Welcome

Welcome to the db4o tutorial. In this tutorial we explore the basics of db4o and how to work with it.

Community Help

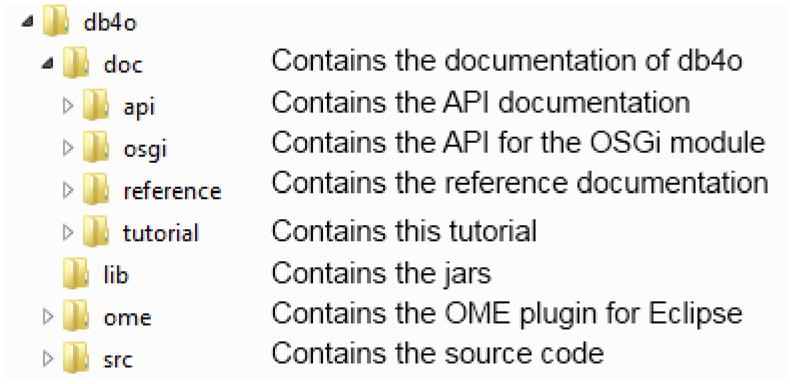
If you have any questions ask on the db4o community forums. You will also find news and documentation there.

Getting db4o

There are two ways how you can get db4o. Either you can download a distribution from db4o website or you can use Maven to get it.

Downloading db4o

Download db4o from the developer website. I recommend using the 'production' version of db4o. Choose the download which fits your environment. The download contains the binaries, documentation and source.



db4o	
doc	Contains the documentation of db4o
api	Contains the API documentation
osgi	Contains the API for the OSGi module
reference	Contains the reference documentation
tutorial	Contains this tutorial
lib	Contains the jars
ome	Contains the OME plugin for Eclipse
src	Contains the source code

Using Maven

Another way to grab db4o is via Maven. For the tutorial we just grab the full db4o distribution.

```
<dependency>
    <groupId>com.db4o</groupId>
    <artifactId>db4o-full-java5</artifactId>
    <version>8.1-SNAPSHOT</version>
</dependency>
```

Additionally we need to add the db4o Maven repository.

```
<repositories>
   <repository>
        <id>db4o</id>
        <name>Db4o</name>
        <url>https://source.db4o.com/maven/</url>
   </repository>
</repositories>
```

Adding db4o to a Project

When you are using Maven it will add the needed dependencies for your project. When you downloaded the distribution add following dependencies for this tutorial:

- db4o-X.XX-all-java5.jar

A detailed overview of the dependencies is available in the reference documentation.

db4o Area of Use

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

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

Basics

Let's start with the basics. First we create a tiny domain model, consisting out of drivers and cars. (Click here to see the example domain model classes)

```
public  class Car {
    private String carName;

    public Car(String carName) {
        this.carName = carName;
    }

}
```

Car.java: Domain model for cars

```
public  class Driver {
    private String name;
    private Car mostLovedCar;

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

    public Driver(String name, Car mostLovedCar) {
        this.name = name;
        this.mostLovedCar = mostLovedCar;
    }

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

    public String getName() {
        return name;
    }

    public Car getMostLovedCar() {
        return mostLovedCar;
    }
}
```

Driver.java: Domain model for drivers

Open a Database

For all operations we need to open the database first. To open the database we pass a file-path to db4o. db4o will open or create that database-file an return an object container. The object container is the connection to our database. All operations go through it.

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

BasicOperations.java: Open and close db4o

Store an Object

To store an object in the database, we simply call the store-method on the object container.

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

BasicOperations.java: Store an object

Query the Database

After storing objects we want to query for our stored objects. We can do this with native queries.

```java
ObjectContainer container = Db4oEmbedded.openFile("databaseFile.db4o");
try {
    List<Driver> drivers = container.query(new Predicate<Driver>() {
        public  boolean match(Driver d) {
            return d.getName().equals("Joe");
        }
    });
    System.out.println("Stored Pilots:");
    for (Driver driver : drivers) {
        System.out.println(driver.getName());
    }
} finally {
    container.close();
}
```

BasicOperations.java: Query for objects

Update an Object

In order to update an object we first query for it. Then we change it and store it again. db4o will recognize the updated object and store the changes in the database.

```java
ObjectContainer container = Db4oEmbedded.openFile("databaseFile.db4o");
try {
    List<Driver> drivers = container.query(new Predicate<Driver>() {
        public  boolean match(Driver d) {
            return d.getName().equals("Joe");
        }
    });
    Driver driver = drivers.get(0);
    System.out.println("Old name" +driver.getName());
    driver.setName("John");
    System.out.println("New name" +driver.getName());
    // update the pilot
    container.store(driver);
} finally {
    container.close();
}
```

BasicOperations.java: Update an object

Delete an Object

Deleting an object is easy. First we query for the object and then delete it.

```java
ObjectContainer container = Db4oEmbedded.openFile("databaseFile.db4o");
try {
    List<Driver> drivers = container.query(new Predicate<Driver>() {
        public  boolean match(Driver d) {
            return d.getName().equals("Joe");
        }
    });
    Driver driver = drivers.get(0);
    System.out.println("Deleting " +driver.getName());
    container.delete(driver);
} finally {
    container.close();
}
```

BasicOperations.java: Delete an object

Queries

Let's take a look at a few examples queries. (Click here to see the example domain model classes)

```
public  class Car {
    private String carName;
    private  int horsePower;

    public Car(String carName, int horsePower) {
        this.carName = carName;
        this.horsePower = horsePower;
    }

    public String getCarName() {
        return carName;
    }

    public  int getHorsePower() {
        return horsePower;
    }

    @Override
    public String toString() {
        return  "Car{" +
                "carName='" + carName + '\'' +
                ", horsePower=" + horsePower +
                '}';
    }
}
```

Car.java: Domain model for cars

```
public  class Driver {
    private String name;
    private  int age;
    private Car mostLovedCar;

    public Driver(String name,int age) {
        this.name = name;
        this.age = age;
    }

    public Driver(String name,int age, Car mostLovedCar) {
        this.name = name;
        this.age = age;
        this.mostLovedCar = mostLovedCar;
    }

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

    public String getName() {
        return name;
    }

    public  int getAge() {
        return age;
    }

    public Car getMostLovedCar() {
        return mostLovedCar;
    }

    @Override
    public String toString() {
        return  "Driver{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", mostLovedCar=" + mostLovedCar +
                '}';
    }
}
```

