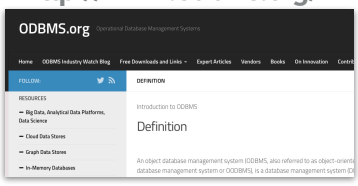
```java
/**
 *
 * @author Prof. Mauro Hemerly (Hämmerli) Gazzani
 */
public class Curso {
     private String nomeCurso = "Engenharia de Computação";
     private String IES = "UEMG - Universidade do Estado de Minas Gerais";
     private String disciplina = "Banco de Dados II";
     private String assuntoDisciplina = "Banco de Dados Orientado a Objetos";
     private String lpoo = "Java";
     private String sgbdoo = "db4o";
     private String semestre = "2019/1";
```

db4objects

UNIVERSIDADE
DO ESTADO DE MINAS GERAIS
UEMG

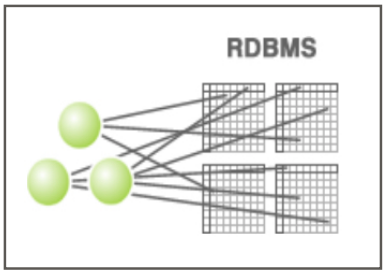
ODMG - Object Database Management Group

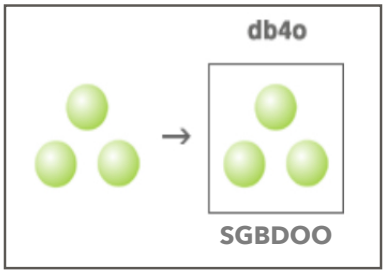
http://www.odbms.org/

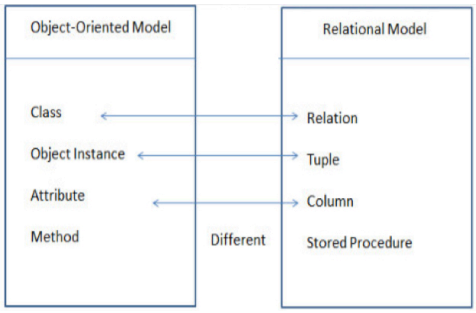


- Padrão para **SGBDOO**
- Consórcio de pesquisadores e fabricantes
- Objetivo
  - integração e padronização de funcionalidades de **BD** a uma **LPOO**
- Componentes do padrão
  - modelo de objeto
  - linguagem de definição de dados (**ODL**)
  - linguagem de consulta (**OQL**)

Modelo Relacional x Orientado a Objetos

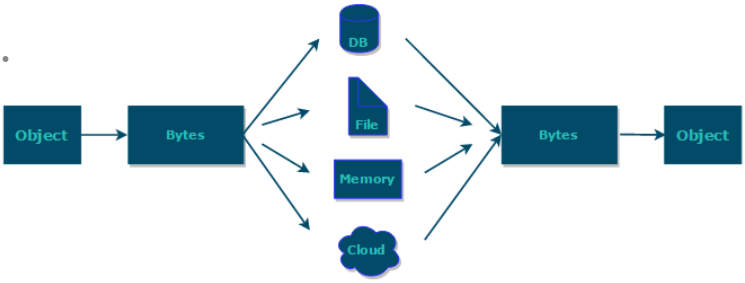






Persistência de Objetos

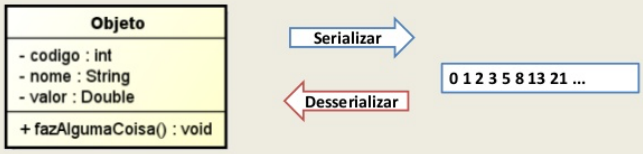
- A capacidade dos objetos de sobreviverem além do tempo de execução de uma aplicação é chamada de **Persistência de Objetos**.
- A persistência precisa armazenar o estado dos objetos em algum repositório para futuramente recuperá-los.
- Os repositórios podem ser:
  - Um **BD relacional** (mais comum).
  - **Arquivos** do sistema.
  - Um **BDOO**.

