConfiguration(), "localhost", port, user, password);
    List<Car> result = client1.queryByExample(new Car(null));
    Car car = result.queryByExample(0);
    car.setPilot(new Pilot("Someone else", 0));
    client1.store(car);
    listResult(client1.queryByExample(new Car(null)));
    listResult(client2.queryByExample(new Car(null)));
    client1.rollback();
    client1.ext().refresh(car, 2);
    listResult(client1.queryByExample(new Car(null)));
    listResult(client2.queryByExample(new Car(null)));
    client1.close();
    client2.close();
}

```

Out-of-band Signaling

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

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

```

StartServer.java: runServer
/** */
 * opens the ObjectServer, and waits forever until close() is called or a
 * StopServer message is being received.
 */
public void runServer() {
    synchronized (this) {
        // Using the messaging functionality to redirect all
        // messages to this.processMessage
        ServerConfiguration config = Db4oClientServer.newServerConfiguration();
        config.networking().messageRecipient(this);

        ObjectServer db4oServer = Db4oClientServer.openServer(
            config, FILE, PORT);
        db4oServer.grantAccess(USER, PASS);

        // to identify the thread in a debugger
        Thread.currentThread().setName(this.getClass().getName());
    }
}

```

```

// We only need low priority since the db4o server has
// it's own thread.
Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
try {
    if (!stop) {
        // wait forever for notify() from close()
        this.wait(Long.MAX_VALUE);
    }
} catch (Exception e) {
    e.printStackTrace();
}
db4oServer.close();
}
}

```

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

```

StartServer.java: processMessage
/** */
* messaging callback
*
* @see com.db4o.messaging.MessageRecipient#processMessage(ObjectContainer,
*      Object)
*/
public void processMessage(MessageContext context, Object message) {
    if (message instanceof StopServer) {
        close();
    }
}

```

Db4o allows a client to send an arbitrary signal or message to a server by sending a plain user object to the server. The server will receive a callback message, including the object that came from the client. The server can interpret this message however it wants.

```

StopServer.java: main
/** */
* stops a db4o Server started with StartServer.
*
* @throws Exception
*/
public static void main(String[] args) {
    ObjectContainer objectContainer = null;
    try {
        // connect to the server
        objectContainer = Db4oClientServer.openClient(Db4oClientServer

```

```

        .newClientConfiguration(), HOST, PORT, USER, PASS);

    } catch (Exception e) {
        e.printStackTrace();
    }

    if (objectContainer != null) {

        // get the messageSender for the ObjectContainer
        MessageSender messageSender = objectContainer.ext().configure()
            .clientServer().getMessageSender();

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

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

```

Simple db4o Server

Let's implement a simple standalone db4o server with a special client that can tell the server to shut itself down gracefully on demand.

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

```

ServerSetup.java:

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

/**
 * EmbeddedConfiguration used for {@link StartServer} and {@link StopServer}.
 */
public interface ServerSetup {

    /**
     * the host to be used.
     * If you want to run the client server examples on two computers,
     * enter the computer name of the one that you want to use as server.
     */
    public String    HOST = "localhost";
    /**
     * the database file to be used by the server.
     */
    public String    FILE = "reference.db4o";
}

```

```

    /**
     * the port to be used by the server.
     */
    public int      PORT = 0xdb40;

    /**
     * the user name for access control.
     */
    public String   USER = "db4o";

    /**
     * the password for access control.
     */
    public String   PASS = "db4o";
}

```

Now we'll create the server:

```

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

import com.db4o.ObjectContainer;
import com.db4o.ObjectServer;
import com.db4o.cs.Db4oClientServer;
import com.db4o.cs.config.ServerConfiguration;
import com.db4o.messaging.MessageContext;
import com.db4o.messaging.MessageRecipient;

/** */
* starts a db4o server with the settings from {@link ServerSetup}. <br>
* <br>
* This is a typical setup for a long running server. <br>
* <br>
* The Server may be stopped from a remote location by running StopServer. The
* StartServer instance is used as a MessageRecipient and reacts to receiving an
* instance of a StopServer object. <br>
* <br>
* Note that all user classes need to be present on the server side and that all
* possible Db4oEmbedded.configure() calls to alter the db4o configuration need
* to be executed on the client and on the server.
*/
public class StartServer implements ServerSetup, MessageRecipient {

    /** */
    * setting the value to true denotes that the server should be closed
    */
    private boolean stop = false;

    /**

```

```

 * starts a db4o server using the configuration from
 * {@link ServerSetup}.
 */
public static void main(String[] arguments)  {
    new StartServer().runServer();
}

// end main

/** */
* opens the ObjectServer, and waits forever until close() is called or a
* StopServer message is being received.
*/
public void runServer()  {
    synchronized (this)  {
        // Using the messaging functionality to redirect all
        // messages to this.processMessage
        ServerConfiguration config = Db4oClientServer.newServerConfiguration();
        config.networking().messageRecipient(this);

        ObjectServer db4oServer = Db4oClientServer.openServer(
            config, FILE, PORT);
        db4oServer.grantAccess(USER, PASS);

        // to identify the thread in a debugger
        Thread.currentThread().setName(this.getClass().getName());

        // We only need low priority since the db4o server has
        // it's own thread.
        Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
        try  {
            if (!stop)  {
                // wait forever for notify() from close()
                this.wait(Long.MAX_VALUE);
            }
            } catch (Exception e)  {
                e.printStackTrace();
            }
            db4oServer.close();
        }
    }

// end runServer

/** */
* messaging callback
*
* @see com.db4o.messaging.MessageRecipient#processMessage(ObjectContainer,
*          Object)
*/
public void processMessage(MessageContext context, Object message)  {
    if (message instanceof StopServer)  {
        close();
    }
}

```

```

        }

    }

// end processMessage

/** */
 * closes this server.
 */
public void close() {
    synchronized (this) {
        stop = true;
        this.notify();
    }
}
// end close
}

```

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

```

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

import com.db4o.*;
import com.db4o.cs.Db4oClientServer;
import com.db4o.messaging.*;

/** */
* stops the db4o Server started with {@link StartServer}. <br>
* <br>
* This is done by opening a client connection to the server and by sending a
* StopServer object as a message. {@link StartServer} will react in it's
* processMessage method.
*/
public class StopServer implements ServerSetup {

/** */
* stops a db4o Server started with StartServer.
*
* @throws Exception
*/
public static void main(String[] args) {
    ObjectContainer objectContainer = null;
    try {

        // connect to the server
        objectContainer = Db4oClientServer.openClient(Db4oClientServer
            .newClientConfiguration(), HOST, PORT, USER, PASS);

    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

```

    }

    if (objectContainer != null)  {

        // get the messageSender for the ObjectContainer
        MessageSender messageSender = objectContainer.ext().configure()
            .clientServer().getMessageSender();

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

        // close the ObjectContainer
        objectContainer.close();
    }
}
// end main
}

```

Keywords:

start remote instance

Communication Timeouts

Native Queries In Client-Server Mode

A quite subtle problem may occur if you're using Native Queries as anonymous (or just non-static) inner classes in Client/Server mode. Anonymous/non-static inner class instances carry a synthetic field referencing their outer class instance - this is just Java's way of implementing inner class access to fields or final method locals of the outer class without introducing any notion of inner classes at all at the bytecode level. If such a non-static inner class predicate cannot be converted to S.O.D.A. form on the client, it will be wrapped into an evaluation and passed to the server, introducing the same problem already mentioned in the [evaluation chapter](#): db4o will try to transfer the evaluation, using the standard platform serialization mechanism. If this fails, it will just try to pass it to the server via db4o marshalling. However, this may fail, too, for example if the outer class references any local db4o objects like O ObjectContainer , etc., resulting in an ObjectNotStorableException.

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

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 [Object Manager](#).

Unfortunately in a server without persistent classes mode there are still some limitations:

- evaluation queries won't work
- native queries will only work if they can be optimized
- LINQ queries will only work if they can be optimized
- multidimensional arrays can not be stored.

The following topics provide examples of a server without persistent classes usage. We will use a server from the [ClientServer](#) example, a separate project for a client and deploy persistent classes only on the client side. In .NET you will need to test the examples on separate machines as the assembly information is loaded in CLR and shared.

More Reading:

- [Saving Objects](#)
- [QBE](#)
- [SODA](#)
- [Native Queries](#)
- [LINQ](#)
- [Evaluations](#)
- [Multidimensional Arrays](#)

Saving Objects

```
Client.java: savePilots
private static void savePilots() throws IOException {
    System.out.println("Saving Pilot objects without Pilot class on the server");
    ObjectContainer oc = Db4o.openClient("localhost", 0xdb40, "db4o",
```

```

    "db4o");
try {
    for (int i = 0; i < COUNT; i++) {
        oc.store(new Pilot("Pilot #" + i, i));
    }
} finally {
    oc.close();
}
}

```

You can check the saved objects using ObjectManager or the querying examples.

QBE

```

Client.java: getPilotsQBE
private static void getPilotsQBE() throws IOException {
    System.out.println("Retrieving Pilot objects: QBE");
    ObjectContainer oc = Db4o.openClient("localhost", 0xdb40, "db4o","db4o");
    try {
        ObjectSet result = oc.queryByExample(new Pilot(null,0));
        listResult(result);
    } finally {
        oc.close();
    }
}

```

SODA

```

Client.java: getPilotsSODA
private static void getPilotsSODA() throws IOException {
    System.out.println("Retrieving Pilot objects: SODA");
    ObjectContainer oc = Db4o.openClient("localhost", 0xdb40, "db4o","db4o");
    try {
        Query query = oc.query();

        query.constrain(Pilot.class);
        query.descend("points").constrain(5);

        ObjectSet result = query.execute();
        listResult(result);
    } finally {
        oc.close();
    }
}

```

Native Queries

We have checked in the SODA topic that SODA queries can work with a server without application classes deployed. So we can expect optimized Native Queries (converted to SODA under the hood) to work as well:

```
Client.java: getPilotsNative
private static void getPilotsNative() throws IOException {
    System.out.println("Retrieving Pilot objects: Native Query");
    ObjectContainer oc = Db4o.openClient("localhost", 0xdb40, "db4o","db4o");
    try {
        List <Pilot> result = oc.query(new Predicate<Pilot>() {
            public boolean match(Pilot pilot) {
                return pilot.getPoints() == 5;
            }
        });
        listResult(result);
    } finally {
        oc.close();
    }
}
```

Unfortunately, if the query cannot be optimized to SODA, the server needs to instantiate the classes to run the query. This is not possible if the class is not available

This query won't work:

```
Client.java: getPilotsNativeUnoptimized
private static void getPilotsNativeUnoptimized() throws IOException {
    System.out.println("Retrieving Pilot objects: Native Query Unoptimized");
    ObjectContainer oc = Db4o.openClient("localhost", 0xdb40, "db4o","db4o");
    try {
        List <Pilot> result = oc.query(new Predicate<Pilot>() {
            public boolean match(Pilot pilot) {
                return pilot.getPoints() % 2 == 0;
            }
        });
        listResult(result);
    } finally {
        oc.close();
    }
}
```

If you want to use unoptimized Native Queries in a client-server environment, you must make sure, that the Native Query classes are deployed on the server side. You can do so by implementing all predicate classes in a separate library and deploying it on both the client and the server side. See also [Usage Pitfalls](#).

Evaluations

Evaluations need to retrieve the actual object instance to be evaluated. That is why they do not work on a server without persistent classes:

```
Client.java: getPilotsEvaluation
private static void getPilotsEvaluation() throws IOException {
    System.out.println("Retrieving Pilot objects: Evaluation");
    ObjectContainer oc = Db4o.openClient("localhost", 0xdb40, "db4o", "db4o");
    try {
        Query query = oc.query();

        query.constrain(Pilot.class);
        query.constrain(new Evaluation() {
            public void evaluate(Candidate candidate) {
                Pilot pilot = (Pilot) candidate.getObject();
                candidate.include(pilot.getPoints() % 2 == 0);
            }
        });
        ObjectSet result = query.execute();
        listResult(result);
    } finally {
        oc.close();
    }
}
```

Multidimensional Arrays

Currently you can't use multidimensional array fields in a server without persistent classes setup.

```
RecordBook.java
/**/* Copyright (C) 2004 - 2006 Versant Inc. http://www.db4o.com */

package com.db4odoc.noclasses.client;

public class RecordBook {
    private String[][] notes;
    private int recordCounter;

    public RecordBook() {
        notes = new String[20][3];
        recordCounter = 0;
    }

    public void addRecord(String period, String pilotName, String note) {
        String[] tempArray = {period, pilotName, note};
        notes[recordCounter] = tempArray;
        recordCounter++;
    }
}
```

```

    }

    public String toString() {
        String temp;
        temp = "Record book: \n";
        for (int i=0; i<recordCounter;i++ ) {
            temp = temp + notes[i][0] + "/" + notes[i][1] + "/" + notes[i][2] + "\n";
        }
        return temp;
    }
}

```

```

Client.java: saveMultiArray
private static void saveMultiArray() throws IOException {
    System.out.println("Testing saving an object with multidimensional array field");
    ObjectContainer oc = Db4o.openClient("localhost", 0xdb40, "db4o","db4o");
    try {
        RecordBook recordBook = new RecordBook();
        recordBook.addRecord("September 2006", "Michael Schumacher", "last race");
        recordBook.addRecord("September 2006", "Kimi Raikkonen", "no notes");
        oc.store(recordBook);
    } finally {
        oc.close();
    }
}

```

```

Client.java: getMultiArray
private static void getMultiArray() throws IOException {
    System.out.println("Testing retrieving an object with multidimensional array field");
    ObjectContainer oc = Db4o.openClient("localhost", 0xdb40, "db4o","db4o");
    try {
        ObjectSet result = oc.queryByExample(new RecordBook());
        listResult(result);
    } finally {
        oc.close();
    }
}

```

Semaphores

db4o Semaphores are named flags that can only be owned by one client/transaction 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.

Semaphores can be set and released with the following two methods:

Java:

```
ExtObjectContainer#setSemaphore  
(String name, int waitMilliSeconds);  
ExtObjectContainer#releaseSemaphore(String name);
```

OO Languages Semaphores

The concept of db4o semaphores is very similar to the concept of synchronization in OO programming languages:

Java:

```
synchronized(monitorObject) {  
//           exclusive      code      block      here  
}
```

db4o Semaphore

Java:

```
if(objectContainer.ext().setSemaphore(semaphoreName,  
//           exclusive      code      block      here  
objectContainer.ext().releaseSemaphore(semaphoreName);  
}  
1000){
```

Although the principle of semaphores is very simple they are powerful enough for a wide range of usecases.

