# Object-Oriented Databases

## db4o: Part 2

- Configuration and Tuning, Distribution and Replication
- Schema Evolution: Refactoring, Inheritance Evolution
- Callbacks and Translators

# Summary: db4o Part 1

- Managing databases with an object container
- Retrieving objects
  - query by example
  - native queries
  - SODA queries
- Updating and deleting simple and complex objects
  - configuration of update, delete and activation depth
  - inconsistencies between in-memory and stored objects
- Transactions
  - commit and rollback
  - concurrent transactions, collision detection and avoidance

# Configuration and Tuning

- Configuration interface
  - global configuration set through `Db4o.configure()`
  - current global settings are cloned when object container or object server opened
  - further changes of global configuration not propagated to already existing object containers and object servers
- External tools
  - performance tuning
  - database diagnostics
- Indexes
  - optimise query evaluation

# Configuration Interface

- Represented by `com.db4o.config.Configuration`
- Methods rather than properties files
- Configuration setting groups
  - object-related methods
  - file-related methods
  - reflection-related methods
  - communication-related methods
  - logging-related methods
  - miscellaneous configuration methods
- Configuring an existing object container or object server
  - access settings with `ExtObjectContainer#configure()` or `ExtObjectServer#configure()`, respectively

# External Tools

- Defragment
  - removes unused fields and management information
  - compacts database file and provides faster access
  - initiated from command line or from within application
- Statistics
  - computes and outputs statics about a database file
  - executed from command line or programmatically
- Logger
  - logs all objects in a database file
  - logs all objects of a given class
  - run from command line

# Indexes

- Trade-off between increased query performance and decreased storage, update and delete performance
- Support for B-Tree indexes on single object fields
  - enabled or disabled using configuration interface
  - internal field **`i_indexed`** is set to true or false for indexed field
  - index created or removed automatically when object container or object server is opened
- Example

```
// create an index
Db4o.configure().objectClass(...).objectField(...).indexed(true);
// remove an index
Db4o.configure().objectClass(...).objectField(...).indexed(false);
```

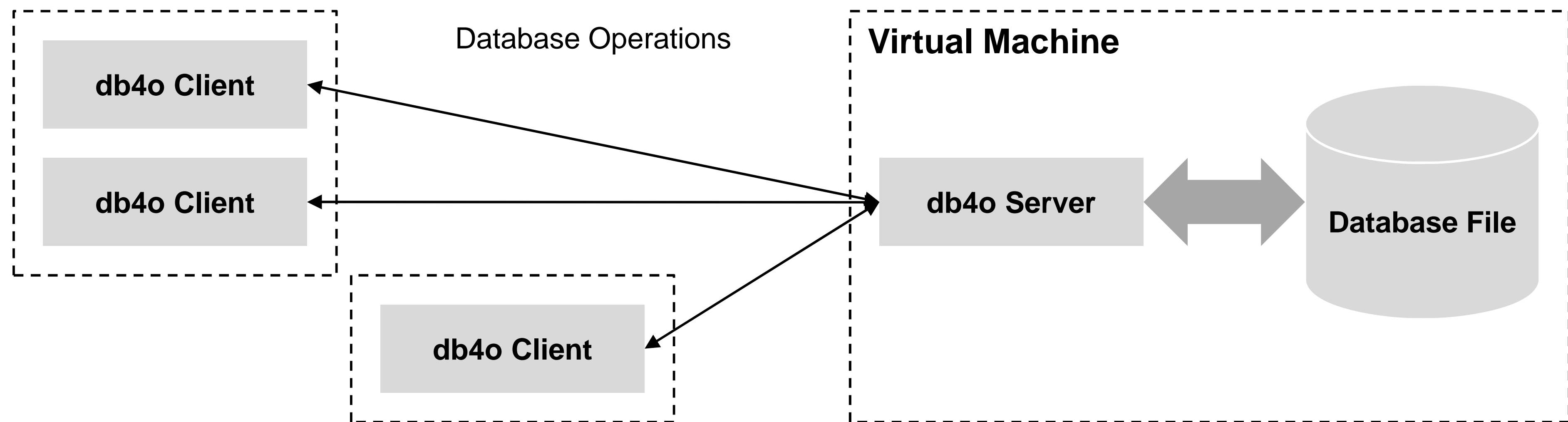
# Tuning for Speed

- Heuristics to improve performance of db4o
- Object loading
  - use appropriate activation depth
  - use multiple object containers
  - disable weak references if not required (no updates performed)
- Database tests
  - disable detection of schema changes
  - disable instantiation tests of persistent classes at start-up
- Query evaluation
  - set field indexes on most used objects to improve searches
  - optimise native queries