Driver.java: Domain model for drivers

Native Queries

Native queries are the preferred way to query a db4o database. To query the database, we pass in a predicate which filters objects by their properties. Here are two example queries:

Query for a Certain Person

This query finds all drivers named Joe:

```
List<Driver> drivers = container.query(new Predicate<Driver>() {
    @Override
    public  boolean match(Driver driver) {
        return driver.getName().equals("Joe");
    }
});
```

Queries.java: Query for drivers named Joe

More Complex Query

We can combine many criteria in a query by using logical operators. For example we query for all adult drivers with powerful cars by combining multiple criteria.

```
List<Driver> drivers = container.query(new Predicate<Driver>() {
    @Override
    public  boolean match(Driver driver) {
        return driver.getMostLovedCar().getHorsePower()>=150 && driver.getAge()>=18;
    }
});
```

Queries.java: Query for people with powerful cars

Native Query Optimization

db4o tries to optimize native queries by translating them to the db4o low level query API. In cases which that doesn't work db4o actually instantiates all objects and runs them through the predicate. Unfortunately that's an order of magnitude slower than an optimized query.

How do we find out that a query couldn't be optimized? The simplest way is to use the debugger. We add a break-point in query method. When the debugger stops there, then query couldn't be optimized. In that case try to simplify our query or use the low level query API directly.

```
List<Driver> drivers = container.query(new Predicate<Driver>() {
    @Override
    public  boolean match(Driver driver) {
        // Add a break point here. If the debugger stops, the query couldn't be optimized
        // That means it runs very slowly and we should find an alternative.
        // This example query cannot be optimized because the hash code isn't a stored in database
        return driver.hashCode() == 42;
    }
});
```

Queries.java: Unoptimized query

SODA-Queries

SODA is db4o's low level query API. Prefer native queries over this API. We create SODA queries by creating a query object. Then we add different criteria to that query object. Finally we execute the query and get the result. Let's take a look at two example queries:

Query for a Certain Person

Here we query for a driver named Joe with SODA:

```
Query query = container.query();
query.constrain(Driver.class);
query.descend("name").constrain("Joe");
List<Driver> drivers = query.execute();
```

Queries.java: Query for drivers named Joe with SODA

More Complex Query

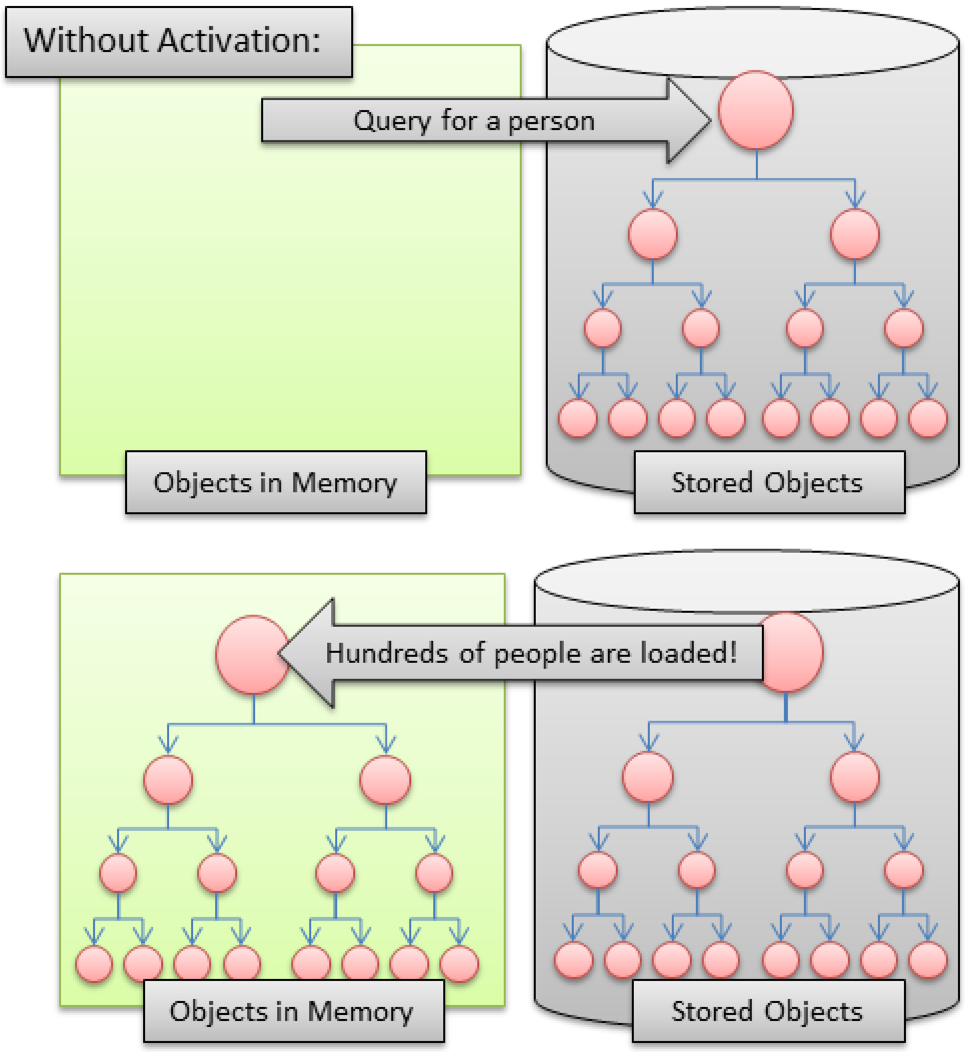
Here's a more complex query in SODA. We query for all adult drivers with powerful cars:

```
Query query = container.query();
query.constrain(Driver.class);
query.descend("mostLovedCar").descend("horsePower").constrain(150).greater();
query.descend("age").constrain(18).greater().equal();
List<Driver> drivers = query.execute();
```

