

Object-Oriented Databases

db4o: Part 1

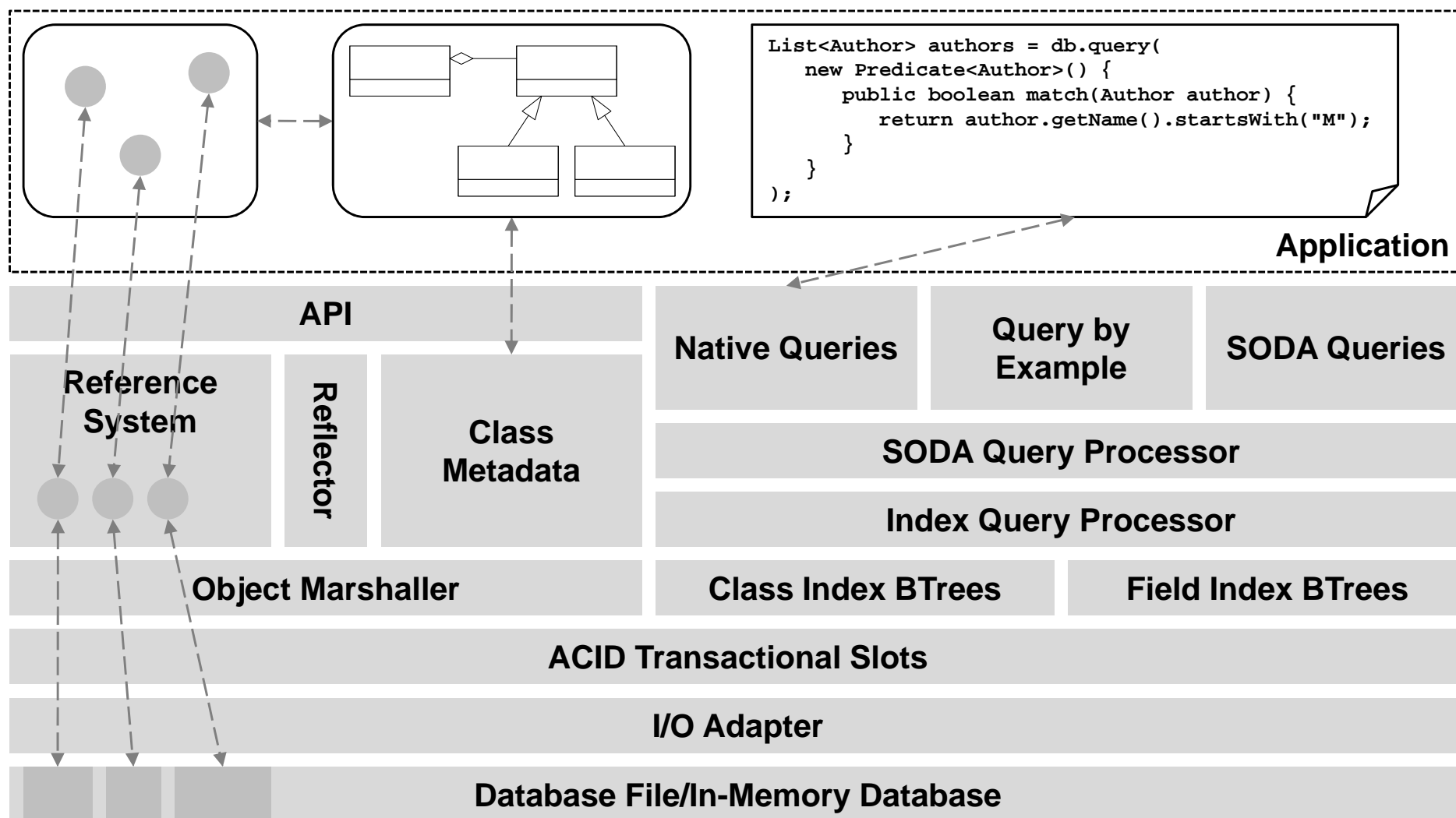
- Managing Databases, Storing and Retrieving Objects
- Query by Example, Native Queries, SODA
- Simple and Complex Objects, Activation, Transactions



