

Tecnologias Java

JDBC

Marcio Seiji Oyamada
msoyamada@gmail.com



JDBC

- JDBC: Java Database Connectivity
 - API Java para acessar dados armazenados em um Banco de Dados
- Conectar a um banco dados
- Enviar consultas e atualizações para o banco de dados
- Obter e processar os resultados obtidos a partir de consultas ao banco de dados

Componentes JDBC

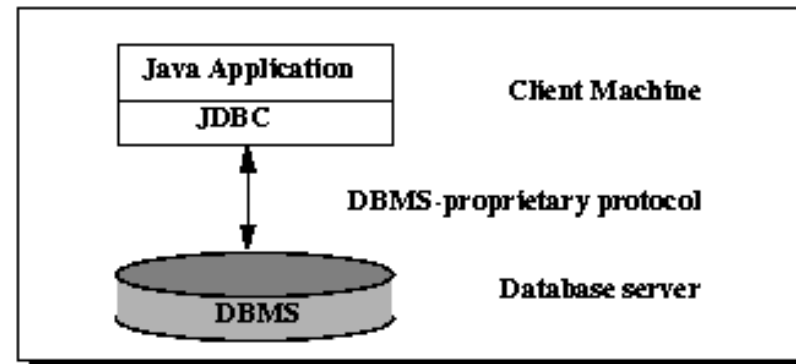
- **API JDBC:** operações sobre o banco de dados
 - java.sql
 - javax.sql
- **JDBC Driver Manager**
 - DriverManager: fornece a conexão de uma aplicação Java com o Banco de dados
 - <http://developers.sun.com/product/jdbc/drivers>
- **JDBC-ODBC Bridge**
 - Fornece uma ponte da conexão JDBC com o *driver* ODBC

Drivers JDBC

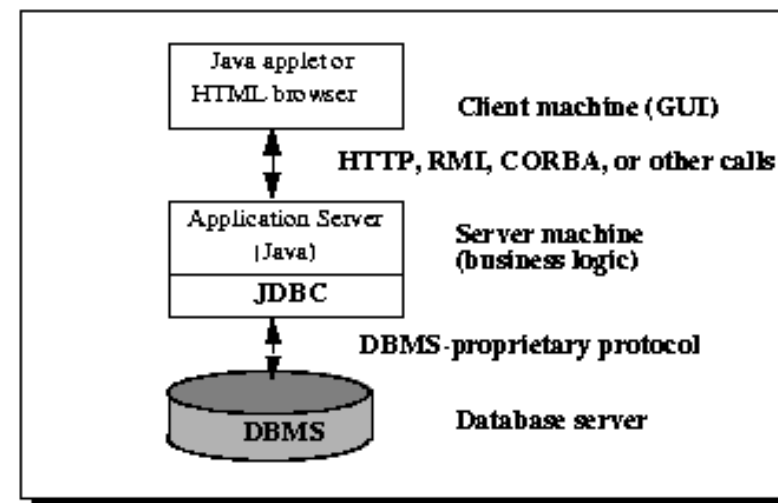
- **Type 1** – drivers que implementam a API JDBC e o mapeamento para uma outra API (ODBC). Normalmente dependente de uma biblioteca nativa. Ex: JDBC-ODBC Bridge
- **Type 2** – drivers parcialmente implementados em Java e com algumas bibliotecas escritas em código nativo. Portabilidade limitada
- **Type 3** – drivers utilizam somente a linguagem Java no cliente e se comunica com o middleware do servidor de base de dados através de um protocolo independente. O middleware então se comunica com a base de dados e responde ao cliente
- **Type 4** – drivers implementados em Java, utilizando um protocolo específico de rede, para acessar diretamente a base de dados

Modelos de integração com Banco de dados

- Duas camadas



- Três camadas



Banco de dados relacional

- Database Management System (DBMS): gerencia a forma de armazenar, recuperar e realizar a manutenção das tabelas
- Tabela: coleção de “objetos” do mesmo tipo.
Ex: Cliente
 - Colunas: Campos da tabela
 - Linha: Um registro em particular
 - Chave primária: diferenciar elementos de uma tabela