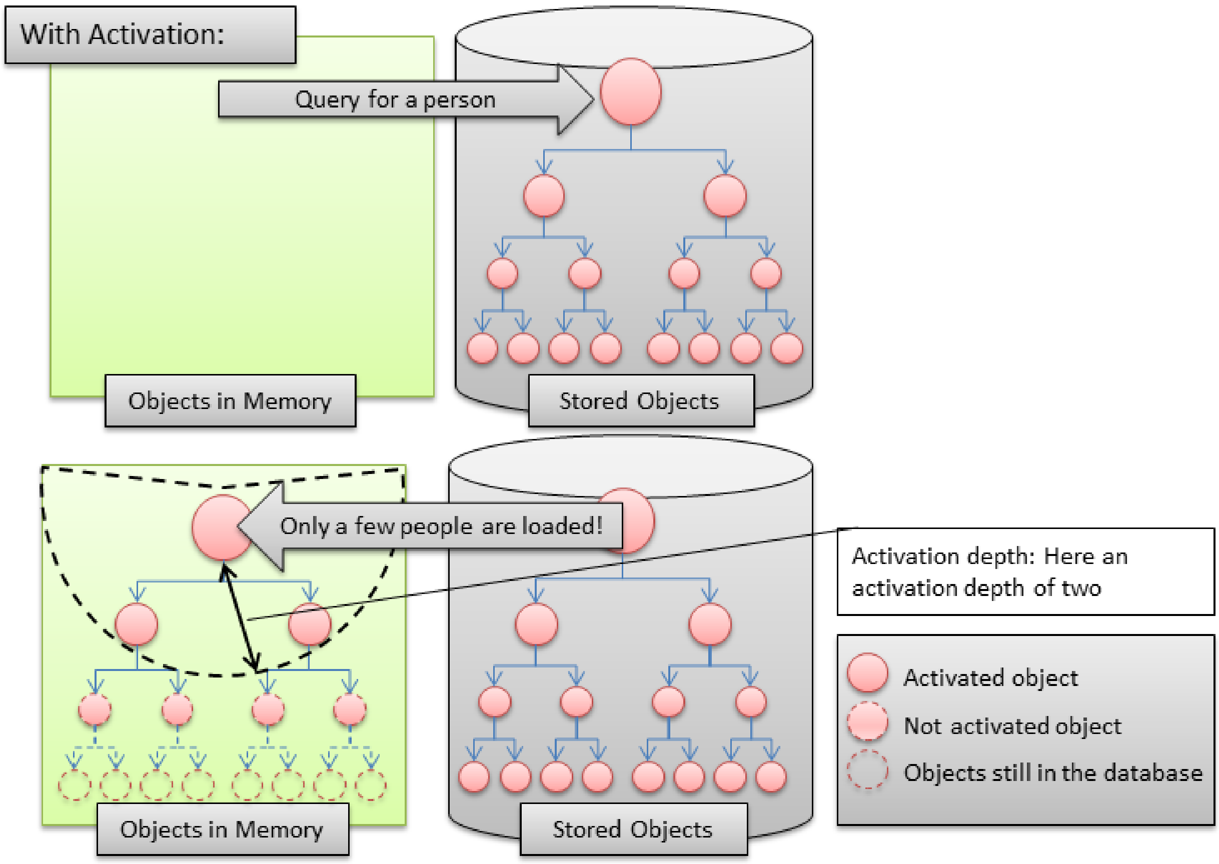
Queries.java: Query for people with powerful cars with SODA

Activation-Concept

Activation is a mechanism which controls object instantiation. Why is this necessary? Let's look at an example. We store person-objects with its parents. Each person object has a reference to his parents, the parents again have references to their parents and so on and so forth. Now we load a person object from the database, what does db4o do? Does it load the person object, its parents and the parents of the parents? Then it probably loads all objects from the database and will use up all memory.



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



Activation in Action

Let's see db4o's activation in action. To do so we need a deep object-graph. We create a person class with a mother-field and then create a deep hierarchy of people, for example a hierarchy of seven people. (Click here to see the example domain model classes)

```java
public  class Person {
    private String name;
    private Person mother;

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

    public String getName() {
        return name;
    }

    public Person getMother() {
        return mother;
    }

    @Override
    public String toString() {
        return  "Person{" +
                "name='" + name +
                "'}";
    }
}
```

Person.java: Domain model for people

```java
Person eva = new Person("Eva",null);
Person julia = new Person("Julia",eva);
Person jennifer = new Person("Jennifer",julia);
Person jamie = new Person("Jamie",jennifer);
Person jill = new Person("Jill",jamie);
Person joanna = new Person("Joanna",jill);

Person joelle = new Person("Joelle",joanna);
container.store(joelle);
```

ActivationConcept.java: Store a deep object hierarchy

After that we close the database, reopen it and query for a person. Then we start to traverse the object graph. At a certain point we will reach objects which aren't activated anymore. In those objects all fields are null or have the default value:

```java
Person joelle = queryForJoelle(container);
Person jennifer = joelle.getMother().getMother().getMother().getMother();
System.out.println("Is activated: " + jennifer);
// Now we step across the activation boundary
// therefore the next person isn't activate anymore.
// That means all fields are set to null or default-value
Person julia = jennifer.getMother();
System.out.println("Isn't activated anymore"+julia);
```

ActivationConcept.java: Activation depth in action

Since not activate objects have set all their fields to null, they can cause null pointer exceptions:

```java
String nameOfMother = julia.getMother().getName();
```

ActivationConcept.java: NullPointer exception due to not activated objects

Deal with Activation

We've seen that activation can cause issues and we have to deal with it. There are different strategies for that.