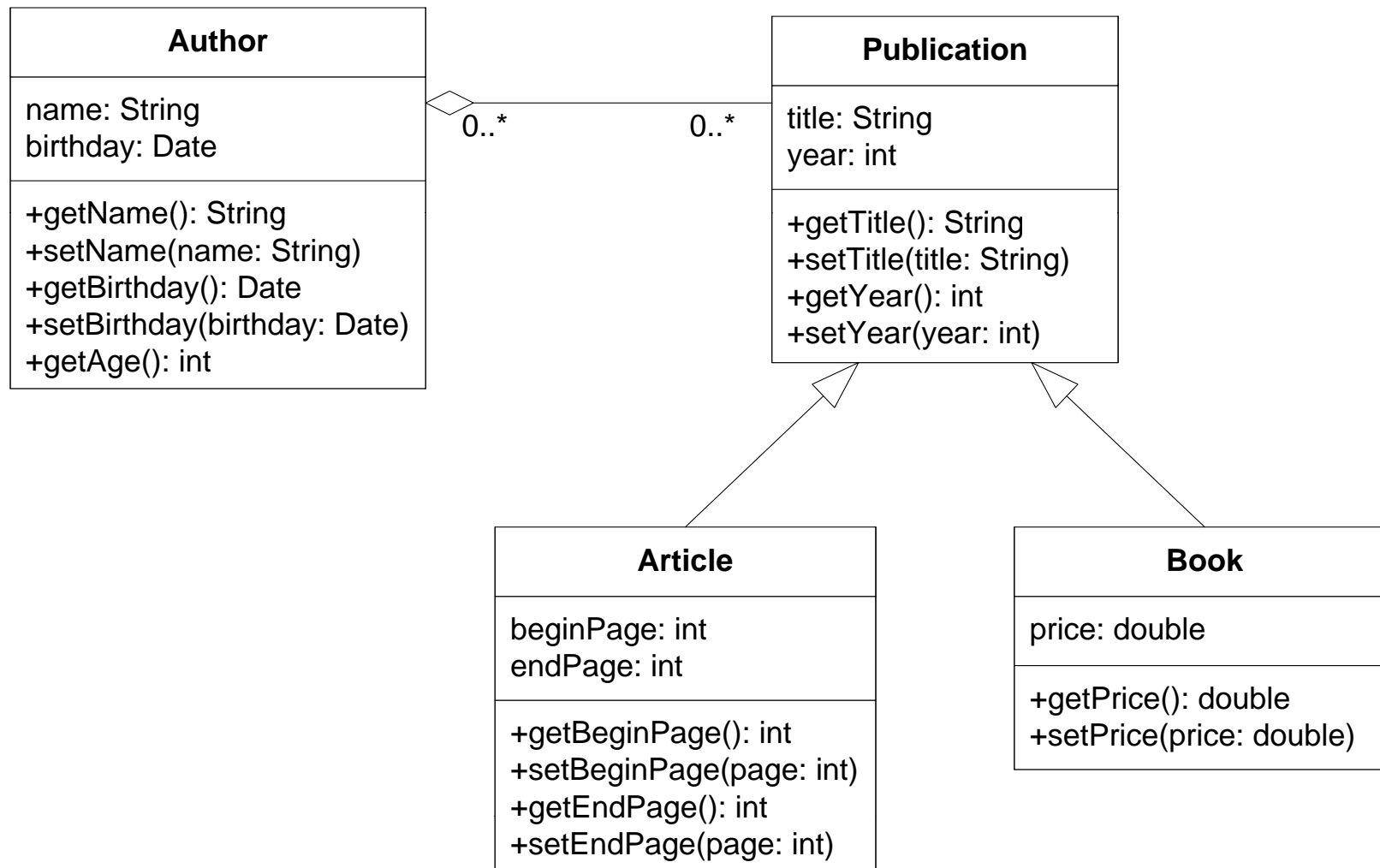
Introducing db4o

- Open source native object database for Java and .NET
- Key features
 - No conversion or mapping needed
 - No changes to classes to make objects persistent
 - One line of code to store objects of any complexity
 - Works in local or client/server mode
 - ACID transaction model
 - Object caching and integration with native garbage collection
 - Automatic management and versioning of database schema
 - Seamless Java or .NET language binding
 - Small memory foot-print (single 500Kb library)

