

# Conexão Java e BDs

Prof. Dr. José de Jesús Pérez Alcázar  
EACH - USP

# Agenda

---

- ▶ Introdução
- ▶ JDBC
- ▶ DAO
- ▶ Mapeamento Objeto-Relacional
- ▶ Hibernate



# Programação por meio de funções: SQL/CLI e JDBC

---

- ▶ SQL Embutida é referida como uma abordagem estática de programação de Bds.
- ▶ O uso de chamados de função é mais dinâmico – outra abordagem.
- ▶ Uma biblioteca de funções , também conhecida como uma interface para a programação de aplicações (API).
- ▶ Embora, proporcione maior flexibilidade, uma vez que não é necessário nenhum pré-processador, tem desvantagens:
  - ▶ Verificação de sintaxe dos comandos SQL precisa ser feita em tempo de execução.
  - ▶ É necessária uma programação mais complexa para acessar o resultado da consulta, porque pode ser que o resultado da consulta não seja conhecido antecipadamente.



# Programação por meio de funções: SQL/CLI e JDBC

---

- ▶ Duas interfaces de chamada de função:
  - ▶ SQL/CLI (Call Level Interface), parte do padrão SQL. Continuação de ODBC (Open Data Base Connectivity). Exemplos com C como hospedeira.
  - ▶ JDBC, interface para JAVA. Parte integral de Java desde a versão 1.1.
- ▶ A principal vantagem do uso de funções é a maior facilidade de acesso a diversos Bds dentro de um mesmo programa, até mesmo se eles estiverem armazenados em SGBDs diferentes.