Banco de dados relacional (2)

Id	Nome	Endereco	Cidade
1	Joao	R. Brasil	Cascavel
2	Jose	R. Parana	Toledo
3	Maria	Praca XV	Cascavel

SQL: Structured Query Language

- Linguagem de consulta para acesso à banco de dados
- Subconjunto “padrão” implementado por todos os banco de dados
 - SELECT
 - FROM
 - WHERE
 - GROUP BY
 - ORDER BY
 - INNER JOIN
 - INSERT
 - UPDATE
 - DELETE

SELECT

- **SELECT:** consulta, obter informações de uma tabela
 - Quais as tabelas e os campos que deverão ser selecionados
 - Ex: Selecionar todos os dados da tabela Clientes

```
SELECT *
```

```
FROM Clientes
```

- Ex: Selecionar os campos Id e Nome da tabela Clientes

```
SELECT Id, Nome
```

```
FROM Clientes
```

WHERE

- **WHERE:** fornece um critério de seleção no comando **SELECT**

- **Selecionar:** especificando o campo

```
SELECT *  
FROM Clientes  
WHERE Id= 1
```

- **Selecionar:** especificando um “range”

```
SELECT *  
FROM Clientes  
WHERE Id > 1
```

```
SELECT *  
FROM Clientes  
WHERE Id > 1 and Id < 3
```

- **Selecionar utilizando máscaras**

```
SELECT *  
FROM Clientes  
WHERE Nome LIKE 'J%'
```

ORDER BY

- ORDER BY:
 - ASC
 - DESC

```
SELECT *  
  FROM Clientes  
  ORDER BY Nome ASC
```

```
SELECT *  
  FROM Clientes  
  ORDER BY Nome DESC
```

INSERT

```
INSERT INTO Clientes (Nome, Endereco, Estado)  
VALUES ('Ana', 'R. das Camélias', 'Cascavel')
```

- O campo Id é um campo incrementado automaticamente (auto-increment)

DELETE

```
DELETE FROM Clientes
```

```
WHERE Id=1
```

```
DELETE FROM Clientes
```

```
WHERE Nome='Joao' and Cidade='Cascavel'
```

- Pode ser definido um conjunto de valores

UPDATE

- UPDATE Clientes
SET Cidade='Pato Branco'
WHERE Id= 1

JOIN

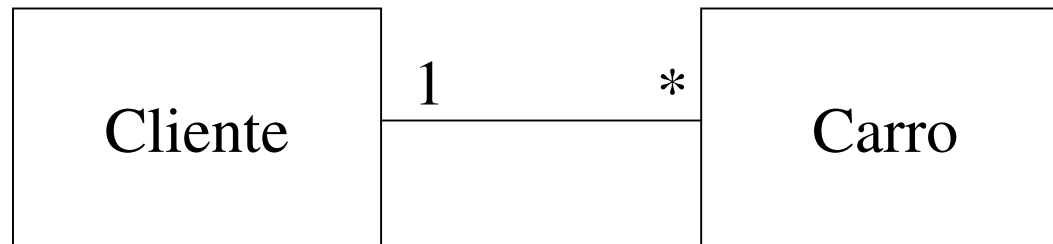
```
SELECT Cliente.ClienteID, Nome, Endereco, Modelo  
FROM Cliente  
INNER JOIN Carro  
    ON Cliente.ClienteID= Carro.ClienteID  
ORDER BY Nome ASC
```

Outros comandos SQL

- **CREATE TABLE** — cria uma tabela com as colunas e seus respectivos tipos de dados fornecidos pelo usuário. Os tipos de dados, variam conforme o SGBD, portanto é necessário utilizar os metadados para estabelecer quais os tipos de dados suportados pelo SGBD.
- **DROP TABLE** — apaga todas as linhas de uma tabela e remove a definição do banco de dados.
- **ALTER TABLE** — adiciona ou remove colunas de uma tabela.