db4o Architecture



Example Class Hierarchy



Example Java Classes

```
public class Author {  
  
    private String name;  
    private Date birthday;  
    private Set<Publication> pubs;  
  
    public Author(String name) {  
        this.name = name;  
        this.pubs =  
            new HashSet<Publication>();  
    }  
    public String getName() {  
        return this.name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    ...  
  
}
```

```
public class Publication {  
  
    private String title;  
    private int year;  
    private List<Author> authors;  
  
    public Publication(String title) {  
        this.title = title;  
        this.authors =  
            new ArrayList<Author>();  
    }  
    public String getTitle() {  
        return this.title;  
    }  
    public void setTitle(String title) {  
        this.title = title;  
    }  
  
    ...  
  
}
```

Object Container

- Represents db4o databases
 - supports local file mode or client connections to db4o server
- Owns one transaction
 - all operations are executed transactional
 - transaction is started when object container is opened
 - after commit or rollback next transaction is started automatically
- Maintains references to stored and instantiated objects
 - manages object identities
 - is able to achieve a high level of performance
- Lifecycle
 - intended to be kept open as long as programs work against it
 - references to objects in RAM will be discarded when closed

Storing Objects

```
// create a publication
Publication concepts =
    new Publication("Information Concepts for Content Management");

// create authors
Author michael = new Author("Michael Grossniklaus");
Author moira = new Author("Moira C. Norrie");

// assign authors to publication
concepts.addAuthor(michael);
concepts.addAuthor(moira);

// store complex object
database.store(concepts);
```

- Objects stored using method `set` of `ObjectContainer`
- Stores objects of arbitrary complexity
- Persistence by reachability

Retrieving Objects

- db4o supports three query languages
- Query by Example
 - simple method based on prototype objects
 - selects exact matches only
- Native Queries
 - expressed in application programming language
 - type safe
 - transformed to SODA and optimised
- Simple Object Data Access (SODA)
 - query API based on the notion of a query graph
 - methods for descending graph and applying constraints

Query by Example

```
ObjectContainer database = Db4o.openFile("test.db");

// get author "Moira C. Norrie"
Author proto = new Author("Moira C. Norrie");
ObjectSet<Author> authors = database.queryByExample(proto);
for (Author author: authors) {
    System.out.println(author.getName());
}

// get all publications
ObjectSet<Publication> publications = database.query(Publication.class);
for (Publication publication: publications) {
    System.out.println(publication.getTitle());
}
```

Native Queries

```
ObjectContainer database = Db4o.openFile("test.db");

// find all publications after 1995
ObjectSet<Publication> publications = database.query(

    new Predicate<Publication>() {
        public boolean match(Publication publication) {
            return publication.getYear() > 1995;
        }
    }

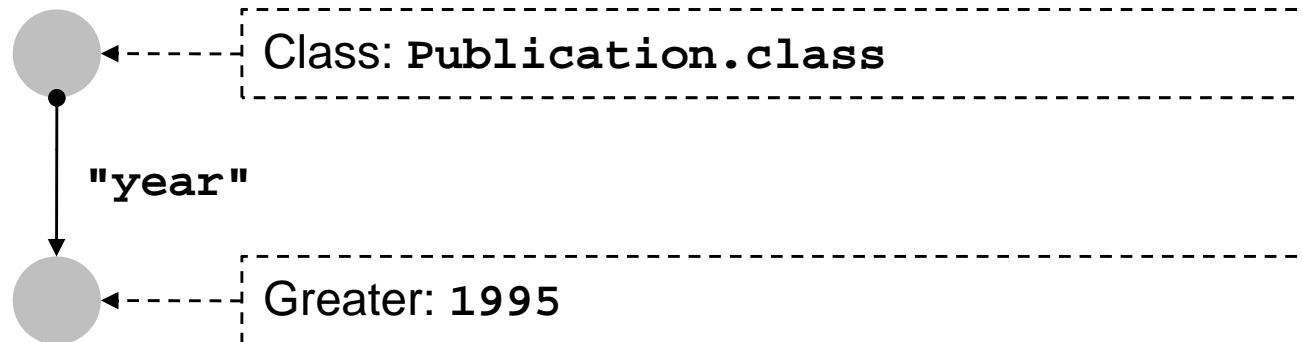
);
for (Publication publication: publications) {
    System.out.println(publication.getTitle());
}
```