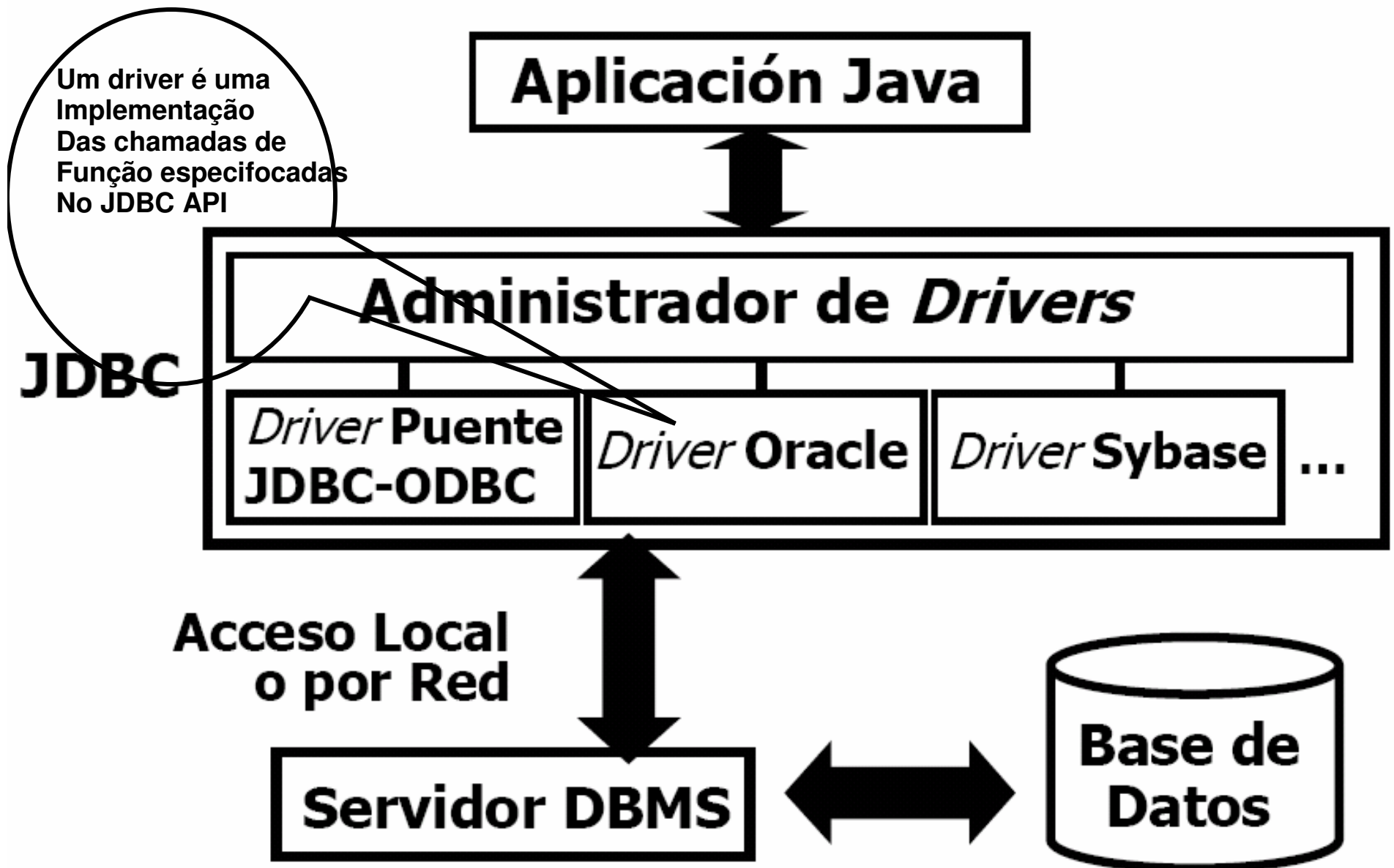


# **JDBC: Chamadas de Função SQL em Programação JAVA**

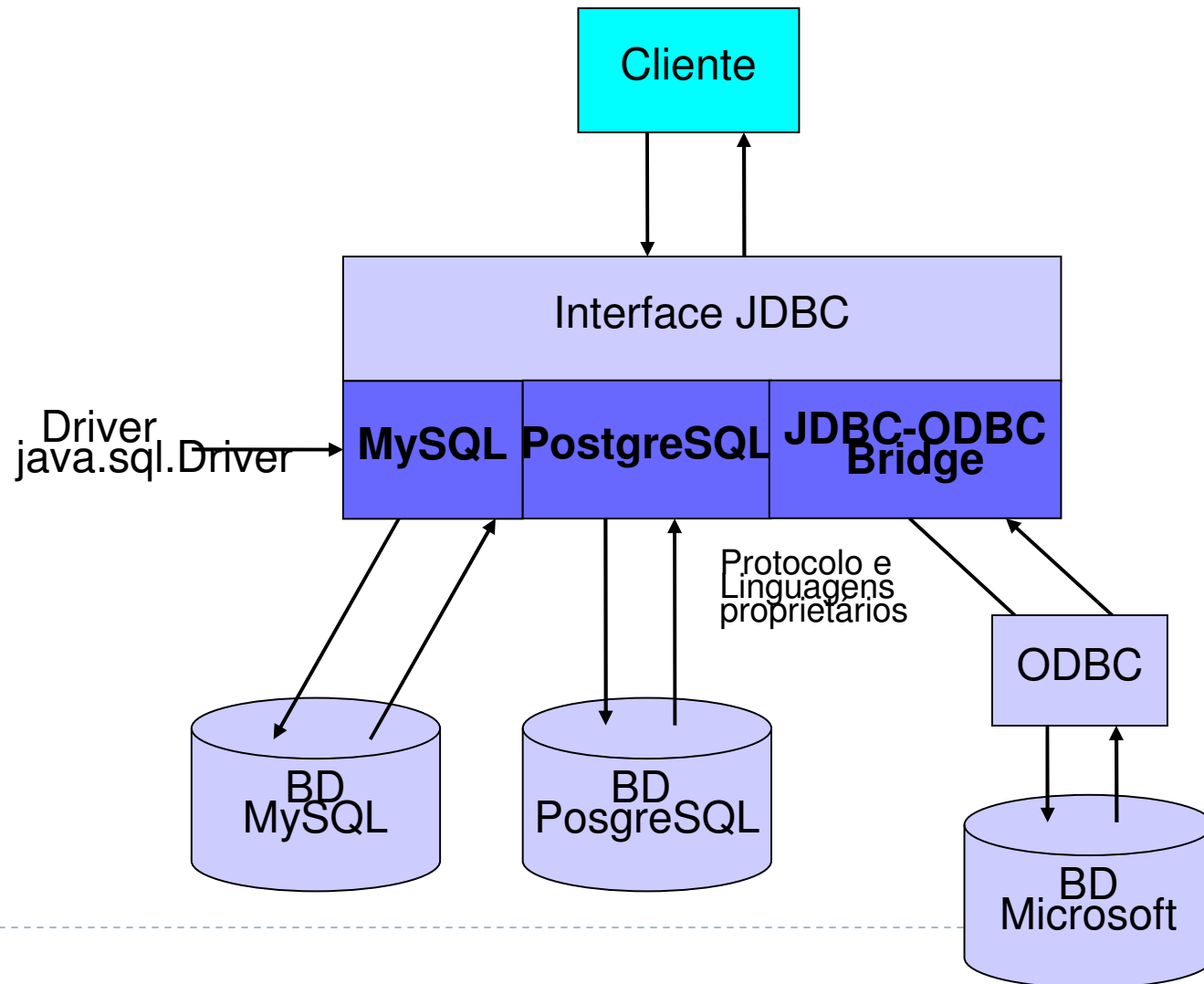
---

- ▶ JDBC, é um API de Java para executar enunciados SQL.
- ▶ Consiste de um conjunto de classes e interfaces escritas para se ter acesso a dados contidos em BDs relacionais
- ▶ É parte integral de Java desde a versão 1.1





# Conexão em Java



# JDBC

---

- ▶ Antes de poder processar as chamadas de função JDBC de JAVA, é necessário importar as bibliotecas de classes JDBC, que são chamadas `java.sql.*`. Podem ser instaladas e carregadas pela web.
- ▶ Foi projetada para permitir que um programa JAVA pudesse conectar-se a vários BDs diferentes (“fontes de dados”)
- ▶ Portanto, são necessários drivers de diferentes fabricantes → papel do gerente de drivers. Este registra o driver antes de ser usado. Operações ou métodos: `getDriver`, `registerDriver` e `deregisterDriver`.





# JDBC

---

- ▶ Para carregar um driver JDBC, uso de uma função JAVA genérica.
    - ▶ `Class.forName("oracle.jdbc.driver.OracleDriver")`
  - ▶ Os passos típicos são (veja figura):
    1. A biblioteca de classes JDBC deve ser importada. Na linha 1 `"import java.sql.*"` com as outras classes.
    2. Carregue o driver JDBC: linhas 4 e 7. A exceção da linha 5 acontece se não for devidamente carregado.
    3. Crie as variáveis apropriadas para o programa: linhas 8 e 9.
    4. Um objeto de declaração é criado no programa. Classe básica de declaração, `Statement`, com duas subclasses especializadas: `PreparedStatement` e `CallableStatement`. O exemplo ilustra como os objetos de `PreparedStatement` são criados e utilizados. Linhas 14 e 15.
-

# JDBC

---

- 1 O programador deve optar pelo objeto PreparedStatement se uma consulta for executada diversas vezes.
  - 2 O ponto de interrogação (?) da linha 14 representa um parâmetro de declaração. Valor determinado em tempo de execução. Podem haver vários diferenciados pela ordem em que aparecem.
  - 3 Antes de executar uma consulta com PreparedStatement, todos os parâmetros devem ser carregados em variáveis do programa. Dependendo do tipo: setInteger, setString, etc. Veja linha 18.
  - 4 Seguindo estes procedimentos podemos executar a declaração SQL pelo objeto p usando a função executeQuery (linha 19). Existe executeUpdate.
  - 5 Na linha 19, o resultado é devolvido em um objeto r do tipo ResultSet. Semelhante a um cursor de SQL embutido.
    - Ao contrário de outras técnicas o JDBC não diferencia entre as consultas que devolvem uma única tupla e várias tuplas.
- 