More Reading:

- [Locking Objects](#)
- [Ensuring Singletons](#)
- [Limiting the Number of Users](#)
- [Controlling Login Information](#)

Locking Objects

```
LockManager.java  
/**/* Copyright (C) 2004 - 2007 Versant Inc. http://www.db4o.com */  
  
package com.db4odoc.semaphores;  
  
import com.db4o.*;
```

```

import com.db4o.ext.*;

/** /**
 * This class demonstrates a very rudimentary implementation of
 * virtual "locks" on objects with db4o. All code that is intended to
 * obey these locks will have to call lock() and unlock().
 */
public class LockManager {

    private final String SEMAPHORE_NAME = "locked: ";

    private final int WAIT_FOR_AVAILABILITY = 300; // 300
                                                   // milliseconds

    private final ExtObjectContainer _objectContainer;

    public LockManager(ObjectContainer objectContainer) {
        _objectContainer = objectContainer.ext();
    }

    public boolean lock(Object obj) {
        long id = _objectContainer.getID(obj);
        return _objectContainer.setSemaphore(SEMAPHORE_NAME + id,
            WAIT_FOR_AVAILABILITY);
    }

    public void unlock(Object obj) {
        long id = _objectContainer.getID(obj);
        _objectContainer.releaseSemaphore(SEMAPHORE_NAME + id);
    }
}

```

Ensuring Singletons

```

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

package com.db4odoc.semaphores;

import com.db4o.*;
import com.db4o.query.*;

/** /**
 * This class demonstrates the use of a semaphore to ensure that only
 * one instance of a certain class is stored to an ObjectContainer.
 *
 * Caution !!! The getSingleton method contains a commit() call.
 */
public class Singleton {

    /** /**

```

```

* returns a singleton object of one class for an ObjectContainer.
* <br>
* <b>Caution !!! This method contains a commit() call.</b>
*/
public static Object getSingleton(
    ObjectContainer objectContainer, Class clazz)  {

    Object obj = queryForSingletonClass(objectContainer, clazz);
    if (obj != null)  {
        return obj;
    }

    String semaphore = "Singleton#getSingleton_"
        + clazz.getName();

    if (!objectContainer.ext().setSemaphore(semaphore, 10000))  {
        throw new RuntimeException("Blocked semaphore "
            + semaphore);
    }

    obj = queryForSingletonClass(objectContainer, clazz);

    if (obj == null)  {

        try  {
            obj = clazz.newInstance();
        } catch (InstantiationException e)  {
            e.printStackTrace();
        } catch (IllegalAccessException e)  {
            e.printStackTrace();
        }

        objectContainer.store(obj);

        /**
         * !!! CAUTION !!! There is a commit call here.
         *
         * The commit call is necessary, so other transactions can
         * see the new inserted object.
         */
        objectContainer.commit();
    }

    objectContainer.ext().releaseSemaphore(semaphore);
}

return obj;
}

private static Object queryForSingletonClass(
    ObjectContainer objectContainer, Class clazz)  {
    Query q = objectContainer.query();
    q.constrain(clazz);
    ObjectSet objectSet = q.execute();
}

```

```

        if (objectSet.size() == 1)  {
            return objectSet.next();
        }
        if (objectSet.size() > 1)  {
            throw new RuntimeException(
                "Singleton problem. Multiple instances of: "
                + clazz.getName());
        }
        return null;
    }

}

```

Limiting the Number of Users

```

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

package com.db4odoc.semaphores;

import com.db4o.*;

/**
 * This class demonstrates the use of semaphores to limit the number
 * of logins to a server.
 */
public class LimitLogins {

    static final String HOST = "localhost";

    static final int PORT = 4455;

    static final String USER = "db4o";

    static final String PASSWORD = "db4o";

    static final int MAXIMUM_USERS = 10;

    public static ObjectContainer login()  {

        ObjectContainer objectContainer;
        try  {
            objectContainer = Db4o.openClient(HOST, PORT, USER,
                PASSWORD);
        } catch (Exception e)  {
            return null;
        }

        boolean allowedToLogin = false;

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

```

```

        if (objectContainer.ext().setSemaphore(
            "max_user_check_" + i, 0)) {
            allowedToLogin = true;
            break;
        }
    }

    if (!allowedToLogin) {
        objectContainer.close();
        return null;
    }

    return objectContainer;
}
}

```

Pluggable Sockets

db4o allows to customize client-server communication by using pluggable socket implementations:

Java:

```

public static ObjectServer openServer(Configuration config, String databaseFileName, int
port, NativeSocketFactory socketFactory);

public static ObjectContainer OpenClient(Configuration config, String hostName, int port,
String user, String password, NativeSocketFactory socketFactory);

```

Here NativeSocketFactory interface should be used to provide client and server sockets, which may implement any custom way of communication.

An example of such customary implementation can be [encrypted communication](#).

Note, that this API is in the development stage and is subject to future changes.

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. This can be fixed with a [new pluggable socket](#) client/server implementation. Let's look at a simple example - we will use SSL protocol to protect our communication channel.

Basically the task is to create a NativeSocketFactory implementation that will create sockets able to communicate through an encrypted channel. In Java these would be SSLServerSocket and SSLSocket.

```

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

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

import javax.net.ssl.SSLContext;

import com.db4o.config.NativeSocketFactory;

public class SecureSocketFactory implements NativeSocketFactory {

    private SSLContext _context;

    public SecureSocketFactory(SSLContext context) {
        _context = context;
    }

    public ServerSocket createServerSocket(int port) throws IOException {
        System.out.println("SERVER on " + port);
        return _context.getServerSocketFactory().createServerSocket(port);
    }

    public Socket createSocket(String hostName, int port) throws IOException {
        System.out.println("CLIENT on " + port);
        return _context.getSocketFactory().createSocket(hostName, port);
    }

    public Object deepClone(Object context) {
        return this;
    }

}

```

In order for this class to work correctly we need to provide a correctly initialized SSLContext. For those who are not familiar with Java SSL implementation it is recommended to get acquainted with Java documentation for SSLContext, TrustManager, KeyStore and KeyManagerFactory APIs.

For encryption purposes we will need to create a new keystore. This can be done with the following command:

```
keytool -genkey -keystore SSLcert -storepass password
```

This command creates a file SSLcert, which contains a keystore protected by "password" password. For a test keystore you can provide any answers to the keytool questions and leave the key password the same as the keystore password. For easy access copy the SSLcert file into your projects directory.

Now we are ready to create a SecureSocketFactory

```

SSLSSocketsExample.java: createSecureSocketFactory
private static SecureSocketFactory createSecureSocketFactory() throws Exception {
    SSLContext sc;

    //Create a trust manager that does not validate certificate chains
    TrustManager[] trustAllCerts = createTrustManager();

    // Install the all-trusting trust manager
    sc = SSLContext.getInstance("SSLv3");
    KeyStore ks = KeyStore.getInstance(KEYSTORE_ID);
    ks.load(new FileInputStream(KEYSTORE_PATH), null);
    KeyManagerFactory kmf = KeyManagerFactory.getInstance(
    KeyManagerFactory.getDefaultAlgorithm());
    kmf.init(ks, KEYSTORE_PASSWORD.toCharArray());

    sc.init(kmf.getKeyManagers(), trustAllCerts, new java.security.SecureRandom());
    return new SecureSocketFactory(sc);
}

```

```

SSLSSocketsExample.java: createTrustManager
private static TrustManager[] createTrustManager() {
    return new TrustManager[] {
        new X509TrustManager() {
            public java.security.cert.X509Certificate[] getAcceptedIssuers() {
                return null;
            }
            public void checkClientTrusted(
                java.security.cert.X509Certificate[] certs, String authType) {
            }
            public void checkServerTrusted(
                java.security.cert.X509Certificate[] certs, String authType) {
            }
        };
    };
}

```

Starting a server and opening client connections with the new socket factory is as simple as usual:

```

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

    // Create a SecureSocketFactory for the SSL context
    socketFactory = createSecureSocketFactory();

    Configuration config = Db4o.newConfiguration();
    ObjectServer db4oServer = Db4o.openServer(config, FILE, PORT,
        socketFactory);
    db4oServer.grantAccess(USER, PASSWORD);
    try {
        storeObjectsRemotely(HOST, PORT, USER, PASSWORD);
        queryRemoteServer(HOST, PORT, USER, PASSWORD);
    } finally {
        db4oServer.close();
    }
}

```

```

SSLocketExample.java: storeObjectsRemotely
private static void storeObjectsRemotely(String host,
int port, String user, String password) throws IOException {
    Configuration config = Db4o.newConfiguration();
    ObjectContainer client=Db4o.openClient(config,
"localhost",port,user,password, socketFactory);
    Pilot pilot = new Pilot("Fernando Alonso", 89);
    client.store(pilot);
    client.close();
}

```

```

SSLocketExample.java: queryRemoteServer
private static void queryRemoteServer(String host, int port, String user, String password) throws IOE
    Configuration config = Db4o.newConfiguration();
    ObjectContainer client=Db4o.openClient(config, "localhost",port,user,password, socketFactor
        listResult(client.queryByExample(new Pilot(null, 0)));
    client.close();
}

```

Remote Code Execution

Sometimes you will need your client to update many objects in a similar way on the server. The solution that is sought in this case is to give a server a criteria for selecting objects and instructions on what to do with them. In this way you will avoid the overhead of getting the objects over network and sending the updated set back.

More Reading:

- [Remote Execution Through Evaluation API](#)
- [Using Messaging API For Remote Code Execution](#)

Remote Execution Through Evaluation API

One of the ways to do that is using evaluation API. Evaluation/candidate classes are serialized and sent to the server. That means that if we will put the selection criteria and update code inside evaluation class, we will have that code on the server and executing a query using evaluation on the client side will run the update code on the server side.

For easier query execution we can use database singleton - a class that have only one instance saved in the database. That actually can be the class calling the query itself.

Let's fill up our server database:

```

RemoteExample.java: setObjects
private static void setObjects() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {

```

```

        for (int i = 0; i < 5; i++) {
            Car car = new Car("car" + i);
            container.store(car);
        }
        container.store(new RemoteExample());
    } finally {
        container.close();
    }
    checkCars();
}

```

Now we can update the cars using specially designed singleton:

```

        RemoteExample.java: updateCars
private static void updateCars() {
    // triggering mass updates with a singleton
    // complete server-side execution
    Configuration configuration = Db4o.newConfiguration();
    configuration.messageLevel(0);
    ObjectServer server = Db4o.openServer(configuration, DB4O_FILE_NAME, 0);
    try {
        ObjectContainer client = server.openClient();
        Query q = client.query();
        q.constrain(RemoteExample.class);
        q.constrain(new Evaluation() {
            public void evaluate(Candidate candidate) {
                // evaluate method is executed on the server
                // use it to run update code
                ObjectContainer objectContainer = candidate
                    .objectContainer();
                Query q2 = objectContainer.query();
                q2.constrain(Car.class);
                ObjectSet objectSet = q2.execute();
                while (objectSet.hasNext()) {
                    Car car = (Car) objectSet.next();
                    car.setModel("Update1-" + car.getModel());
                    objectContainer.store(car);
                }
                objectContainer.commit();
            }
        });
        q.execute();
        client.close();
    } finally {
        server.close();
    }
    checkCars();
}

```

This method has its pros and cons.

Pros:

- any arbitrary code can be executed;
- the code is executed on the server side;

Cons:

- the code will have to be serialized and sent over a network connection;
- changing the code will require update of all clients

Using Messaging API For Remote Code Execution

Messaging API gives you an easy and powerful tool for remote code execution. The short overview of the API is in [Messaging chapter](#).

All you will need to do is to define specific message class or classes (should be shared between client and server).

The client side can issue messages using:

Java: `MessageSender#send(message)`

The server side should register a message listener:

Java: `Configuration#setMessageRecipient(MessageRecipient)`

Message recipient should define a response to the different messages received in

Java: `processMessage(ObjectContainer objectContainer, Object message)`

method. `ObjectContainer` parameter gives full control over the database.

Let's reset our database and try updating using special `UpdateServer` message.

```
RemoteExample.java: setObjects
private static void setObjects() {
    new File(DB4O_FILE_NAME).delete();
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        for (int i = 0; i < 5; i++) {
            Car car = new Car("car" + i);
            container.store(car);
        }
    }
```

```
    container.store(new RemoteExample());
} finally {
    container.close();
}
checkCars();
}
```

You can also put some information in the message being sent (UpdateServer class).

The advantage of this method is that having predefined message types you can make any changes to the handling code only on the server side. In the situations with many distributed clients it can really save you lots of time on support.

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.

More Reading:

- [Optimistic Locking](#)
- [Pessimistic Locking](#)

Optimistic Locking

In optimistic locking system no locks are used to prevent collision: any user can read an object into the memory and work on it at any time. However, before the client can save its modifications back to the database, a check should take place verifying that the item did not change since the time of initial read (no collision occurred). If a collision is detected it should be resolved according to your application logic. Typical solutions are:

- Rollback
- Display the problem and let the user decide
- Merge the changes

- Log the problem so someone can decide later
- Ignore the collision and overwrite

Let's look at an example realization.

We will use a db4o database containing objects of Pilot class and a separate thread to create a client connection to the database, retrieve and modify objects.

```

OptimisticThread.java: run
public void run()  {
    try  {
        List<Pilot> result = _container.query(Pilot.class);
        for (Pilot pilot: result) {
            /**/* We will need to set a lock to make sure that the
             * object version corresponds to the object retrieved.
             * (Prevent other client committing changes
             * at the time between object retrieval and version
             * retrieval )
            */
            if (!_container.ext().setSemaphore("LOCK_"+
_container.ext().getID(pilot), 3000)) {
                System.out.println("Error. The object is locked");
                continue;
            }
            long objVersion = _container.ext().getObjectInfo(pilot).getVersion();
            _container.ext().refresh(pilot, Integer.MAX_VALUE);
            _container.ext().releaseSemaphore("LOCK_"
+ _container.ext().getID(pilot));

            /**/* save object version into _idVersions collection
             * This will be needed to make sure that the version
             * originally retrieved is the same in the database
             * at the time of modification
            */
            long id = _container.ext().getID(pilot);
            _idVersions.put(id, objVersion);

            System.out.println(getName() + "Updating pilot: "
+ pilot+ " version: "+objVersion);
            pilot.addPoints(1);
            _updateSuccess = false;
            randomWait();
            if (!_container.ext().setSemaphore("LOCK_"+
_container.ext().getID(pilot), 3000)) {
                System.out.println("Error. The object is locked");
                continue;
            }
            _container.store(pilot);
            /**/* The changes should be committed to be
             * visible to the other clients
            */
            _container.commit();
            _container.ext().releaseSemaphore("LOCK_"+

```

```

    _container.ext().getID(pilot));
        if (_updateSuccess) {
            System.out.println(getName() + "Updated pilot: " + pilot);
        }
        System.out.println();
        /**/* The object version is not valid after commit
         * - should be removed
         */
        _idVersions.remove(id);
    }

} finally {
    _container.close();
}
}

```

A semaphore is used for locking the object before saving and the lock is released after commit when the changes become visible to the other clients. The semaphore is assigned a name based on object ID to make sure that only the modified object will be locked and the other clients can work with the other objects of the same class simultaneously.

Locking the object for the update only ensures that no changes will be made to the object from the other clients during update. However the object might be already changed since the time when the current thread retrieved it. In order to check this we will need to implement an event handler for the updating event:

```

OptimisticThread.java: registerCallbacks
public void registerCallbacks() {
    EventRegistry registry = EventRegistryFactory.
forObjectContainer(_container);
    // register an event handler to check collisions on update
    registry.updating().addListener(new EventListener4() {
        public void onEvent(Event4 e, EventArgs args) {
            CancellableObjectEventArgs queryArgs = ((CancellableObjectEventArgs) args);
            Object obj = queryArgs.object();
            // retrieve the object version from the database
            long currentVersion = _container.ext().getObjectInfo(obj).getVersion();
            long id = _container.ext().getID(obj);
            // get the version saved at the object retrieval
            long initialVersion = _idVersions.queryByExample(id);
            if (initialVersion != currentVersion) {
                System.out.println(getName() +"Collision: ");
                System.out.println(getName() +"Stored object: version: "
+ currentVersion);
                System.out.println(getName() +"New object: " + obj+
" version: "+ initialVersion);
                queryArgs.cancel();
            } else {
                _updateSuccess = true;
            }
        }
    });
}

```

```
        }
    });
}
```

In the above case the changes are discarded and a message is sent to the user if the object is already modified from another thread. You can replace it with your own strategy of collision handling.

Note: the supplied example has random delays to make the collision happen. You can experiment with the delay values to see different behavior.

```
OptimisticThread.java: randomWait
private void randomWait()  {
    try  {
        Thread.sleep((long) (5000*Math.random()));
    } catch(InterruptedException e)  {
        System.out.println("Interrupted!");
    }
}
```

Pessimistic Locking

Pessimistic locking is an approach when an entity is locked in the database for the entire time that it is in application memory. This means that an object should be locked as soon as it is retrieved from the database and released after commit.

```
PessimisticThread.java: run
public void run()  {
    try  {
        List<Pilot> result = _container.query(Pilot.class);
        for (Pilot pilot: result) {
            /**/* with pessimistic approach the object is locked as soon
             * as we get it
            */
            if (!_container.ext().setSemaphore("LOCK_"+
_container.ext().getID(pilot), 0)) {
                System.out.println("Error. The object is locked");
            }

            System.out.println(getName() + "Updating pilot: " + pilot);
            pilot.addPoints(1);
            _container.store(pilot);
            /**/* The changes should be committed to be
             * visible to the other clients
            */
        }
    }
}
```

```

        _container.commit();
        _container.ext().releaseSemaphore("LOCK_"+
_container.ext().getID(pilot));
        System.out.println(getName() + "Updated pilot: " + pilot);
        System.out.println();
    }
} finally {
    _container.close();
}
}

```

As you see this approach is considerably easier to implement. Another advantage is that it guarantees that your changes to the database are made consistently and safely.