SODA Queries

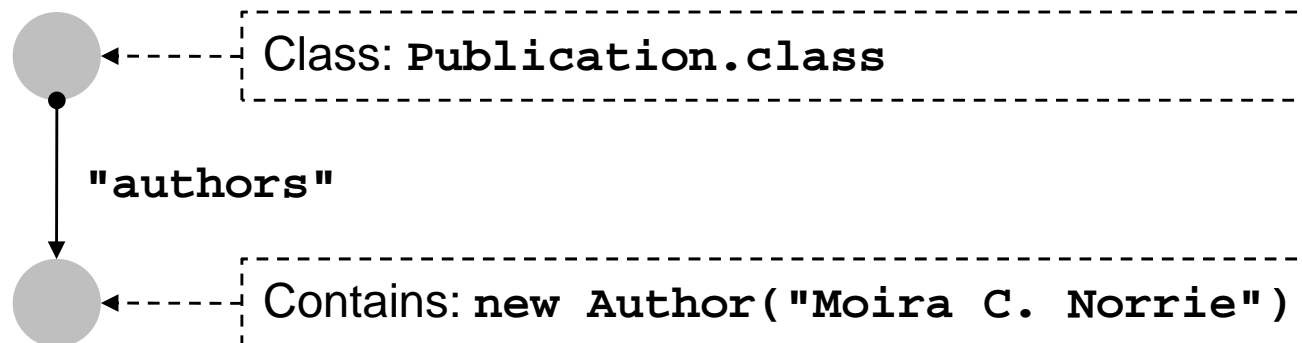
- Expressed using `Query` objects
 - `descend` adds or traverses a node in the query tree
 - `constrain` adds a constraint to a node in the query tree
 - `sortBy` sorts the result set
 - `orderAscending` and `orderDescending`
 - `execute` executes the query
- Interface `Constraint`
 - `greater` and `smaller` comparison modes
 - `identity`, `equal` and `like` evaluation modes
 - `and`, `or` and `not` operators
 - `startsWith` and `endsWith` string comparisons
 - `contains` to test collection membership

SODA Queries

- Find all publications published after 1995



- Find all publications of author "Moira C. Norrie"



SODA Queries

```
ObjectContainer database = Db4o.openFile("test.db");

// find all publications after 1995
Query query = database.query();
query.constrain(Publication.class);
query.descend("year").constrain(Integer.valueOf(1995)).greater();
ObjectSet<Publication> publications = query.execute();
for (Publication publication : publications) {
    System.out.println(publication.getTitle());
}

// find all publications of author "Maira C. Norrie"
Query query = database.query();
query.constrain(Publication.class);
Author proto = new Author("Maira C. Norrie");
query.descend("authors").constrain(proto).contains();
ObjectSet<Publication> publications = query.execute();
for (Publication publication : publications) {
    System.out.println(publication.getTitle());
}
```

Updating Objects

- Update procedure for persistent object
 - retrieve desired object from the database
 - perform the required changes and modification
 - store object back to the database by calling the `store` method
- Background
 - db4o uses IDs to maintain connections between in-memory objects and corresponding stored objects
 - IDs are cached as weak references until database is closed
 - fresh reference is required to update objects
 - querying for objects ensures fresh reference
 - creating and setting objects ensures fresh reference
 - db4o uses reference to find and update stored object automatically when `store` method is called

Updating Objects

```
ObjectContainer database = Db4o.openFile("test.db");

Author michael =
    (Author) database.get(new Author("Michael Grossniklaus")).next();

Calendar calendar = Calendar.getInstance();
calendar.set(1976, Calendar.JUNE, 22);
michael.setBirthday(calendar.getTime());

database.store(michael);
```

Deleting Objects

```
ObjectContainer database = Db4o.openFile("test.db");

// retrieving author "Moirra C. Norrie"
Author moira =
    (Author) database.queryByExample(new Author("Moirra C. Norrie")).next();

// deleting author "Moirra C. Norrie"
database.delete(moira);
```

- Similar to updating objects
 - fresh reference required
 - established by querying or by creating and setting
- Method `delete` of `ObjectContainer` removes objects
- What happens to referenced objects?

