

# 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
  - transparent activation and persistence
- Transactions
  - commit and rollback
  - concurrent transactions, collision detection and avoidance

# Configuration and Tuning

- Configuration interface
  - configuration obtained through
    - `Db4oEmbedded#newConfiguration()`
    - `Db4oClientServer#newClientConfiguration()`
    - `Db4oClientServer#newServerConfiguration()`
  - configuration set when object container, client or server opened
  - further changes to configuration do not affect already opened object containers, clients and servers
- External tools
  - performance tuning
  - database diagnostics
- Indexes
  - optimise query evaluation

# Configuration Interface

- Represented by **CommonConfiguration**, **Configuration** and their subclasses
- 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 statistics 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
CommonConfiguration#objectClass(...).objectField(...).indexed(true);

// remove an index
CommonConfiguration#objectClass(...).objectField(...).indexed(false);
```

# Tuning for Speed

- Heuristics to improve performance of db4o
  - weak references, BTree node size, freespace manager, locking, flushing, callbacks, caches, ...
- 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

- Embedded mode
  - database accessed by clients in the same virtual machine
  - direct file access: one user and one thread at a time
  - client session: one user and multiple threads
- Client/Server mode
  - clients in multiple virtual machines access database on server
  - server listens for and accepts connections
  - clients connect to server to perform database tasks