# Distribution and Replication

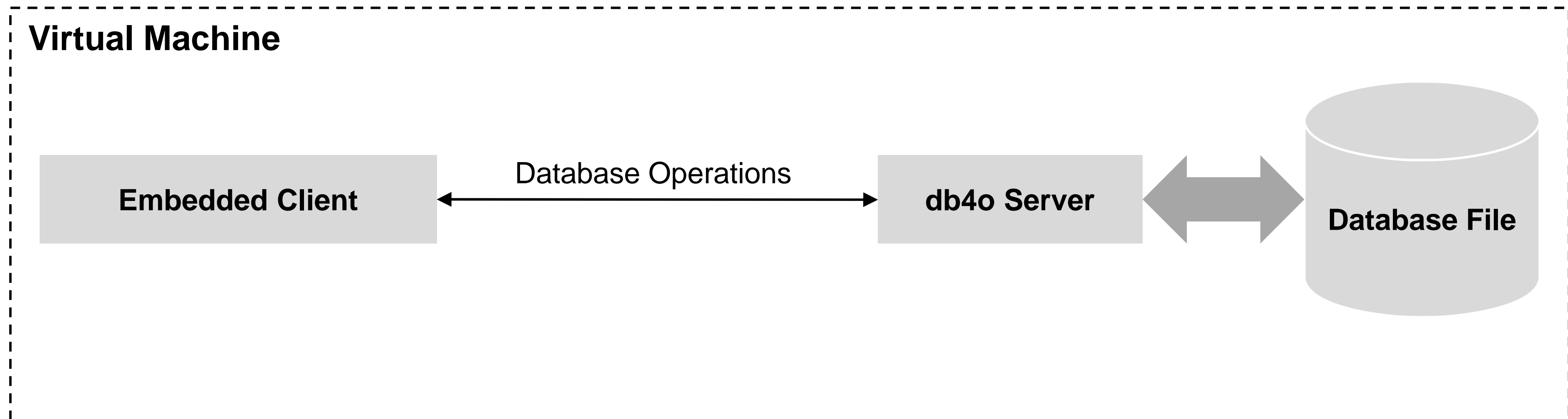
- Local mode
  - standalone database
  - database file opened and accessed directly
  - one user, one process or one thread at a time
- Client/Server mode
  - multiple clients interact with one central server
  - server listens for and accepts connections
  - clients connect to server to perform database tasks
- Replication
  - multiple server manage redundant copies of a database
  - changes are replicated from master to client servers
  - replicated databases need to be kept consistent