Simple Structured Objects

- Storing of new objects using the **store** method
 - object graph is traversed and all referenced objects are stored
 - persistence by reachability
- Updating of existing objects using the **store** method
 - by default update depth is set to one
 - only primitive and string values are updated
 - object graph is not traversed for reasons of performance
- Deleting existing objects using the **delete** method
 - by default delete operations are not cascaded
 - referenced objects have to be deleted manually
 - cascading delete can be configured for individual classes

Updating Simple Structured Objects

```
ObjectContainer database = Db4o.openFile("test.db");

// retrieving author "Moira C. Norrie"
Author moira =
    (Author) database.queryByExample(new Author("Moira C. Norrie")).next();

// updating all publications
for (Publication publications: moira.getPublications()) {
    publication.setYear(2007);
}

// storing author "Moira C. Norrie" has no effect on publications
database.store(moira);

// storing updated publications
for (Publication publications: moira.getPublications()) {
    database.store(publication);
}
```

Updating Simple Structured Objects

- Cascading updates can be configured per class using method `cascadeOnUpdate` from `ObjectClass`
- Update depth can be configured
 - method `store(object, depth)` from `ExtObjectContainer` updates referenced fields to the given depth
 - method `updateDepth(depth)` from `ObjectClass` defines a sufficient update depth for a class of objects
 - method `updateDepth(depth)` from `Configuration` sets global update depth for all persisted objects
- Global update depth not flexible enough for real-world objects having different depth of reference structures

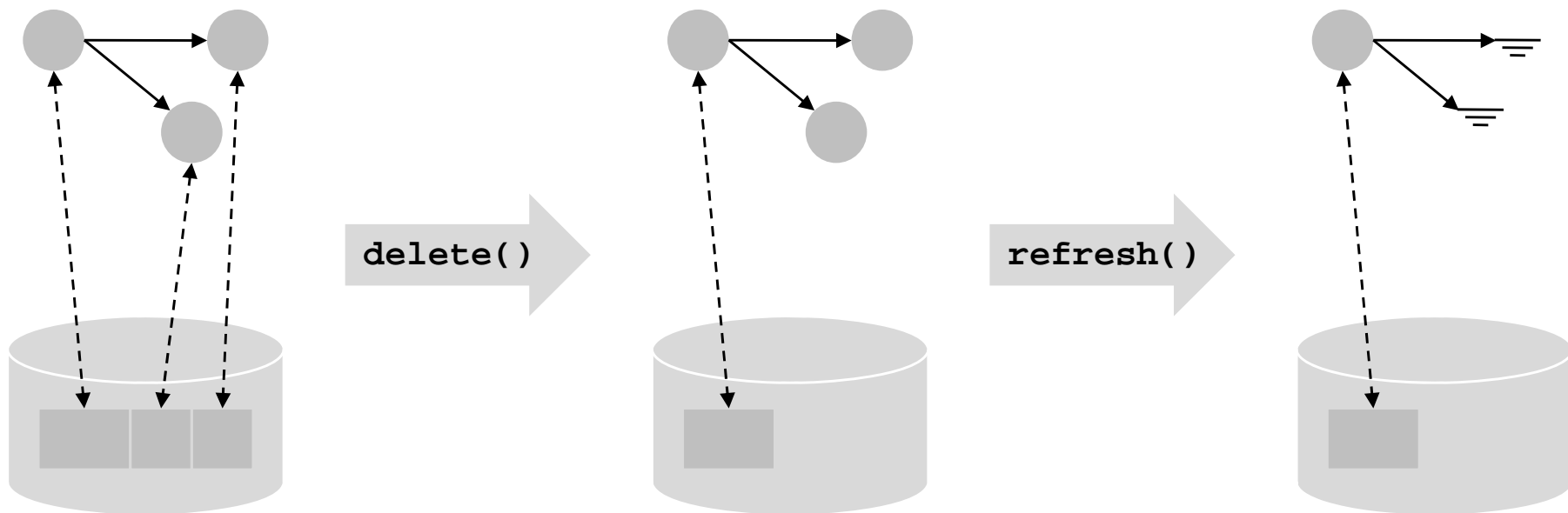
Deleting Simple Structured Objects

```
ObjectContainer database = Db4o.openFile("test.db");  
// configuration of cascading deletes for Author objects  
Db4o.configure().objectClass(Author.class).cascadeOnDelete(true);  
  
// retrieving author "Moira C. Norrie"  
Author moira =  
    (Author) database.queryByExample(new Author("Moira C. Norrie")).next();  
  
// deleting author "Moira C. Norrie"  
database.delete(moira);
```