Directly Activating Objects

We can explicitly activate objects or ask about their activation state. This way we can activate objects when needed. However as soon as we need to activate objects explicitly all over the place in our code we should look for alternatives.

```
boolean isActivated = container.ext().isActive(julia);
```

ActivationConcept.java: Check if an instance is activated

```
container.activate(julia,5);
```

ActivationConcept.java: Activate instance to a depth of five

Changing Activation Depth

Alternatively we can increase the activation depth globally or for certain types. However we need to be aware that a high activation depth decreases performance. That's why we should be very careful with increasing the activation depth.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().activationDepth(10);
ObjectContainer container = Db4oEmbedded.openFile(configuration,DATABASE_FILE);
```

ActivationConcept.java: Increase the activation depth to 10

We can even more fine tune the activation depth. However the fundamental issue is the same. When we increase the activation depth it will hurt performance.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
// At least activate persons to a depth of 10
configuration.common().objectClass(Person.class).minimumActivationDepth(10);
// Or maybe we just want to activate all referenced objects
configuration.common().objectClass(Person.class).cascadeOnActivate(true);
```

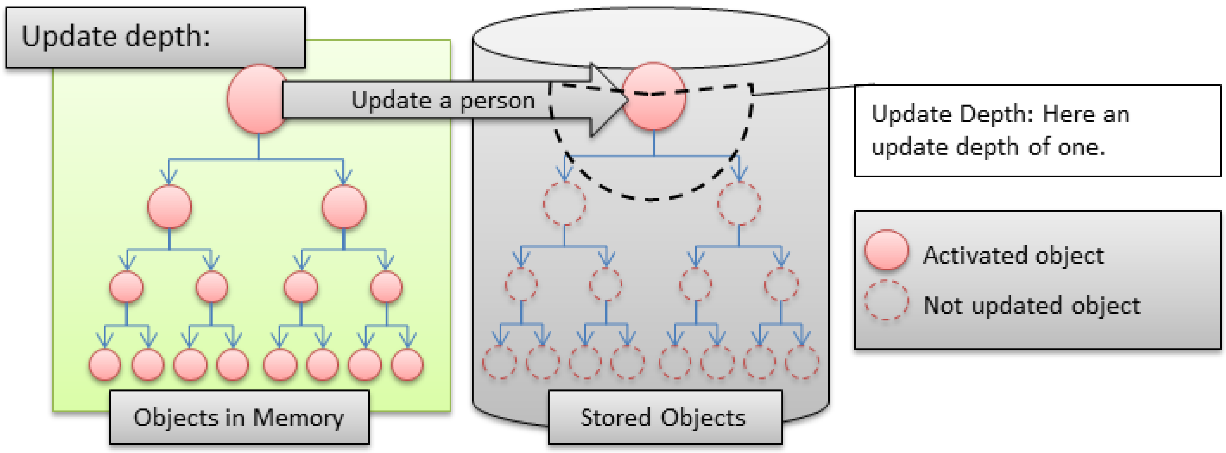
ActivationConcept.java: More activation options

Transparent Activation / Persistence

The last option we have is transparent activation. Transparent activation takes care of all activation issues. It activates objects as soon as they are needed. To get transparent activation we need to enhance our classes at compile time. Read this section of the tutorial where we use transparent activation.

Update Concept

How does updating objects work with db4o? Similar to the activation depth, db4o uses an update depth. When we store an object again, db4o only updates the objects up to the given update-depth. This prevents db4o from traversing the whole object graph in order to store the changes.



Update Depth in Action

Let's see db4o's update strategy in action. We store a driver with his cars and his most loved car. (Click here to see the example domain model classes)

```java
public  class Car {
    private String carName;

    public Car(String carName) {
        this.carName = carName;
    }

    public String getCarName() {
        return carName;
    }

    public  void setCarName(String carName) {
        this.carName = carName;
    }

    @Override
    public String toString() {
        return  "Car{" +
                "carName='" + carName + '\'' +
                '}';
    }
}
```

Car.java: Domain model for cars

```java
public  class Driver {
    private String name;
    private  final List<Car> ownedCars = new ArrayList<Car>();
    private Car mostLovedCar;

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

    public Driver(String name, Car mostLovedCar) {
        this.name = name;
        this.mostLovedCar = mostLovedCar;
    }

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

    public String getName() {
        return name;
    }

    public Car getMostLovedCar() {
        return mostLovedCar;
    }

    public Iterable<Car> getOwnedCars() {
        return ownedCars;
    }

    public  boolean addOwnedCar(Car car) {
        return ownedCars.add(car);
    }
}
```

Driver.java: Domain model for drivers

```java
Car beetle = new Car("VW Beetle");
Car ferrari = new Car("Ferrari");

Driver driver = new Driver("John", ferrari);
driver.addOwnedCar(beetle);
driver.addOwnedCar(ferrari);

container.store(driver);
```

UpdateConcept.java: Store a driver and his cars

After that we update the driver, change the name of his most loved car and add another car to his collection. Then we update the car in the database.

```java
Driver driver = queryForDriver(container);
driver.setName("Johannes");
driver.getMostLovedCar().setCarName("Red Ferrari");
driver.addOwnedCar(new Car("Fiat Punto"));
container.store(driver);
```

UpdateConcept.java: Update the driver and his cars

Then we close the database, reopen it and check if the objects have been updated. Unfortunately only the car itself has been updated. That's because db4o by default uses an update-depth of one. That means only the object we pass to the store-method is updated. All referenced objects are not updated. In our case that means that the referenced car and the referenced list of cars isn't updated.

```
Driver driver = queryForDriver(container);
// Is updated
System.out.println(driver.getName());
// Isn't updated at all
System.out.println(driver.getMostLovedCar().getCarName());
// Also the owned car list isn't updated
for (Car car : driver.getOwnedCars()) {
    System.out.println(car);
}
```

UpdateConcept.java: Check the updated objects

Deal with Updates

We've seen that by default only the object passed to the store method is updated. That means we need a strategy to deal with this.

Explicitly Storing Every Changed Object

One way to deal with this is to store every changed object. In our case that would be the driver, the most loved car and the list of owed cars. However in practice this strategy needs too much effort.

```
container.store(driver);
container.store(driver.getMostLovedCar());
container.store(driver.getOwnedCars());
```

UpdateConcept.java: Update everything explicitly

Change the Activation Depth

A simple strategy is to increase the update depth, for example increase the depth to two. This is quite a reasonable update-depth, especially since it will update collections of an object.

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
configuration.common().updateDepth(2);
```

UpdateConcept.java: Increase the update depth to 2

We can control the update-depth also per class:

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
// Update all referenced objects for the Driver class
configuration.common().objectClass(Driver.class).cascadeOnUpdate(true);
```

UpdateConcept.java: More update options

Transparent Persistence

The last and most elaborate option is to use transparent persistence. In that case db4o manages all updates for us. We just can change objects and db4o will track all changes for us. To get transparent persistence we need to enhance our classes at compile time. Read this section of the tutorial where we use transparent persistence.

Transparent Persistence

In the previous topics we've seen that activation and the update depth can be annoying and cause unexpected errors. That's why we want to get rid of it. That's where transparent persistence comes to our rescue. It manages the activation and updating of our updates. Let's get started.