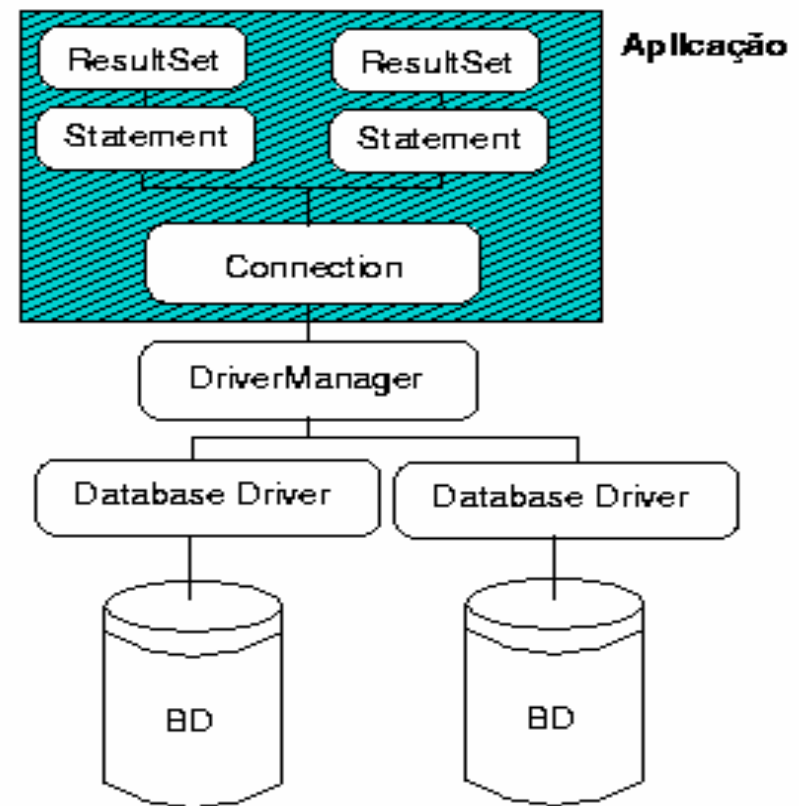


## Segmento de programa JDBC1, um segmento de programa JAVA com o JDBC.

```
-- //Programa JDBC1:
0) import java.io.* ;
1) import java.sql.*
   ...
2) class getEmpInfo {
3)     public static void main (String args []) throws SQLException, IOException {
4)         try { Class.forName("oracle.jdbc.driver.OracleDriver")
5)         } catch (ClassNotFoundException x) {
6)             System.out.println ("Driver nao pode ser carregado") ;
7)         }
8)         String dbacct, senha, ssn, unome ;
9)         Double salario ;
10)        dbacct = readentry("Entre com conta do banco de dados:") ;
11)        passwrđ = readentry("Entre com a senha:") ;
12)        Connection conn = DriverManager.getConnection
13)            ("jdbc:oracle:oci8:" + dbacct + "/" + passwrđ) ;
14)        String stmt1 = "select UNOME, SALARIO from EMPREGADO where SSN = ?" ;
15)        PreparedStatement p = conn.prepareStatement(stmt1) ;
16)        ssn = readentry("Entre com o Numero do Seguro Social: ") ;
17)        p.clearParameters() ;
18)        p.setString(1, ssn) ;
19)        ResultSet r = p.executeQuery() ;
20)        while (r.next()) {
21)            unome = r.getString(1) ;
22)            salario = r.getDouble(2) ;
23)            system.out.println(unome + salario) ;
24)        } }
25) }
```

# Arquitetura

```
rs.getInt("id_fornecedor")  
  
statement.executeQuery("select * from ...")  
  
connection.createStatement()  
  
DriverManager.getConnection(  
    "jdbc:postgresql://localhost:5432/teste?..."  
)  
  
Class.forName("org.postgresql.Driver");
```



## Segmento de programa JDBC2, um segmento de programa JAVA que usa o JDBC para uma consulta que tem, como resultado, uma coleção de tuplas

```
//Segmento de Programa JDBC2:
0)   import java.io.* ;
1)   import java.sql.*
    ...
2)   class printDepartmentEmps {
3)       public static void main (String args []) throws SQLException, IOException {
4)       try { Class.forName("oracle.jdbc.driver.OracleDriver")
5)       } catch (ClassNotFoundException x) {
6)           System.out.println ("O Driver nao pode ser carregado") ;
7)       }
8)       String dbacct, senha, unome ;
9)       Double salario ;
10)      Integer dno ;
11)      dbacct = readentry("Entre com a conta do banco de dados:") ;
12)      passwd = readentry("Entre com a senha:") ;
13)      Connection conn = DriverManager.getConnection
14)          ("jdbc:oracle:oci8:" + dbacct + "/" + passwd) ;
15)      dno = readentry("Entre com o Numero do Departamento: ") ;
16)      String q = "select UNOME, SALARIO from EMPREGADO where DNO = " +
17)          dno.toString() ;
18)      Statement s = conn.createStatement() ;
19)      ResultSet r = s.executeQuery(q) ;
20)      while (r.next()) {
21)          unome = r.getString(1) ;
22)          salary = r.getDouble(2) ;
23)          system.out.println(unome + salario) ;
24)      } }
```

Executa a consulta diretamente sem preparação.

## Como acessar de forma organizada o BD?

---

- ▶ Os programas devem saber se comunicar com o banco de dados. Como fazer isso de maneira adequada? Uma alternativa muito viável é usar o pattern DAO (Data Access Object).
- ▶ DAO: o intermediário entre os mundos
- ▶ Abstração: o DAO abstrai a origem e o modo de obtenção / gravação dos dados, de modo que o restante do sistema manipula os dados de forma transparente, sem se preocupar com o que acontece por trás dos panos.
- ▶ É muito comum em códigos de programadores iniciantes vermos o banco de dados sendo acessada em diversos pontos da aplicação, de maneira extremamente explícita e repetitiva.
- ▶ O DAO também nos ajuda a resolver este problema, provendo pontos unificados de acesso a dados. Desse modo, a lógica de interação com o banco de dados fica em lugares específicos e especializados nisso, além de eliminar códigos redundantes,
- ▶ facilitando a manutenção e futuras migrações.

## Vantagens do Padrão DAO

---

- ▶ Permite transparência
- ▶ Permite migração mais fácil
- ▶ Reduz complexidade dos códigos nos objetos de negócios
- ▶ Centraliza todo acesso de dados em uma nova camada
- ▶ Não útil para a persistência gerenciada pelo container



# Fábrica de conexões

---

Às vezes queremos controlar um processo muito repetitivo

```
public class ConnectionFactory {  
    public Connection getConnection() {  
        try {  
            return DriverManager.getConnection("jdbc:mysql://localhost:3306/db", "root", "root");  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

Até a versão 3 do JDBC, antes de chamar o `DriverManager.getConnection()` era necessário registrar o driver JDBC que iria ser utilizado através do método `Class.forName("com.mysql.jdbc.Driver")`

`getConnection()` é uma fábrica de conexões

```
Connection con = new ConnectionFactory().getConnection();
```





# Exemplo.

---

```
create table contatos (  
    id BIGINT NOT NULL AUTO_INCREMENT,  
    nome VARCHAR(255),  
    email VARCHAR(255),  
    endereco VARCHAR(255),  
    dataNascimento DATE,  
    primary key (id)  
);
```

Javabeans são classes que possuem o construtor sem argumentos e métodos de acesso do tipo get e set.



# Javabeans

---


```
public class Contato {  
  
    private Long id;  
    private String nome;  
    private String email;  
    private String endereco;  
    private Calendar dataNascimento;  
  
    // métodos get e set para id, nome, email, endereço e dataNascimento  
  
    public String getNome() {  
        return this.nome;  
    }  
    public void setNome(String novo) {  
        this.nome = novo;  
    }  
  
    public String getEmail() {  
        return this.email;  
    }  
    public void setEmail(String novo) {  
        this.email = novo;  
    }  
}
```



# JavaBeans

---

```
public String getEndereco() {  
    return this.endereco;  
}  
public void setEndereco(String novo) {  
    this.endereco = novo;  
}  
  
public Long getId() {  
    return this.id;  
}  
public void setId(Long novo) {  
    this.id = novo;  
}  
  
public Calendar getDataNascimento() {  
    return this.dataNascimento;  
}  
public void setDataNascimento(Calendar dataNascimento) {  
    this.dataNascimento = dataNascimento;  
}  
}
```

 }

- 
- ▶ Inserir código SQL dentro das classes de lógica é algo nem um pouco elegante e muito menos viável quando você precisa manter o seu código.
  - ▶ Que tal se pudéssemos chamar um método que adiciona um Contato ao banco?

```
// adiciona um contato no banco  
Misterio bd = new Misterio();
```

```
// método muito mais elegante  
bd.adiciona(contato);
```

Com este código seremos capazes de acessar o BD. Com esta idéia vamos isolar todo o acesso ao BD em classes bem simples. Precisamos de uma classe ContatoDAO com um método adiciona.