The main disadvantage is the lack of scalability. Time waiting for the lock to be released can become unacceptable for a system with many users or long transactions. This limits the practical implementations of pessimistic locking.

You may want to select pessimistic locking in cases when the cost of loosing the transaction results due to a collision is too high.

Messaging

In client/server mode the TCP connection between the client and the server can also be used to send messages from the client to the server.

Possible usecases could be:

- shutting down and restarting the server
- triggering server backup
- using a customized login strategy to restrict the number of allowed client connections

Here is some example code how this can be done.

First we need to decide on a class that we want to use as the message. Any object that is storables in db4o can be used as a message, but strings and other simple types will not be carried (as they are not storables in db4o on their own). Let's create a dedicated class:

```

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

package com.db4odoc.messaging;

class MyClientServerMessage {

```

```

private String info;

public MyClientServerMessage(String info) {
    this.info = info;
}

public String toString() {
    return "MyClientServerMessage: " + info;
}

}

```

Now we have to add some code to the server to react to arriving messages. This can be done by configuring a MessageRecipient object on the server. Let's simply print out all objects that arrive as messages. For the following we assume that we already have an ObjectServer object, opened with [Db4o.openServer\(\)](#).

```

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

package com.db4odoc.messaging;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectServer;
import com.db4o.config.Configuration;
import com.db4o.messaging.MessageContext;
import com.db4o.messaging.MessageRecipient;
import com.db4o.messaging.MessageSender;

public class MessagingExample {
    private final static String DB4O_FILE_NAME="reference.db4o";

    public static void configureServer() {
        Configuration configuration = Db4o.newConfiguration();
        configuration.clientServer().setMessageRecipient(new MessageRecipient() {
            public void processMessage(MessageContext context,
                Object message) {
                // message objects will arrive in this code block
                System.out.println(message);
            }
        });
        ObjectServer objectServer = Db4o.openServer(configuration, DB4O_FILE_NAME, 0);

        try {
            ObjectContainer clientObjectContainer = objectServer.openClient();
            // Here is what we would do on the client to send the message
            MessageSender sender = clientObjectContainer.ext().configure().
clientServer().getMessageSender();
        }
    }
}

```

```

        sender.send(new MyClientServerMessage("Hello from client."));
        clientObjectContainer.close();
    } finally {
        objectServer.close();
    }
}
// end configureServer

}

```

The MessageSender object on the client can be reused to send multiple messages.

Batch Mode

Client-server batch messaging mode was introduced in db4o version 6.1 and was enabled by default in version 7.4.

This mode allows to increase the performance by reducing client/server communication.

Java:

```
ClientConfiguration#networking().batchMessages(true);
```

How it works? Db4o client communicates with the server by means of messaging. If Batch-Messages is set to false db4o client sends a message with an instruction to the server and waits for the response. This might be quite inefficient when there are many small messages to be sent (like bulk inserts, updates, deletes): network communication becomes a bottleneck. Batch messaging mode solves this problem by caching the client messages on the client before sending them to the server.

The advantages of the batch messaging mode are:

- reduced network load;
- increased performance for bulk operations.

The downside is:

- increased memory consumption on both the client and the server.

More Reading:

- [Controlling Memory Consumption](#)
- [Batch Messaging Example](#)

Controlling Memory Consumption

In order to control the increasing memory usage in batch messaging mode the following configuration method is provided:

Java:

```
ClientConfiguration#networking().maxBatchQueueSize(size);
```

When this configuration method is used, client messages are flushed to the server when the message queue reaches the specified size in bytes.

By default maximum batch queue size is equal to the maximum integer number.

Batch Messaging Example

Let's create a small example to test batch messaging mode behavior. We will use bulk insert with and without batch messaging configuration.

```
BatchExample.java: fillUpDb
private static void fillUpDb()
    throws IOException {
    System.out.println("Testing inserts");
    ClientConfiguration configuration =
Db4oClientServer.newClientConfiguration();
    // Uncomment the following line to test CS mode without batching
    //configuration.networking().batchMessages(false);
    // Uncomment the following line to test maximum batch
    // queue size (batch mode should be enabled)
    //configuration.networking().maxBatchQueueSize(1024);
    ObjectContainer container = Db4oClientServer.openClient(
        configuration, HOST, PORT, USER,
        PASS);
    try {
        long t1 = System.currentTimeMillis();
        for (int i = 0; i < NO_OF_OBJECTS; i++) {
            Pilot pilot = new Pilot("pilot #" + i, i);
            container.store(pilot);
        }
        long t2 = System.currentTimeMillis();
        long diff = t2 - t1;
        System.out.println("Operation time: " + diff + " ms.");
    } finally {
        container.close();
    }
}
```

You can run this example with the server from [Simple db4o Server](#) chapter. In order to see the influence of batch mode configuration try commenting and uncommenting line disabling batch

mode. In addition you can try different values of `NO_OF_OBJECTS` constant.

If the value of `NO_OF_OBJECTS` is high ($>1,000,000$) you may notice that the memory consumption increases a lot. In order to decrease it, uncomment the line setting the maximum batch queue size.

db4o Replication System (dRS)

db4o Replication System (dRS) provides functionality to periodically synchronize databases that work disconnected from each other, such as remote autonomous servers or handheld devices synchronizing with central servers.

Before you start, please make sure that you have downloaded the latest dRS distribution from the [download area](#) of the [db4objects website](#).

The list of currently supported replication features currently include:

Java Platform

- db4o to db4o replication
- db4o to Hibernate replication
- Hibernate to Hibernate replication
- Array Replication
- Inheritance hierarchies
- Standard Primitive Types
- One-To-One Relations
- One-To-Many Relations
- Many-To-One Relations
- UUID and Version generation for db4o and Hibernate
- Queries for changed objects
- Parent-to-Child traversal along changed objects
- Collection Replication
- Replication Event System
- Replication of deleted objects

For the list of unsupported and planned features, please, see dRS project in our [Jira tracking system](#).

We invite you to join the db4o community in the public [db4o forums](#) to ask for help at any time. You may also find the db4o [knowledgebase](#) helpful for keyword searches.

More Reading:

- [Getting Started](#)
- [Db4o Databases Replication](#)

- [Replication With RDBMS](#)
- [Advanced Replication Strategies](#)
- [Deployment Instructions](#)
- [License](#)

Getting Started

This topic will give you some initial information about dRS and will help to get your environment ready to work with dRS.

Download Contents

.NET Platform

The distribution comes as a zip archive. The contents need to be extracted to any folder before you start to use the dRS.

You can find the offline version of this documentation in **/docs/reference** folder.

The API documentation is located in **/docs/api** folder.

db4o and [dRS online](#) documentation resources are located in the [reference documentation](#).

Java Platform

The distribution comes as a zip archive. The contents need to be extracted to any folder before you start to use the dRS.

You can find the offline version of this documentation in **/doc/reference** folder.

The API documentation is located in **/doc/api** folder.

db4o and [dRS online](#) documentation resources are located in the [reference documentation](#).

Requirements

Java version

dRS is dependent on the JDK 5 version of db4o object database. You will also need a corresponding db4o version (compatible db4o and dRS have the same version number). To use the Hibernate Replication functionality, the Hibernate core version 3.1 files are required. These files and their dependencies have been included in the **/lib** folder for your convenience. If you require

more complicated Hibernate mapping configurations, you may wish to download the full Hibernate documentation from the [Hibernate project website](#). To run the provided build scripts dRS requires [Apache Ant](#) 1.6 or later.

.NET version

dRS requires Microsoft .NET framework version 2.0 to work. A compatible version of db4o database should be used (compatible db4o and dRS have the same version number).

To work with the source code and examples you will need Microsoft Visual Studio 2005.

Installation

Java Platform Installation

dRS functionality is delivered in a single jar file. If you add dRS.jar to your CLASSPATH, dRS is installed. For using db4o replication to another database (RDBMS) you will also need hibernate library, which is included in the /lib folder in the distribution.

In case you work with an integrated development environment like [Eclipse](#) you can copy the dRS jar to a /lib/ folder under your project and add dRS to your project as a library.

Here is how to add the dRS library to an Eclipse project

- create a folder named "lib" under your project directory, if it doesn't exist yet
- copy dRS.jar to this folder
- Right-click on your project in the Package Explorer and choose "refresh"
- Right-click on your project in the Package Explorer again and choose "properties"
- select "Java Build Path" in the treeview on the left
- select the "Libraries" tabpage.
- click "Add Jar"
- the "lib" folder should appear below your project
- choose dRS.jar in this folder
- hit OK twice

You might need to repeat the same for hibernate3.jar

.NET Platform Installation

dRS engine comes as a dll. To use it in a development project, you only need to add the appropriate version of dRS to your project references. dRS version number corresponds to db4o version number, i.e dRS-6.0 will work with db4o-6.0.

Here is an example of how to add dRS to a Visual Studio 2005 project:

- copy drs.dll to your VS.NET project folder
- right-click on "References" in the Solution Explorer
- choose "Add Reference"
- select "Browse"
- find the drs.dll in your project folder
- click "Open"
- click "OK"

Db4o Databases Replication

db4o-to-db4o replication is the simplest replication and is supported by both java and .NET versions.

In the following examples we will use Pilot class from the db4o documentation.

```

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

package com.db4odoc.replication;

public class Pilot {
    private String name;

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

    public String getName() {
        return name;
    }

    public String toString() {
        return name;
    }
}

```

More Reading:

- [Initial Setup](#)
- [Simple Example](#)
- [Bi-Directional Replication](#)
- [Selective Replication](#)

Initial Setup

When replicating objects to and from a db4o database, we need to enable UUIDs and VersionNumbers.

UUIDs are object IDs that are unique across all databases created with db4o. That is achieved by having the database creation timestamp as part of its objects UUIDs. The db4o UUID contains two parts. The first part contains an object ID. The second part identifies the database that originally created this ID. More information on the UUIDs can be found in the [db4o reference documentation](#). The replication system will use the version number to invisibly tell when an object was last replicated, and if any changes have been made to the object since it was last replicated. An object's version number indicates the last time an object was modified. It is the database version at the moment of the modification.

```
ReplicationExample.java: configureReplication
public static Configuration configureReplication() {
    Configuration db4oConfig = Db4o.newConfiguration();
    db4oConfig.generateUUIDs(ConfigScope.GLOBALLY);
    db4oConfig.generateVersionNumbers(ConfigScope.GLOBALLY);
    return db4oConfig;
}
```

The above settings can also be applied to a specific class object, which needs to be replicated. This can help to improve the performance if only selected classes need to be replicated:

```
ReplicationExample.java: configureReplicationPilot
public static Configuration configureReplicationPilot() {
    Configuration db4oConfig = Db4o.newConfiguration();
    db4oConfig.objectClass(Pilot.class).generateUUIDs(true);
    db4oConfig.objectClass(Pilot.class).generateVersionNumbers(true);
    return db4oConfig;
}
```

Simple Example

The following example does a simple replication from a handheld database to the desktop database:

```
ReplicationExample.java: replicate
public static void replicate() {
    ObjectContainer desktop=Db4o.openFile(configureReplication(), DTFILENAME);
    ObjectContainer handheld=Db4o.openFile(configureReplication(), HHFILENAME);
    //      Setup a replication session
    ReplicationSession replication = Replication.begin(handheld, desktop);
```

```

/***/
 * There is no need to replicate all the objects each time.
 * objectsChangedSinceLastReplication methods gives us
 * a list of modified objects
 */
ObjectSet changed = replication.providerA().objectsChangedSinceLastReplication();

while (changed.hasNext()) {
    replication.replicate(changed.next());
}

replication.commit();
handheld.close();
desktop.close();
}

```

We start by opening two Object Containers. The next line creates the ReplicationSession. This object contains all of the replication-related logic.

After creating the session, there is an interesting line:

Java:

```
ObjectSet changed = replication.providerA().objectsChangedSinceLastReplication();
```

This line of code will get the provider associated with the first of our sources (the handheld ObjectContainer in this case). Then it finds all of the objects that have been updated or created. The new/modified objects will be returned in an enumerable ObjectSet.

After that comes a simple loop where the resulting objects are replicated one at a time. The `replication.commit()` call at the end is important. This line will save all of the changes we have made, and end any needed transactions. Forgetting to make this call will result in your replication changes being discarded when your application ends, or your ObjectContainers are closed. The `#commit()` calls also mark all objects as replicated. Therefore, changed/new objects that are not replicated in this session will be marked as replicated.

Bi-Directional Replication

The previous example copied all new or modified objects from the handheld device to the desktop database. What if we want to go the other way? Well, we only have to add one more loop:

```

ReplicationExample.java: replicateBiDirectional
public static void replicateBiDirectional() {
    ObjectContainer desktop=Db4o.openFile(configureReplication(), DTFILENAME);
}

```

```

ObjectContainer handheld=Db4o.openFile(configureReplication(), HHFILENAME);
ReplicationSession replication = Replication.begin(handheld, desktop);
ObjectSet changed = replication.providerA().objectsChangedSinceLastReplication();
while (changed.hasNext()) {
    replication.replicate(changed.next());
}
// Add one more loop for bi-directional replication
changed = replication.providerB().objectsChangedSinceLastReplication();
while(changed.hasNext()) {
    replication.replicate(changed.next());
}

replication.commit();
handheld.close();
desktop.close();
}

```

Now our handheld contains all of the new and modified records from our desktop.

Easy, isn't it? Now, if there had been any modifications made to the destination database, the two are now both in sync with each other.

Selective Replication

What if the handheld doesn't have enough memory to store a complete set of all of the data objects? Well, then we should check, which objects are to be replicated:

```

ReplicationExample.java: replicatePilots
public static void replicatePilots() {
    ObjectContainer desktop=Db4o.openFile(configureReplicationPilot(), DTFILENAME);
    ObjectContainer handheld=Db4o.openFile(configureReplicationPilot(), HHFILENAME);
    ReplicationSession replication = Replication.begin(handheld, desktop);
    ObjectSet changed = replication.providerB().objectsChangedSinceLastReplication();

    /**/* Iterating through the changed objects,
     * check if the name starts with "S" and replicate only those items
     */
    while (changed.hasNext()) {
        if (changed instanceof Pilot) {
            if (((Pilot)changed).getName().startsWith("S")) {
                replication.replicate(changed.next());
            }
        }
    }

    replication.commit();
    handheld.close();
    desktop.close();
}

```

Now, only Pilots whose name starts with "S" will be replicated to the handheld database.

Replication With RDBMS

db4o replication can be used to synchronize db4o database with an RDBMS. This is achieved by using [Hibernate](#).

The following topics will show how to configure and run Hibernate enabled replication.

More Reading:

- [Configuration](#)
- [Running DRS](#)
- [Replication Examples](#)
- [Hibernate Replication Internals](#)

Configuration

hibernate.cfg.xml

Hibernate requires a xml configuration file (hibernate.cfg.xml) to run. In order to run dRS with Hibernate, the user has to set some parameters in the configuration file.

```
hibernate.cfg.xml
<hibernate-configuration>      <session-factory>
    <!-- Database connection settings -->
    <property name="hibernate.connection.driver_class">
        oracle.jdbc.driver.OracleDriver</property>
    <property name="hibernate.connection.url">
        jdbc:oracle:thin:@ws-peterv:1521:websys</property>
        <property name="hibernate.connection.username">db4o</property>
        <property name="hibernate.connection.password">db4o</property>
        <!-- JDBC connection pool (use the built-in) -->
        <property name="hibernate.connection.pool_size">1</property>
        <!-- SQL dialect -->
        <property name="hibernate.dialect">
            org.hibernate.dialect.OracleDialect</property>
        <!-- Echo all executed SQL to stdout -->
        <property name="hibernate.show_sql">false</property>
        <!-- Update the database schema if out of date -->
        <property name="hibernate.hbm2ddl.auto">update</property>

        <property name="hibernate.jdbc.batch_size">0</property>
    </session-factory>
</hibernate-configuration>
```

For the property "hibernate.connection.pool_size", dRS requires only 1 connection to the RDBMS, increasing it will not have any effect. "hibernate.jdbc.batch_size" is set to 0 is for easier debugging. You may increase it to batch SQL statements to potentially increase performance.

