



# Tehnologii Java

*Curs -*

Cristian Frăsinaru

`acf@infoiasi.ro`

Facultatea de Informatică

**Universitatea "Al. I. Cuza" Iași**





# **Baze de date orientate obiect (db4o)**



# Cuprins

---

- Introducere
- Prima aplicație
- Caracteristici
- Persistența relațiilor
- Mecanisme de interogare
- Modelul client/server
- Transmiterea de mesaje
- Aspecte legate de configurare



# Introducere



# Nivelul datelor

- **Relațional ← RDBMS**
  - + : viteză, suport teoretic, răspândire largă
  - - : dificil de utilizat din aplicații OO
- **Obiectual-Relațional ← RDBMS, OORDBMS**
  - + : separarea modelelor obiectual-relațional
  - - : "impedance mismatch", performanță
- **Obiectual ← OODMS**
  - + : simplu de utilizat din aplicații OO
  - - : răspândire redusă

# Problemele modelului OO

- Crearea unui mecanism de serializare eficient
- Posibilitatea schimbării structurii obiectelor unei anumite clase
- Necesitatea salvării unor ierarhii (grafuri) de obiecte
- Implementarea unui mecanism intern de asignare de ID-uri
- Definirea unei modalități eficiente de interogare
- Gestiunea memoriei

# Refactoring

*Refactoring* = Modificarea sintactică a unui element software fără alterarea semanticii, cu scopul de a aderara la diverse convenții sau standarde. Poate fi aplicat la nivel:

- **obiectual**: redenumirea claselor, metodelor, pachetelor, etc din codul sursă; ușor de realizat; există instrumente specializate
- **relațional**: redenumirea tabelelor, coloanelor, procedurilor stocate, etc din baza de date; greu de realizat; nu există instrumente specializate; Implicații negative asupra interogărilor:

```
statement.executeQuery("select nume from persons where id=1");
```

# Un nou tip de bază de date

- **OODMS** → *Database for Objects*
- Disponibilă pe platformele standard, micro și enterprise
- Consum minimal de resurse
- Performanță: scriere, citire
- Simplitate în utilizare: *"as simple as possible, but no simpler"*
- Portabilitate: separare funcțională între implementare și accesare
- Siguranță: *ACID*

# Avantaje

- Complet OO  $\rightarrow$   $\nexists$  "impedance mismatch"
- Modelul obiectual == schema bazei de date
- Elimină compromisul OO - *performanță*
- Simplitate în utilizare: "The One-Line-of-Code-Database"
  - Elimină nivelul intermediar de mapare
  - Interogările sunt OO
- Reducerea timpului de dezvoltare a aplicațiilor
- Performanță
- Utilizabil în aplicații standard, Web, pe dispozitive mobile, sisteme în timp-real.

# Prima aplicație (DB4O)

# Componentele aplicației

- Modelul obiectual: clasa `Person`
- Modelul relațional: -
- Fișierul de mapare: -
- Fișierul de configurare: -
- Baza de date: fișierul `database.yap`
- Aplicația în sine

# Clasa Person

```
public class Person {
    private String name;

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

    public String getName() {
        return name;
    }

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

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

# Aplicația

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

```
Person duffy = new Person("Duffy Duck");  
db.set(duffy);
```

```
Person mickey = new Person("Mickey Mouse");  
db.set(mickey);
```

```
Person proto = new Person(null);  
ObjectSet result=db.get(proto);
```

```
System.out.println(result.size());  
while(result.hasNext()) {  
    System.out.println(result.next());  
}
```

```
db.close();
```

# API-ul de bază

- **Deschiderea unei baze de date**  
`Db4o.openFile` → `ObjectContainer`
- **Salvarea obiectelor:** `set`
- **Regăsirea obiectelor:** `get`  
*Query By Example:* `get` → `ObjectSet`
- **Actualizarea obiectelor:** `get + set`
- **Ștergerea obiectelor:** `delete`
- **Persistența obiectelor:** `commit`, `rollback`
- **Inchiderea bazei de date:** `close`



# Persistența relațiilor



# Asocieri simple

Salvarea unei ierarhii de obiecte se face apelând metoda `set` pentru obiectul părinte. Salvare va fi realizată automat în **cascadă**.