```
public class ContatoDAO {

    // a conexão com o banco de dados
    private Connection connection;

    public ContatoDAO() {
        this.connection = new ConnectionFactory().getConnection();
    }

    public void adiciona(Contato contato) {
        String sql = "insert into contatos (nome,email,endereco,dataNascimento) values (?,?,?,?)";

        try {
            // prepared statement para inserção
            PreparedStatement stmt = connection.prepareStatement(sql);

            // seta os valores
            stmt.setString(1,contato.getNome());
            stmt.setString(2,contato.getEmail());
            stmt.setString(3,contato.getEndereco());
            stmt.setDate(4, new Date( contato.getDataNascimento().getTimeInMillis() ));

            // executa
            stmt.execute();
            stmt.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

## Em outro pacote:

```
public class TestaInsere {  
    public static void main(String[] args) {  
        // pronto para gravar  
        Contato contato = new Contato( );  
        contato.setNome("EACH-USP");  
        contato.setEmail("each@usp.br");  
        contato.setEndereco("R. Arlindo Bettio, 1000");  
        contato.setDataNascimento(Calendar.getInstance());  
  
        // grave nessa conexão  
        ContatoDAO dao = new ContatoDAO( );  
  
        // método elegante  
        dao.adiciona(contato);  
  
        System.out.println("Gravado!");  
    }  
}
```

---



# Fazendo buscas no BD.

Crie o método getLista na classe ContatoDAO. Importe List de java.util:

```
1 public List<Contato> getLista() {  
2     try {  
3         List<Contato> contatos = new ArrayList<Contato>();  
4         PreparedStatement stmt = this.connection.prepareStatement("select * from contatos");  
5         ResultSet rs = stmt.executeQuery();  
6  
7         while (rs.next()) {  
8             // criando o objeto Contato  
9             Contato contato = new Contato();  
10            contato.setNome(rs.getString("nome"));  
11            contato.setEmail(rs.getString("email"));  
12            contato.setEndereco(rs.getString("endereco"));  
13  
14            // montando a data através do Calendar  
15            Calendar data = Calendar.getInstance();  
16            data.setTime(rs.getDate("dataNascimento"));  
17            contato.setDataNascimento(data);  
18  
19            // adicionando o objeto à lista  
20            contatos.add(contato);  
21        }  
22        rs.close();  
23        stmt.close();  
24        return contatos;  
25    } catch (SQLException e) {  
26        throw new RuntimeException(e);  
27    }  
28 }
```

Em outro pacote:

```
public class TestaLista {  
    public static void main(String[] args) {  
        // Cria um ContatoDAO  
        ContatoDAO dao = new ContatoDAO( );  
  
        // Liste os contatos com o DAO  
        List<Contato> contatos = dao.getLista( );  
  
        // Itere nessa lista e imprime as informações dos contatos  
        for (Contato contato : contatos) {  
            System.out.println("Nome: " + contato.getNome( ));  
            System.out.println("Email: " + contato.getEmail( ));  
            System.out.println("Endereço: " + contato.getEndereco( ));  
            System.out.println("Data de Nascimento " +  
contato.getDataNascimento( ).getTime() + "\n");  
        }  
    }  
}
```

---





# Outros Métodos para o seu DAO

---

Veja primeiro o método altera, que recebe um contato cujos valores devem ser alterados:

```
1 public void altera(Contato contato) {
2     String sql = "update contatos set nome=?, email=?, endereco=?, dataNascimento=? where id=?";
3
4     try {
5         PreparedStatement stmt = connection.prepareStatement(sql);
6         stmt.setString(1, contato.getNome());
7         stmt.setString(2, contato.getEmail());
8         stmt.setString(3, contato.getEndereco());
9         stmt.setDate(4, new Date(contato.getDataNascimento().getTimeInMillis()));
10        stmt.setLong(5, contato.getId());
11        stmt.execute();
12        stmt.close();
13    } catch (SQLException e) {
14        throw new RuntimeException(e);
15    }
16 }
```



# Outros Métodos para o seu DAO

---

Agora o código para remoção: começa com uma query baseada em um contato, mas usa somente o id dele para executar a query do tipo delete:

```
1 public void remove(Contato contato) {
2     try {
3         PreparedStatement stmt = connection.prepareStatement("delete from contatos where id=?");
4         stmt.setLong(1, contato.getId());
5         stmt.execute();
6         stmt.close();
7     } catch (SQLException e) {
8         throw new RuntimeException(e);
9     }
10 }
```



# Mapeamento Objeto-Relacional

---

- ▶ Com a popularização do Java em ambientes corporativos se faz necessário fazer conexões das classes com os BDs relacionais.
- ▶ Proposta inicial: a criação de SGBDs orientados a objetos puros. Comercialmente não deu certo.
- ▶ Segunda proposta SGBDs relacionais orientados a objetos.
- ▶ Terceira proposta fazer o mapeamento manual de objetos para relações. A POO difere muito do esquema relacional . Essa diferença gera bastante trabalho: a todo momento devemos “transformar” objetos em linhas e linhas de objetos, sendo que essa relação não é um-para-um.
- ▶ Quarta proposta: ferramentas de mapeamento objeto-relacional → Hibernate



# Mapeamento Objeto-Relacional

---

- ▶ Ferramentas/técnicas para armazenar e recuperar objetos de um banco de dados
  - ▶ Da perspectiva do código comporta-se como um BD de objetos
  - ▶ Uma solução ORM completa deveria fornecer:
    - ▶ Funcionalidade básica CRUD
    - ▶ Uma Facilidade de Consulta Orientada a Objetos
    - ▶ Suporte de Mapeamento de Metadados
    - ▶ Capacidade Transacional
- 