# Client/Server Modes: Networking Mode



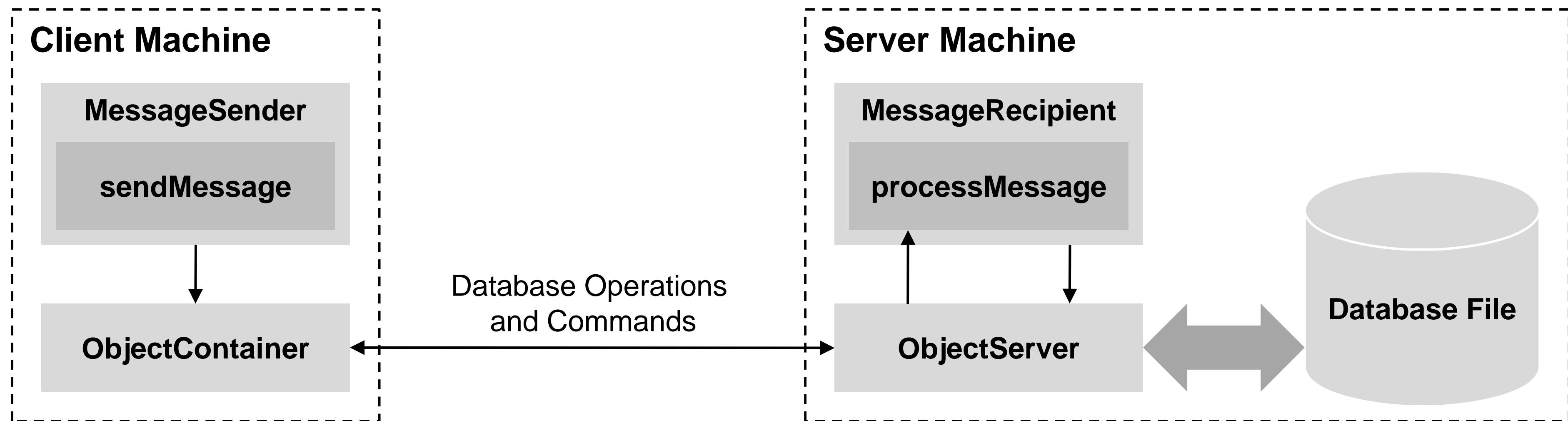

- Client opens TCP/IP connection to server
  - method `Db4o.openServer(filename, port)`
  - method `Db4o.openClient(host, port, user, pass)`
- Client sends query, insert, update and delete instructions to server and receives data from the server

# Client/Server Modes: Embedded Mode




- Not distributed across a network
  - client and server run in the same virtual machine
  - better performance for multi-threaded applications
- Server is started on port 0
- Client is opened using `ObjectServer#openClient()`

# Client/Server Modes: Out-of-Band Signalling

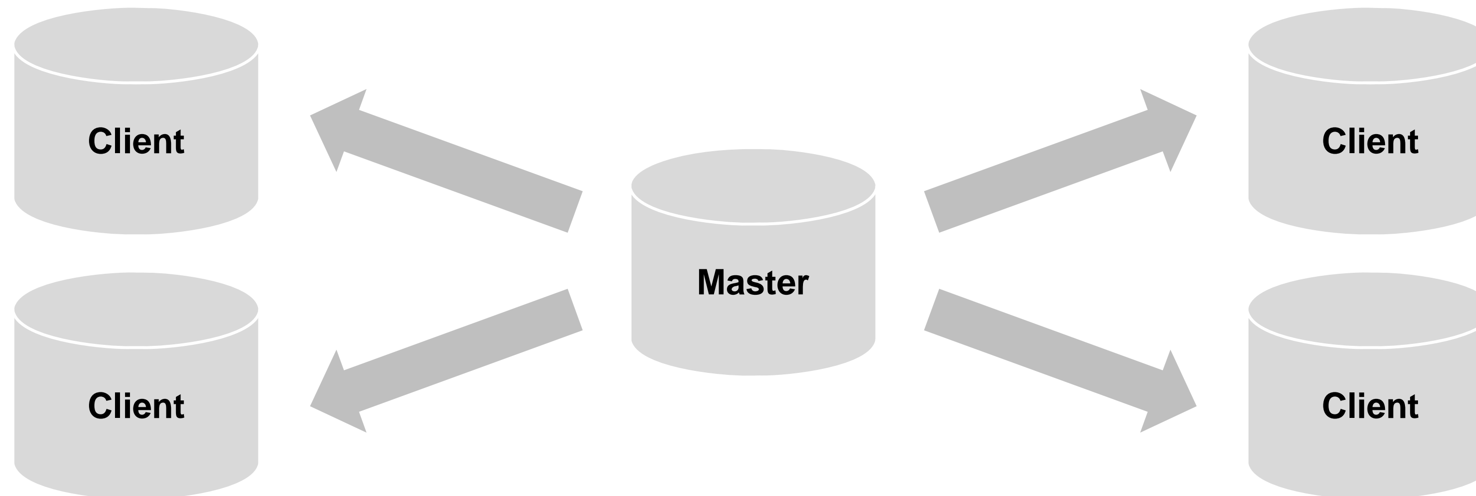



- Basic client/server mode cannot transmit command
  - operations are limited to methods of `ObjectContainer`
- Out-of-band signalling
  - interface `MessageSender`
  - interface `MessageRecipient`