It is a MUST that "hibernate.hbm2ddl.auto" be set to "update" because the system will create some extra tables to store the metadata for replication to work properly.

```
<property name="hibernate.hbm2ddl.auto">update</property>
```

Manual Creation of dRS Tables And Columns

In some situations, you may not have the privilege to create or alter tables. You may need to ask your DBA to create the tables for you before using dRS.

You can turn off the automatic creation of dRS tables and columns by changing the "hibernate.hbm2ddl.auto" property to "validate" in hibernate.cfg.xml. By doing so, dRS will not create or alter any tables. Rather, it will check the existence of the dRS tables before starting replication.

If the required dRS tables do not exist, dRS will throw a RuntimeException notifying the user and the replication will halt.

Hibernate Mapping Files

Persisted Objects are objects that the user wants to store to the database, e.g. cars, pilots, purchase orders. For dRS to operate properly, for each persisted object, the user MUST declare the primary key column of the database table in the hbm.xml mapping file as follow:

```
<id column="typed_id" type="long"> <generator class="native"/></id>
```

- column - The name of the primary key column. The value can be well-formed string . "typed_id" is recommended.
- type - MUST be "long"
- class - MUST be "native"

The "typed_id" column stores the id used by Hibernate. It allows dRS to identify a persisted object by invoking org.hibernate.Session#getIdentifier(Object object).

If you do not define getter/setter for property, make default-access="field". default-lazy="false" default-cascade="save-update" is required for replication to work properly. Note, you should not set the cascade style to "delete", otherwise deletion replication will not work.

Running DRS

This section describes how to configure and run dRS. It is crucial to follow the sequence of actions so that dRS can detect new/changed objects and replicate them during replication sessions.

```
// Read or create the Configuration as usual
Configuration cfg = new Configuration().
configure("your-hibernate.cfg.xml");
// Let the ReplicationConfigurator adjust
// the configuration
ReplicationConfigurator.configure(cfg);
//
```

```

Create the SessionFactory as usual
SessionFactory sessionFactory = cfg.buildSessionFactory();
// Create the Session as usual
Session session = sessionFactory.openSession();
// Let the ReplicationConfigurator install the
// listeners to the Session
ReplicationConfigurator.install(session, cfg);
//Update objects as usual
Transaction tx = session.beginTransaction();
Pilot john = (Pilot) session.createCriteria(Pilot.class)

```

Some precautions to take into consideration:

1. Do not open more than one dRS replication session against the same RDBMS concurrently. Otherwise data corruption will occur.
2. When dRS is in progress, do not modify the data in the RDBMS by using SQL or Hibernate. Otherwise data corruption will occur.

Replication Examples

The following topics represent some typical replication patterns.

More Reading:

- [Simple Example](#)
- [Collections](#)

Collections

This section covers examples on Collection, including array, Set, List and Map.

As an experienced db4o user, you may know that db4o treats Collection as first class object, which means it assigns unique UUID to each Collection. Hence a Collection can be shared among many owners. This is different to Hibernate's approach, where Collection does not have a unique ID and they cannot be shared among objects.

To bridge this gap, dRS treats Collections as second class objects and does not assign UIDs to them. When a shared Collection is replicated from db4o to Hibernate using dRS, it is automatically cloned. Each owner of the Collection receives a copy of the Collection. Further modifications to the db4o copy will not be replicated to cloned copies. Therefore, you cannot share Collections if you want to perform RDBMS replications with dRS.

In the following examples, we will use Car as the element in the following examples.

```

Car.java
/**/* Copyright (C) 2004 - 2008 Versant Inc. http://www.db4o.com

```

This file is part of the db4o open source object database.

db4o is free software; you can redistribute it and/or modify it under
the terms of version 2 of the GNU General Public License as published

```
by the Free Software Foundation and as clarified by db4objects' GPL interpretation policy, available at http://www.db4o.com/about/company/legalpolicies/gplinterpretation/ Alternatively you can write to db4objects, Inc., 1900 S Norfolk Street, Suite 350, San Mateo, CA 94403, USA.
```

```
db4o is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
```

```
You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. */ package f1.collection;
```

```
public class Car { public String brand; public String model; }
```

```
Car.hbm.xml
<?xml version="1.0"?>

<!-- Copyright (C) 2004 - 2008 Versant Inc. http://www.db4o.com -->

<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping default-access="field" default-lazy="false"
  default-cascade="save-update">
  <class name="f1.collection.Car">
    <id column="typed_id" type="long">
      <generator class="native"/>
    </id>
    <property name="brand"/>
    <property name="model"/>
  </class>
</hibernate-mapping>
```

More Reading:

- [Array](#)
- [List](#)
- [Set](#)
- [Map](#)

Array

Hibernate version used with dRS supports one dimensional arrays but does not support multidimensional arrays.

```
Pilot.java
/** Copyright (C) 2004 - 2008 Versant Inc. http://www.db4o.com

This file is part of the db4o open source object database.

db4o is free software; you can redistribute it and/or modify it under
the terms of version 2 of the GNU General Public License as published
by the Free Software Foundation and as clarified by db4objects' GPL
interpretation policy, available at
http://www.db4o.com/about/company/legalpolicies/gplinterpretation/
Alternatively you can write to db4objects, Inc., 1900 S Norfolk Street,
Suite 350, San Mateo, CA 94403, USA.
```

db4o is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. */

```
package f1.collection.array;

import f1.collection.Car;

public class Pilot {
    String name;
    Car[] cars;
}
```

```
Pilot.hbm.xml
<?xml version="1.0"?>

<!-- Copyright (C) 2004 - 2008 Versant Inc. http://www.db4o.com -->

<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping default-access="field"
default-lazy="false" default-cascade="save-update">
    <class name="f1.collection.array.Pilot">
        <id column="typed_id" type="long">
            <generator class="native"/>
        </id>

        <property name="name"/>

        <array name="cars" table="cars">
            <key column="pilotId"/>
            <list-index column="sortOrder"/>
            <one-to-many class="f1.collection.Car"/>
        </array>
    </class>
</hibernate-mapping>
```

Add Pilot and Car to hibernate.cfg.xml

```
<mapping resource="f1/collection/array/Pilot.hbm.xml"/>  
<mapping resource="f1/collection/array/Car.hbm.xml"/>
```

Save the pilot as usual and start replication:

```
ArrayExample.java: main
```

```
public static void main(String[] args) {  
    new File("ArrayExample.yap").delete();  
    System.out.println("Running Array example.");  
    com.db4o.config.Configuration db4oConfig = Db4o.newConfiguration();  
    db4oConfig.generateUUIDs(ConfigScope.GLOBALLY);  
    db4oConfig.generateVersionNumbers(ConfigScope.GLOBALLY);  
    ObjectContainer objectContainer = Db4o.openFile(db4oConfig, "ArrayExample.yap");  
    Pilot pilot = new Pilot();  
    pilot.name = "John";  
    Car car1 = new Car();  
    car1.brand = "BMW";  
    car1.model = "M3";  
    Car car2 = new Car();  
    car2.brand = "Mercedes Benz";  
    car2.model = "S600SL";  
    pilot.cars = new Car[]{car1, car2};  
    objectContainer.store(pilot);  
    objectContainer.commit();  
    Configuration config = new Configuration().configure("f1/collection/array/hibernate.cfg.xml");  
    ReplicationSession replication = HibernateReplication.begin(objectContainer, config);  
    ObjectSet changed = replication.providerA().objectsChangedSinceLastReplication();  
    while (changed.hasNext())  
        replication.replicate(changed.next());  
    replication.commit();  
    replication.close();  
    objectContainer.close();  
    new File("ArrayExample.yap").delete();  
}
```

List

Similar to array, you can replicate a List of Cars.

```
Pilot.java
/**/* Copyright (C) 2004 - 2008 Versant Inc. http://www.db4o.com

This file is part of the db4o open source object database.

db4o is free software; you can redistribute it and/or modify it under
the terms of version 2 of the GNU General Public License as published
by the Free Software Foundation and as clarified by db4objects' GPL
interpretation policy, available at
http://www.db4o.com/about/company/legalpolicies/gplinterpretation/
Alternatively you can write to db4objects, Inc., 1900 S Norfolk Street,
Suite 350, San Mateo, CA 94403, USA.

db4o is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. */
package f1.collection.list;

import java.util.List;

public class Pilot {
    String name;
    List cars;
}
```

Map the List using the list tag in Pilot.hbm.xml

```
Pilot.hbm.xml
<?xml version="1.0"?>

<!-- Copyright (C) 2004 - 2008 Versant Inc. http://www.db4o.com -->

<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping default-access="field"
default-lazy="false" default-cascade="save-update">
    <class name="f1.collection.list.Pilot">
        <id column="typed_id" type="long">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <list name="cars" table="cars">
```

```

<key column="pilotId"/>
<list-index column="sortOrder"/>
<one-to-many class="f1.collection.Car"/>
</list>
</class>
</hibernate-mapping>

```

Replicate the pilot:

```

ListExample.java: main
1

```

Map

Replication supports replicating a Map of objects.

```

Pilot.java
/**/* Copyright (C) 2004 - 2008 Versant Inc. http://www.db4o.com

This file is part of the db4o open source object database.

db4o is free software; you can redistribute it and/or modify it under
the terms of version 2 of the GNU General Public License as published
by the Free Software Foundation and as clarified by db4objects' GPL
interpretation policy, available at
http://www.db4o.com/about/company/legalpolicies/gplinterpretation/
Alternatively you can write to db4objects, Inc., 1900 S Norfolk Street,
Suite 350, San Mateo, CA 94403, USA.

db4o is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. */
package f1.collection.map;

import java.util.Map;

public class Pilot {
    String name;
    Map cars;
}

```

Use the map element to define the Map in the hbm file:

```

Pilot.hbm.xml
<?xml version="1.0"?>

<!-- Copyright (C) 2004 - 2008 Versant Inc. http://www.db4o.com -->

<!DOCTYPE hibernate-mapping PUBLIC

```

```

"-->Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping default-access="field" default-lazy="false"
default-cascade="save-update">
  <class name="f1.collection.map.Pilot">
    <id column="typed_id" type="long">
      <generator class="native"/>
    </id>

    <property name="name"/>

    <map name="cars" table="cars">
      <key column="pilotId"/>
      <map-key type="string"/>
      <one-to-many class="f1.collection.Car"/>
    </map>
  </class>
</hibernate-mapping>

```

Do the replication:

```

MapExample.java: main
1

```

Set

Replicating a Set of objects is simple and is similar to the [List](#) example.

```

Pilot.java
/**/* Copyright (C) 2004 - 2008 Versant Inc. http://www.db4o.com

This file is part of the db4o open source object database.

db4o is free software; you can redistribute it and/or modify it under
the terms of version 2 of the GNU General Public License as published
by the Free Software Foundation and as clarified by db4objects' GPL
interpretation policy, available at
http://www.db4o.com/about/company/legalpolicies/gplinterpretation/
Alternatively you can write to db4objects, Inc., 1900 S Norfolk Street,
Suite 350, San Mateo, CA 94403, USA.

db4o is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. */
package f1.collection.set;

import java.util.Set;

public class Pilot {

```

```
    String name;  
    Set cars;  
}
```

Use the set tag.

```
Pilot.hbm.xml  
<?xml version="1.0"?>  
  
<!-- Copyright (C) 2004 - 2008 Versant Inc. http://www.db4o.com -->  
  
<!DOCTYPE hibernate-mapping PUBLIC  
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  
  
<hibernate-mapping default-access="field"  
default-lazy="false" default-cascade="save-update">  
  <class name="f1.collection.set.Pilot">  
    <id column="typed_id" type="long">  
      <generator class="native"/>  
    </id>  
  
    <property name="name"/>  
  
    <set name="cars" table="cars">  
      <key column="pilotId"/>  
      <one-to-many class="f1.collection.Car"/>  
    </set>  
  </class>  
</hibernate-mapping>
```

Do the replication.

```
SetExample.java: main  
1
```

Simple Example

This is a one-to-one association example.

The following persistent classes are used:

```
Helmet.java  
/**/* Copyright (C) 2004 - 2008 Versant Inc. http://www.db4o.com  
  
This file is part of the db4o open source object database.  
  
db4o is free software; you can redistribute it and/or modify it under  
the terms of version 2 of the GNU General Public License as published  
by the Free Software Foundation and as clarified by db4objects' GPL  
interpretation policy, available at  
http://www.db4o.com/about/company/legalpolicies/gplinterpretation/  
Alternatively you can write to db4objects, Inc., 1900 S Norfolk Street,  
Suite 350, San Mateo, CA 94403, USA.
```

```
db4o is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
for more details.
```

```
You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. */
package f1.one_to_one;
```

```
public class Helmet {
    String model;
}
```

```
Pilot.java
/** Copyright (C) 2004 - 2008 Versant Inc. http://www.db4o.com
```

```
This file is part of the db4o open source object database.
```

```
db4o is free software; you can redistribute it and/or modify it under
the terms of version 2 of the GNU General Public License as published
by the Free Software Foundation and as clarified by db4objects' GPL
interpretation policy, available at
http://www.db4o.com/about/company/legalpolicies/gplinterpretation/
Alternatively you can write to db4objects, Inc., 1900 S Norfolk Street,
Suite 350, San Mateo, CA 94403, USA.
```

```
db4o is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
for more details.
```

```
You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. */
package f1.one_to_one;
```

```
public class Pilot {
    String name;
    Helmet helmet;
}
```

A one-to-one association to another persistent class is declared using a one-to-one element:

```
Pilot.hbm.xml
<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping default-access="field" default-lazy="false"
                    default-cascade="save-update">
    <class name="f1.one_to_one.Pilot">
        <id column="typed_id" type="long">
```

```

<generator class="foreign">
    <param name="property">helmet</param>
</generator>
</id>
<property name="name"/>
<one-to-one name="helmet"/>
</class>
</hibernate-mapping>

```

Remember to add mappings in hibernate.cfg.xml:

```

<mapping resource="f1/one_to_one/Pilot.hbm.xml"/>
<mapping resource="f1/one_to_one/Helmet.hbm.xml"/>

```

The code to run the replication is provided below:

```

OneToOneExample.java: main
1

```

Hibernate Replication Internals

So far we have seen that dRS allows you to replicate objects between db4o and relational database. You maybe curious about how dRS keeps track of the identity of objects in relational database and how dRS knows which objects are changed since the last round of replication. Read on and you will see how dRS does that.

dRS internal objects keep information used by replication. Each internal object is associated with a Hibernate mapping file (.hbm.xml). Hibernate reads these files and understands how to store / retrieve these internal objects to / from the RDBMS. Each type of internal object maps to one table in RDBMS. If such table does not exist, Hibernate creates it automatically.

ProviderSignature, MySignature and PeerSignature

ProviderSignature uniquely identifies a ReplicationProvider. MySignature and PeerSignature are the subclasses of ProviderSignature. A HibernateReplicationProvider has a MySignature to serve as its own identity. PeerSignature identifies the peer ReplicationProvider during a ReplicationSession.

Record

Record contains the version of the RDBMS during a ReplicationSession. Near the end of a ReplicationSession, two ReplicationProviders synchronize their versions.

Record allows dRS to detect changed objects. dRS detects changed objects by comparing the version of an object (v) with the maximum version of all Records (m). An object is updated if v > m.

UUID

UUID uniquely identifies a persisted object in dRS. Each persisted object is identified by a "typed_id" in Hibernate. This "typed_id" is unique only with its type of that object (i.e. A car has an "typed_id" of 1534, a Pilot can also have an "typed_id" of 1534) and within the current RDBMS. How do we identify "a Pilot that is originated from Oracle instance pi2763" ? To do so, we need two parameters: 1. an id that is unique across types 2. association between this id and the ProviderSignature of the RDBMS (The RDBMS that owns this object)

```
class UUID {long longPart;ProviderSignature provider;}
```

Collectively, 1 and 2 forms the "UUID".