Utilizando o Netbeans

- Criar um banco de dados
- Inicializar o servidor de banco de dados
- Criar as tabelas
 - Executar comando



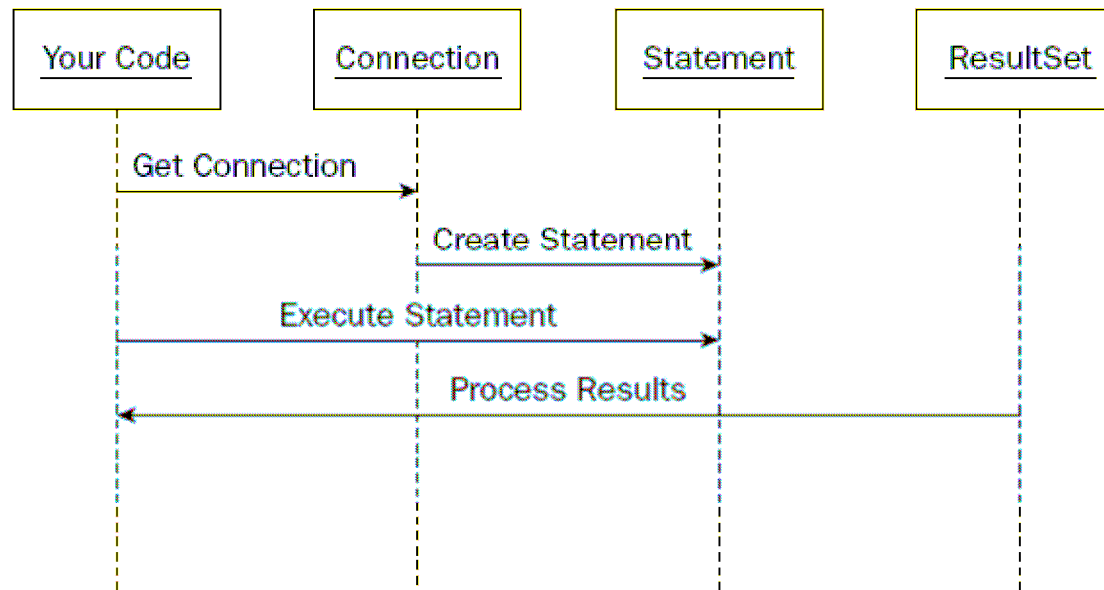
Criando as tabelas

```
create table Cliente (  
  ClienteID INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,  
  Nome varchar(40),  
  Endereco varchar(40),  
  Cidade varchar(40),  
  Estado varchar(2));
```

```
create table Carro (  
  CarroID INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,  
  Marca varchar(40),  
  Modelo varchar(40),  
  Ano varchar(4),  
  Placa varchar(8),  
  ClienteID INT NOT NULL,  
  FOREIGN KEY (ClienteID) REFERENCES Cliente(ClienteID));
```

Conectando-se a um banco de dados via JDBC

- Carregar o driver JDBC adequado
- Criar uma conexão
- Enviar consultas SQL



Carregando o driver

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
```

- Cria automaticamente uma instância do driver e registra no DriverManager
- Lembre-se de colocar a biblioteca derbyclient.jar no seu projeto
 - Netbeans: propriedade do projeto -> bibliotecas
 - Derbyclient.jar -> c:\Program Files\java\jdk1.6\db\lib

Criar a conexão

- URL
 - Protocolo: JDBC
 - Subprotocolo: depende da base de dados /Localização da base de dados

MySQL	<code>com.mysql.jdbc.Driver</code>	<code>jdbc:mysql://nomeDoHost/ nomeDoBancoDeDados</code>
ORACLE	<code>oracle.jdbc.driver.OracleDriver</code>	<code>jdbc:oracle:thin:@nomeDoHost: numeroDaPorta:nomeDoBancoDeDados</code>
DB2	<code>COM.ibm.db2.jdbc.net.DB2Driver</code>	<code>jdbc:db2:nomeDoHost:numeroDaPorta/ nomeDoBancoDeDados</code>
Sybase	<code>com.sybase.jdbc.SybDriver</code>	<code>jdbc:sybase:Tds:nomeDoHost: numeroDaPorta/nomeDoBancoDeDados</code>

- Exception: SQLException
 - `String url = "jdbc:derby://localhost:1527/Clientes";`
 - `String user= "John";`
 - `String pass= "FreeAccess";`
 - `Connection con = DriverManager.getConnection(url, user, pass);`

Enviar consultas SQL

- Statement
 - Statement
 - PreparedStatement: parâmetros IN
 - CallableStatement: parâmetros IN e OUT (stored procedures)

Statement

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
Properties props= new Properties();
props.put("user", User);
props.put("password", Pass);
String dbUrl= "jdbc:derby://localhost:1527/Clientes";
try {
    dbConnection = DriverManager.getConnection(dbUrl, props);
    Statement sStatement = dbConnection.createStatement();
    ResultSet rsResults = sStatement.executeQuery("SELECT * FROM
Clientes");
    while (rsResults.next()) {
        // Opera sobre o resultado da consulta
        System.out.println("Nome: "+results.getString("Nome"));
    }
}catch (Exception e) {
    e.printStackTrace();
}
```