# Hibernate

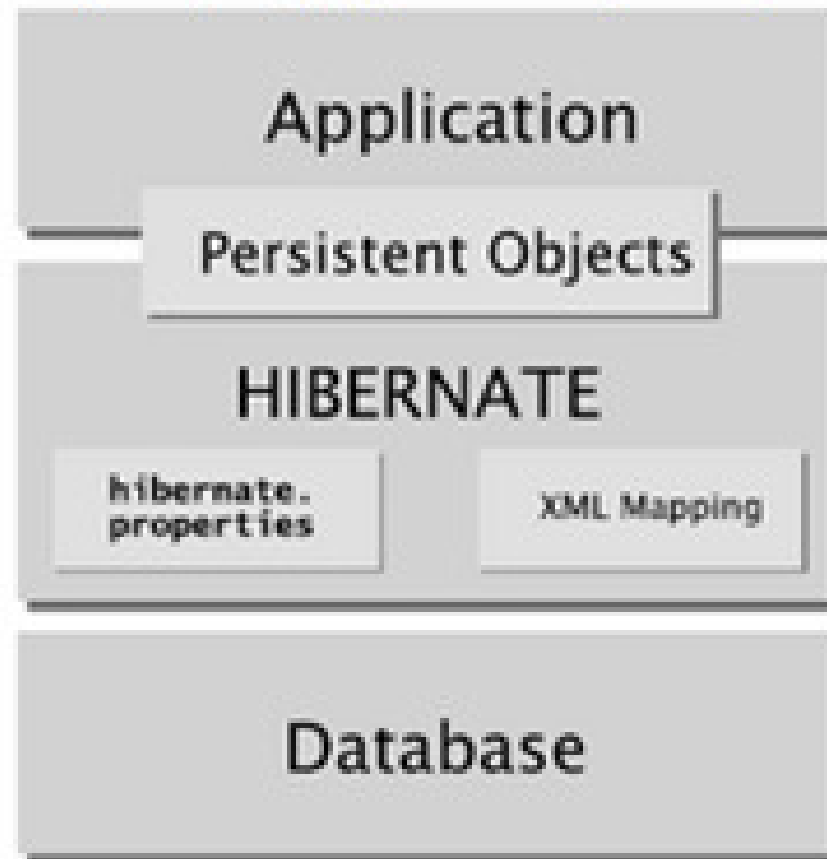
---

- ▶ O Hibernate abstrai o seu código SQL que será gerado em tempo de execução. A ferramenta troca de SGBD, sem precisar de alterar o código.
- ▶ Ele faz o mapeamento de classes Java para tabelas de BDs e de tipos de dados SQL.
- ▶ Ele fornece consultas e facilidades para retorno dos dados que reduzem significativamente o tempo de desenvolvimento.
- ▶ Ele gera o SQL para a aplicação, não necessitando o tratamento dos “result sets”, faz a conversão entre registros e objetos e permite portabilidade (Veja acima).



# Hibernate

---



**Figura 1.** Atuação do Hibernate dentro de uma aplicação

---



# Hibernate

---

- ▶ **Histórico.**
- ▶ Foi desenvolvido por uma equipe de programadores liderada por Gavin King → 2004.
- ▶ Resolver problemas relacionados a persistência em EJB 2.0 → Complexo.
- ▶ Depois foi tomado pelo JBoss Group.
- ▶ Atualmente o projeto está na versão 3.X. Ele é gratuito e distribuído sob a licença de software livre LGPL.



# Hibernate

---

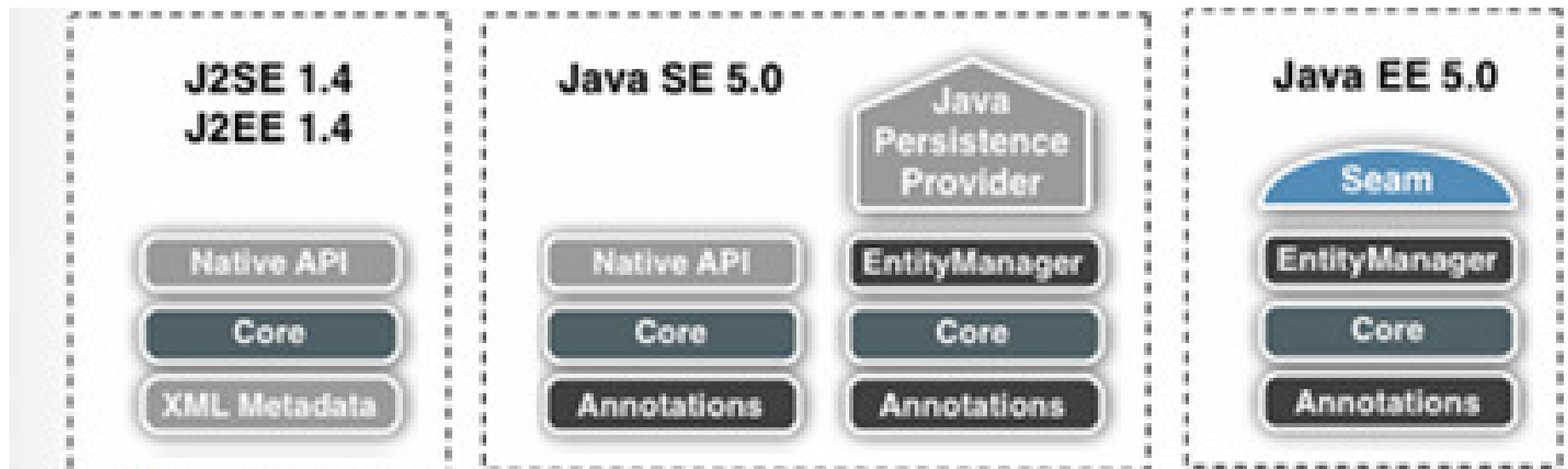
- ▶ **Arquitetura**
- ▶ O projeto Hibernate é composto por vários pacotes Java.
- ▶ Cada pacote → funcionalidade específica. **Veja Fig.**
- ▶ O único pacote que deve estar presente independente da plataforma Java utilizada é o Core.
- ▶ Além desses pacotes pacotes existem outros: **Veja Fig.**
- ▶ Nesta aula analisaremos o Hibernate Core, Annotations e EntityManager, os quais serão apresentados em um exemplo.





# Hibernate

---



**Figura 2.** Arquitetura do Hibernate para a plataforma Java utilizada



# Hibernate

---

Pacote	Descrição
Hibernate Core	Hibernate para Java, APIs nativas e metadados XML
Hibernate Annotations	Mapeia as classes com anotações do JDK 5.0
Hibernate EntityManager	API de persistência Java padrão para Java SE e Java EE
Hibernate Shards	Framework para particionamento horizontal de dados
Hibernate Validator	API para anotação e validação da integridade dos dados
Hibernate Search	Integração do Hibernate com o Lucene para indexação e consulta a dados.
Hibernate Tools	Ferramentas de desenvolvimento para Eclipse e Ant.
JBoss Seam	Framework para aplicações JSF, Ajax e EJB 3.0/Java EE 5.0.

Lista dos pacotes Hibernate e uma descrição resumida de cada um deles

---



## Hibernate Core

---

- ▶ Este componente provê persistência transparente: o único requisito para uma classe ser persistente é ter declarado um construtor default, ou seja, sem argumentos. Ele oferece ainda opções de consulta sofisticadas, seja com SQL diretamente, ou uma linguagem independente HQL (Hibernate Query Language), ou ainda com uma API de consultas útil para gerar consultas dinamicamente.



# Hibernate Annotations

---

- ▶ Requer metadados que governem as transformações dos dados de uma representação para outra. Estes eram tradicionalmente fornecidos por arquivos XML, mas a partir da versão 3.2 do Hibernate pode-se usar anotações do JDK 5.0 para fazer o mapeamento.
- ▶ Para utilizar, necessário, no mínimo, o JDK 5.0 e o Hibernate Core, sem a necessidade de um servidor de aplicação ou container EJB 3.0.



# Hibernate EntityManager

---

- ▶ EJB 3.0 estabelece uma API padrão inspirada no Hibernate.
- ▶ O Entity Manager é o objeto principal da API EntityManager. Ele é usado para acessar um BD de uma determinada unidade de persistência (classes persistentes para as quais o Entity Manager provê suporte CRUD).
- ▶ Uma aplicação poderá definir várias unidades persistentes.