```
PersonType type = new PersonType("Cartoons");
```

```
Person walt = new Person("Walt Disney");  
walt.setType(type);
```

```
Person mickey = new Person("Mickey Mouse");  
mickey.setType(type);  
mickey.setBoss(boss);
```

```
db.set(mickey)
```

# Restaurarea în cascadă

Restaurarea în cascadă este controlată de **nivelul de activare** asociat fiecărei clase. Implicit:

- Nivelul de activare este 5
- Referințele de după al cincilea nivel vor fi `null`

Nivelul de activare poate fi configurat înainte de deschiderea bazei de date:

```
Db4o.activationDepth(10);  
Db4o.configure().objectClass(Person.class).minimumActivationDepth(10);  
Db4o.configure().objectClass(Person.class).cascadeOnActivate(true);
```

# Actualizarea în cascadă

Actualizarea în cascadă este controlată de **nivelul de persistență (update depth)** asociat fiecărei clase.

Implicit:

- Nivelul de persistență este 0
- Doar tipurile primitive și `String` vor fi actualizate

Nivelul de persistență poate fi configurat înainte de deschiderea bazei de date:

```
Db4o.configure().objectClass("Person").cascadeOnUpdate(true);  
Db4o.configure().objectClass("Person").updateDepth(3);  
Db4o.configure().objectClass("Person").cascadeOnDelete(true);
```

# Tablouri și colecții

Tablourile și colecțiile se comportă exact ca orice alt obiect.

```
public class Person {  
    private List<Task> tasks;  
    private String[] addrs;  
    ...  
}
```

```
Person duke = new Person("Duke");  
List<Task> tasks = new ArrayList<Task>();  
tasks.add(new Task("compile"));  
tasks.add(new Task("run"));  
duke.setTasks(tasks);  
duke.setAddrs(new String[]{"duke@sun.com", "duke@infoiasi.ro"});  
db.set(duke);
```

# Moștenirea

```
public interface Human {... }  
public class Person { ... }  
public class Employee extends Person implements Human {...}  
public class Executive extends Employee {...}
```

```
Person proto=new Person(null);  
ObjectSet result=db.get(proto);
```

```
Person proto=new Executive(null);  
ObjectSet result=db.get(proto);
```

```
ObjectSet result=db.get(Human.class);
```

# Mecanisme de interogare

# Tipuri de interogări

- **Simple Object Database Access (SODA) Query API**
- **Query-By-Example (QBE)**
- **Native Queries (NQ)**

# Query by Example

## Interogare pe baza unui **prototip**

```
Person proto = new Person(null);  
ObjectSet result = db.get(proto);
```

### Dezavantaje:

- Limitări severe în exprimarea interogării
- Crearea unui obiect prototip poate avea efecte secundare
- Unele proprietăți pot să nu fie specificate în constructori
- Nu putem crea interogări pentru valorile implicite (`null`)

# SODA Query API



Crearea unui **graf de interogare** format din obiecte și proprietăți.

```
Query query=db.query();  
query.constrain(Person.class);
```

```
query.descend("name").constrain("Duke");  
ObjectSet result = query.execute();
```

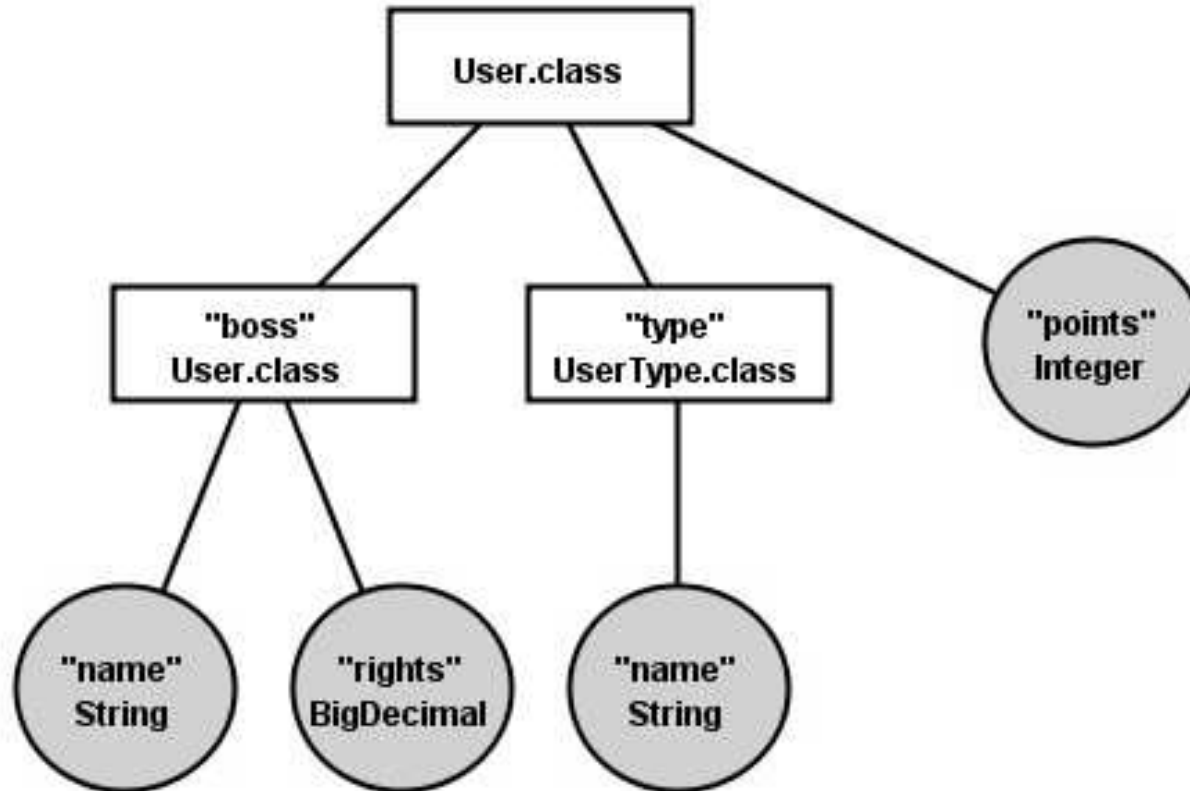
```
query.descend("type").descend("name").constrain("Cartoons");  
ObjectSet result = query.execute();
```