PreparedStatement

```
PreparedStatement stmtGetCliente;  
ResultSet results;  
stmtGetCliente =  
    dbConnection.prepareStatement("SELECT * FROM Cliente  
    WHERE ClienteID=?");  
stmtGetCliente.clearParameters();  
stmtGetCliente.setInt(1);  
results= stmtGetCliente.executeQuery();  
  
if (results.next()) {  
    System.out.println("Nome "+results.getString("Nome"));  
}
```

Result Sets and Cursors

- As linhas que satisfazem as condições de uma consulta são chamadas de “result set”.
- O número de linhas pode ser maior ou igual a zero.
- O resultado pode ser acessado linha por linha, e o *cursor* fornece a posição lógica dentro do “result set”.
 - O cursor permite ao usuário processar cada linha do “result set”, do começo ao fim, e portanto pode ser utilizado para processar todo o resultado da consulta
 - As versões mais novas do JDBC permite que cursor seja movimentado para frente ou para trás, ou para uma linha específica

ResultSet

- **ResultSet**: quando criado, o *cursor* é posicionado na linha anterior ao primeiro resultado.
- Métodos para movimentação do *cursor*
 - `next()` – move para a próxima linha. Retorna *true* se conseguiu movimentar, ou *false* se o cursor está na ultima linha.
 - `previous()` – move para linha anterior. Retorna *false* se estava na primeira linha
 - `first()` – move para a primeira linha. Retorna *false* se o objeto ResultSet não contém nenhuma linha.
 - `last()` – move para a ultima linha. Retorna *false* se o objeto ResultSet não contém nenhuma linha
 - `beforeFirst()` – posiciona o cursor no início do ResultSet, antes da primeira linha.
 - `afterLast()` – posiciona o cursor depois no final do ResultSet, depois da última linha.
 - `relative(int rows)` – move o cursor em n linhas, relativas a posição atual
 - `absolute(int row)` – move o cursor para a n linha do ResultSet

Mapeamento SQL JDBC

Java Object/Type	JDBC Type
Int	INTEGER
Short	SMALLINT
Byte	TINYINT
Long	BIGINT
Float	REAL
Double	DOUBLE
java.math.BigDecimal	NUMERIC
Boolean	BOOLEAN or BIT

Java Object/Type	JDBC Type
String	CHAR, VARCHAR, or LONGVARCHAR
Clob	CLOB
Blob	BLOB
Struct	STRUCT
Ref	REF
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.net.URL	DATALINK
Array	ARRAY
byte[]	BINARY, VARBINARY, or LONGVARBINARY
Java class	JAVA_OBJECT

Padrão DAO

- Data Access Object
 - Separar o acesso aos dados da lógica de negócio
 - Pode ser utilizado para acessar diferentes tipos de armazenamento (hibernate, jdbc, etc..)

```
public interface UserDao {  
    public void save (User user);  
    public void delete (User user);  
    public List list ();  
    public User find (String name);  
}
```