# Desenvolvendo com o Hibernate

---

- ▶ Requer metadados que governem a transformação dos dados de uma representação para outra.
- ▶ Hibernate 2.x mapeamento em arquivos XML.
- ▶ Hibernate 3.x permite que o mapeamento passe a ser descrito na própria classe Java através de *annotations*.
- ▶ As entidades são classes Java comuns (POJOs) e o conceito de entidade do EJB 3 é o mesmo utilizado pelas entidades persistentes do Hibernate.
- ▶ Duas categorias de anotações:
  - ▶ Anotações de mapeamento lógico: associações entre classes, etc.
  - ▶ Anotações de mapeamento físico: tabelas colunas, índices, etc.



## Requisitos de utilização

---

- ▶ Descompactar Hibernate Entity Manager;
- ▶ Instalar versão do JDK 5.0 ou superior.



# Mapeando Classes e Atributos com Anotações

---

- ▶ Toda classe Java que será persistida em um BD é considerada entidade:

**Declaração  
de entidade**

@Entity

```
public class Elementos Serializable {
```

@Id

**Declaração da  
chave primária**

```
    Long id;
```

```
    public Long getId() { return id; }
```

```
    public void setId(Long id) { this.id = id; }
```

```
}
```

---





# Mapeando Classes e Atributos com Anotações

---

- ▶ É possível definir a estratégia para geração graças à anotação `@GeneratedValue`. 4 Tipos:
  1. TABLE - utiliza uma tabela com a informação dos últimos IDs para geração dos próximos;
  2. IDENTITY – utilizado quando não é suportado sequence, mas suportam colunas identidade;
  3. SEQUENCE - utilizando uma sequence para gerar a chave primária;
  4. AUTO - Utiliza uma das estratégias acima de acordo com o banco de dados.

Uma coluna identidade: - identifica cada tupla; - composta de valores gerados pelo SGBD.



# Mapeando Classes e Atributos com Anotações

---

Ex: `@id @GeneratedValue`  
`(strategy=GenerationType SEQUENCE,`  
`generator="SEQ_STORE")`  
`public integer getid( ... )`

- ▶ A anotação `@Table` definida a nível de classe, referencia a tabela que será utilizada para o mapeamento da entidade.

- ▶ Default → o próprio nome da classe. Ex:

`@Entity`

`@Table(name="contacts")`

`public class Contato implements`

- 
- ▶ `Serializable { ... }`

# Mapeando Classes e Atributos com Anotações

---

- ▶ É possível mapear as colunas de uma tabela com a anotação `@Column`. Sobrescreve o valor do nome do próprio atributo. Ex:

`@Entity`

```
public class Vôo implements Serializable {
```

```
...
```

```
@Column(updatable = false, name = "nome_vôo",  
        nullable = false, length = 50)
```

```
public String petName() { ... }
```



# Mapeando Herança com Anotações

---

- ▶ O Hibernate suporta 3 tipos de mapeamentos de herança:
  - ▶ Estratégia Tabela por Classe: utiliza a tag <unio-class> para descrever a estratégia de mapeamento;
  - ▶ Estratégia Única Tabela por Hierarquia de Classes: utiliza a tag <subclass> para descrever a estratégia de mapeamento;
  - ▶ Estratégia Sub-classes interligadas: utiliza a tag <joined-subclass> para descrever a estratégia de mapeamento.
- ▶ A escolha da estratégia é declarada na classe, acima da definição da entidade, usando a anotação @inheritance.
- ▶ A anotação para interfaces não é suportada.

## Estratégia Tabela por Classe

---

- ▶ Esta estratégia tem muitas desvantagens (especialmente com consultas que envolvem superclasses e subclasses e suas associações). Ex:
  - ▶ `@Entity`
  - ▶ `@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)`
  - ▶ `public class Flight implements Serializable {`
- ▶ Esta estratégia suporta uma ou várias associações desde que elas sejam bidirecionais, ou seja, uma classe pode acessar os dados da outra e vice-versa. Não suporta anotações para geração do id (IDENTITY): o id tem de ser compartilhado.



# Estratégia Única Tabela por Hierarquia e Classes

- ▶ Todas as propriedades de todas as super e subclasses são mapeadas em uma mesma tabela, anotação especial para as colunas que distingue as instâncias → **@DiscriminatorColumn**
  - Define a estratégia de Herança
- ▶ @Entity
- ▶ @Inheritance(strategy = InheritanceType.SINGLE\_TABLE)
- ▶ @DiscriminatorColumn(
  - Nome da coluna que distingue
- ▶ name="planetype",
- ▶ discriminatorType= DiscriminatorType.STRING
- ▶ )
- ▶ @DiscriminatorValue("Plane")
- ▶ public class Plane { ... }
  - superclass
- ▶ @Entity
- ▶ @DiscriminatorValue("A320")
  - Subclasse
- ▶ public class A320 extends Plane { ... }

# Estratégia Sub-classes interligadas

- ▶ Cria uma tabela para cada subclasse, as quais são interligadas às superclasse através da chave primária.
- ▶ As anotações `@PrimaryKeyJoinColumn` e `@PrimaryKeyJoinColumns` → definem a(s) chave(s) primária(s) da superclasse que interliga as subclasses.

- ▶ `@Entity`

- ▶ `@Inheritance(strategy=InheritanceType.JOINED)`

- ▶ `public class Boat implements Serializable {`

- ▶ `@Entity`

- ▶ `public class Ferry extends Boat {`

- ▶ `@Entity`

- ▶ `@PrimaryKeyJoinColumn(name="BOAT_ID")`

Ferry é interligada com Boat usando os mesmos nomes da chave primária

É interligada à Boat usando a condição Boat.id = AmericaCupClass.BOAT\_ID

- ▶ `public class AmericaCupClass extends Boat { ... }`

# Mapeando Associações com Anotações

---

- ▶ Para cada tipo de Associação existe uma anotação:

Associações Um-para-Um

Associações Muitos-para-Um

Associações Um-para-Muitos

Associações Muitos-para-Muitos





## Associação Um-para-Um

---

- ▶ É possível associar entidades usando a anotação **@OneToOne**. Existem três casos: entidades associadas compartilham os mesmos valores da chave primária; ou uma chave estrangeira é armazenada por uma das entidades (com valor único); ou uma tabela associativa é usada para armazenar o link entre as duas entidades (valor único em cada FK)



# Exemplo de mapeamento associação

## Um-para-Um com chave compartilhada

---

@Entity

public class Body {

  @Id

  public Long getId() { return id;

  @OneToOne(cascade = CascadeType.ALL)

  @PrimaryKeyJoinColumn

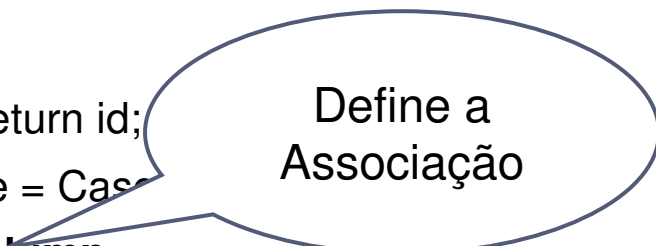
  public Heart getHeart() {

    return heart;

  }

  ...