ObjectReference

ObjectReference contains the UUID and the version of a persisted object. It also contains the class-Name and the typed_id of that persisted object. UUID forms an 1 to 1 relationship with {className, typedId}.

```
class ObjectReference {String className;long typedId;Uuid uuid;long version;}
```

List of dRS tables

drs_providers

Column	Type	Function
id	long	synthetic, auto-increment primary key
is_my_sig	char(1)	't' if MySignature, 'f' if PeerSignature
signature	binary	holds the unique identifier - byte array
created	long	legacy field used by pre-dRS db4o replication code

drs_history

Column Type Function

provider_id	long	primary key, same as the PK of a PeerSignature
time	long	the version of the RDBMS during a ReplicationSession

drs_objects

Column	Type	Function
id	long	synthetic, auto-increment primary key
created	long	the UUID long part of this ObjectReference
provider_id	long	specifies the originating ReplicationProvider of this ObjectReference
class_name	varchar	the type of the referenced object
typed_id	long	the id used by Hibernate, which is only unique within its type
modified	long	the version of the referenced object

Advanced Replication Strategies

This chapter contains information on some advanced dRS strategies.

More Reading:

- [Events](#)
- [Deleted Objects Replication](#)

Events

Replication events provide you with an ability to add your custom logic into the replication process. Below you will find some of the usage examples. For more information see [ReplicationEvent API](#).

More Reading:

- [Conflict Resolution](#)

Conflict Resolution

The most popular usage of replication callback events is conflict resolution. When an object was changed in both replicating databases you must have a strategy to decide, which value must be final. In the simplest case one of the databases is made "dominant" and the changes in this database always override other changes. The following example demonstrates this behavior:

```
EventsExample.java: conflictResolutionExample
private static void conflictResolutionExample() {
    Configuration config = Db4o.newConfiguration();
    config.generateUUIDs(ConfigScope.GLOBALLY);
    config.generateVersionNumbers(ConfigScope.GLOBALLY);

    // Open databases
    ObjectContainer desktop = Db4o.openFile(config, "desktop.db4o");
    ObjectContainer handheld = Db4o.openFile(config, "handheld.db4o");

    Pilot pilot = new Pilot("Scott Felton", 200);
    handheld.store(pilot);
    handheld.commit();
    /**/* Clean the reference cache to make sure that objects in memory
     * won't interfere
    */
    handheld.ext().refresh(Pilot.class, Integer.MAX_VALUE);

    /**/* Replicate changes from handheld to desktop
     * Note, that only objects replicated from one database to another will
     * be treated as the same. If you will create an object and save it to both
     * databases, dRS will count them as 2 different objects with identical
     * fields.
```

```

/*
ReplicationSession replication = Replication.begin(handheld, desktop);
ObjectSet changedObjects = replication.providerA().objectsChangedSinceLastReplication();
while (changedObjects.hasNext())
    replication.replicate(changedObjects.next());
replication.commit();

// change object on the handheld
pilot = (Pilot)handheld.query(Pilot.class).next();
pilot.setName("S.Felton");
handheld.store(pilot);
handheld.commit();

// change object on the desktop
pilot = (Pilot)desktop.query(Pilot.class).next();
pilot.setName("Scott");
desktop.store(pilot);
desktop.commit();

/**/* The replication will face a conflict: Pilot object was changed on the
 * handheld and on the desktop.
 * To resolve this conflict we will add an event handler, which makes
 * desktop changes dominating.
*/
ReplicationEventListener listener;
listener = new ReplicationEventListener() {
    public void onReplicate(ReplicationEvent event) {
        if (event.isConflict()) {
            ObjectState chosenObjectState = event.stateInProviderB();
            event.overrideWith(chosenObjectState);
        }
    }
};

replication = Replication.begin(handheld, desktop, listener);

//The state of the desktop after the replication should not change, as it dominates
changedObjects = replication.providerA().objectsChangedSinceLastReplication();
while (changedObjects.hasNext())
    replication.replicate(changedObjects.next());

//Commit
replication.commit();
replication.close();

// Check what we've got on the desktop
ObjectSet result = desktop.query(Pilot.class);
System.out.println(result.size());
while (result.hasNext()) {
    System.out.println(result.next());
}
handheld.close();
desktop.close();

```

```
new File("handheld.db4o").delete();
new File("desktop.db4o").delete();

}
```

Deleted Objects Replication

In addition to replicating changed/new objects, dRS is able to replicate deletions of objects. When an object is deleted since last replication in one database and you would like to replicate these changes to another database you can use the following method to do this:

Java:

```
replication.replicateDeletions(Car.class);
```

VB:

```
replication.ReplicateDeletions(GetType(Car))
```

dRS traverses every Car object in both providers. For instance, if a deletion is found in one provider, the deletion will be replicated to the other provider. During the traversal replication events will be generated and can be used as usual. By default, in a case of a conflict the deletion will prevail. You can choose the counterpart of the deleted object to prevail using the event.

Note, that the deletions of a Parent will not be cascaded to child objects. For example, if a Car contains a child object, e.g. Pilot, Pilot will not be traversed and the deletions of Pilot will not be replicated.

Deployment Instructions

When you deploy dRS with your application, you must first ensure that the [requirements for db4o](#) deployment are met.

dRS-Java Deployment

In order to deploy dRS in the most simple db4o-to-db4o replication scenario you will only need **dRS-7.12-core.jar**.

For support of replication to relational databases you will need **hibernate3.jar** and its dependencies:

- antlr-2.7.6.jar
- asm.jar
- asm-attrs.jar
- cglib-2.1.3.jar
- commons-collections-2.1.1.jar
- commons-logging-1.0.4.jar
- dom4j-1.6.1.jar
- ehcache-1.2.jar
- jta.jar
- log4j-1.2.13.jar
- xerces-2.6.2.jar

Please, refer to [hibernate](#) documentation for an explanation of these dependencies.

If you are going to use dRS with HSQLDB you will need hsqldb-1.8.0.7.jar in addition to hibernate.

License

dRS

[Versant Inc.](#) supplies the db4o Replication System (dRS) under the General Public License (GPL).

Under the GPL dRS is free to be used:

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

You should have received a copy of the GPL in the file dRS.license.htm together with the dRS distribution.

Bundled 3rd Party Licenses

The dRS distribution comes with several 3rd party libraries, located in the /lib/ subdirectory together with the respective license files.

Db4o Testing Framework

db4ounit is a minimal xUnit (JUnit, NUnit) style testing framework. 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-6.0-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:

- [Db4ounit Methods](#)
- [Creating A Sample Test](#)
- [Running The Tests](#)

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.

private transient Db4oFixture _fixture; - determines an environment for the test execution and gives an access to the test database. The environment can be local (derived from AbstractSoloDb4oFixture) or client/server (AbstractClientServerDb4oFixture).

You can always access the fixture from your test class using

Java:

```
public void fixture(Db4oFixture fixture)
```

Methods for working with a database:

Java:

```
public ExtObjectContainer db()
```

Returns an instance of object container for the current environment.

Java:

```
protected void reopen() throws Exception
```

This function will close the database and open it again. It also performs an implicit commit on close.

Java:

```
protected Reflector reflector()
```

Returns current reflector.

Java:

```
protected Transaction trans()  
protected Transaction systemTrans()  
protected Transaction newTransaction()
```

Methods to get transaction object for the current environment.

Various methods to work with persistent objects:

Java:

```
protected Query newQuery()  
protected Query newQuery(Class clazz)  
protected Query newQuery(Transaction transaction, Class clazz)  
protected Query newQuery(Transaction transaction)
```

Create a new query object.

Java:

```
protected Object retrieveOnlyInstance(Class clazz)
```

Checks if only one object of a class is stored in the database

Java:

```
protected int countOccurrences(Class clazz)
```

Returns the amount of objects of the specified class in the database.

Java:

```
protected void foreach(Class clazz, Visitor4 visitor)
```

This method goes through the ObjectSet of the specified class objects in the database calling Visitor4.visit() method. Visitor4 is an interface specifying a visit method:

Java:

```
public void visit(Object obj);
```

Java:

```
protected void deleteAll(Class clazz)
```

Deletes all the instances of the specified class in the database.

Java:

```
protected ReflectClass reflectClass(Class clazz)
```

Returns a ReflectClass instance for the specified class.

Java:

```
protected void indexField(Configuration config, Class clazz, String fieldName)
```

Adds field index into specified configuration.

Java:

```
public final void setUp() throws Exception
```

This method:

- deletes the used database;
- configures and opens a new one (see Configure method).
- Calls db4oSetupBeforeStore
- Calls store()
- Commits and reopens the database
- Calls db4oSetupAfterStore

More details about the mentioned above methods:

Java:

```
protected void configure(Configuration config)
```

Use this method to create your custom configuration for a test. Config parameter is the current default test configuration, which can be modified.

Java:

```
protected void db4oSetupBeforeStore() throws Exception
```

This method is a placeholder for any custom setup actions that need to be taken before filling up the database with objects.

Java:

```
protected void store() throws Exception {}
```

This method is supplied for creating and storing the objects, which you are going to use for your test.

Java:

```
protected void db4oSetupAfterStore() throws Exception
```

This method is a placeholder for any custom setup actions that need to be taken after the database is filled up with objects.

Methods for running tests:

Java:

```
public int runSoloAndClientServer()
```

Will run the test in both modes

Java:

```
public int runSolo()
```

Only local mode.

Java:

```
public int runClientServer()
```

Db4ounit.Assert

Db4ounit.Assert class - provides a variety of methods for controlling code execution. Some of the methods are presented below. For more information please refer to the source code.

Java:

```
public static void expect(Class exception, CodeBlock block)
```

This method runs a specified method (block parameter) and throws an exception if the block runs without any exception.

Java:

```
public static void isTrue(boolean condition)  
public static void isTrue(boolean condition, String msg)
```

This method checks the condition and throws an exception if the condition is false. Msg parameter can be used to customize exception message.

Java:

```
public static void areEqual(Object expected, Object actual)
```

Checks if the supplied parameters are equal and throws an exception otherwise.

Similar methods are provided for null, lesser, greater and other checking, please refer to Assert class code for full information.

FrameworkTestCase

FrameworkTestCase class provides methods to run your test suite and check if its results.

Java:

```
public static void runTestAndExpect(Test test, int expFailures)
```

This method will run the test specified and throw an exception if the number of expected failures (expFailures parameter) is not equal to the number of experienced failures.

For more information please refer to the source code of FrameworkTestCase class.

Creating A Sample Test

Let's create a simple test case to check if the cascade on update setting is working as expected.

We will use a simple linked list class:

```
CascadeOnUpdate.java: Atom
public static class Atom {
    public Atom child;
    public String name;

    public Atom() {
    }

    public Atom(Atom child) {
        this.child = child;
    }

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

    public Atom(Atom child, String name) {
        this(child);
        this.name = name;
    }
}
```

Custom configuration should be added to configure method:

```
CascadeOnUpdate.java: configure
```

```

protected void configure(Configuration conf) {
    conf.objectClass(Atom.class).cascadeOnUpdate(false);
}

```

To prepare the database we will need to store some Atom objects:

```

CascadeOnUpdate.java: store
protected void store() {
    Atom atom = new Atom(new Atom(new Atom("storedGrandChild"), "storedChild"), "parent");
    db().store(atom);
    db().commit();
}

```

Now we are ready to test the expected behavior:

```

CascadeOnUpdate.java: test
public void test() throws Exception {
    Query q = newQuery(Atom.class);
    q.descend("name").constrain("parent");
    ObjectSet objectSet = q.execute();
    Atom atom = null;
    while (objectSet.hasNext()) {
        // update child objects
        atom = (Atom) objectSet.next();
        ((Atom) atom.child).name = "updated";
        ((Atom) atom.child).child.name = "notUpdated";
        // store the parent object
        db().store(atom);
    }

    // commit and refresh to make sure that the changes are saved
    // and reference cash is refreshed
    db().commit();
    db().refresh(atom, Integer.MAX_VALUE);

    q = newQuery(Atom.class);
    q.descend("name").constrain("parent");
    objectSet = q.execute();
    while (objectSet.hasNext()) {
        atom = (Atom) objectSet.next();
        Atom child = (Atom) atom.child;
        // check if the child objects were updated
        Assert.AreEqual("updated", child.name);
        Assert.AreNotEqual("updated", child.child.name);
    }
}

```

Running The Tests

The TestSuite can be built:

- from the list of the arguments supplied to Db4oUnitTestMain class (only for Java version) .
- using Db4oTestSuiteBuilder to specify test classes to be run.

If you are using java version you can test your class immediately (assuming the class is created in TestDb4oUnit package):

```
java -cp db4o- 6.0- db4ounit.jar;your_class_folder db4ounit.UnitTestingMain your_class_package.AssertTestCase
```

The second option is to list the test class in Db4oTestSuiteBuilder argument. You can use db4ounit.extensions.tests.AllTests class. Just add the following method:

```
AllTests.java: testCascadeOnUpdate
public void testCascadeOnUpdate() {
    Db4oFixture fixture=new ExcludingInMemoryFixture
(new IndependentConfigurationSource());
    TestSuite suite = new Db4oTestSuiteBuilder
(fixture, new Class[] {CascadeOnUpdate.class}).build();
    FrameworkTestCase.runTestAndExpect(suite,0);
}
```

Now running the AllTests class will include your custom class too.

Object Manager Enterprise

Starting from db4o version 7.8 db4o distribution package includes Object Manager Enterprise - graphical tool for browsing and managing db4o databases. The tool is available in 2 flavors: Eclipse plugin for Java users and Visual Studio 2005/2008 plugin for .NET users.

More Reading:

- [OME For Visual Studio](#)
- [OME For Eclipse](#)

OME For Eclipse

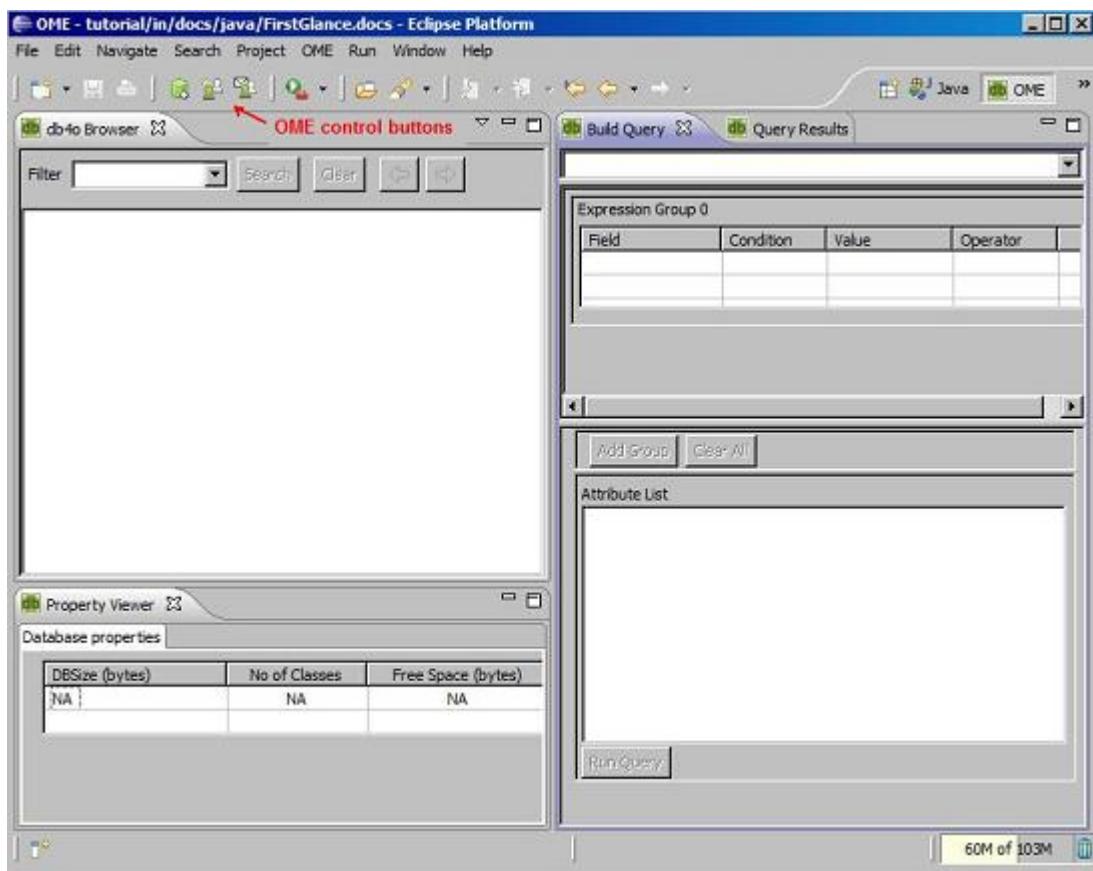
Object Manager Enterprise (OME) is an object browser for db4o databases. OME installation can be found in /ome folder of the distribution. The zip file in this folder contains the Eclipse plugin version of OME. To install the plugin, you need to have a version of Eclipse >= 3.3 installed. Unzip the file to a folder of your choice. Then open Eclipse, select 'Help' -> 'Software Updates...' -> 'Available Software' from the menu. Choose 'Add Site...' -> 'Local...' and select the unzipped folder. Follow the Eclipse Update Manager instructions for the OME feature from here on. The actual menu structure may vary over Eclipse versions. (The above applies to Eclipse 3.4 Ganymede.) When in doubt, please refer to the Eclipse documentation on Software Updates. Alternatively, you can install the plugin manually by simply copying the contents of the 'plugins' and 'features' folders from the unzipped folder to the corresponding subfolders in the root folder of your Eclipse installation.