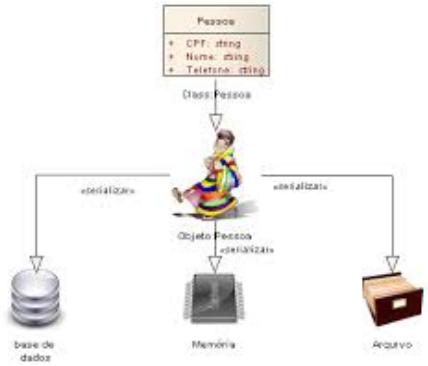


Persistência de Objetos: Arquivos do sistema

O que é Serialização?

É a técnica que permite converter objetos em bytes (**colocando-os em série**) e uma vez que eles são bytes, eles podem ser salvos em disco ou enviados através de um **stream** (via HTTP, via socket, entre outros).

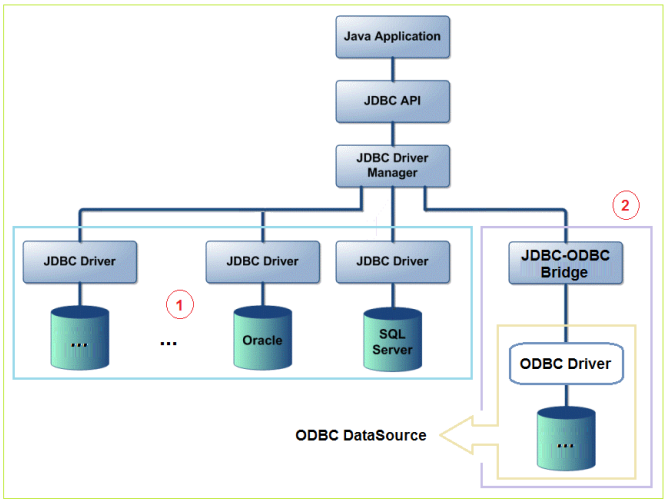




Persistência de Objetos: BD relacional

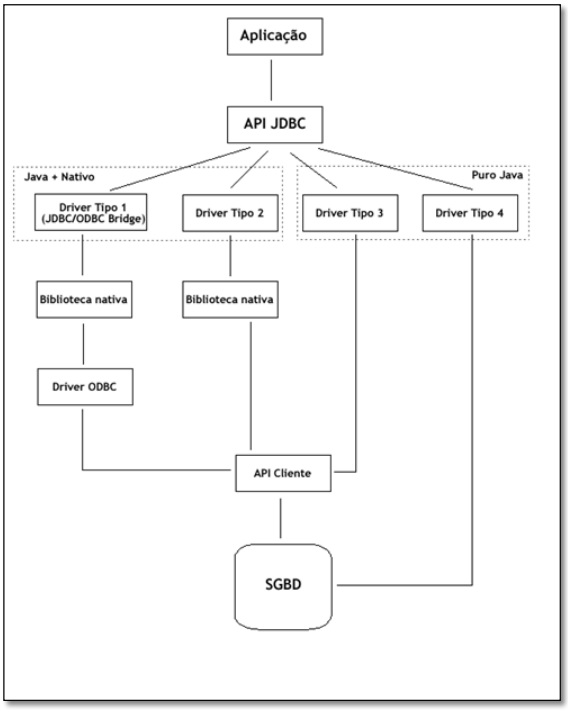
- A partir de um código nativo as aplicações **Java** podiam utilizar qualquer banco de dados que tivesse um driver **ODBC** disponível.

ODBC (acrônimo para **O**pen **D**atab**a**se **C**onnectivity) é um padrão para acesso a sistemas gerenciadores de bancos de dados (SGBD)



- Existe um driver **ODBC** para praticamente qualquer banco de dados de mercado.