Enhance Persistent Classes

The first step for transparent persistence it to enhance the persisted classes. For that we introduce an Annotation:

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

TransparentPersisted.java: Annotation to mark persisted classes

After that we can mark the persistent classes with the Annotation. (Click here to see the example domain model classes)

```
@TransparentPersisted
public  class Car {
    private String carName;

    public Car(String carName) {
        this.carName = carName;
    }

    public String getCarName() {
        return carName;
    }

    public  void setCarName(String carName) {
        this.carName = carName;
    }
}
```

Car.java: Domain model for cars

```java
@TransparentPersisted
public  class Driver {
    private String name;
    private  final List<Car> ownedCars = new ArrayList<Car>();
    private Car mostLovedCar;

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

    public Driver(String name, Car mostLovedCar) {
        this.name = name;
        this.mostLovedCar = mostLovedCar;
    }

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

    public String getName() {
        return name;
    }

    public Car getMostLovedCar() {
        return mostLovedCar;
    }
    public Iterable<Car> getOwnedCars() {
        return ownedCars;
    }

    public  boolean addOwnedCar(Car car) {
        return ownedCars.add(car);
    }
}
```

Driver.java: Domain model for drivers

The next step it to create a class filter which reports all classes which should be enhanced. In our example we check for the presence of the annotation. In practice you can implement your own class filter as you like.

```java
public  final  class AnnotationFilter implements ClassFilter{

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

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

}
```

AnnotationFilter.java: Build a filter

Enhancing Classes Using Ant

The next step is to run the enhancement step when compiling the source. You can do this with a Ant task.

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

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

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

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

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

        <transparent-activation-step>
            <annotation-filter/>
        </transparent-activation-step>
    </db4o-instrument>
</target>
```
enhance.xml: Ant target for enhancing your classes after building them

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

Enhance Classes Using Maven

The next step is to run the enhancement step when compiling the source. You can do this by adding a small Ant task to your Maven file:

```
<plugin>
    <artifactId>maven-antrun-plugin</artifactId>
    <version>1.6</version>
    <dependencies>
        <!-- We need the db4o tooling for enhancing stuff -->
        <dependency>
            <groupId>com.db4o</groupId>
            <artifactId>db4o-tools-java5</artifactId>
            <version>8.1-SNAPSHOT</version>
        </dependency>
    </dependencies>
    <executions>
        <execution>
            <phase>compile</phase>
            <configuration>
                <target>
                    <!-- We enhance with an additional Ant-run step. You can put this also in an extra file -->
                    <typedef  resource="instrumentation-def.properties"
                              classpathref="maven.compile.classpath"/>

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

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

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

pom.xml: Enhance persisted classes during the build

Check Enhancement

Finally we check if the enhancement actually worked. We just check if the classes implement the interface. If the interface is present everything worked. We can make this check a part of our test suite.

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

TransparentPersistence.java: Check for enhancement

Using Transparent Persistence

In order to use transparent persistence we need to configure it first:

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

TransparentPersistence.java: Configure transparent persistence

After that transparent persistence is active and manages the activation and updates for us.

```
Driver driver = queryForDriver(container);
// Transparent persistence will activate objects as needed
System.out.println("Is activated? "+container.ext().isActive(driver));
String nameOfDriver = driver.getName();
System.out.println("The name is "+nameOfDriver);
System.out.println("Is activated? "+container.ext().isActive(driver));
```

TransparentPersistence.java: Transparent persistence manages activation

Transparent persistence also manages updates. You just can change existing objects as you please. Later when you commit the transaction all updates are persisted. You don't need to store objects in order to update them. You only need to call store once for objects which are new and are not referenced by an existing object.

```
Driver driver = queryForDriver(container);
driver.getMostLovedCar().setCarName("New name");
driver.setName("John Turbo");
driver.addOwnedCar(new Car("Volvo Combi"));
// Just commit the transaction. All modified objects are stored
container.commit();
```

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

Transactions

db4o has ACID properties and supports transactions. Now we take a look at db4o's transaction support. (Click here to see the example domain model classes)

```java
@TransparentPersisted
public  class Car {
    private String carName;

    public Car(String carName) {
        this.carName = carName;
    }

    public String getCarName() {
        return carName;
    }
}
```

Car.java: Domain model for cars

```java
@TransparentPersisted
public  class Driver {
    private String name;
    private Car mostLovedCar;

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

    public Driver(String name, Car mostLovedCar) {
        this.name = name;
        this.mostLovedCar = mostLovedCar;
    }

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

    public String getName() {
        return name;
    }

    public Car getMostLovedCar() {
        return mostLovedCar;
    }
}
```

Driver.java: Domain model for drivers

Commit and Rollback

db4o starts a transaction as soon as you open an object container and implicitly commits it when you close the object container. In between we can control the transaction explicitly by calling the commit or rollback method. After the commit everything is persisted and another transaction is started.

```java
container.commit();
```

Transactions.java: Committing changes

The same applies to rollbacks. We can rollback the transaction any time. After a rollback the next transaction is started.

```
container.rollback();
```

Transactions.java: Rollback changes

Object State after Rollback

What happens to objects in memory when we rollback a transaction? By default the objects in memory keep their state. If you want to rollback the objects in memory you need to reload them.

```
Driver driver = queryForDriver(container);
driver.setName("New Name");
System.out.println("Name before rollback " + driver.getName());
container.rollback();
// Without transparent persistence objects keep the state in memory
System.out.println("Name after rollback " + driver.getName());
// After refreshing the object is has the state like in the database
container.ext().refresh(driver, Integer.MAX_VALUE);
System.out.println("Name after rollback " + driver.getName());
```

Transactions.java: Without transparent persistence objects in memory aren't rolled back

With Transparent Persistence

With transparent activation we can use a rollback strategy. That way we can rollback the state of the objects in memory.

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

Transactions.java: Rollback strategy for the transaction

```
Driver driver = queryForDriver(container);
driver.setName("New Name");
System.out.println("Name before rollback " + driver.getName());
container.rollback();
// Thanks to transparent persistence with the rollback strategy
// the object state is rolled back
System.out.println("Name after rollback " + driver.getName());
```

Transactions.java: Transparent persistence rolls back objects in memory

Multiple Transactions

For more complex applications we can use multiple transactions. For that we need to create multiple object containers, each having its own transaction.