- Cascading deletes similar to cascading updates
 - configured per object class
 - method `objectClass` from `Configuration`
 - method `cascadeOnDelete` from `ObjectClass`
- What happens if deleted objects referenced elsewhere?

Deleting Simple Structured Objects

- Inconsistencies between in-memory and stored objects
 - cache and disk can become inconsistent when deleting objects
 - method `refresh` of `ExtObjectContainer` syncs objects
 - restores memory objects to committed values on disk



Object Hierarchies

- db4o handles complex object structures automatically
 - hierarchies, composite hierarchies
 - inverse associations
 - inheritance and interfaces
 - multi-valued attributes, arrays and collections
- Configuration of cascading operation applies
- db4o database-aware collections
 - `ArrayList4` and `ArrayMap4` implement Collections API
 - part of transparent persistence activation framework
 - additional configuration methods from `Activatable` interface
 - complex object implementation becomes db4o dependant

Transparent Persistence

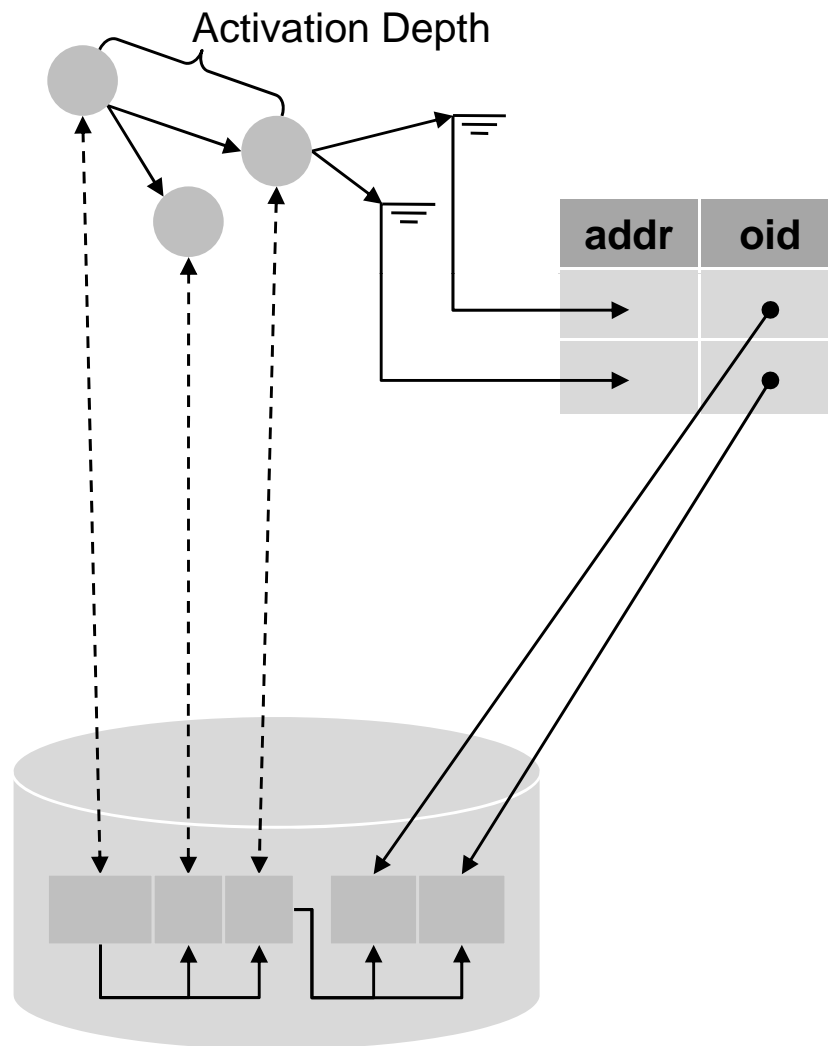
- Make persistence transparent to application logic
 - store objects in database once using the `store` method
 - avoid more class to `store` method as database manages objects
- Logic of the transparent persistence framework
 - transparently persistent objects implement `Activatable` interface
 - instances are initially made persistent using `store` method
 - objects are bound to the transparent persistence framework when stored or retrieved using the `bind` method
 - upon commit, the transparent persistence framework scans for modified persistent objects and implicitly invokes the `store` method
- Enabling transparent persistence

```
configuration.add(new TransparentPersistenceSupport());
```