Persistência de Objetos: BD relacional



Persistência de Objetos: BD relacional

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionFactory {

    public static Connection createConnection() throws SQLException{
        String url = "jdbc:mysql://localhost:3306/loja"; //Nome da base de dados
        String user = "root"; //nome do usuário do MySQL
        String password = "root"; //senha do MySQL

        Connection conexao = null;
        conexao = DriverManager.getConnection(url, user, password);

        return conexao;
    }
}
```

JDBC

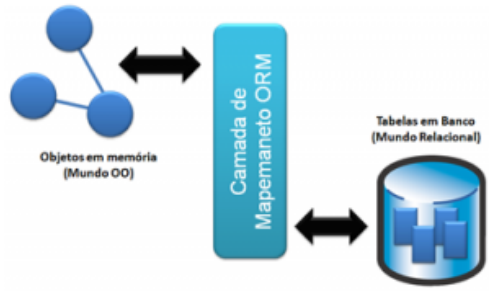
```
conn = ConnectionFactory.createConnection();

String sql = "SELECT codigo, nome, sobrenome, idade, salario FROM funcionario";
ps = conn.prepareStatement(sql);

//Executa o comando de consulta aonde guarda os dados retornados dentro do ResultSet.
//Pelo fato de gerar uma lista de valores, é necessário percorrer os dados através do laço while
ResultSet rs = ps.executeQuery();
//Faz a verificação de enquanto conter registros, percorre e resgata os valores
while(rs.next()){
    //Recupera valor referente ao nome da coluna
    int codigo = rs.getInt("codigo");
    //Recupera o índice do campo referente ao campo nome
    String nome = rs.getString(2);
    String sobreNome = rs.getString("sobrenome");
    String nomeCompleto = nome.concat(" "+sobreNome);
    int idade = rs.getInt("idade");
    Double salario = rs.getDouble("salario");
    System.out.printf("Código %d: %s - %d | Salário: %f \n",codigo, nomeCompleto, idade, salario);
}
```

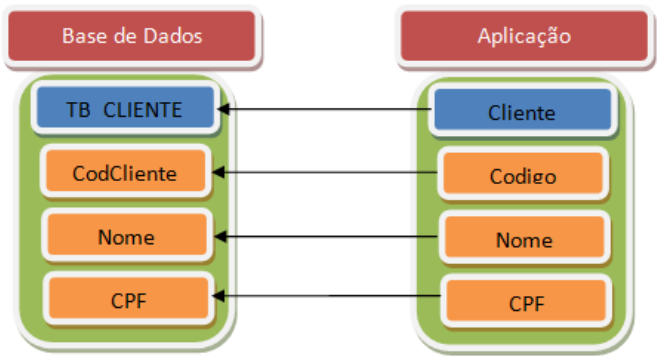
Um BD relacional: Mapeamento Objeto Relacional (ORM)

- O desenvolvimento de aplicações que usam **linguagens OO** e **DB Relacional** enfrentam o problema da **incompatibilidade conceitual** (impedance mismatch).
- Para superar este problema é importante conhecer:
  - O processo de mapeamento objeto-relacional.
  - Como implementar mapeamento objeto-relacional.



MUITOS DESENVOLVEDORES NÃO SE SENTEM A VONTADE EM ESCREVER CÓDIGO SQL E PELA PRODUTIVIDADE QUE ESTA TÉCNICA NOS PROPORCIONA. EXISTEM ÓTIMOS ORM´S COMO HIBERNATE, NHIBERNATE, ENTITY FRAMEWORK E ETC.

Mapeamento Objeto Relacional (ORM): uma classe para uma tabela

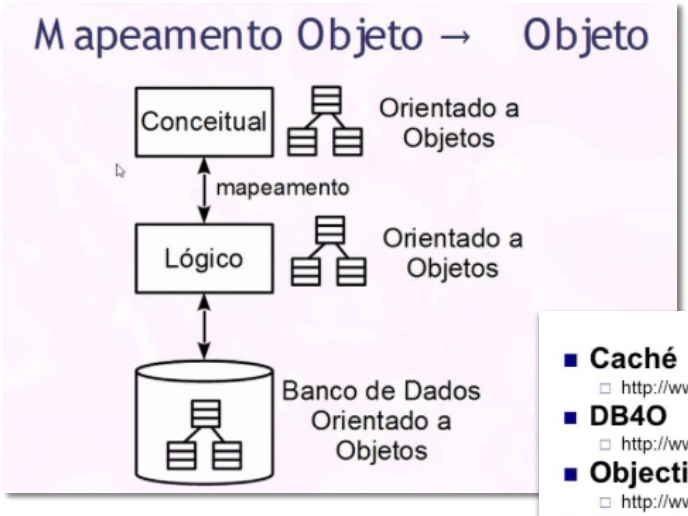