- Replication
  - multiple servers manage redundant copies of a database
  - changes are replicated from master to client servers
  - replicated databases need to be kept consistent

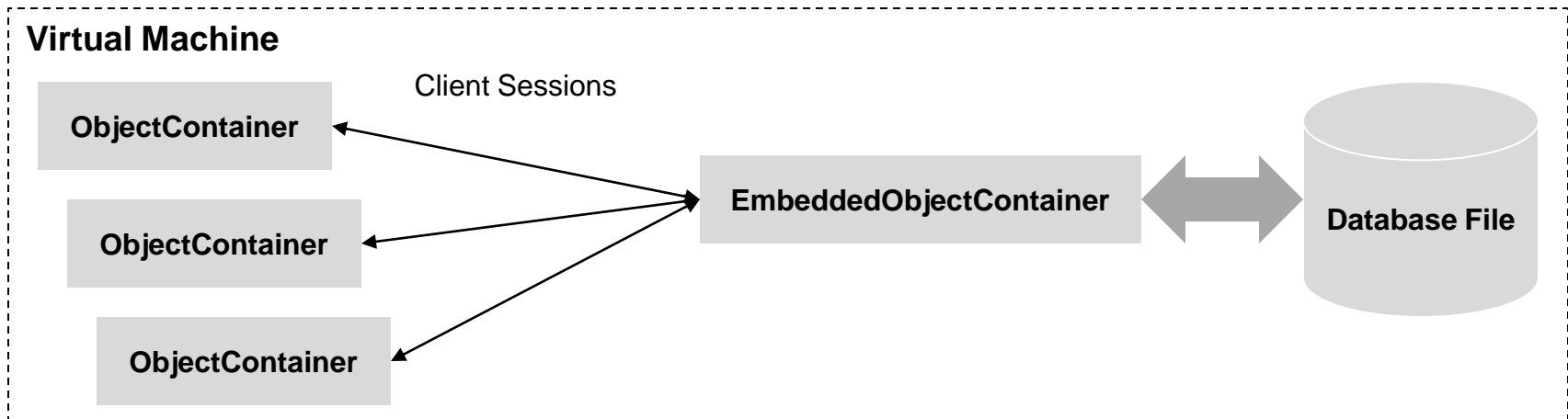


# Embedded Mode: Direct File Access



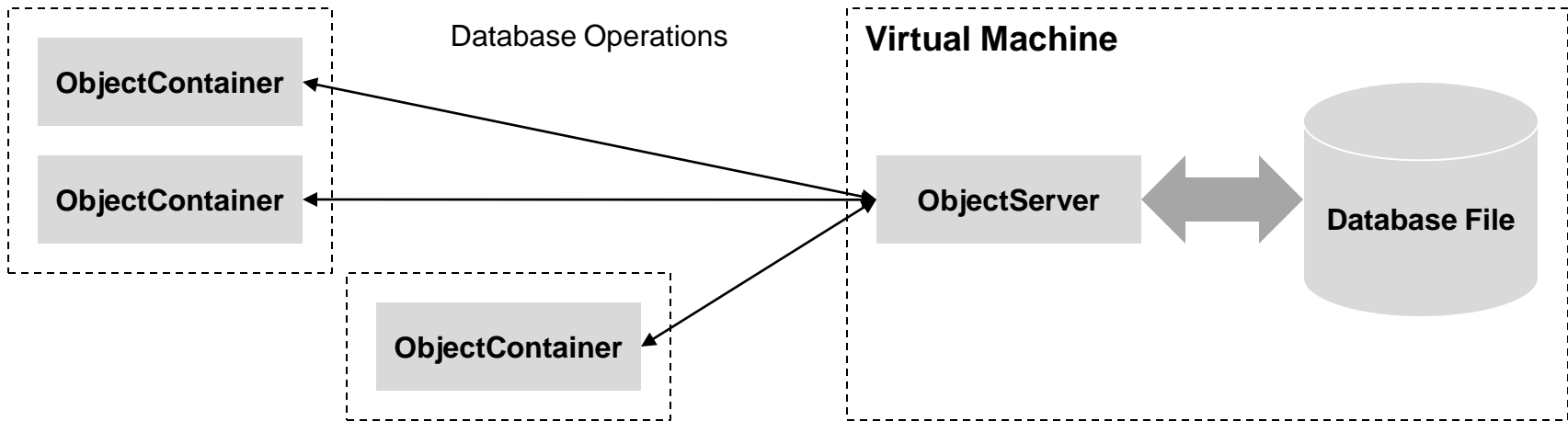
- Application and database in the same virtual machine
- Database file opened, locked and accessed directly
  - `Db4oEmbedded.openFile(config, name)`
  - database operations performed on embedded object container
- Single user and single thread

# Embedded Mode: Client Session



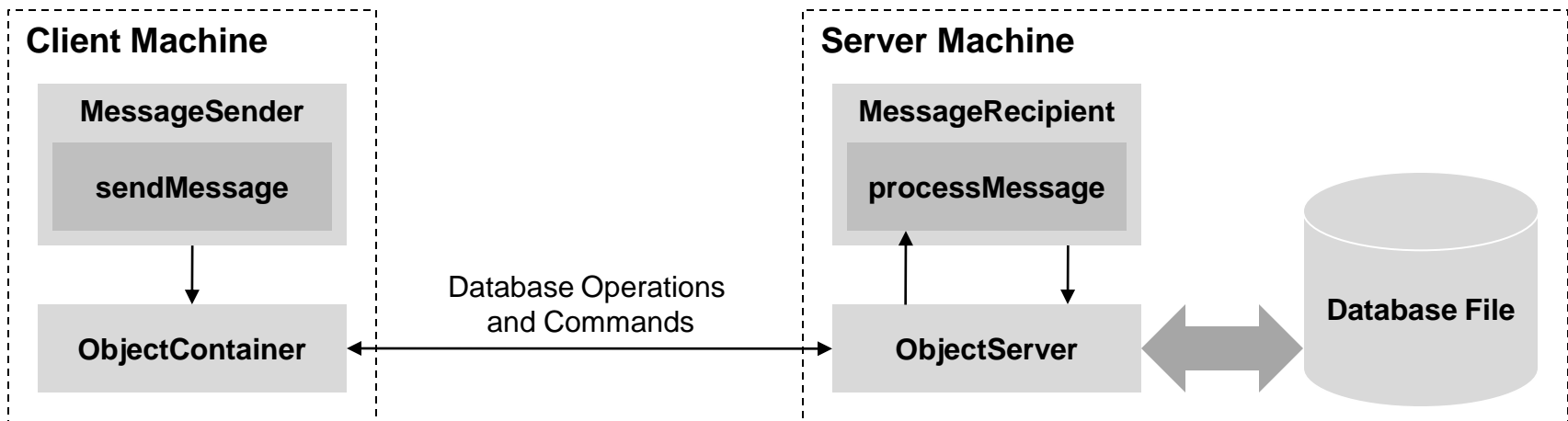
- Application and database in same virtual machine
- Database file accessed through client session
  - `EmbeddedObjectContainer#openSession()`
  - database operation performed on session object container
- Single user and multiple threads

# Client/Server Modes: Networking Mode



- Client opens TCP/IP connection to server
  - `Db4oClientServer.openServer(filename, port)`
  - `Db4oClientServer.openClient(host, port, user, pass)`
- Client sends query, insert, update and delete instructions to server and receives data from the server

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

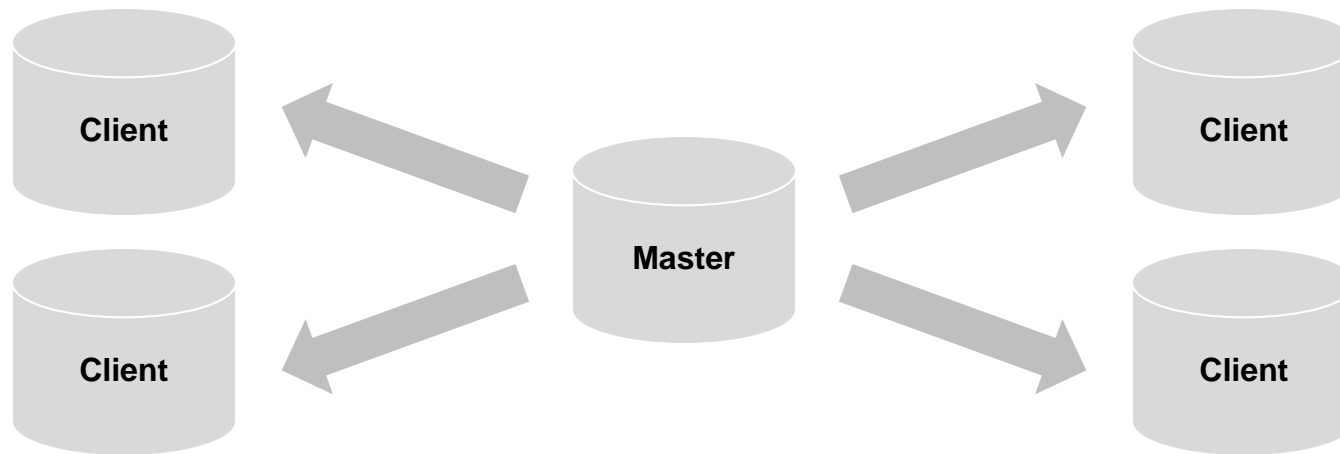


- Basic client/server mode cannot transmit commands
  - operations are limited to methods of **ObjectContainer**
- Out-of-band signalling
  - **MessageSender#send(object)**
  - **MessageRecipient#processMessage(context, message)**

# Replication

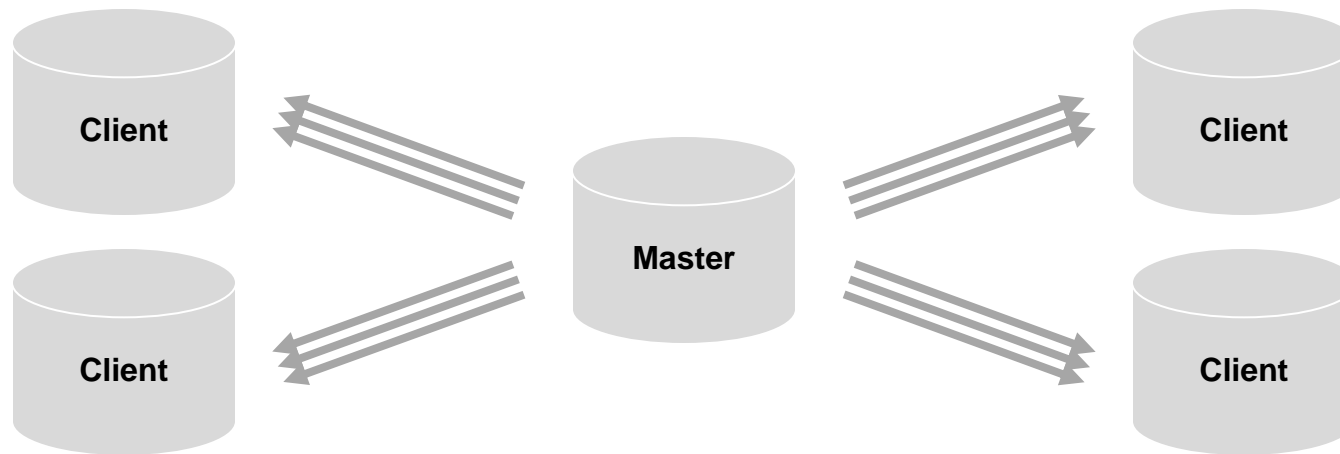
- Database managed by redundant servers
  - data changes on masters or publishers
  - changes replicated to clients of subscribers