# Replication

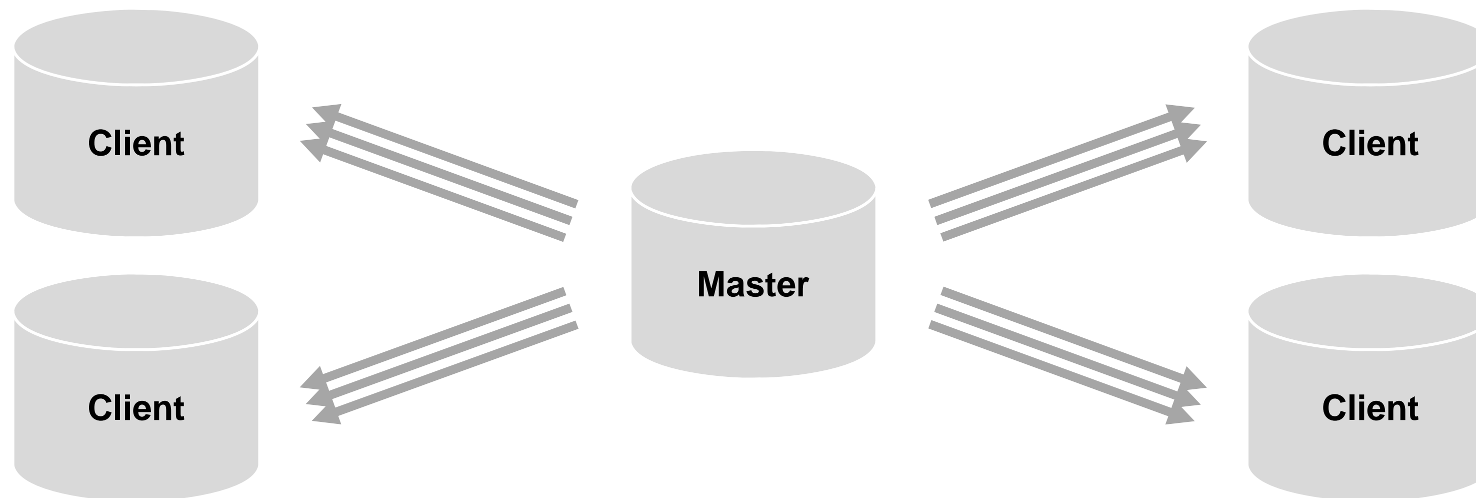
- Database managed by redundant servers
  - data changes on masters or publishers
  - changes replicated to clients of subscribers
- Several forms of replication supported
  - snapshot replication
  - transactional replication
  - merge replication
- Replication in db4o has to be coded into application and cannot be configured on an administrative level