```
01    using System;
02    using System.Collections.Generic;
03    using System.Linq;
04    using System.Text;
05    using NHibernate.Mapping.Attributes;
06
07    namespace DevMedia.Exemplo
08    {
09            [Class(Table = "TB_CLIENTE")]
10            public class Cliente
11            {
12             [Id(Column = "CodCLiente")]
13             public Double Codigo { get; set; }
14
15             [Property]
16             public Double Nome { get; set; }
17
18             [Property]
19             public Double CPF { get; set; }
20            }
21    }
```

Exemplo de mapeamento com anotações com NHibernate e C#

Uma **tabela cliente** na base de dados e uma **classe cliente** em nossa aplicação com as setas indicando o mapeamento da classe da aplicação para a tabela da base de dados.

SGBDOO



- **Caché**
  - http://www.intersystems.com/
- **DB4O**
  - http://www.db4o.com/ (código aberto)
- **Objectivity/DB**
  - http://www.objectivity.com/
- **Object Store**
  - http://www.progress.com/objectstore/index.ssp
- **Ozone**
  - http://www.ozone-db.org./frames/home/what.html (código aberto)
- **Versant Object Database**
  - http://www.versant.com/

BDOO e db4o

- Em 1995, Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, e Stanley Zdonik publicaram um artigo com o título ***The Object-Oriented Database System Manifesto***, definindo como deveria ser um **SGBDOO** e descrevem as principais características em 3 grupos.
- No artigo descrevem as 5 principais características:
  - Persistência de objetos
  - Armazenamento: como deveria gerenciar grandes bd
  - Concorrência: usuários usuários concorrentes
  - Recuperação de falhas de hardware e software
  - Consultas: facilidade na consulta de dados.

BDOO e db4o

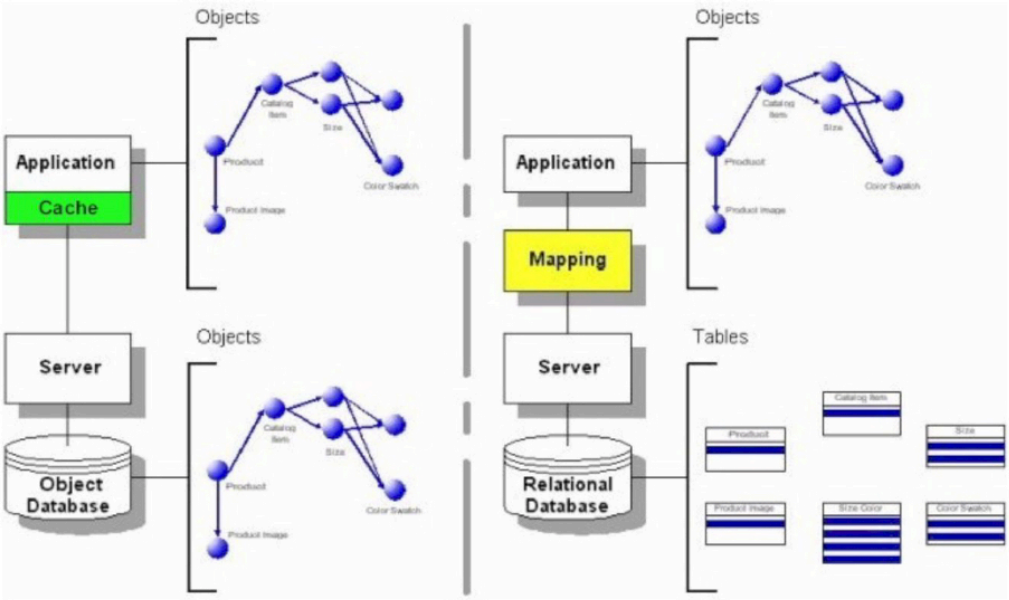
- Algumas funcionalidades disponíveis no **db4o** são:
  - Sessões – Início e fim.
  - Arquivos de base de dados – Criar, Abrir, Fechar e Apagar.
  - Transações – Commit e Rollback
  - Objetos – Armazenar, recuperar, alterar (inclusive cascata), replicar, apagar (inclusive cascata).

BDOO e db4o

- O desenvolvimento do **SGBDOO db4o** iniciou em 2000 pela empresa Actian, finalizando-o em 2001 e comercializado em 2004.
- db4o é um bd não-relacional focado especificamente em persistência de objetos. Realiza persistências e consultas de forma nativa para plataformas **Java** e **.NET**. Portanto, não há nenhuma API como uma camada de software para realizar o mapeamento: os objetos são persistidos diretamente como objetos.

"Using tables to store objects is like driving your car home and then disassembling it to put it in the garage. It can be assembled again in the morning, but one eventually asks whether this is the most efficient way to park a car."

BDOO e db4o



db4o: CRUD