- Several forms of replication supported
  - snapshot replication
  - transactional replication
  - merge replication
- Replication in db4o has to be coded into application and cannot be configured on an administrative level
  - replication only occurs on demand, i.e. not automatically
  - client/master semantics introduced by developer
  - db4o provides one interface to support all forms of replication

# Replication Modes: Snapshot Replication



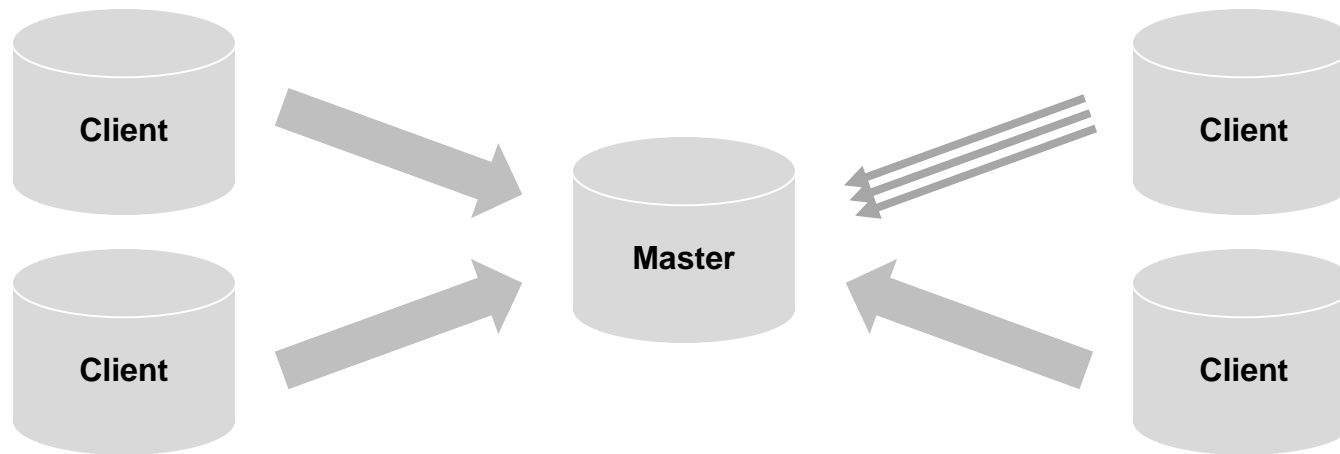
- Snapshots of the master database replicated to client
  - state-based
  - periodical schedule
- Support in db4o
  - special SODA query to detect all new and updated objects

# Replication Modes: Transactional Replication



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

# 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



# 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
- Transfers data between replication providers
- Supported replication providers
  - db4o to db4o
  - db4o to Hibernate, Hibernate to db4o
  - Hibernate to Hibernate

# db4o Replication System

- Requires three steps
  - generating unique IDs and version numbers
  - creating a **ReplicationSession** object
  - replicating objects
- Replication mode is dependent on implementation
- Replication is bidirectional by default
  - replication can be configured to be unidirectional using method **ReplicationSession#setDirection(from, to)**
  - on **ReplicationSession#replicate(object)** newer version of object will be transferred to the database with older version
- Replication has object granularity
  - also traverses new or changed members in object graph

# db4o Replication System