Activation

- Activation controls instantiation of object fields
 - object field values are loaded into memory only to a certain depth when a query retrieves objects
 - activation depth denotes the length of the reference chain from an object to another
 - fields beyond the activation depth are set to null for object references or to default values for primitive types
- Activation occurs in the following cases
 - method `next` is called on an `ObjectSet` retrieved in a query
 - explicit object activation by `activate` from `ObjectContainer`
 - a db4o collection element is accessed
 - members of Java collections are activated automatically when collection is activated

Activation



- Fields beyond activation depth are not loaded into memory
- Weak references are used to later activate these fields
 - table instead of direct reference
 - memory address mapped to persistent id of inactive object
- Inactive objects are activated using mapping table

Activation

- Activation depth trade-off
 - if set to maximum, whole object graphs are loaded into memory for every retrieved object and no manual activation needed
 - if set to minimum, memory consumption is reduced to the lowest level but all the activation logic is left to the application code
- Controlling activation
 - default activation depth is 5
 - methods `activate` and `deactivate` of `ObjectContainer`
 - per class configuration

```
ObjectClass#minimumActivationDepth(minDepth)  
ObjectClass#maximumActivationDepth(maxDepth)  
ObjectClass#cascadeOnActivate(bool)  
ObjectClass#objectField(...).cascadeOnActivate(bool)
```

Transparent Activation

- Make activation transparent to application logic
 - activate fields automatically when they are accessed
 - ease maintenance of multi-level activation strategies
- Logic of the transparent activation framework
 - transparently activated objects implement **Activatable** interface
 - when an object is instantiated, the database registers itself with the object using the **bind** method
 - instances are not activated automatically
 - upon access the **activate** method is used to check whether the field has been activated and, if not, load the value
- Enabling the transparent activation framework

```
configuration.add(new TransparentActivationSupport());
```

Transparent Persistence and Activation Example

```
public class Author implements Activatable {  
  
    // activator  
    transient Activator activator;  
    ...  
  
    public Author() {  
        // empty constructor for instantiation  
    }  
  
    // bind instance to the framework  
    public void bind(Activator a) {  
        if (this.activator == a) {  
            return;  
        }  
        if (a != null && this.activator != null) {  
            throw new IllegalStateException();  
        }  
        this.activator = a;  
    }  
}
```

```
// field activation  
public void activate(ActivationPurpose p) {  
    if (this.activator == null) {  
        return;  
    }  
    this.activator.activate(purpose);  
}  
  
// read activation  
public Date getBirthday() {  
    this.activate(ActivationPurpose.READ);  
    return this.birthday;  
}  
  
// write activation  
public void setBirthday(Date birthday) {  
    this.activate(ActivationPurpose.WRITE);  
    this.birthday = birthday;  
}  
  
...  
}
```

db4o Transactions

- ACID transaction model
- Data transaction journaling
 - zero data loss in case of system failure
 - automatic data recovery after system failure
- db4o core is thread-safe for simultaneous operations
- All work within db4o **ObjectContainer** is transactional
 - transaction implicitly started implicitly when container opened
 - current transaction committed implicitly when container closed
 - explicit commit using method **commit** of **ObjectContainer**
 - explicit abort using method **rollback** of **ObjectContainer**