```
ObjectContainer container = Db4oEmbedded.openFile("databaseFile.db4o");
try {
    // store a new pilot
    Pilot pilot = new Pilot("Joe");
    container.store(pilot);

    // query for pilots
    List<Pilot> pilots = container.query(new Predicate<Pilot>() {
        @Override
        public   boolean match(Pilot pilot) {
            return pilot.getName().startsWith("Jo");
        }
    });

    // update pilot
    Pilot toUpdate = pilots.get(0);
    toUpdate.setName("New Name");
    container.store(toUpdate);

    // delete pilot
    container.delete(toUpdate);
} finally {
    container.close();
}
```

C

R

U

D

db4o: consultas

- As consultas são bem diferentes que em **SQL**. Por exemplo, se você deseja obter os dados de um gerente e de todos os seus subordinados associados, você precisa realizar um **JOIN** entre as duas tabelas em **SQL**. Em um **bdoo** você precisa somente realizar uma consulta a um objeto e retornar todos os objetos associados.
- **db4o** disponibiliza 3 opções de consultas: Query By Example (**QBE**), Native Queries (**NQ**) e **SODA** Query API.
- Ao contrário da maioria dos **DBMS**, **db4o** não foi concebido como um servidor mas sim como uma biblioteca.

db4o: desenvolvimento de projetos

- Indra Sistemas (Espanha): sistema de controle de trens de alta velocidade (processamento de mais de 200.000 objetos heterogêneos por segundo)
- Bosch Packaging Technology Group (The Netherlands): complexo sistema de controle de empacotamento por robôs
- Clarity Medical Systems' Retcam II (UK): sistema de diagnóstico oftalmológico de recém-nascidos
- Boeing (U.S. Navy): selecionado para o projeto P-8A Multi-Mission Maritime Aircraft
- E vários outros projetos pelo mundo.

db4o: criando uma classe de dados (POJO)

```
package db4o;
import com.db4o.Db4o;
import com.db4o.ObjectContainer;
public class Person {

    private String _name;
    private int _age;
    public Person(){}

    public Person(String name, int age) {
      _name = name;
      _age = age;
    }

    public int getAge(){
        return _age;
    }

    public void setAge(int value){
        _age = value;
    }

    public String getName(){
        return _name;
    }

    public void setName(String value){
        _name = value;
    }

    public String toString(){
        return "[" + _name + ";"+ _age + "]";
    }
}
```


db4o: acessando um banco de dados

- Para acessar o banco de dados ou criar um novo, pasta invocar o método **Db4o.openFile()** e fornecer como parâmetro o caminho do bd para obter uma instância **ObjectContainer**. **ObjectContainer** representa simplesmente o banco de dados.

```
ObjectContainer mydb = null;
mydb = Db4o.openFile("C:/myDB/myDB.txt");
```

O arquivo de banco de dados **db4o** é por convenção dado pela extensão **.yap**. Esta extensão significa "**y**et **a**nother **p**rotocol" e obviamente não possui nenhum significado, mas é uma extensão não utilizada por nenhuma outra aplicação, até onde se saiba. Isto auxilia na diferenciação de banco de dados db4o em seu sistema de arquivos. Se você desejar, você pode, entretanto, dar qualquer extensão ao seu bd, como, por exemplo, no trecho de código acima, ou até mesmo **.db4o**.

db4o: persistindo um objeto

- Para persistir um objeto, o método **store()** deverá ser invocado. Mas primeiro um objeto deverá ser instanciado para que seja passado como parâmetro para o método de persistência.