More Reading:

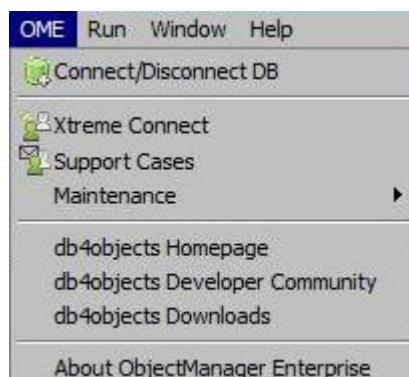
- [OME Interface](#)
- [Browsing A Database](#)
- [Querying](#)

OME Interface

Once the Object Manager (OM) is installed you can see it in Eclipse by selecting Window->Open Perspective->Other and choosing "OME". Typically, OME window should look similar to this:



In the OME perspective you can see:



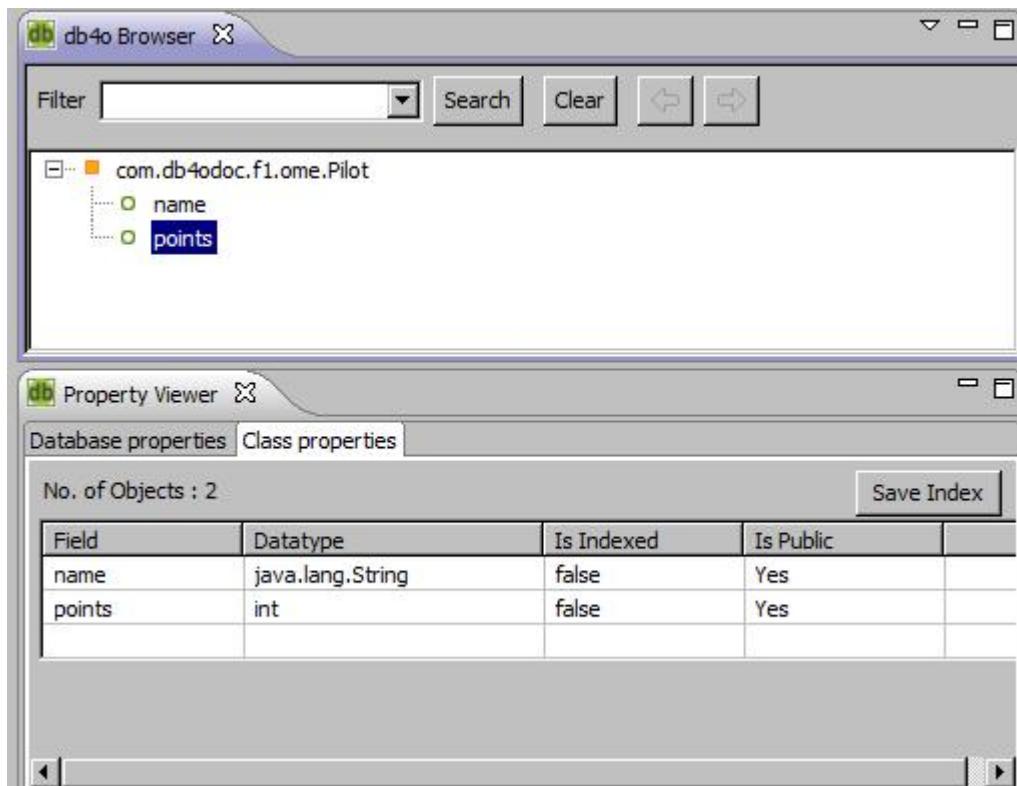
OME menu:

- OME toolbar buttons to access the frequently used functionality fast
- Db4o Browser: window displaying the contents of the open db4o database
- Property Viewer: window displaying the properties of the open database or the properties of the selected database class
- Build Query: windows allowing to build a query using drag&drop functionality
- Query Results: window to browse the results of the query execution

Browsing A Database

Suppose you have a simple database file containing Pilot and Car objects. Please select OME->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:

The screenshot shows the db4o IDE interface. At the top, there are two tabs: "Build Query" and "Query Results". The "Query Results" tab is active, displaying a list titled "com.db4odoc.f1.ome.Pilot". The list contains two rows:

Row Id	name	points
1	Michael Schumacher	100
2	Rubens Barrichello	99

Below the list are buttons for "Save", "Delete", "Refresh", and navigation arrows. The number "1" is displayed next to "of 1", indicating one result page. The "Object 2" section below shows the details of the selected row (Row Id 2):

Field	Value
(G) com.db4odoc.f1.ome.Pilot	(G) com.db4odoc.f1.ome.Pilot
name	Rubens Barrichello
points	99

A "Save" button is located at the bottom left of the "Object 2" panel.

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.

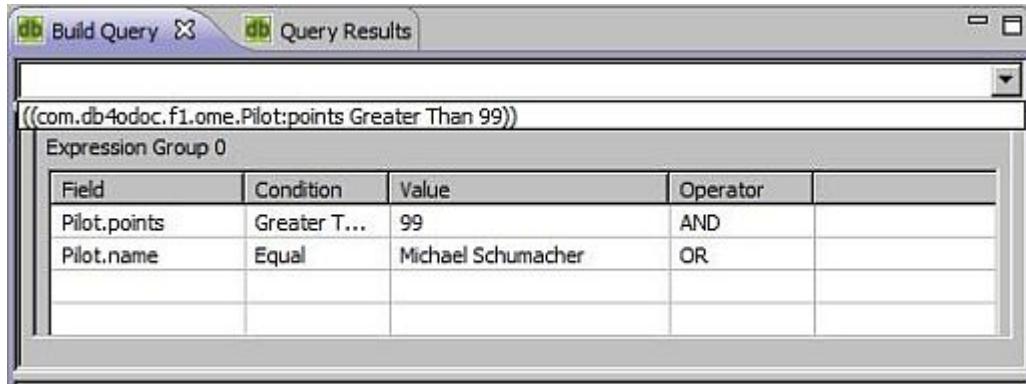
More complex queries can be done by using Build Query view:

The screenshot shows the db4o IDE interface with two main windows open:

- db4o Browser:** This window displays a tree view of objects. Under "com.db4odoc.f1.ome.Pilot", there are two children: "name" and "points".
- Build Query:** This window is used for creating complex queries. It includes sections for "Expression Group 0" (with a table for Field, Condition, Value, and Operator), "Add Group" and "Clear All" buttons, and an "Attribute List" section. At the bottom is a "Run Query" button.

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, you can close the connection by using OME->Connect/Disconnect DB menu command or by using the equivalent button on the toolbar.

Sharpen

Sharpen is an Eclipse plugin 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](#)
- [Example 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

into the same workspace as your java project, which you want to convert. (This is not required but the referencing will be easier to type and maintain).

You will need to install sharpen plugin and define sharpen tasks. Sharpen can be installed manually from within Eclipse:

- Select and right click the freshly checked out projects in the "Package Explorer" and choose "Export" from the context menu;
- Expand the "Plug-in Development" folder and select "Deployable plug-ins and fragments";
- Set "Destination" to the root folder of your eclipse installation and click "Finish";

Alternatively you can use ant script to define sharpen installation task and sharpen conversion. Add the following files to your java project folder:

sharpen.properties

```

#           eclipse      workspace      dir.workspace=e:/db4o/trunk
#       java      executable      file.jvm.jdk1.5=e:/Java/jdk1.6.0_02/bin/java.exe
#           Eclipse      home      directory      eclipse.home=e:/Eclipse
# Eclipse startup jar  eclipse.startup.jar=${eclipse.home}/plugins/org.eclipse.equinox.launcher_
1.0.0.v20070606.jar
# Sandcastle can be used to convert javadoc to .NET xml comments
#                               dir.lib.sandcastle=e:/sandcastle/
# sharpen      compile      directory      dir.dist.classes.sharp=dist/sharpen
# Eclipse plugins home plugins.home=${eclipse.home}/plugins

```

(The paths should be updated to match your environment)

```

Sharpen-Common.Xml
<project name="sharpen common">
<property file="sharpen.properties" />

<macrodef name="reset-dir">
<attribute name="dir" />
<sequential>
<delete dir="@{dir}" />
<mkdir dir="@{dir}" />
</sequential>
</macrodef>

<macrodef name="sharpen">
<attribute name="workspace" />
<attribute name="resource" />

<element name="args" optional="yes" />

<sequential>
<exec executable="${file.jvm.jdk1.5}"
failonerror="true" timeout="1800000">
<arg value="-Xms256m" />
<arg value="-Xmx512m" />
<arg value="-cp" />
<arg value="${eclipse.startup.jar}" />
<arg value="org.eclipse.core.launcher.Main" />
<arg value="-data" />
<arg file="@{workspace}" />
<arg value="-application" />
<arg value="sharpen.core.application" />
<arg value="-header" />
<arg file="config/copyright_comment.txt" />
<arg value="@{resource}" />

<args />

</exec>
</sequential>
</macrodef>

```

```

<target name="install-sharpen-plugin">

    <property name="sharpen.core.dir" location="../../sharpen.core" />
    <reset-dir dir="${dir.dist.classes.sharp}" />

    <echo>${eclipse.home}/plugins</echo>
    <javac fork="true" debug="true" target="1.5" source="1.5"
destdir="${dir.dist.classes.sharp}" srccdir="${sharpen.core.dir}/src" encoding="UTF-8">
        <classpath>
            <fileset dir="${eclipse.home}/plugins">
                <include name="org.eclipse.osgi_*/osgi.jar" />
                <include name="org.eclipse.core.resources_*/resources.jar" />
                <include name="org.eclipse.core.runtime_*/runtime.jar" />
                <include name="org.eclipse.jdt.core_*/jdtcore.jar" />
                <!-- redundant entries: in newer eclipse installs those reside in jars -->
                <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>

    <property name="plugin.dir" value="${plugins.home}/sharpen.core_1.0.0" />
    <reset-dir dir="${plugin.dir}" />
    <jar destfile="${plugin.dir}/sharpen.jar" basedir="${dir.dist.classes.sharp}" />
    <copy todir="${plugin.dir}" file="${sharpen.core.dir}/plugin.xml" />

</target>

</project>

```

Sharpen.properties file defines some common properties that are used to install sharpen and then to convert java to c# using sharpen.

In sharpen-common.xml we define install-sharpen-plugin target, which will install sharpen plugin. After the plugin is installed we can use sharpen task to convert java files. Sharpen task takes 2 arguments:

- workspace - defines the workspace where the conversion will take place
- resource - defines the folder where java resources are located

Example Conversion

Now that we have a [script to install sharpen](#) we can try to do a simple conversion.

For example let's create a new project SharpenTutorial with [PersistentExample.Java](#) and [Pilot.Java](#) classes from reference query examples. Try to run PersistentExample in Eclipse to make sure that the project builds correctly.

As the next step we should add sharpen.properties and sharpen-common.xml files from the [previous chapter](#) to SharpenTutorial.

Now we are ready to create a build script, which will convert our java project to c#:

```
Build-Net.Xml
<?xml version="1.0"?>
<project name="Db4objects.Db4odoc" default="sharpen-docs">
    <property file="sharpen.properties" />
    <import file="sharpen-common.xml" />

    <taskdef name="updatecsharpproject"
    classname="com.db4o.devtools.ant.UpdateCSharpProjectAntTask">
        <classpath>
            <pathelement location="lib" />
            <path path="${path.classpath.full}" />
        </classpath>
    </taskdef>

    <!-- Main target -->
    <target name="sharpen-docs" depends="install-sharpen-plugin, clean">

        <property name="target.dir" location="sharpen" />

        <!-- Copy java files to the resource folder -->
        <copy todir="${target.dir}/sharpened_examples/src">
            <fileset dir="src">
                <include name="**/*.java" />
            </fileset>
        </copy>

        <!-- Sharpen java files -->
        <sharpen workspace="${target.dir}" resource="sharpened_examples/src">
            <args>
                <!-- classpath needed for java sources compilation -->
                <arg value="-cp" />
                <arg path="lib/db4o-7.2.37.10417-java5.jar" />
                <!-- Sharpen options are defined in a separate file -->
                <arg value="@sharpen-all-options" />
            </args>
        </sharpen>

        <!-- Define locations for the converted resources -->
        <property name="dir.proj" location="../sharpen-tutorial.net/Db4objects.Db4odoc" />
        <property name="dir.proj.src" location="../sharpen-tutorial.net/Db4objects.Db4odoc" />
        <property name="dir.sharpen.src" location="sharpen/sharpened_examples.net" />

        <!-- Task for copying converted resources -->
        <macrodef name="copy-sharpened-sources">
            <attribute name="dir" />
```

```

<element name="files" />
<sequential>
    <copy todir="@{dir}">
        <files />
    </copy>
</sequential>
</macrodef>

<!-- Copy sharpened file to the destination folder -->
<copy-sharpened-sources dir="${dir.proj.src}">
    <files>
        <fileset dir="${dir.sharpen.src}/src">
            <include name="**/*.cs" />
        </fileset>
    </files>
</copy-sharpened-sources>

<!-- Define c# resources -->
<fileset id="core.net.files" dir="${dir.proj}">
    <include name="**/*.cs" />
</fileset>

<!-- If we already have a csproj file, we can just update it to include
all the added c# resources -->
<updatecsharpproject projectfile="${dir.proj}/sharpen-examples.csproj">
    <sources refid="core.net.files" />
</updatecsharpproject>

</target>

<!-- Remove resources from the previous conversion run -->
<target name="clean" description="Delete all generated files">
    <delete failonerror="false" includeemptydirs="true"
description="Removing all generated files">
        <fileset dir="../reference.cs.net">
            <include name="**/*" />
            <exclude name="**/*.csproj" />
            <exclude name="**/*.sln" />
        </fileset>
    </delete>
    <delete failonerror="false" includeemptydirs="true"
description="Removing all generated files">
        <fileset dir="sharpen">
            <include name="**/*" />
            <exclude name="**/lib/*" />
        </fileset>
    </delete>
</target>
</project>

```

As you can see from the build script the sequence of actions is the following:

1. Install sharpen plugin
2. Clean up the destination folder

3. Sharpen java resources and copy the resulted c# files to the destination

sharpen-all-options file provides options for the conversion:

```
@sharpen-collections-mapping -pascalCase+ -nativeTypeSystem -nativeInterfaces -fullyQualifyFile - methodMapping System.out.println System.Console.WriteLine - typeMapping com.db4o.Db4o Db4oObjects.Db4o.Db4oFactory - typeMapping com.db4o.ext.ExtDb4o Db4oObjects.Db4o.Ext.ExtDb4oFactory - namespaceMapping com.db4o Db4oObjects.Db4o - namespaceMapping com.db4odoc Db4oObjects.Db4odoc - namespaceMapping java Sharpen - propertyMapping java.lang.System.out Sharpen.Runtime.Out - propertyMapping java.lang.System.err Sharpen.Runtime.Err
```

It also references another options file for an easier structuring:

```
-methodMapping java.util.Iterator.hasNext -methodMapping java.util.Iterator.next Next - typeMapping com.db4o.ObjectSet Db4oObjects.Db4o.IObjectSet - methodMapping com.db4o.ObjectSet.hasNext Db4oObjects.Db4o.IObjectSet.HasNext - methodMapping com.db4o.ObjectSet.next Db4oObjects.Db4o.IObjectSet.Next - typeMapping java.util.Comparator System.Collections.IEnumerator -typeMapping java.util.ArrayList System.Collections.ArrayList - methodMapping java.util.List.addAll AddRange -methodMapping java.util.AbstractList.addAll AddRange -methodMapping java.util.ArrayList.addAll AddRange
```

Sharpen [command-line arguments](#) and [annotations](#) are discussed in detail in the following chapters.