When we have multiple object containers, an object always needs to be loaded and stored in the same object container.

```java
ObjectContainer container = rootContainer.ext().openSession();
try{
    // We do our operations in this transaction
} finally {
    container.close();
}
```

Transactions.java: Opening a new transaction

Indexes

With indexes we can speed up queries. As soon as we have lots of objects and query them we need indexes.

Indexes are a tradeoff: An index makes queries orders of magnitudes faster, but it slows down updates and inserts of those objects.

Index Annotation

We can add an Annotation on a field in order to index that field.

```java
public  class Car {
    @Indexed
    private String carName;
    private  int horsePower;

    public Car(String carName, int horsePower) {
        this.carName = carName;
        this.horsePower = horsePower;
    }

    public String getCarName() {
        return carName;
    }

    public  int getHorsePower() {
        return horsePower;
    }

}
```

Car.java: Indexing the car name

Configuration

How do we configure db4o? To configure db4o we use db4o's configuration API. The first thing we do is to create a new configuration instance:

```
EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
```

ConfigurationExamples.java: Important configuration switches

Now we can configure db4o by using the different methods on the configuration object. You can read up on the different configuration option on the reference- or the API documentation.

```
// If you are using BigDecimal or BigInteger, add bigmath-support
configuration.common().add(new BigMathSupport());
// If you are using UUIDs, add support for those
configuration.common().add(new UuidSupport());
// Add index
configuration.common().objectClass(Driver.class).indexed(true);
```

ConfigurationExamples.java: A few examples

The final step is to pass in the configuration object to the object container factory.

```
ObjectContainer container = Db4oEmbedded.openFile(configuration, "database.db4o");
```

ConfigurationExamples.java: Finally pass the configuration container factory

Client Server Mode

Let's take a look at the client server modes of db4o.

Embedded Clients aka Session Containers

db4o allows us to open lightweight session containers (aka embedded clients), which have their own transaction. We can use these session containers when we need multiple transactions running at the same time.

```java
ObjectContainer container = rootContainer.ext().openSession();
try{
    // We now can use this session container like any other container
} finally {
    container.close();
}
```

ClientServer.java: Creating a session container

Network Client Server Mode

db4o supports a client server mode over the network. However db4o was designed for embedded use cases. Therefore the client server mode isn't a core functionality but more an add-on. Not all db4o features work in the client server mode and the performance characteristics of db4o aren't optimized for client server mode.

For all these reasons we should prefer the embedded mode with session containers instead of full blown clients. Nevertheless the client server mode can be useful to for some scenarios.

Starting a Server

First we start the server:

```java
ObjectServer server = Db4oClientServer.openServer("database.db4o",8080);
try{
    // allow access to this server
    server.grantAccess("user","password");

    // Keep server running as long as you need it
    System.out.println("Press any key to exit.");
    System.in.read();
    System.out.println("Exiting...");
}finally {
    server.close();
}
```

ClientServer.java: Open server

Connect Clients

As long as the server is running we can connect to it. We need to specify the host, port, username and the password in order to connect. Afterwards we have a regular object container ready to be used.

```
ObjectContainer container
        = Db4oClientServer.openClient("localhost",8080,"user","password");
try{
    // Use the client object container as usual
} finally {
    container.close();
}
```

ClientServer.java: Using the client

Shared Classes

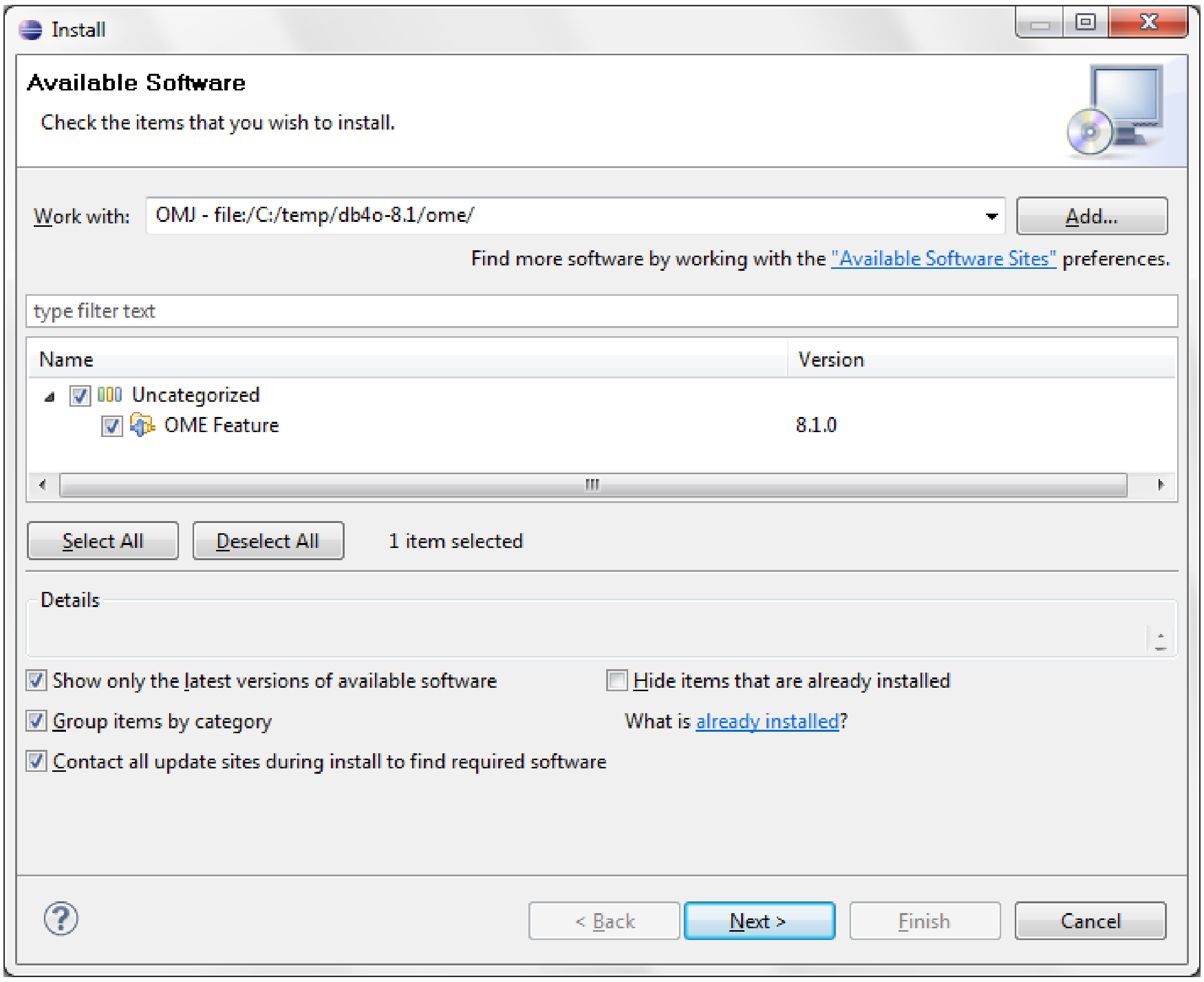
The biggest limitation of the client server mode is that the server and the client both need to have the domain model classes available. The server needs the classes of the stored objects. If we change classes, we need to deploy them to the server and the clients. To do that we should pack the domain model into a jar and ensure that the same version is used on the server and client.