```
//Negatie  
query.descend("name").constrain("Duke").not();
```

```
//Comparare  
query.descend("points").constrain(100).greater();
```



# Graful de interogare



# Conjunții, disjunții

Mai multe obiecte `Constraint` pot fi legate prin metodele `and` sau `or`.

```
Query query = db.query();  
query.constrain(Person.class);
```

```
Constraint active = query.descend("active").constrain(true);  
query.descend("points").constrain(0).greater().and(constr);
```

```
Constraint inactive = query.descend("active").constrain(false);  
query.descend("points").constrain(0).and(constr);
```

# Problema

```
public class Person {  
    private String name;  
    private int age;  
    public String getName(){  
        return name;  
    }  
    public int getAge(){  
        return age;  
    } ...  
}
```

-----

```
Query query = database.query();  
query.constrain(Person.class);  
query.descend("age").constrain(20).smaller();
```

# Objective



Dorim crearea unui mecanism în care interogările să fie:

- **native**
- **orientate-obiect**
- **type-safe**
- **eficiente**



# Native Query

- Soluția **recomandată** pentru exprimarea interogărilor în db4o
- Permit specificarea unei **secvențe de cod (predicat)** care să fie rulată pe toate instanțele unei clase
- Execuția secvenței este **eficientă** - fără a instanția efectiv obiecte (dacă este posibil)
- Se bazează pe SODA Query API.

# Predicate

**Predicatele** sunt folosite pentru exprimarea unor condiții complexe de interogare.

- Sunt clase anonime care extind `Predicate<?>`
- Trebuie implementată metoda `match`

```
List<Person> persons = database.query(  
    new Predicate<Person> () {  
        public boolean match(Person person) {  
            return person.getAge() < 20 &&  
                person.getName().startsWith("A");  
        }  
    }  
);
```



# Accesul la baza de date



# Tranzacții

- La deschiderea unei baze (`Db4o.openFile`) de date este creată o **tranzacție implicită**
- **commit**: asigurarea persistenței obiectelor care au fost salvate (`set`)
- **rollback**: ignorarea modificărilor efectuate de la ultimul `commit`
- **refresh**: reactualizarea stării obiectelor

```
db.set(person);  
db.rollback();  
db.ext().refresh(person, 5); //5=activation depth
```

- La închiderea unei baze de date (`db.close`) este executată automat operațiunea de `commit`

# Modelul Client/Server

- In același proces (VM)
- Intre procese diferite (rețea)

```
int port = 0; // fara retea
ObjectServer server=Db4o.openServer("test.yap", port);
try {
    ObjectContainer client=server.openClient();
    // Do something with this client, or open more clients
    client.close();
}
finally {
    server.close();
}
```

# Accesul în rețea



## Server

```
ObjectServer server = Db4o.openServer("testdb", port);
server.grantAccess(user, password);
/* Serverul lanseaza un fir de executie
   Serverul trebuie pus in starea de asteptare
   Serverul va fi inchis la solicitarea unui client
   la expirarea unui timp limita sau niciodata
*/
```

## Client