PersistentExample.java

```
PersistentExample.java
package com.db4odoc.sharp;

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

public class PersistentExample {
    private final static String DB4O_FILE_NAME = "reference.db4o";

    public static void main(String[] args) {
        new File(DB4O_FILE_NAME).delete();
        accessDb4o();
        new File(DB4O_FILE_NAME).delete();
        ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
        try {
            storeFirstPilot(container);
            storeSecondPilot(container);
            retrieveAllPilots(container);
            retrievePilotByName(container);
            retrievePilotByExactPoints(container);
            updatePilot(container);
            deleteFirstPilotByName(container);
        }
    }
}
```

```

        deleteSecondPilotByName(container);
    } finally {
        container.close();
    }
}
// end main

private static void accessDb4o() {
    ObjectContainer container = Db4o.openFile(DB4O_FILE_NAME);
    try {
        // do something with db4o
    } finally {
        container.close();
    }
}
// end accessDb4o

private static void storeFirstPilot(ObjectContainer container) {
    Pilot pilot1 = new Pilot("Michael Schumacher", 100);
    container.store(pilot1);
    System.out.println("Stored " + pilot1);
}
// end storeFirstPilot

private static void storeSecondPilot(ObjectContainer container) {
    Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
    container.store(pilot2);
    System.out.println("Stored " + pilot2);
}
// end storeSecondPilot

private static void retrieveAllPilotQBE(ObjectContainer container) {
    Pilot proto = new Pilot(null, 0);
    ObjectSet result = container.queryByExample(proto);
    listResult(result);
}
// end retrieveAllPilotQBE

private static void retrieveAllPilots(ObjectContainer container) {
    ObjectSet result = container.queryByExample(Pilot.class);
    listResult(result);
}
// end retrieveAllPilots

private static void retrievePilotByName(ObjectContainer container) {
    Pilot proto = new Pilot("Michael Schumacher", 0);
    ObjectSet result = container.queryByExample(proto);
    listResult(result);
}
// end retrievePilotByName

private static void retrievePilotByExactPoints(
    ObjectContainer container) {
    Pilot proto = new Pilot(null, 100);
    ObjectSet result = container.queryByExample(proto);
}

```

```

        listResult(result);
    }
    // end retrievePilotByExactPoints

    private static void updatePilot(ObjectContainer container) {
        ObjectSet result = container.queryByExample(new Pilot(
            "Michael Schumacher", 0));
        Pilot found = (Pilot) result.next();
        found.addPoints(11);
        container.store(found);
        System.out.println("Added 11 points for " + found);
        retrieveAllPilots(container);
    }
    // end updatePilot

    private static void deleteFirstPilotByName(
        ObjectContainer container) {
        ObjectSet result = container.queryByExample(new Pilot(
            "Michael Schumacher", 0));
        Pilot found = (Pilot) result.next();
        container.delete(found);
        System.out.println("Deleted " + found);
        retrieveAllPilots(container);
    }
    // end deleteFirstPilotByName

    private static void deleteSecondPilotByName(
        ObjectContainer container) {
        ObjectSet result = container.queryByExample(new Pilot(
            "Rubens Barrichello", 0));
        Pilot found = (Pilot) result.next();
        container.delete(found);
        System.out.println("Deleted " + found);
        retrieveAllPilots(container);
    }
    // end deleteSecondPilotByName

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

```

Pilot.java

```

Pilot.java
package com.db4odoc.sharp;

public class Pilot {
    private String name;

```

```

private int points;

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

public int getPoints()  {
    return points;
}

public void addPoints(int points)  {
    this.points += points;
}

public String getName()  {
    return name;
}

public String toString()  {
    return name + "/" + points;
}
}

```

Sharpen Command-Line Arguments

Sharpen 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="db4o/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
-namesMapping	Maps a java package name to a .NET namespace. For example:
paceMapping	-namespaceMapping com.db4o Db4objects.Db4o
-methodMapping	Maps a java method name to a .NET method (can be method in another class). For example: -methodMapping java.util.Date.getTime Sharpen.Runtime.ToDateTime
-typeMapping	Maps a java class to .NET type: -typeMapping com.db4o.Db4o Db4objects.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.
-header	For a complete list of all the method that can be mapped to the runtime class see Configuration#runtimeMethod call hierarchy. Header comment to be added to all converted files.
<arg value="-header" />	
<arg file="config/copyright_comment.txt" />	
-xml/doc	Specifies an xml-overlay file, which overrides javadoc documentation for specific classes: <arg value="-xml/doc" /> <arg file="config/sharpen/ApiOverlay.xml" />

Sharpen Annotations

Sharpen 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

@sharpen.enum
@sharpen.rename

@sharpen.private

@sharpen.internal

@sharpen.event

@sharpen.event.add

Meaning

Mark java class to be processed as a .NET enum

Specifies a different name for the converted type, takes a single name argument. For example:

@sharpen.rename Db4oFactory

Specifies that the element must be declared private in the converted file, though it can be not private in the java source:

```
* @sharpen.private
```

```
*/
```

```
public List4 _first;
```

Specifies that the element must be declared internal in the converted file:

```
/**
```

```
* @sharpen.internal
```

```
*/
```

```
public abstract int size();
```

Links an event to its arguments. For example:

Java:

```
/**
```

```
* @sharpen.event com.db4o.events.QueryEventArgs
```

```
*/
```

```
public Event4 queryStarted();
```

is converted to:

c#:

```
public delegate void QueryEventHandler(object sender, Db4oObjects.Db4o.Events.QueryEventArgs
```

```
args);
```

```
.....
```

```
event Db4oObjects.Db4o.Events.QueryEventHandler QueryStarted;
```

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>

<code>@sharpen.event.onAdd</code>	Valid for event declaration only (SHARPEN_EVENT). Configures the method to be invoked whenever a new event handler is subscribed to the event.
<code>@sharpen.if</code>	Add <code>#if <expression>#endif</code> declaration:
<code>@sharpen.property</code>	<code>@sharpen.if <expression></code> Convert a java method as a property: <code>/** * @sharpen.property */</code>
<code>@sharpen.indexer</code>	<code>public abstract int size();</code> Marks an element as an indexer property
<code>@sharpen.ignore</code>	Skip the element while converting
<code>@sharpen.ignore.extends</code>	Ignore the extends clause in Java class definition
<code>@sharpen.ignore.implements</code>	Ignore the implements clause in Java class definition
<code>@sharpen.extends</code>	Adds an extends clause to the converted class definition. For example:
	Java: <code>/** * @sharpen.extends System.Collections.IList */</code>
	<code>public interface ObjectSet {...</code> converts to c#: <code>public interface IObjectSet : System.Collections.IList</code> Marks the converted class as partial
<code>@sharpen.partial</code>	Marks a method invocation that should be removed
<code>@sharpen.remove</code>	Marks class to be converted as c# struct
<code>@sharpen.struct</code>	

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:

- [Using The Sources From Db4o Distribution](#)
- [Using The Repository](#)
- [Db4o Directory Structure](#)
- Configuring Java System Libraries
- [Building Full Distribution](#)
- [Building Java Version](#)
- [Sharpen Set-Up For Db4o Build](#)
- [Testing Db4o](#)
- [Patch Submission](#)
- [Project Dependencies](#)
- [Coding Style](#)

Using The Sources From Db4o Distribution

The source code is available in /src folder in your db4o distribution.

In Java version the following projects are available:

bloat

[Byte-code optimization library](#)

db4o.instrumentation

Bytecode instrumentation layer on top of bloat

db4o osgi

db4o OSGI bundle.

db4o osgi test

Tests for db4o OSGI bundle.

db4oj

db4o core.

db4oj.tests

collection of db4o regression tests.

db4ojdk1.2

db4o sources for JDK1.2-1.4

db4ojdk5

db4o sources for JDK 5-6

db4otaj

Transparent Activation instrumentation library

db4otools

Bytecode instrumentation tools

db4onqopt

db4o native query optimization library

db4ounit

[unit test framework](#) for db4o.

db4ounit.extensions

Db4oUnit extensions for [testing db4o](#).

We recommend [Eclipse](#) to work with the sources.

First you will need to configure Java System Libraries. Having done that use the following steps to import the source project into the development environment:

- open a workspace;
- select File/New/Java Project;
- type in a project name (for example "db4o");

- select "Create project from existing source" and browse to the src/db4o folder in your installation (you may want to copy it to another location first);
- click "Next" and "Finish".

Further reading:

[Db4o Directory Structure](#)

[Building Java Version](#)create it

[Building .NET Version](#)create it

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.

If you are using [Eclipse](#) it may be convenient for you to use [Subversive plugin](#) as the Subversion client.

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. If a top-level module acts as a container for related projects rather than as a standalone project, its name should be suffixed with **-projects** by convention, indicating that you'll have to look inside to find the actual project folders to check out.

bloat

Bytecode instrumentation library

cruisecontrol

Cruisecontrol project for db4o build

dashboard

db4o eclipse utility to watch db4o cruisecontrol build status

db4o-update-site

db4o OSGI plugin installation and update site

db4o.archives

Collection of db4o versions.

db4o.cs

db4o client/server specific sources

db4o.instrumentation

Bytecode instrumentation for db4o.

db4o.net

db4o for .NET sources including dRS

db4oME db4o for J2ME CLDC (spike project, to be integrated with regular production)

db4obuild

Resources and scripts to build db4o

db4obuild.tests

Tests for db4obuild

db4odebugutils

Debug utilities for db4o **db4oj** db4o core

db4oj.optional

optional db4o components (separate project to keep the core smaller)

db4oj.tests

Collection of db4o regression tests

db4onqopt

Native query optimizer for Java

db4opolepos

[Poleposition](#) benchmark for db4o

db4otaj

Instrumentation for db4o TA

db4otaj.tests.integration

Transparent Activation tests

db4otestecclipse

db4ounit integration in Eclipse

db4otools

Instrumentation tools for db4o

db4ounit

[Unit-test framework for db4o](#) (no db4o dependencies)

db4ounit.extensions

db4o-specific extensions for [db4o testing framework](#)

decaf

decaf - Java SDK version converter

decaf.functional.tests - decaf functional tests

decaf.tests - decaf tests

docWiki

Offline version of [db4o reference documentation](#)

doctor Internal db4o documentation system

dRS

db4o replication system

drspolepos

Polepolition benchmarks for dRS

eiffel

Tests with Eiffel for .NET

enterprise

omj Object Manager Enterprise for Java (Eclipse plugin)

omn Object Manager Enterprise for .NET (Visual Studio plugin)

objectmanager

Old Object Manager sources

objectmanager-api Object Manager API

objectmanager-swing Object Manager (database browser)

osgi-projects
osgi_db4o - OSGi db4o bundle with factory service **osgi_db4o_test** - bundle to run db4ojdk1.2 test suite in OSGi context

reference

Collection of examples for the reference documentation

sandbox

Sandbox

sharpen

sharpen.core - java to c# converter
sharpen.builder - java to c# converter
sharp.ui - Eclipse UI
java to c# converter extension **sharp.ui.tests** - tests for sharpen.ui

spikes

Placeholder for db4o spike projects

sync4o Client and Server sync sources for the Funambol sync4j system.

tutorial

Tutorial sources

version6converter

Converter for .NET legacy projects to db4o version 6 .NET conventions

The latest development source code is available from TRUNK.

For branches, tags and versions we recommend downloading the distributions from our [download center](#).

Further reading:

[Db4o Directory Structure](#)

[Building Java Version](#)

Building .NET Version

Db4o Directory Structure

This topic will explain the directory structure of the db4o project.

In order to keep db4o core library small and suitable for installation on devices with limited memory, db4o projects are separated into core, client/server and optional.

Core project

Java:

db4oj

This project contains the smallest, yet fully-functional db4o distribution. It is recommended for use in embedded mode on devices.

The following folders are used within the db4o core project.

Activation - Transparent Activation support classes.

Collections - db4o-enabled collections

Config - contains configuration interface and other classes and interfaces used for db4o tuning and configuration.

annotations - annotations, which can be used to configure db4o

Encoding - string encoding implementations

Constraints -unique constraints code.

Defragment - defragmentation code and related service classes and interfaces.

Diagnostic - diagnostics classes and interfaces.

Events - external events implementation.

Ext - extended db4o functionality .

Foundation - db4o base classes and interfaces.

Io - file system related code.

Internal - internal db4o logic.

Activation - internal activation implementation

Btree- b-Tree implementation. Used for indexing, freespace, defragment etc.

Caching - contains code for cache mechanisms that can be used on db4o IO level

Callbacks - callbacks definitions.

ClassIndex- class index implementation.

Collections - fast collections development classes.

Config - configuration implementations

Convert - version converter.

Delete - deletion context used in typehandlers

Diagnostic - diagnostic processor.

Encoding - string encoding implementations

Events- internal events implementation.

Fieldhandlers - field handler implementation for typrhandlers

Fieldindex - field index logic

Fileheader - classes for handling db4o file header.

Freespace - freespace management code.

Handlers - different type handlers (array, byte, char etc).

Io - block size implementation

Mapping - internal mapping implementation for defragment.

Marshall - different type marshallers.

Query - query logic implementation.

Reflect - reflection layer implementation

Replication - deprecated replication logic.

Slots - classes dealing with slots in db4o file.

TransactionLog - transaction log implementation

IO - db4o IoAdapter implementations.

Marshal - marshalling interfaces.

Messaging - messaging interfaces for client/server communications.

Query - classes and interfaces for different query types.

Reflect - reflection interfaces, generic reflector implementation.

Core - abstract reflect classes.

Generic - generic reflector implementation.

Jdk - wrapper classes to JDK reflection.

Replication- db4o replication code

Ta - transparent activation code

Typehandlers - typehandler interfaces

Types - db4o specific types

Client/Server project

Java:

db4o.cs

This is a client-server version of db4o (depends on db4o core project).

The following folders are included:

CS - client/server object container

Config - configuration for C/S version

Foundation - db4o base classes and interfaces.

Network - classes for network communications (buffer, socket).

Internal - internal db4o logic.

CS - client/server implementation code

Optional Project

Java:

db4oj.optional

This project contains optional db4o components (depends on db4o core).

The following folders are included:

Cluster - contains cluster source code, which allows queries against several databases. Work in progress.

Internal - internal db4o logic.

Cluster - internal cluster code.

IO - db4o IoAdapter implementations.

Reflect - reflection interfaces, generic reflector implementation.

Self - reflector implementations for JDK platforms without reflection support.