Object Manager

The object manager allows us to open and explore db4o databases.

Installation

The object manager is an Eclipse plugin. In the db4o distribution there's a subfolder 'ome', which contains a zip-file. Unzip that file and then start up Eclipse. There we choose 'Help'->'Install New Software...'. Then we add the location of the unzipped object manager as an update site and continue the installation.



Work with: OMJ - file:/C:/temp/db4o-8.1/ome/

Find more software by working with the "Available Software Sites" preferences.

Name	Version
☑ Uncategorized	
☑ OME Feature	8.1.0

1 item selected

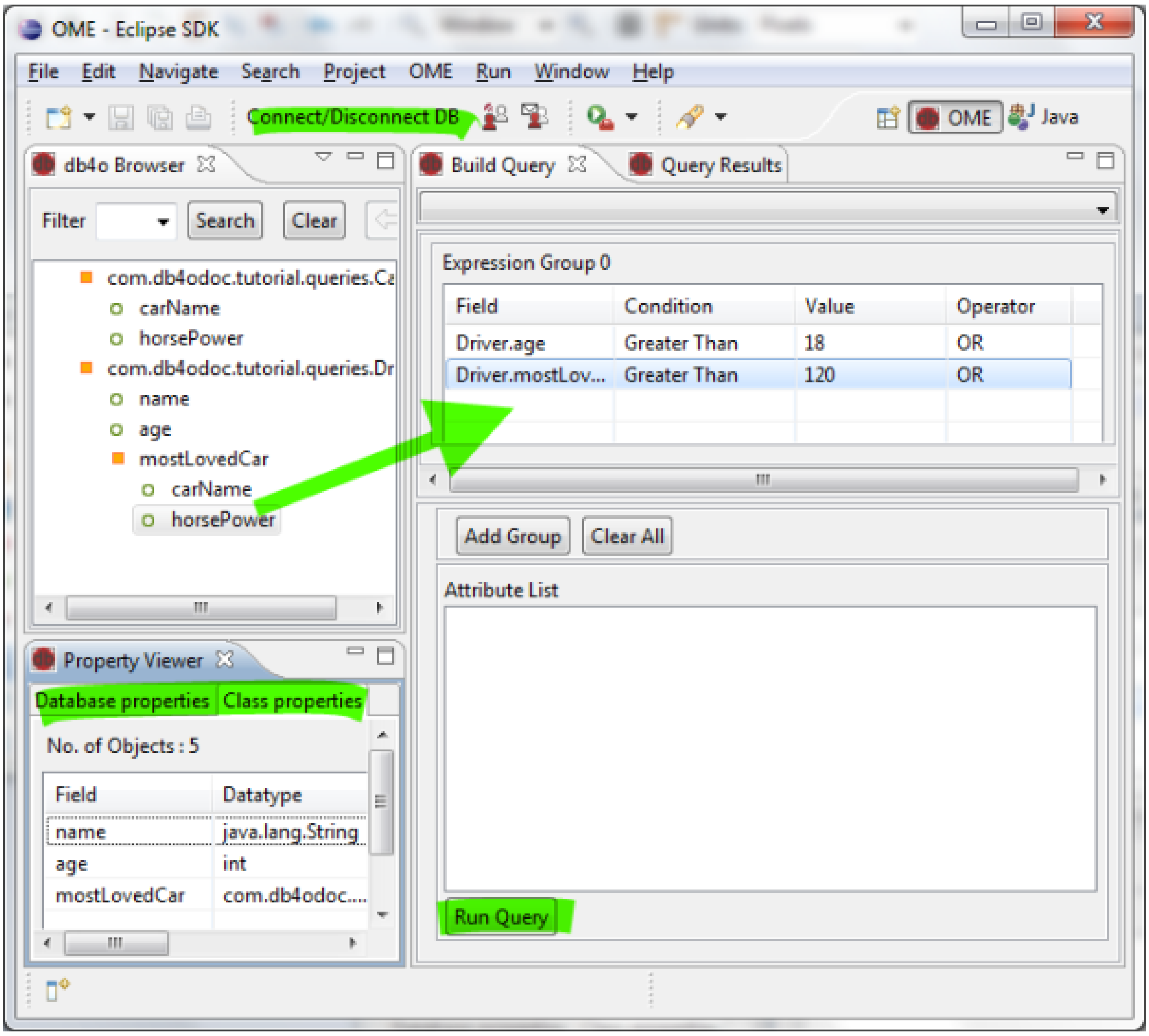
☑ Show only the latest versions of available software
☐ Hide items that are already installed
☑ Group items by category
What is already installed?
☑ Contact all update sites during install to find required software

Using the Object Manager

To use the object manager we first open the OME perspective ('Window'->'Open Perspective'->'Other...'. Then we can open a db4o database.

After connecting we see a list of stored classes on the right. Below that we can inspect a stored class and see it details.

To query the database we drag and drop the fields we're interested in to the query builder. Then we change the condition and values on that query and run it.