```
Person pinar = new Person("Pinar",24);
Person anneMarie = new Person("AnneMarie", 23);
mydb.store(pinar);
mydb.store(anneMarie);
```


db4o: listando o banco de dados

```
public static void listResult(ObjectSet result) {
    System.out.println(result.size());
    while (result.hasNext()) {
        System.out.println(result.next());
    }
}
```

```
public static void listResultNQ(List<Person> person) {
    while ((ObjectSet)person.hasNext()) {
        System.out.println((ObjectSet)person.next());
    }
}
```

db4o: Query by Example (QBE)

- Quando usando **Query-By-Example**, você deverá criar um protótipo de objeto para **db4o** usar como um exemplo na consulta ao banco de dados. **db4o** retornará todos os objetos do banco de dados que contém os mesmos atributos dados como exemplo no protótipo.
- Os resultados serão retornados como uma instância da classe **ObjectSet**. Usaremos o método **listResult** visto anteriormente para listar o conteúdo do nosso **ObjectSet**.

```
public static void listResult(ObjectSet result) {
    System.out.println(result.size());
    while (result.hasNext()) {
        System.out.println(result.next());
    }
}
```


db4o: SELECT statement with WHERE clause

- **db4o** usando **QBE** para consulta

```
//finds the objects whose name value is "Pinar" and returns it/them.
Person proto = new Person("Pinar", 0);
ObjectSet result = mydb.queryByExample(proto); //Query-by-Example
listResult(result);
```

- Consulta equivalente em **SQL**

```
//Same query in SQL
SELECT *
FROM Person
WHERE name = "Pinar"
```

db4o: atualizando um objeto

- **db4o** usando **QBE** para consulta

```
ObjectSet result = mydb.queryByExample(new Person("AnneMarie",0)); //updating an object
Person found = (Person)result.next();
found.setAge(25);
mydb.store(found);
retrieveAllPersons(mydb);
```

- Consulta equivalente em **SQL**

```
UPDATE Person
SET age =25
WHERE name=”AnneMarie”
```


db4o: excluindo um objeto

- **db4o** usando **QBE** para consulta

```
ObjectSet test = mydb.queryByExample(new Person("Esteban",0)); //deleting an object
Person found2 = (Person)test.next();
mydb.delete(found2);
```

- Consulta equivalente em **SQL**

```
DELETE FROM Person
WHERE name = "Esteban";
```

db4o: exemplos de consultas QBE

- Consulta de piloto **por nome**

```
Pilot theExample = new Pilot();
theExample.setName("John");
final ObjectSet<Pilot> result = container.queryByExample(theExample);
```

- Consulta de piloto **por idade**

```
Pilot theExample = new Pilot();
theExample.setAge(33);
final ObjectSet<Pilot> result = container.queryByExample(theExample);
```

- Consulta de piloto **por nome e idade**

```
Pilot theExample = new Pilot();
theExample.setName("Jo");
theExample.setAge(29);
final ObjectSet<Pilot> result = container.queryByExample(theExample);
```


db4o: exemplos de consultas QBE

- Retornará **todos** os objetos do **tipo Pilot**

```
Pilot example = new Pilot();
final ObjectSet<Pilot> result = container.queryByExample(example);
```

- Ou alternativamente:

```
final ObjectSet<Pilot> result = container.queryByExample(Pilot.class);
```

- Retornará **todos os objetos** do bd

```
final ObjectSet<Pilot> result = container.queryByExample(null);
```

- Retornará **todos os carros** com o **piloto Jenny**

```
Pilot pilotExample = new Pilot();
pilotExample.setName("Jenny");

Car carExample = new Car();
carExample.setPilot(pilotExample);
final ObjectSet<Car> result = container.queryByExample(carExample);
```