}



Define a  
Associação

@Entity

public class Heart {

  @Id

  public Long getId() { ...}

---

  }

# Exemplo de mapeamento associação Um-para-Um com chave estrangeira

---

@Entity

```
public class Customer implements Serializable {  
    @OneToOne(cascade = CascadeType.ALL)  
    @JoinColumn(name="passport_fk")  
    public Passport getPassport() {  
        ...  
    }  
}
```

@Entity

```
public class Passport implements Serializable {  
    @OneToOne(mappedBy = "passport")  
    public Customer getOwner() {  
        ...  
    }  
}
```

---

## Associação Muitos-para-Um

---

- ▶ São declaradas em nível de propriedade com a anotação `@ManyToOne`

`@Entity()`

`public class Flight implements`

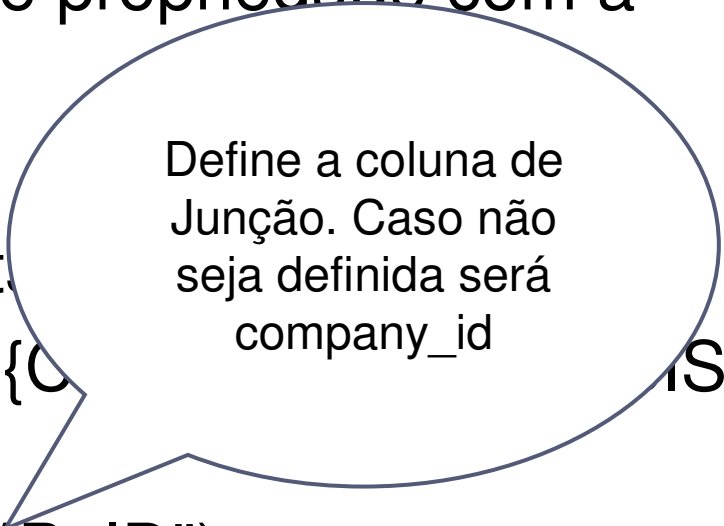
`@ManyToOne( cascade = {C  
CascadeType.MERGE} )`

`@JoinColumn(name="COMP_ID")`

`public Company getCompany() {  
 return company;  
}`

`...`

`▶ }`



Define a coluna de  
Junção. Caso não  
seja definida será  
company\_id

IST,

# Associação Um-para-Muitos

---

- ▶ Declaradas em nível de propriedades com a anotação `@OneToMany`. Ela é bidirecional.

`@Entity`

```
public class Troop {  
    @OneToMany(mappedBy="troop")  
    public Set<Soldier> getSoldiers() {  
        ...  
    }  
}
```

`@Entity`

```
public class Soldier {  
    @ManyToOne  
    @JoinColumn(name="troop_fk")  
    public Troop getTroop() {  
        ...  
    }  
}
```



## Associação Muitos-para-Muitos

---

- ▶ Definida logicamente usando a anotação `@ManyToMany`. Também descreve-se uma tabela associativa e a condição de ligação usando a anotação `@JoinTable`.
- ▶ Se a associação é bidirecional, um lado poderá atualizar as informações do relacionamento na tabela associativa enquanto o outro lado será ignorado.



# Mapeando Enumerações com Anotações

---

- ▶ Anotações do Hibernate também suportam coleções de tipos básicos (integer, String, enums, ...). Uma Coleção de Elementos é anotada como **@CollectionOfElements** . Para definir a tabela coleção, a anotação **@JoinTable** é usada na propriedade da associação.
- ▶ Ex: Veja listagem.



@Entity

```
public class Boy {  
    private Integer id;  
    private Set nickNames =
```

---

```
    new HashSet();  
    private int[] favoriteNumbers;  
    private Set favoriteToys =  
    new HashSet();  
    private Set characters =
```

```
    new HashSet();
```

```
    ...
```

**@CollectionOfElements**

```
public Set getCharacters() {  
    return characters;  
}
```

```
    ...
```

```
public enum Character {  
    GENTLE,  
    NORMAL,  
    AGGRESSIVE,  
    ATTENTIVE,  
    VIOLENT,  
    CRAFTY
```

---

```
    }  
}
```



# Configurando o Hibernate Entity Manager

---

- ▶ A configuração do Entity Manager tanto em um servidor de aplicação como em uma standalone reside em um arquivo de persistência → JAR que contém um arquivo *persistence.xml* na pasta *META-INF*. Todas as classes persistentes (as anotadas com `@Entity`) e todos os arquivos do Hibernate (*hbm.xml*) são empacotados no arquivo JAR.
- ▶ Exemplo do arquivo [persistence.xml](#)
- ▶ O Entity Manager Factory é uma fábrica de Entity Managers. Configurado no arquivo *persistence.xml* através do tag **persistence-unit** e é definido para apontar para um único BD e para mapear um grupo de entidades. Ponto de entrada para criar e gerenciar EntityManager.

# Configurando o Hibernate Entity Manager

---

- ▶ EJB3 prevê uma classe que permite o acesso ao **EntityManager** → **javax.persistence. Persistence**.

```
EntityManagerFactory emf =  
    Persistence.create  
        EntityManagerFactory("manager1")  
EntityManager em =  
    emf.createEntityManager();  
em.close();  
...  
emf.close();
```



# Tornando um objeto persistente com Hibernate Entity Manager

---

- ▶ Uma vez criada uma instância da entidade (usando o operador **new**) ele está em um estado de inserção. Pode ser tornado persistente associando-o ao **EntityManager**.

```
DomesticCat fritz = new DomesticCat();  
fritz.setColor(Color.  
fritz.setSex(M);  
fritz.setName("Fritz");  
em.persist(fritz);
```

Se tiver um gerador de ID automático, o valor é associado aqui, senão deve ser setado antes



# Carregando um objeto com Hibernate Entity Manager

---

- ▶ Para carregar uma instância da entidade pelo valor do ID, utilizamos o método **find()**. Ex:

```
long catId = 1234;
```

```
em.find( Cat.class, new Long(catId) );
```

- ▶ Deletando um objeto: Primeiro carregá-lo e então executar o comando **EntityManager.remove()**
- ▶ Removerá o objeto do BD, mas a instância do objeto continuará existindo na aplicação. Necessário descartar o objeto.



## Modificando um objeto.

---

- ▶ Uso do comando **flush()**.
- ▶ Busca a instância do objeto através do `find()`; fazer as alterações necessárias e executar o comando `flush()` para persistir a informação.