Exemplo: ClienteDAO

```
class ClienteDAO {  
    ClienteDAO();  
    ClienteDAO(String DatabaseURL, String User, String  
    Pass);  
    boolean connect();  
    void disconnect();  
    boolean deleteRecord(Cliente record);  
    boolean deleteRecord(int id);  
    int saveRecord(Cliente record);  
    boolean editRecord(Cliente record);  
    Cliente getRecord(int index);  
    List<Cliente> getListEntries();  
    Vector<Carro> getCarros();  
    // atributos  
    // definição das consultas  
}
```

ClienteDAO: método Connect

```
public boolean connect() {  
    Class.forName("org.apache.derby.jdbc.ClientDriver");  
    Properties props= new Properties();  
    props.put("user", User);  
    props.put("password", Pass);  
    String dbUrl= DatabaseURL;  
    try {  
        dbConnection = DriverManager.getConnection(dbUrl, props);  
        stmtSaveNewRecord =  
            dbConnection.prepareStatement(strSaveCliente,  
                Statement.RETURN_GENERATED_KEYS);  
        stmtUpdateExistingRecord =  
            dbConnection.prepareStatement(strUpdateCliente);  
        stmtGetCliente =  
            dbConnection.prepareStatement(strGetCliente);  
        stmtDeleteCliente =  
            dbConnection.prepareStatement(strDeleteCliente);  
  
        isConnected = dbConnection != null;  
    } catch (SQLException ex) {  
        isConnected = false;  
    }  
  
    return isConnected;  
}
```

ClienteDAO: método deleteRecord

```
public boolean deleteRecord(int id) {
    boolean bDeleted = false;
    try {
        stmtDeleteCliente.clearParameters();
        stmtDeleteCliente.setInt(1, id);
        stmtDeleteCliente.executeUpdate();
        bDeleted = true;
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }

    return bDeleted;
}
```

```
public boolean deleteRecord(Cliente record) {
    int id = record.getClienteID();
    return deleteRecord(id);
}
```

ClienteDAO: método SaveRecord

```
public int saveRecord(Cliente record) {
    int id = -1;
    try {
        stmtSaveNewRecord.clearParameters();

        stmtSaveNewRecord.setString(1, record.getNome());
        stmtSaveNewRecord.setString(2, record.getEndereco());
        stmtSaveNewRecord.setString(3, record.getCidade());
        stmtSaveNewRecord.setString(4, record.getEstado());

        int rowCount = stmtSaveNewRecord.executeUpdate();
        ResultSet results =
stmtSaveNewRecord.getGeneratedKeys();
        if (results.next()) {
            id = results.getInt(1);
        }

    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
    return id;
}
```

ClienteDAO: método editRecord

```
public boolean editRecord(Cliente record) {
    boolean bEdited = false;
    try {
        stmtUpdateExistingRecord.clearParameters();
        stmtUpdateExistingRecord.setString(1, record.getNome());
        stmtUpdateExistingRecord.setString(2, record.getEndereco());
        stmtUpdateExistingRecord.setString(3, record.getCidade());
        stmtUpdateExistingRecord.setString(4, record.getEstado());
        stmtUpdateExistingRecord.setInt(5, record.getClienteID());
        stmtUpdateExistingRecord.executeUpdate();
        bEdited = true;
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
    return bEdited;
}
```

ClienteDAO: método getRecord

```
public Cliente getRecord(int index) {
    Cliente cliente = null;
    try {
        stmtGetCliente.clearParameters();
        stmtGetCliente.setInt(1, index);
        ResultSet result = stmtGetCliente.executeQuery();
        if (result.next()) {
            cliente= new Cliente();
            cliente.setClienteID(result.getInt("ClienteID"));
            cliente.setNome(result.getString("Nome"));
            cliente.setEndereco(result.getString("Endereco"));
            cliente.setCidade(result.getString("Cidade"));
            cliente.setEstado(result.getString("Estado"));
            cliente.setCarros(getCarros(index));
        }
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }

    return cliente;
}
```