Tools - additional db4o tools

Building Full Distribution

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 - bytecode optimization library, required for db4o NQ optimizer

db4o.cs - db4o client/server library code

db4o.instrumentation - db4o instrumentation

db4o.net - db4o .NET native sources (.NET specific code that cannot be converted from Java)

db4obuild - db4o build tool, contains build scripts and other data necessary to build the distro

db4obuild.tests - tests for db4obuild project

db4oj - db4o core

db4oj.optional - db4o optional library code

db4oj.tests - db4o tests package

db4onqopt - db4o NQ optimizer

db4otaj - TA instrumentation

db4otaj.tests.integration - TA instrumentation tests

db4otools - bytecode instrumentation for db4o

db4ounit - db4o unit-test framework

db4ounit.extensions - db4o unit-test framework extensions

decaf /decaf, decaf/annotations, decaf/decaf.functional.tests, decaf/decaf.tests - JDK versions converter, allows generating db4o libraries for JDK1.1 - JDK1.6 from a single code base.

docWiki - reference documentation for db4o

doctor - documentation generation tool, required for building interactive tutorial

osgi-projects/db4o-osgi - db4o OSGI library

osgi-projects/db4o-osgi-tests - tests for db4o OSGI

sharpen\sharpen.builder, sharpen\sharpen.core - java to c#converter plugin for Eclipse. These are not needed if you are only interested in building java distribution

tutorial - interactive 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.

If you do not have a JDK 1.3 installed and you want to use JDK 5 or JDK 6 to build db4o for Java 1.1, you will need this line:

```
file.compiler.jdk1.3.args.optional=-source 1.3
```

An example of the minimum machine.properties file:

```
dir.workspace=c:/workspaces/db4obuild    file.compiler.jdk1.3=c:/java/jdk/jdk1.6.0/bin/javac.exe  
file.jvm.jdk1.5=C:/java/jdk1.6.0/bin/java    dir.compactframework=C:/Program Files/Microsoft.NET/SDK/CompactFramework  
compiler.jdk1.3.args.optional=-    source      1.3    eclipse.startup.jar=c:/java/eclipse-3.3.1/plugins/org.eclipse.equinox.launcher_1.0.1.R33x_v20070828.jar    font.pdf.base=C-  
:WINDOWS/Fonts/VERDANA.TTF no.ftp=true
```

Build Preparation

First you will need to run some preparation scripts. This is done only once per workspace and should not be repeated in the future.

Run build-db4obuild.xml, this will compile some of the tools used in the build process.

You will need to generate a key to sign the tutorial applet. Use the following commands:

```
keytool -genkey -alias db4objects -keyalg rsa
```

```
keytool -export -alias db4objects -file [path]/db4obuild/config/db4objects.crt
```

Use "kistoa" (without quotes) as your keypass and storepass.

Replace [path] with the path to db4obuild project on your system and make sure that db4objects.crt file is created in db4obuild/config folder.

If you've already generated db4objects key pair before, you will need to delete it before re-generating:

```
keytool -delete -alias db4objects
```

You will need to add ant-contrib.jar to your eclipse ant. You can download ant-contib.jar at:

<http://sourceforge.net/projects/ant-contrib>

Add ant-contib jar to ant folder in eclipse/plugins. After this is done go to Window->Preferences menu in Eclipse. Select Ant->Runtime in the list. Then select "Ant Home Entries", press "Add External Jar" and select ant-contib.jar location in the plugins folder.

Running The Build

Now everything is ready to run db4o build. Right-click build.xml file and select "Run As/Ant Build". You will need to run "buildall" target to generate java and .NET distribution.

Building Java Version

In order to build Java distribution only you should follow the instructions in [Building Full Distribution](#) and then start the build by right-clicking build.xml file and selecting "Run As/Ant Build". You will need to run "buildjava" target.

The ready distribution can be found in /dist folder of db4obuild project

Sharpen Set-Up For Db4o Build

Sharpen is used to translate Java version of db4o to c#. The basic process is defined in **sharpen-alltarget** of **db4obuild\build-dotnet.xml**. In the primitive case defining sharpen task and applying it to java source should be enough:

```
Sharpen.Xml
<!-- Define sharpen task-->
<macrodef name="sharpen">
    <attribute name="workspace" />
    <attribute name="resource" />

    <element name="args" optional="yes" />

    <sequential>
        <echo>java -cp ${eclipse.startup.jar}
org.eclipse.core.launcher.Main -data ${workspace} -application
sharpen.core.application ${resource}</echo>
```

```

<exec executable="${file.jvm.jdk1.5}" failonerror="true"
timeout="1800000">
    <arg value="-Xms256m" />
    <arg value="-Xmx512m" />
    <arg value="-cp" />
    <arg value="${eclipse.startup.jar}" />
    <arg value="org.eclipse.core.launcher.Main" />
    <arg value="-data" />
    <arg file="@{workspace}" />
    <arg value="-application" />
    <arg value="sharpen.core.application" />
    <arg value="-header" />
    <arg file="${dir.config}/copyright_comment.txt" />
    <arg value="@{resource}" />

    <args />

</exec>
</sequential>
</macrodef>

<!-- Call sharpen to convert db4oj/core/src into dist.sharpen -->
<sharpen workspace="dist.sharpen" resource="db4oj/core/src" />

```

However, the complexity of the translation requires some fine tuning. Let's look at the options and annotations used to make db4o conversion working and produce a reasonable result.

Conversion Scope

Though Sharpen provides rather sophisticated framework, some of the java language structures cannot be translated to c# without losing some of the readability and nativeness of the c# code. One of the examples is IO API. In these cases a full c# class is re-written manually. These classes are excluded from the conversion by using @sharpen.ignore annotation. However this solves only half of the problem, because the manually-translated classes still should be placed correctly in the corresponding namespaces. This is achieved by creating a c# project structure containing only manually translated classes. The result of the sharpening is then copied into this structure, resulting in a full and valid c# project.

Naming

Naming conventions differ for java and c#, but db4o code should look native in both environments. There are several options that help to achieve this:

-pascalCase+ - enforces Pascal case for identifies and namespaces

-nativeTypeSystem - use .NET typesystem

-nativeInterfaces - add "I" in front of the interface name

Sharpen options also include different re-namings to make c# names look more natural:

-typeMapping com.db4o.Db4o Db4objects.Db4o.Db4oFactory

```
-namespaceMapping com.db4o Db4oObjects.Db4o
```

and others.

c# supports a concept of type properties whereas Java getter and setter methods are used. To convert those properly propertyMapping option is defined:

```
-propertyMapping com.db4o.foundation.Iterator4.current Current
```

For more information on these and other options see [Sharpen Command-Line Arguments](#).

Testing Db4o

After you've modified the sources and built the library it is always a good idea to test your new version thoroughly. For this purpose, you can use db4o regression tests.

The full Java test suite is run as part of [normal java build](#). You can also run db4o tests separately as java classes or use "run.tests.*" targets from build-java.xml.

For more information see [Db4o Testing Framework](#).

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.

Project Dependencies

Project	Dependencies
db4o_osgi	JDK6 db4o-7.12-db4ounit-java1.2.jar org.eclipse.osgi_3.4.0.v20080605-

	1900.jar
db4o osgi test	JDK6 db4o-7.12-all-java1.2.jar org.eclipse.osgi_3.4.0.v20080605-1900.jar db4o-osgi db4o-tests.jar db4o-tests-nq.jar
db4o.cs	JDK1.5 db4oj
db4o.instrumentation	JDK1.5 bloat db4oj db4ounit db4ounit.extensions ant.jar
db4o.net	.NET2.0 .NET3.5 .NET2.0.CF .NET3.5.CF
db4oj	JDK1.5 decaf-annotations
db4oj.optional	JDK1.5 db4oj
db4oj.tests	JDK1.5 easymock.jar db4o.cs db4oj db4oj.optional db4ounit db4ounit.-extensions
db4onqopt	JDK5 bloat db4o.instrumentation db4oj db4oj.optional db4oj.tests db4ounit db4ounit.extensions
db4otaj	JDK5 bloat db4o.instrumentation db4oj db4oj.optional db4oj.tests db4ounit db4ounit.extensions
db4otools	JDK5 bloat db4o.instrumentation db4oj db4onqopt db4otaj db4ounit db4ounit.extensions
db4ounit	JRE5 db4oj
db4ounit.extensions	JRE5 db4o.cs db4oj db4ounit
decaf	JRE5 eclipse plugins sharpen.core decaf.annotations
decaf.annotations	JRE5
decaf.tests	JRE5
decaf.tests.functional	JRE5 ant.jar asm-3.1.jar JUnit3
doctor	JRE5 ant.jar itext.jar junit.jar
docWiki	None
drs	db4o.cs db4oj db4ounit ant.jar antlr-2.7.6.jar asm.jar asm-attrs.jar cglib-2.1.3.jar commons-collections-2.1.1.jar commons-logging-1.0.4.jar db2_license_cu.jar db4cc.jar derby.jar doctor.jar dom4j-1.6.1.jar ehcache-1.2.jar hibernate3.jar hsqldb.jar jText.jar jta.jar junit.jar log4j-1.2.13.jar mysql-connector-java.5.0.4-bin.jar odbc14.jar postgresql-8.1-407.jdbc3.jar sqljdbc.jar xerces-2.6.2.jar
drspolepos	JDK1.5 db4oj db4ojdk1.2 db4ojdk5 db4oj.tests db4ounit db4ounit.-extensions drs polepos
omj	JDK1.5 other libraries included in svn
omn	.NET2.0 .NET3.5 other libraries included in svn
sharpen.builder	JRE5 xstream-1.1.2.jar eclipse plugin libraries
sharpen.core	JRE6 eclipse plugin libraries
sharpen.ui	JRE6 sharpen.core sharpen.builder eclipse plugin libraries
sharpen.ui.tests	JRE6 sharpen.core sharpen.builder sharpen.ui eclipse plugin libraries
Tutorial	JRE5 db4oj db4oj.tools

Coding Style

Coding conventions proved to be very important for producing maintainable and reliable code. In

db4o production cycle, coding conventions have a special value, as the code ownership is spread over all the members of the team and any developer is able to work with any piece of code.

In general, we follow Code Conventions for the Java Programming Language, however there are some specifics, which can be useful to know for db4o users and core contributors.

This document is supposed to emphasize some of the java coding style recommendations used by db4o and explain db4o specific coding style requirements.

File Header

All code files must have copyright. The normal db4o copyright notice should be created as the standard template in your Eclipse workspace for db4o development. Window + Preferences + Java + Code Style + Code Templates + Code + New Java Files /* Copyright (C) 2004 - 2009 Versant Inc. http://www.db4o.com */ \${package_declaration} \${typecomment} \${type_declaration}

Naming Conventions

General Naming

All names should be written in English.

English is the preferred language for international development.

Package naming

Package names should be in all lower case.

com.db4o.reflection

Class Naming

Class names should be nouns and written in mixed case starting with upper case.

ObjectContainer, Configuration

Methods Naming

Method names must be verbs and written in mixed case starting with lower case.

isReadOnly(), rename()

Abbreviations and Acronyms

Abbreviations and acronyms should not be uppercase when used as name.

IoAdapter(); // NOT: IOAdapter

Using all uppercase for the base name will give conflicts with the naming conventions given above and will reduce the readability.

Type Naming

Type names must be nouns and written in mixed case starting with upper case.

```
Db4oList, TransientClass
```

Variable Naming

Variable names must be in mixed case starting with lower case. Variables should have full sensible name, reflecting their purpose.

```
listener, objectContainer
```

- Private Variables

Private class variables should have underscore prefix.

```
public abstract class IoAdapter {  
    private int _blockSize;  
    ....  
}
```

Underscore prefix will help a programmer to distinguish private class variables from local scratch variables.

- Scratch Variables

Scratch variables used for iterations or indices should be kept short. Common practice is to use i, j, k, m, n for numbers and c, d for characters.

Constants

Constants names (final variables) must be all uppercase using underscore to separate words.

```
ACTIVATION_DEPTH, READ_ONLY
```

It is a good practice to add methods to retrieve constant values for user interface:

```
boolean isReadOnly() {  
    return _config.getAsBoolean(READ_ONLY);  
}
```

Getters/Setters

Normally we do not use *get/set* prefix for methods accessing attributes directly, unless such usage adds valuable information:

```
public void setStateDirty() {}
```

In other cases feel free to access attributes by names:

```
clientServer(), configuration()
```

Boolean Methods And Variables

is(can, has, should) prefix should be used for boolean variables and methods.

```
isDirty(), canHold(reflectClass)
```

Using the *is(can, has, should)* prevents choosing bad names like *status* or *flag*. *isStatus* or *isFlag* simply doesn't fit, and the programmer is forced to choose more meaningful names.

Setter methods for boolean variables must have *set* prefix as in:

```
public void setStateDirty() {}
```

Initialize

The term *initialize* can be used where an object or a concept is established. `classIndex.initialize(_targetDb);`

Abbreviations like *init* must be avoided.

Complementary Names

Complementary names must be used for complementary entities:

get/set, add/remove, create/destroy, start/stop, insert/delete, increment/decrement, old/new, begin/end, first/last, up/down, min/max, next/previous, old/new, open/close, show/hide, suspend/resume, etc. For example:

```
startServer();  
stopServer();
```

This convention helps to distinguish the borders of a logical operation and to recognize opposite action methods.

Abbreviations

Abbreviations in names should be avoided.

```
copyIdentity(); // NOT: cpIdentity, NOT:copyId
```

However some well established and commonly used acronyms or abbreviations must be preferred to full names:

```
html // NOT: HypertextMarkupLanguagecpu // NOT: CentralProcessingUnit
```

Named Constants

Named constants should be used instead of:

- **magic numbers:**
`if (blockSize > MAX_BLOCK_SIZE) // NOT: blockSize > 256`
- **fixed phrases:**
`if (fieldname == CREATIONTIME_FIELD) // NOT if (fieldname == "i_uuid")`

This convention gives a programmer an idea about the meaning of the constant value. At the same time, it makes it easier to change the constant value: the change must be made only in one place.

Code Organization

Package Structure

Internal class implementations should be placed in com.db4o.internal package. This helps to keep the top-level API smaller and more understandable.

`com.db4o.query.Evaluation`

has implementation in

`com.db4o.internal.query.PredicateEvaluation`

Classes and Interfaces

Class and Interface declarations should be organized in the following manner:

1. Class/Interface documentation.
2. `class` or `interface` statement.
3. Class (static) variables in the order `public, protected, package`(no access modifier), `private`.
4. Instance variables in the order `public, protected, package`(no access modifier), `private`.
5. Constructors.
6. Methods.

Methods

Group class methods by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding of the code easier.

Wait & Notify

Use `Lock4#snooze()`, `#awake()` instead of `Object#wait` directly for CF reason.

Blank Lines

Use blank lines to separate methods, class variable declarations of different scope, and logical units within a block of code. Consider extracting logical blocks of code into separate methods

Import Declarations

Import declarations should only include package name:..

```
import java.util.*; // NOT: import java.util.List;
```

Modern IDEs, such as Eclipse, provide an automated way to create correct import statements (see "Source/Organize Imports" command in Eclipse).

Initialization

Local variables should appear at the beginning of a code block. (A block is any code surrounded by curly braces "{" and "}".) Try to initialize the variable immediately to prevent using uninitialized values.

The exception to the rule is indexes of `for` loops, which in Java can be declared in the `for` statement:

```
for (int i = 0; i < maxLoops; i++) { ... }
```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;  
...  
myMethod() {  
    if (condition) {  
        int count = 0; // AVOID!  
        ...  
    }  
    ...  
}
```

Exception Handling

Never use `exception#printStackTrace()` in db4o productive code.

There are three possible choices that can be used:

- throw (the base type is `Db4oException`)
- swallow silently
- swallow with a `com.db4o.diagnostic` message to the user

Comments

Supply your code with javadoc comments. All public classes and public and protected functions within public classes should be documented. This makes it easy to keep up-to-date online code documentation. If a class or a method is not part of public API use `@exclude` tag. For further details, see "How to Write Doc Comments for Javadoc"

All comments should be written in English. In an international environment, English is the preferred language.

Avoid using comments to explain tricky code, rather rewrite it to make it self-explanatory.

For more information see: Code Conventions for the Java Programming Language.

This document was compiled based on db4o team coding practices and the following documents:

<http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.htm>

<http://geosoft.no/development/javastyle.htm>

<http://java.sun.com/j2se/javadoc/writingdoccomments/index.htm>

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

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

Support

Visit our [free Community Forums](#)

or log into your [Customer Portal](#) (dDN Members Only).

Careers

career@db4o.com

Partnering

partner@db4o.com