## Consultando objetos

---

- ▶ Consultas em EJB3QL e SQL são representadas por uma instância de `javax.persistence.Query`. Interface oferece métodos para converter parâmetros, executar as consultas e gerar os resultados. Consultas são sempre criadas utilizando o `EntityManager` corrente. Ex:

```
List cats = em.createQuery  
    ( "select cat from Cat as  
      cat where cat.birthdate < ?1")  
.setParameter(1, date,  
    TemporalType.DATE)  
.getResultList();
```



# Desenvolvendo uma aplicação bancária com o Hibernate

---

- ▶ Cadastro de agências, clientes, contas-corrente e poupança.
- ▶ O exemplo a ser construído irá contemplar as seguintes funcionalidades:
  - CRUD de agências;
  - CRUD de clientes;
  - CRUD de contas-corrente e poupança;
  - Operações de depósito, saque e transferência em uma conta, tanto conta-corrente como em poupança;
  - Execução das seguintes consultas:
    - o Todas as contas de um cliente;
    - o Todos os clientes em dívida no cheque especial;
    - o Todos os clientes vips, ou seja, com poupança maior que 100.000;



# Criando as classes do modelo

---

- ▶ Todas as classes receberão as anotações JPA para a persistência de dados. As anotações da classe Agência são descritas na [Listagem 1](#).
- ▶ A classe Cliente (veja [Listagem 2](#)), contém basicamente os mesmos tipos de mapeamento. Destacar campo **sexo**, que é um tipo **Enumeration**.
- ▶ A classe **Conta** ([Listagem 3](#)) é um pouco mais elaborada, ela é uma classe abstrata que é a base para as implementações **ContaCorrente** e **Poupança**. Ela possui os métodos de negócio **sacar()**, **depositar()** e **transferir()** e o método abstrato **getSaldoTotal()**, que é implementado adicionando o limite do cheque especial no caso de conta-corrente e adicionado o valor dos rendimentos no caso de uma poupança.





# Classe Agência com seus mapeamentos.

---

```
package bancohibernate.model;

...

@Entity
@Table (name="AGENCIA")
public class Agencia {
    @Id
    @GeneratedValue (strategy=GenerationType.AUTO)
    @Column(name="ID_AGENCIA")
    private Long codigo;
    @Column(name="NM_NUM_AGENCIA")
    private String numero;
    @Column(name="NM_AGENCIA")
    private String nome;

    public Long getCodigo() {
        return codigo;
    }

    ...
}
```



# Classe Cliente com seus mapeamentos

```
package bancohibernate.model;
```

```
...
```

```
@Entity
```

```
@Table (name="CLIENTE")
```

```
public class Cliente {
```

```
    public enum Sexo {MASCULINO, FEMININO};
```

```
    @Id
```

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

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

```
    private Long codigo;
```

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

```
    private String nome;
```

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

```
    private String cpf;
```

```
    @Enumerated
```

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

```
    private Sexo sexo;
```

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

```
    private Date dataNascimento;
```

```
    public Long getCodigo() {
```

```
        return codigo;
```

```
    }
```

```
...
```

```
}
```



# Implementação das sub-classes da Conta

---

- ▶ Será utilizado o mecanismo de herança do tipo “Join”. Os dados persistem em 3 tabelas diferentes. As tabelas filhas terão como identificador o mesmo identificador da tabela pai.
- ▶ As classes ContaCorrente e Poupança (veja [Listagem 4](#) e [Listagem 5](#)) possuem o mapeamento **@PrimaryKeyJoinColumn (name="ID\_CONTA")**, que indica que herdarão o identificador da classe **Conta**.



## Configurando o persistence.xml

---

- ▶ Apresentado na [Listagem 6](#), fornece a configuração necessária para a persistência dos dados. Várias unidades de persistência, diferenciadas pelo nome, e cada uma com as propriedades seguintes:
  - ▶ Driver do banco de dados, usuário e senha de acesso, url de acesso ao BD e dialeto utilizado. No exemplo o dialeto será HSQLDB configurado com a propriedade **org.hibernate.dialect.HSQLDialect**
  - ▶ O Hibernate fornece uma propriedade **hibernate.hbm2ddl.auto** → define que as tabelas do BD serão automaticamente pelo Hibernate. Duas formas:
    - ▶ **Create-drop:** BD sempre recriado na inicialização
    - ▶ **Update:** BD é atualizado apenas se houver alteração no modelo (novos mapeamentos ou modificações nos mapeamentos)

# Criando classes genéricas para os CRUDs

---

- ▶ Para evitar repetições nos códigos de CRUD (inserção, leitura, atualização e exclusão) iremos criar uma interface CRUD genérica, **CrudDAO** ([Listagem 7](#)), e uma implementação desta utilizando JPAHibernate, **CrudDAOJPA** ([Listagem 8](#)). Usa da classe parametrizada (conceito de Generics do Java 5) para poder executar os métodos genéricos do JPA (find(), persist(), merge(), remove(), etc.). Métodos:
    - ▶ **Public List<T> findAll():** retorna uma lista de objetos do tipo T;
    - ▶ **Public T findById(Serializable id):** retorna um objeto do tipo T do BD a partir do seu identificador;
    - ▶ **Public void addEntity(T entity):** inclui um objeto do tipo T no BD; **updateEntity(T entity)** e **removeEntity(T entity)**.
- 



# Criando DAO's para os objetos persistentes

---

- ▶ Vamos criar as interfaces e implementações DAO para os objetos persistentes da nossa aplicação. Com a criação das classes DAO genéricas para operações CRUD, as interfaces herdarão de **CrudDAO** e adicionarão os outros.
- ▶ Para a entidade **Agencia** são necessários apenas os métodos CRUD. Todo o trabalho será executado pela classe **CrudDAOImpl** ([Listagem 9](#))
- ▶ Já a interface **ClienteDAO**, além dos métodos CRUD, conterá os métodos `findAllClientesDividaChequeEspecial` e `findAllClientesVips` ([Listagem 10](#)). A implementação destes irá utilizar HQL (similar a SQL), mas orientado a objetos.



## Criando DAO's para os objetos persistentes

---

- ▶ Para os objetos persistentes Conta, Conta-Corrente e Poupança serão criadas apenas uma interface e uma implementação DAO para toda a hierarquia, existem várias operações em comum ([Listagem 11](#)).
- ▶ Além dos métodos CRUD que serão iguais para as subclasses, temos os métodos **findAllContasCorrente()**, **findAllPoupanças()**, **sacar()**, **depositar()**, **transferir()** e **findAllContasByCliente()**.
- ▶ O método **transferir()** é interessante para visualizarmos o mecanismo de transação. Obtida do próprio **EntityManager** da JPA que possui os métodos **begin()**, **commit()** e **rollback()**.

# Executando as operações de negócios

---

- ▶ Agora é possível criar uma classe de teste que execute:
  1. **testAgencia()**: Cria duas agências e realiza operações CRUD sobre elas;
  2. **testCliente()**: Cria três clientes e realiza operações CRUD sobre eles;
  3. **testCadastroConta()**: Cria uma conta-corrente e uma poupança para cada cliente;
  4. **testOperacoesConta()**: Realiza operações de saque, depósito e transferência sobre as contas criadas;
  5. **testConsultas()**: Realiza as consultas de clientes em dívida no cheque especial e clientes vips após as operações realizadas sobre as contas.





## Conclusões

---

- ▶ Hibernate é um framework bastante difundido e utilizado pela comunidade Java. A idéia é facilitar a tarefa de mapeamento objeto/relacional.
- ▶ Com a utilização de anotações, ganha-se muito tempo na configuração do Hibernate.
- ▶ Um exemplo demonstrou as facilidades do Hibernate.