# Replication Modes: Snapshot Replication



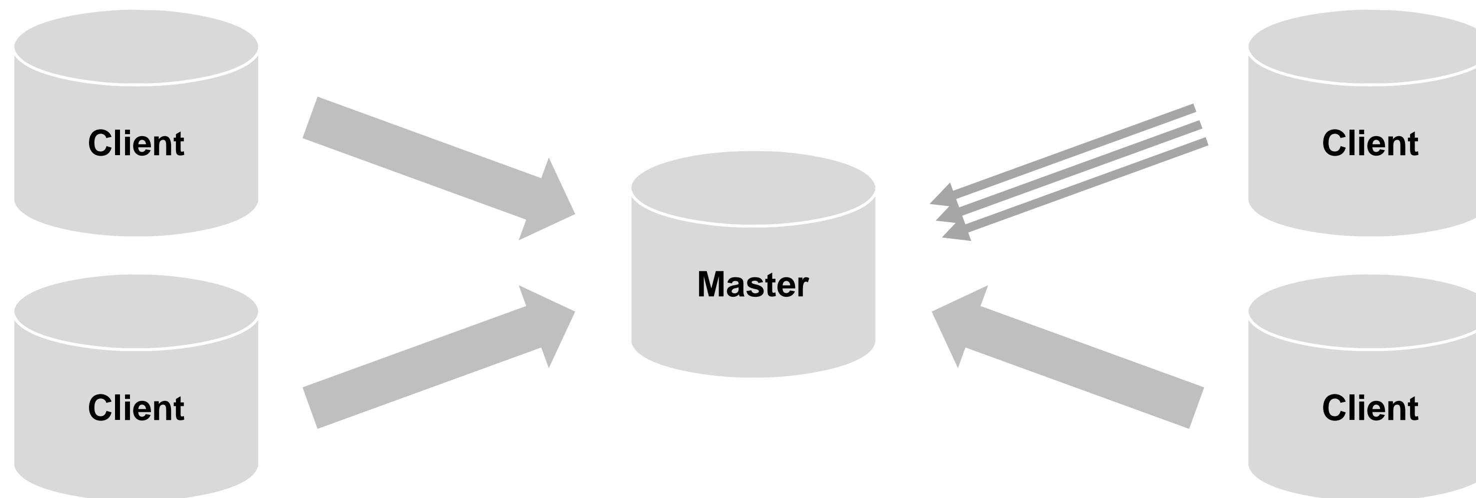

- Snapshots of the master database replicated to client
  - state-based
  - periodical schedule
- Support in db4o
  - special SODA query to detect all new and updated objects

# Replication Modes: Transactional Replication




- Changes are synchronised after transaction
  - operation based
  - changes are replicated immediately
- Support in db4o
  - single object replication with `ReplicationProcess`

# Replication Modes: Merge Replication




- Changes from client are merged to central server
- Other clients are updated to reflect changes
- Can be done either transactionally or on a periodic basis
- Typically occurs if subscribers are occasionally offline

# Core Replication

- Transfers data between peer object containers
  - local database files
  - object servers in networking or embedded mode
- Requires three steps
  - generating unique IDs and version numbers
  - creating a `ReplicationProcess` object
  - replicating objects
- Replication mode is dependent on implementation
- Replication is bidirectional by default
  - replication can be configured to be unidirectional using method `ReplicationProcess#setDirection()`

# Core Replication

```
// configuration
Db4o.configure().generateUUIDs(Integer.MAX_VALUE);
Db4o.configure().generateVersionNumbers(true);
// replication process
ReplicationProcess replication = db1.ext().replicationBegin(db2,
      new ReplicationConflictHandler() {
         public Object resolveConflict(ReplicationProcess p,
               Object a, Object b) {
            return a;
         }
      }
);
replication.setDirection(db1, db2);
// update database and replicate (transactional replication)
Author alex = new Author("Alexandre de Spindler");
db1.store(alex);
replication.replicate(alex);
replication.commit();
```

# Core Replication

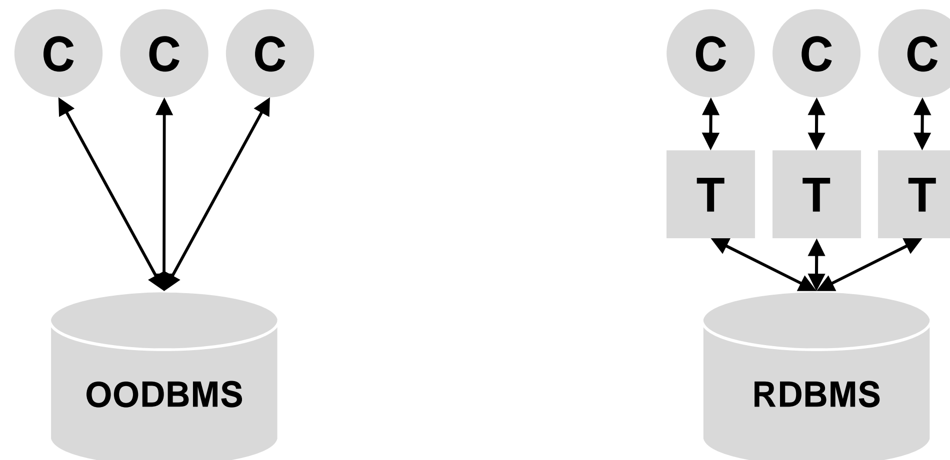
```
// query for changed objects
Query query = db1.query();
query.constrain(Publication.class);
replication.whereModified(query);
ObjectSet<Publication> result = query.execute();

// replicate objects (snapshot replication)
for (Publication publication: result) {
   replication.replicate(publication);
}
replication.commit();
```