ClienteDAO: método getRecord

```
public Vector<Carro> getCarros(int ClienteID){
    Vector<Carro> vectorCarro = new Vector<Carro>();
    Carro carro;
    try {
        stmtGetCarros.clearParameters();
        stmtGetCarros.setInt(1, ClienteID);
        ResultSet result = stmtGetCarros.executeQuery();

        while (result.next()) {
            carro = new Carro();
            carro.setCarroId(result.getInt("CarroID"));
            carro.setMarca(result.getString("Marca"));
            carro.setModelo(result.getString("Modelo"));
            carro.setAno(result.getString("Ano"));
            carro.setPlaca(result.getString("Placa"));
            vectorCarro.add(carro);
        }
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
    return vectorCarro;
}
```

ClienteDAO: método getListEntries

```
public List<Cliente> getListEntries() {
    List<Cliente> listEntries = new ArrayList<Cliente>();
    Statement queryStatement = null;
    ResultSet results = null;
    try {
        queryStatement = dbConnection.createStatement();
        results = queryStatement.executeQuery(strGetListEntries);
        while(results.next()) {
            Cliente c= new Cliente();
            c.setClienteID(results.getInt(1));
            c.setNome(results.getString(2));
            c.setEndereco(results.getString(3));
            c.setCidade(results.getString(4));
            c.setEstado(results.getString(5));
            c.setCarros(getCarros(results.getInt(1)));
            listEntries.add(c);
        }
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
    return listEntries;
}
```

ClienteDAO: método disconnect

```
public void disconnect() {
    if(isConnected) {
        String dbUrl = getDatabaseUrl();
        dbProperties.put("shutdown", "true");
        try {
            DriverManager.getConnection(dbUrl, dbProperties);
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
        isConnected = false;
    }
}
```

Atributos da classe ClienteDAO

```
private Connection dbConnection;  
private boolean isConnected;  
private String DatabaseURL, User, Pass;  
private PreparedStatement stmtSaveNewRecord;  
private PreparedStatement stmtUpdateExistingRecord;  
private PreparedStatement stmtGetListEntries;  
private PreparedStatement stmtGetCliente;  
private PreparedStatement stmtDeleteCliente;
```

Definição das consultas (1)

```
private static final String strCreateClienteTable =
    "create table Cliente(" +
    "    ClienteID    INTEGER NOT NULL PRIMARY KEY
GENERATED ALWAYS AS IDENTITY," +
    "    Nome        VARCHAR(40), " +
    "    Endereco    VARCHAR(40), " +
    "    Cidade      VARCHAR(40), " +
    "    Estado      VARCHAR(2))";

private static final String strGetCliente =
    "SELECT * FROM Cliente " +
    "WHERE ClienteID = ?";

private static final String strSaveCliente =
    "INSERT INTO Cliente" +
    "    (Nome, Endereco, Cidade, Estado) " +
    "    VALUES (?, ?, ?, ?)";
```

Definição das consultas (2)

```
private static final String strGetListEntries =  
    "SELECT ClienteID, Nome, Endereco, Cidade, Estado FROM  
    Cliente" +  
    "ORDER BY LASTNAME ASC";
```

```
private static final String strUpdateCliente =  
    "UPDATE Cliente " +  
    "SET Nome= ?, " +  
    "    Endereco = ?, " +  
    "    Cidade= ?, " +  
    "    Estado= ?"+  
    "WHERE ClienteID = ?";
```

```
private static final String strDeleteCliente =  
    "DELETE FROM Cliente " +  
    "WHERE ClienteID = ?";
```

```
private static final String strGetCarros =  
    "SELECT * FROM Carro "+  
    "WHERE ClienteID= ?";
```

Exercício

- Implemente o ClienteDAO
- Implemente o CarroDAO
- Integrar com a interface gráfica desenvolvida na aula passada

Transações

- Acesso concorrente a base de dados pode ocasionar inconsistências
- Transações
 - Conjunto de um ou mais comandos SQL que formam um unidade lógica de trabalho
 - Ao final de uma transação
 - Commit: torna permanente todas as alterações nos dados
 - Rollback: retorna ao estado anterior ao início da transação
- Lock (bloqueio): proíbe que duas transações manipulem o mesmo dado ao mesmo tempo
 - Em alguns SGBDs uma tabela toda é bloqueada, ou apenas uma única linha.