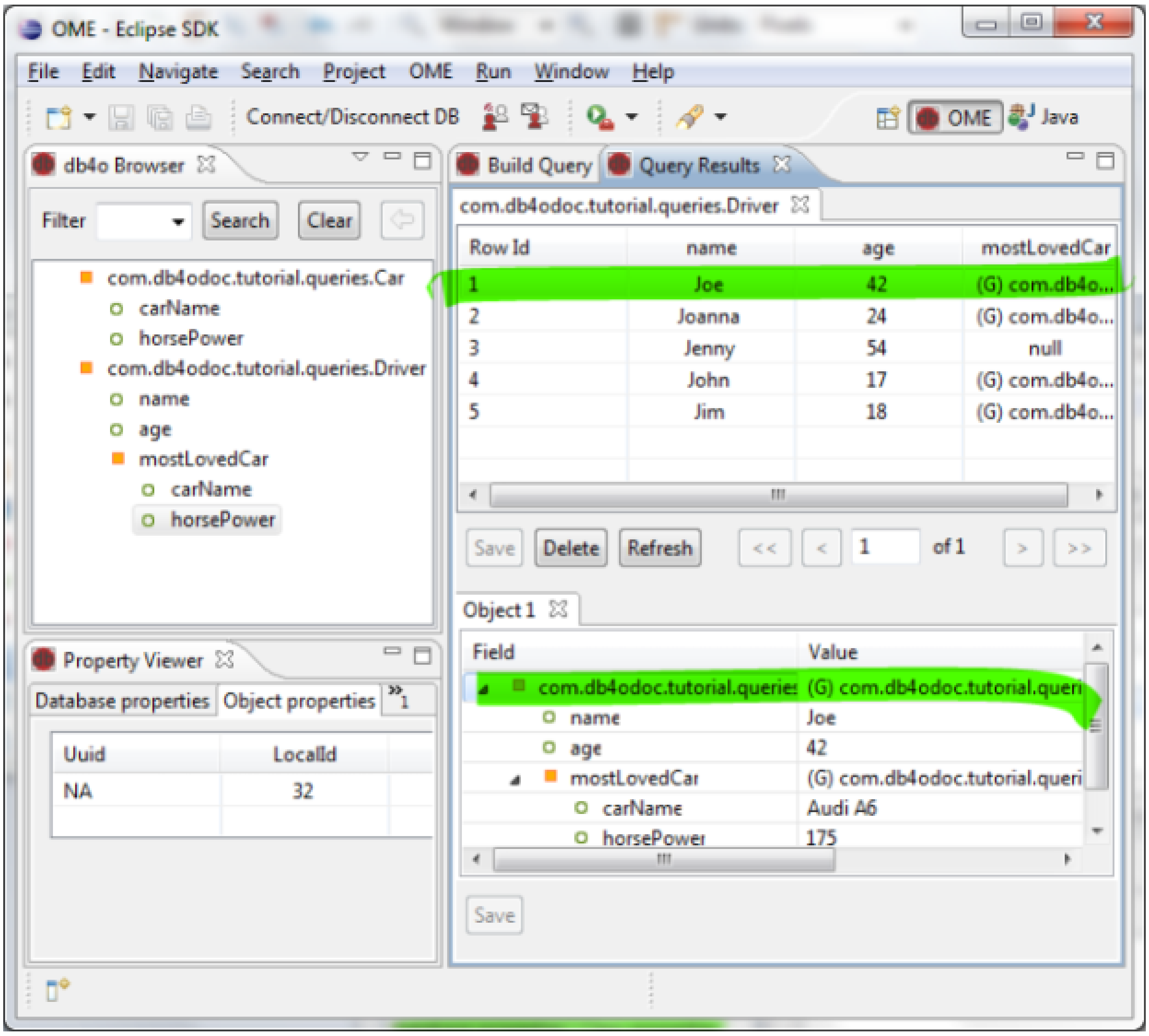


Field	Condition	Value	Operator
Driver.age	Greater Than	18	OR
Driver.mostLov...	Greater Than	120	OR

No. of Objects : 5

Field	Datatype
name	java.lang.String
age	int
mostLovedCar	com.db4odoc....

On the result page we can select the different objects and inspect their values. We also can change values of objects and then store the data.



- com.db4odoc.tutorial.queries.Car
  - carName
  - horsePower
- com.db4odoc.tutorial.queries.Driver
  - name
  - age
  - mostLovedCar
    - carName
    - horsePower

com.db4odoc.tutorial.queries.Driver

Row Id	name	age	mostLovedCar
1	Joe	42	(G) com.db4o...
2	Joanna	24	(G) com.db4o...
3	Jenny	54	null
4	John	17	(G) com.db4o...
5	Jim	18	(G) com.db4o...

Field	Value
com.db4odoc.tutorial.queries	(G) com.db4odoc.tutorial.queri
name	Joe
age	42
mostLovedCar	(G) com.db4odoc.tutorial.queri
carName	Audi A6
horsePower	175

Uuid	LocalId
NA	32

Community

That's it for the tutorial and you now can go wild with db4o. You also can participate in the db4o community and help to improve db4o. The main community website for db4o is http://-community.versant.com. There you can find forums, blog posts, additional resources and the newest db4o releases.

db4o Forums

The best place to ask questions about db4o, discuss and make suggestions is the db4o forum. You can find the db4o forums here: http://community.versant.com/Forums.aspx

Additional Resources on the Web

To stay informed about new features, community activity and other news subscribe to the different db4o blogs: http://community.versant.com/Blogs.aspx.

Or follow us on Twitter: http://twitter.com/db4objects

Report Bugs

Software contains bugs and so does db4o. If you find bugs please report them to us. In case you're not sure if you found a bug, you maybe want to discuss your issue first on the db4o forum. As soon as you're sure that you've found a bug please report it on our bug-tracking system. The login is the same as in the db4o forums.

When possible include a small test program which reproduces the issue. A test case avoids confusion and effort to reproduce the bug.

Contact

Address and Contact Information

Versant Corporation
255 Shoreline Drive, Suite 450
Redwood City, CA 94065
USA

Phone
+1 (650) 232-2436

Fax
+1 (650) 232-2401

Sales
Fill out our sales contact form on the db4o website or mail to sales@db4o.com

Careers

career@db4o.com

Partnering

partner@db4o.com