```
ObjectContainer client =
    Db4o.openClient("localhost", port, user, password);
```

Clasele aplicației trebuie să se găsească atât pe server cât și pe client. Apelurile

`Db4o.configure()` trebuie făcute atât pe server cât și pe client.



# Transmiterea de mesaje

- Clientul poate trimite mesaje serverului:  
MessageSender

```
// get the messageSender for the ObjectContainer
MessageSender messageSender = objectContainer.ext()
    .configure().getMessageSender();
```

```
// send an instance of a StopServer object
messageSender.send(new StopServer());
```

- Serverul poate primi mesaje de la orice client:  
MessageRecipient

```
server.ext().configure().setMessageRecipient(this);
...
public void processMessage(ObjectContainer con, Object message) {
    if(message instanceof StopServer){
        ...
    }
}
```

# Aspecte legate de configurare

# Configurarea Db4o

- Controlul nivelelor de activare, persistență pentru operațiile în cascadă

- Criptarea și parolarea bazei de date

```
Configuration conf = Db4o.configure();  
conf.encrypt(true);  
conf.password("youWillNeverGuessIt");
```

- Deschiderea bazei doar pentru citire: `setReadOnly`
- Selectarea modului verbose: `messageLevel`
- Aspecte legate de performanță

# Folosirea indecșilor

*Index* = facilitate care permite accesul rapid la înregistrările unei baze de date, uzual folosind arbori echilibrați (b-arbori).

## Crearea unui index

```
Db4o.configure().objectClass(Person.class)
    .objectField("name").indexed(true);
```

Spre deosebire de alte configurări, indecșii vor rămâne în baza de date, fiind folosiți automat în sesiunile de lucru următoare.

# Apelul constructorilor

Crearea și popularea obiectelor din baza de date:

- Folosind un constructor al clasei
- Fără a apela vreun constructor
- Folosind un *translator* (permite specificarea explicită a salvării/instanțierii unui obiect)

Configurări:

```
Db4o.configure().callConstructors(true);
```

```
Db4o.configure().objectClass(Person.class).callConstructor(true);
```

```
Db4o.configure().exceptionsOnNotStorable(true);
```

# Identificarea obiectelor

Db4o poate utiliza două mecanisme de identificare a obiectelor:

- **ID-uri interne**

```
long id = objectContainer.ext().getID(object);
```

```
Object obj = objectContainer.ext().getByID(id);  
[objectContainer.activate(object, depth);]
```

- **UUID-uri**

```
Db4o.configure().generateUUIDs(Integer.MAX_VALUE);  
Db4o.configure().objectClass(Person.class).generateUUIDs(true);
```

```
Db4oUUID uuid = objectContainer.ext()  
    .getObjectInfo().getUUID();  
Object obj = objectContainer.ext().getByUUID(uuid);
```

# Replicarea

*Replicare* = Sincronizarea unor baze de date cu aceeași schemă în arhitecturi distribuite. Replicarea datelor către o bază centrală = *Consolidare*.

```
Db4o.configure().generateUUIDs(Integer.MAX_VALUE);  
Db4o.configure().generateVersionNumbers(Integer.MAX_VALUE);
```

```
ReplicationProcess replication =  
    desktop.ext().replicationBegin(  
        handheld, new ReplicationConflictHandler() {  
        public Object resolveConflict(  
            ReplicationProcess replicationProcess,  
            Object a, Object b) {  
            return a;  
        }  
    });  
replication.setDirection(desktop, handheld);
```

# Refactoring ("Evoluția schemei")



## Automat

- Schimbarea interfeței sau API-ului unei clase
- Adăugarea unei noi variabile membre
- Eliminarea unei variabile membre

## Manual

```
//Redenumire clasa
Db4o.configure().objectClass("package.class")
    .rename("newPackage.newClass");

//Redenumire variabile membre
Db4o.configure().objectClass("package.class")
    .objectField("oldField").rename("newField");
```



# Schimbarea structurii clasei



## Interfața **ObjectCallbacks**

```
public class Student {  
    ...  
    String address; // proprietate noua  
    ...  
    public void objectOnActivate(ObjectContainer container) {  
        if (address == null) {  
            address = "Iasi";  
        }  
    }  
}
```



# Mentenanță

- **Defragmentare** - crearea unei noi baze de date, compactare, recreare indecsi, etc
- **Backup**

```
objectContainer.ext().backup(String path)
```