```
// configuration
EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
config.file().generateUUIDs(ConfigScope.GLOBALLY);
config.file().generateVersionNumbers(ConfigScope.GLOBALLY);
ObjectContainer db1 = Db4oEmbedded.openFile(config, "test1.db");
ObjectContainer db2 = Db4oEmbedded.openFile(config, "test2.db");

// replication session
ReplicationSession replication = Replication.begin(db1, db2,
    new ReplicationEventListener() {
        public onReplicate(ReplicationEvent e) {
            if (e.isConflict()) {
                e.overrideWith(e.stateInProviderA());
            }
        }
    }
);
replication.setDirection(db1, db2);
```

# db4o Replication System

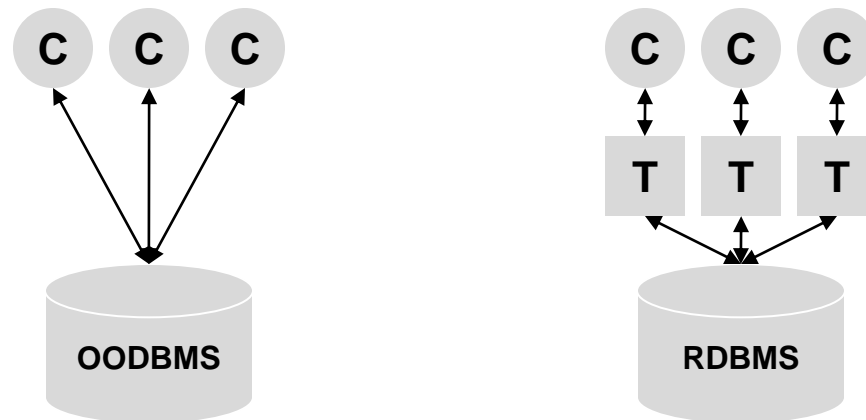
```
// update database and replicate (transactional replication)
Author stanzetta = new Author("Christoph Zimmerli");
db1.store(stanzetta);
replication.replicate(stanzetta);
replication.commit();