Exemplo: Transações

```
public void Teste() throws SQLException {
    Statement queryStatement = null;
    ResultSet results = null;
    Savepoint sv;
    try {
        dbConnection.setAutoCommit(false);
        sv= dbConnection.setSavepoint("Change");
        queryStatement =
        dbConnection.createStatement(ResultSet.TYPE_FORWARD_ONLY,
        ResultSet.CONCUR_UPDATABLE);
        results = queryStatement.executeQuery("SELECT * FROM
CLIENTE FOR UPDATE of Cidade");
        results.next(); //primeiro registro
        results.updateString("Cidade", "Pato Branco");
        results.updateRow();
        dbConnection.rollback(sv);

    }catch (Exception ex){
        ex.printStackTrace();
    }
}
```

* Note que as linhas resultantes da consultas, ficarão bloqueadas até a finalização da transação (commit ou rollback)

DATABASE METADATA

- Informações sobre a base de dados
 - Nomes de colunas, chaves primárias, tipos, etc.
 - Informações sobre o *driver*
 - Funcionalidades implementadas
- JDBC fornece uma interface
DatabaseMetaData

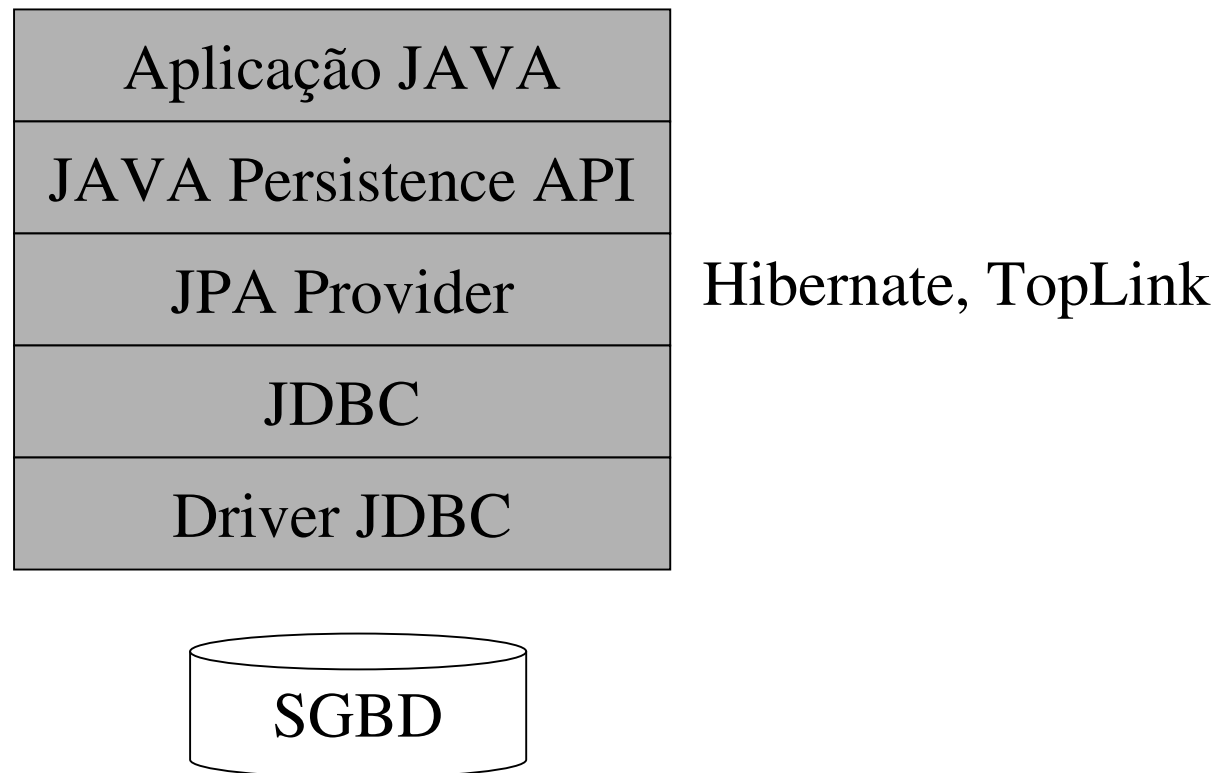
```
DatabaseMetaData metadb;  
metadb= dbConnection.getMetaData();  
metadb.supportsSavepoints();
```