# db4o Replication System

- Introduced in db4o version 5.1
- Replication solution separated from db4o core
  - bridges divide between db4o and relational databases
  - uni- or bidirectional replication
  - replication of relational databases based on Hibernate
- Supported replication providers
  - db4o to db4o
  - db4o to Hibernate, Hibernate to db4o
  - Hibernate to Hibernate
- Concepts similar to core replication
  - `ReplicationSession` instead of `ReplicationProcess`
  - `ConflictResolver` instead of `ConflictHandler`

# Schema Evolution

- Class definitions and inheritance structure can change
  - additional application requirements
  - software refactoring
- Class definitions and hierarchy are database schema
- In object-oriented database schema evolution is simpler as in object-relational mappings as only one data model



# Refactoring Scenarios

- Changes to interface implemented by class
  - supported as db4o only stores data and not implementations
- Removing a field
  - new objects stored in new format
  - additional field ignored in objects stored in old format
- Adding a field
  - new objects stored in new format
  - additional field set to null in objects stored in old format
- Changing the type of a field
  - simply stored as a new field
  - manual migration if old and new type incompatible

# Refactoring Scenarios

- Renaming a field
  - old field is deleted and a new inserted
  - data migration through configuration interface

```
Db4o.configure().objectClass(...).objectField(...).rename(...);
```

- Renaming a class
  - managed through configuration interface

```
Db4o.configure().objectClass(...).rename(...);
```

- Merging fields
- Splitting fields
- Moving fields

} manual using a helper program

# Inheritance Evolution

- Refactoring of inheritance structure
  - deleting classes from inheritance hierarchy
  - inserting classes into inheritance hierarchy
  - swap classes in inheritance hierarchy
- Tools for inheritance evolution are being developed
  - create a type-less transfer database
  - switch classpath manually

# Callbacks

- Set of methods called in response to events (triggers)
- db4o events
  - activate and deactivate
  - new, update and delete
- Methods called before and after event
  - methods starting with **`can`** called before event
  - methods starting with **`on`** called after event
- Methods defined by interface **`ObjectCallbacks`**
  - interface does not have to be implemented explicitly by persistent class to use its functionality
  - any number of methods can be implemented by persistent class

# Callbacks

```
package com.db4o.ext;
public interface ObjectCallbacks {
   public boolean objectCanActivate(ObjectContainer c);
   public boolean objectCanDeactivate(ObjectContainer c);
   public boolean objectCanDelete(ObjectContainer c);
   public boolean objectCanNew(ObjectContainer c);
   public boolean objectCanUpdate(ObjectContainer c);

   public void objectOnActivate(ObjectContainer c);
   public void objectOnDeactivate(ObjectContainer c);
   public void objectOnDelete(ObjectContainer c);
   public void objectOnNew(ObjectContainer c);
   public void objectOnUpdate(ObjectContainer c);
}
```

# Use Cases for Callbacks

- Recording or preventing updates
  - methods `canUpdate()` and `onUpdate()`
- Setting default values after refactoring
  - get values before update using method `canNew()`
- Checking object integrity before storing objects
  - check field values using methods `canNew()` and `canUpdate()`
- Setting transient fields
- Restoring connected state when objects activated
  - display graphical elements or restore network connections
- Creating special indexes
  - detect if a field is queried often and create index automatically

# Controlling Object Instantiation

- No convention imposed for persistent classes by db4o
- Objects are instantiated using one of three techniques
  - using a constructor
  - bypassing the constructor
  - using a translator
- For certain classes it is important which of these methods is used to retrieve objects
  - if available, bypassing the constructor is default setting
  - behaviour can be configured globally or per class
  - for debugging, db4o can be configured to throw an exception if the objects of a class cannot be stored

# Using Constructors

- db4o can use a constructor to instantiate objects
  - if no default public constructor is present, all available constructors are tested to create instances of a class
  - null or default values passed to all constructor arguments
  - first successfully tested constructor is used throughout session
  - if instance of a class cannot be created, the object is not stored
  - by default, execution will continue without any message or error
- Settings adjusted through configuration interface

```
// global setting
Db4o.configure().callConstructors(true)
// per class setting
Db4o.configure().objectClass(...).callConstructors(true)
// exceptions for debugging
Db4o.configure().exceptionsOnNotStorable(true)
```