Database Commit

```
ObjectContainer database = Db4o.openFile("test.db");

// retrieving author "Maira C. Norrie"
Author moira =
    (Author) database.queryByExample(new Author("Maira C. Norrie")).next();

// creating author "Stefania Leone"
Author stefania = new Author("Stefania Leone");

// creating new publication
Publication article = new Publication("Web 2.0 Survey");
article.addAuthor(stefania);
article.addAuthor(moira);

// storing publication
database.store(article);

// committing database
database.commit();
```

Database Rollback

- Modifications are written to temporary memory storage
- Implicit or explicit commit writes the modifications to disk
- Database rollback resets last committed database state

```
ObjectContainer database = Db4o.openFile("test.db");

// retrieving publication
Publication article = (Publication) database.queryByExample(
    new Publication("Web 2.0 Survey")).next();

// updating publication
Author michael = new Author("Michael Grossniklaus");
article.addAuthor(michael);
database.store(article);

// aborting transaction
database.rollback();
```

Database Rollback

- Again, inconsistencies of memory and disk possible
 - method rollback cancels modifications on disk
 - state of the objects in reference cache is not adapted
 - live objects need to be refreshed explicitly

```
ObjectContainer database = Db4o.openFile("test.db");
// retrieving publication
Publication article = (Publication) database.queryByExample(
    new Publication("Web 2.0 Survey")).next();
// updating publication
Author michael = new Author("Michael Grossniklaus");
article.addAuthor(michael);
database.store(article);
// aborting transaction
database.rollback();
// refreshing article to remove author from in-memory representation
database.ext().refresh(article, Integer.MAX_VALUE);
```

Concurrent Transactions

- Digression on isolation levels
 - **read uncommitted:** values modified by other transactions can be read before they are committed
 - **read committed:** only values that have been modified and committed by other transactions can be read
 - **repeatable read:** all read operations within a transaction yield the same result
 - **serialisable:** database state resulting from concurrent execution of transactions could have been obtained from a possible serial execution of the same transactions
- db4o uses the read committed isolation level
- Inconsistencies and collisions can occur!

Collision Detection

- Check if object has changed during transaction before committing a transaction
 - store value of object in local variable at transaction start
 - use `peekPersisted` method of `ExtObjectContainer` to look at the persistent version of the object
 - compare initial value to stored value
 - rollback current transaction if value has changed
- Method `peekPersisted` returns a transient object that has no connection to the database
 - instantiation depth of transient object can be configured
 - method can be used to read either committed or set values

Collision Detection

```
ObjectContainer client = Db4o.openClient("localhost", 3927, "...", "...");

// retrieving author "Maira Norrie"
Author moira =
    (Author) client.queryByExample(new Author("Maira C. Norrie")).next();

// storing initial value of field
Date birthday = moira.getBirthday();

...

// retrieve stored value of field
Author persisted = client.ext().peekPersisted(moira, 9, true);

// compare the values and abort if necessary
if (persisted.getBirthday() != birthday) {
    client.rollback();
} else {
    client.commit();
}
```

Collision Avoidance with Semaphores

- Avoid collisions by locking objects explicitly
- Semaphores can be used protect critical code sections
 - a unique name must be provided
 - time to wait if a semaphore is already owned by another transaction has to be given
- Semaphores form basis to implement custom locking

```
ObjectContainer client = Db4o.openClient("localhost", 3927, "...", "...");

if (client.ext().setSemaphore("SEMAPHORE#1", 1000)) {
    // critical code section
    ...
    // release semaphore after critical section
    client.ext().releaseSemaphore("SEMAPHORE#1");
}
```

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

db4o: Part 2

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