Material Complementar

JPA - Java Persistence API

JPA

- EJB 3.0 e JPA
 - JPA: solução completa para o mapeamento objeto-relacional



Mapeamento OR

- Realizado através de anotações em classes Java
 - Sem necessidade de descritores XML
- POJO (Plain old java object)
 - Sem necessidade de herança ou implementação de interfaces

POJO

- Requisitos
 - Anotado com @Entity
 - Contém ao menos um campo @Id
 - Construtor sem argumentos
 - Não pode ser uma classe final
 - Não pode ser inner class, interface, ou enum
 - Deve implementar a interface Serializable

Anotações

- @Entity: descreve que a classe é mapeada para um banco de dados relacional
- @Table: o nome da tabela no qual a classe está mapeada
- @Column: a coluna no qual o atributo está mapeado
- Relacionamento
 - @ManytoOne
 - @ManytoMany

Exemplo:Cliente

@Entity

@Table(name = "CLIENTE")

```
public class Cliente implements Serializable {
```

```
    @Id
```

```
    @Column(name = "CLIENTEID", nullable = false)
```

```
    @GeneratedValue(strategy=GenerationType.SEQUENCE)
```

```
    private Integer clienteid;
```

```
    @Column(name = "NOME")
```

```
    private String nome;
```

```
    @Column(name = "ENDERECO")
```

```
    private String endereco;
```

```
    @Column(name = "CIDADE")
```

```
    private String cidade;
```

```
    @Column(name = "ESTADO")
```

```
    private String estado;
```

```
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "clienteid")
```

```
    private Collection<Carro> carroCollection;
```

Persistência

- EntityManager
 - Responsável pela comunicação com as camadas inferiores
 - Obtido através de EntityManagerFactory
- EntityManagerFactory
 - Criado a partir de um PersistenceUnit (veja o arquivo persistence.xml)

```
EntityManagerFactory emf=  
Persistence.createEntityManagerFactory("PU");  
EntityManager em= emf.createEntityManager();
```

Inserindo clientes

```
EntityManagerFactory
    emf=Persistence.createEntityManagerFactory("PU");
EntityManager em= emf.createEntityManager();

Cliente c= new Cliente();
c.setNome("Joao");
c.setEndereco("R. Brasil");
c.setCidade("Cascavel");
c.setEstado("PR");

em.getTransaction().begin();
em.persist(c);
em.getTransaction().commit();
```

Obtendo clientes

```
Cliente c= new Cliente();  
em.getTransaction().begin();  
c= em.find(c.getClass(), new Integer(2));  
em.getTransaction().commit();
```

Removendo Cliente

```
Cliente c= new Cliente();  
em.getTransaction().begin();  
c= em.find(c.getClass(), new Integer(2));  
if (c!= null)  
    em.remove(c);  
em.getTransaction().commit();
```

Alterando Cliente

```
Cliente c= new Cliente();  
em.getTransaction().begin();  
c= em.find(c.getClass(), new Integer(2));  
if (c!= null){  
    c.setCidade("Toledo");  
    em.merge(c);  
}  
em.getTransaction().commit();
```

Utilizando o Netbeans

- Novo
 - Categoria Persistência
 - Classe de Entidade de Banco de dados
- Escolha o banco de dados e as tabelas, para que os objetos sejam criados
- Não esqueça de criar também o “Persistence Unit”

Derby

ij

- Connect 'jdbc:derby:teste;create=true';
- Create table
- Insert...
- Exit;
- Verifique que foi criado um directorio chamado teste
- Executando scripts SQL
- Run 'sql.run'

Inicializando servidor Derby

- Inicializar no diretorio onde a base de dados esta armazenada
- `setNetworkServerCP` (Inicializando corretamente o classpath)
- `startNetworkServer`
- `NetworkServerControl start -p 1368`