# Using Constructors

```
public class Person {

   Date birthdate;
   transient Calendar today;

   public Person(Date birthdate) {
      this.birthdate = birthdate;
      // get today's date and store it in a transient field
      this.today = Calendar.getInstance();
   }

   public int getAge() {
      Calendar birth = Calendar.getInstance();
      birth.setTime(this.birthdate);
      // NullPointerException in the next line if constructor not called!
      int years = this.today.get(Calendar.YEAR) - birth.get(Calendar.YEAR);
      int diff = birth.add(Calendar.YEAR, age);
      return (today.before(birth)) ? age-- : age;
   }

}
```

# Bypassing Constructors

- Constructors that cannot handle null or default values must be bypassed
- db4o uses platform-specific mechanisms to bypass constructors
- Not all environments support this feature
  - Sun Java Virtual Machine (only JRE 1.4 and above)
  - Microsoft .NET Framework (except Compact Framework)
- Default setting if supported by current environment
- Breaks classes that rely on constructors being executed

# Bypassing Constructors

```
public class Person {

   Calendar birthdate;
   int age;

   public Person(Calendar birthdate) {
      this.birthdate = birthdate;
      // calculate age
      Calendar today = Calendar.getInstance();
      // NullPointerException in next line if called with null value!
      int years = today.get(Calendar.YEAR) -
            this.birthdate.get(Calendar.YEAR);
      int diff = birth.add(Calendar.YEAR, age);
      this.age = (today.before(birth)) ? age-- : age;
   }

   ...

}
```

# Translators

- Some classes cannot be cleanly reinstantiated by db4o using either method
  - constructor needed to populate transient members
  - constructor fails if called with null or default values
- Translators control loading and storing of such objects
  - Interface **`ObjectTranslator`**

```
public Object onStore(ObjectContainer c, Object appObject);
public void onActivate(
   ObjectContainer c, Object appObject, Object storedObject);
public Class storedClass();
```

  - Interface **`ObjectConstructor`** extends **`ObjectTranslator`**

```
public Object on Instantiate(
   ObjectContainer c, Object storedObject);
```

# Translators

```
public class Person {
   String name;
   Calendar birthdate;
   transient int age;
   public Person(String name, Calendar birthdate) {
      this.name = name;
      this.birthdate = birthdate;
      Calendar today = Calendar.getInstance();
      int years = today.get(Calendar.YEAR) -
            this.birthdate.get(Calendar.YEAR);
      int diff = birth.add(Calendar.YEAR, age);
      this.age = (today.before(birth)) ? age-- : age;
   }
   public String getName() { ... }
   public Calendar getBirthdate() { ... }
   public int getAge() { ... }
   ...
}
```

# Translators

```
public class PersonTranslator implements ObjectConstructor {
   // map Person object to storage representation
   public Object onStore(ObjectContainer c, Object appObject) {
      Person person = (Person) appObject;
      return new Object[] { person.getName(), person.getBirthdate() };
   }
   // reconstruct Person object from storage representation
   public Object onInstantiate(ObjectContainer c, Object storedObject) {
      Object[] raw = (Object[]) storedObject;
      return new Person((String) raw[0], (Calendar) raw[1]);
   }
   public void onActivate(ObjectContainer c, Object appObject,
         Object storedObject) { }
   // return metadata about storage representation
   public Class storedClass() {
      return Object[].class;
   }
}
```

# Literature

- db4o Tutorial
  - http://www.db4o.com/about/productinformation/resources/
- db4o Reference Documentation
  - http://developer.db4o.com/Resources/view.aspx/Reference
- db4o API Reference
  - http://developers.db4o.com/resources/api/db4o-java/
- Jim Paterson, Stefan Edlich, Henrik Hörning, and Reidar Hörning: **The Definitive Guide to db4o**, *APress 2006*

# Next Week
## ODMG Standard

- Object Model, Object Definition Language, Object Query Language
- Programming Language Bindings
- Outlook