// replicate changed publications (snapshot replication)
ReplicationProvider provider1 = replication.providerA();
ObjectSet<Publication> result =
    provider1.objectsChangedSinceLastReplication(Publication.class);
for (Publication publication: result) {
    replication.replicate(publication);
}
replication.commit();

// close the replication session
replication.close();
```

# Schema Evolution

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



# 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

```
CommonConfiguration#objectClass(...).objectField(...).rename(...);
```

- Renaming a class
  - managed through configuration interface

```
CommonConfiguration#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

# 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
- Settings adjusted through configuration interface

```
// global setting (default: depends on environment)
CommonConfiguration#callConstructors(true)

// per class setting (default: depends on environment)
CommonConfiguration#objectClass(...).callConstructors(true)

// exceptions for debugging (default: true)
CommonConfiguration#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 onInstantiate(  
    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;  
    }  
}
```

```
CommonConfiguration#objectClass(Person.class).translate(  
    new PersonTranslator())
```

# Type Handlers

- Instead of Translators

# 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://developer.db4o.com/resources/api/db4o-java/>
  - <http://developer.db4o.com/resources/api/dRS-java/>
- Jim Paterson, Stefan Edlich, Henrik Hörning, and Reidar Hörning: **The Definitive Guide to db4o**, *APress* 2006

# Next Week

## Version Models

- Temporal Databases
- Engineering Databases
- Software Configuration Systems

