

Mapeamento Objeto/Relacional com Hibernate

Tutorial Básico

Licença para uso e distribuição

Este material está disponível para uso não-comercial e pode ser derivado e/ou distribuído, desde que utilizando uma licença equivalente.



Atribuição-Usó Não-Comercial-Compartilhamento pela mesma licença, versão 2.5

<http://creativecommons.org/licenses/by-nc-sa/2.5/deed.pt>

Você pode copiar, distribuir, exibir e executar a obra, além de criar obras derivadas, sob as seguintes condições: (a) você deve dar crédito ao autor original, da forma especificada pelo autor ou licenciante; (b) você não pode utilizar esta obra com finalidades comerciais; (c) Se você alterar, transformar, ou criar outra obra com base nesta, você somente poderá distribuir a obra resultante sob uma licença idêntica a esta.

Objetivos

- Aprender sobre mapeamento objeto/relacional, uma nova forma de realizar persistência;
- Conhecer os conceitos básicos do *framework* Hibernate;
- Capacitar os alunos na construção de sistemas de informação utilizando Hibernate.

Grupo de Usuários de Java do Estado do Espírito Santo



Mapeamento Objeto/Relacional com Hibernate

Parte I: Mapeamento objeto/relacional

O que é persistência?

- Capacidade de preservar os dados entrados pelo usuário após o programa ter sido fechado;
- Em Java, várias formas:
 - Escrita direta em arquivo (texto ou binário);
 - Serialização (com ou sem *framework*);
 - Banco de dados relacionais (SGBDR) com JDBC;
 - Banco de dados orientado a objetos (SGBDOO);
 - Etc.
- Em sistemas de informação, o uso de SGBDRs é o mais comum.

Uso de SGBDRs em Java

- Comunicação por meio de sentenças SQL:
 - Criação e alteração de tabelas;
 - Inserção, atualização e exclusão de dados;
 - Restrições, projeções e junções;
 - Agrupamentos, ordenação e agregação;
 - Etc.
- Conexão ao SGBD por meio de um driver;
- Padronização da API via JDBC;
- Tarefa tediosa e propensa a erros.

Software orientadas a objetos

- Softwares simples podem ser construídos em cima do acesso JDBC:
 - Lógica de negócio trabalha com linhas e colunas.
- Softwares mais complexos possuem um modelo de domínio:
 - Classes que representam objetos do domínio do problema;
 - Utilização de conceitos OO como polimorfismo;
 - Lógica de negócio trabalha com objetos.

A incompatibilidade de paradigmas

- Há 15 anos se discute o *paradigm mismatch*;
- Representação tabular de dados é muito diferente de um grafo de objetos interligados;
- Os problemas:
 - Granularidade: limitada a tabela e coluna;
 - Herança (subtipos): armazenamento e polimorfismo;
 - Identidade: `==` vs. `equals()` vs. chave-primária;
 - Associações: transposição de chaves;
 - Navegação no grafo de objetos: o problema dos `n+1 SELECTs`.

O custo da incompatibilidade

- Aproximadamente 30% do código é feito para manipular dados via SQL/JDBC;
- Estruturas são repetidas em comandos INSERT, UPDATE e SELECT;
- O modelo de objetos geralmente é “torcido” para se adequar ao modelo de dados;
- Software de difícil manutenção.

Alternativas para persistência

- Divisão em camadas é senso comum;
- Alternativas para camada de persistência:

SQL/JDBC codificado à mão	Desperdício de esforço, baixa produtividade e manutenção, desempenho possivelmente inferior às soluções já existentes.
Serialização	Acesso ao grafo como um todo, não permite buscas, problemas de concorrência.
EJBs de Entidade	Torce o modelo de objetos, sem suporte a polimorfismo e herança, não são portáveis na prática, não são serializáveis, modelo intrusivo que dificulta testes unitários.
Bancos de Dados OO	Baixa aceitação pelo mercado, padrão imaturo.

Mapeamento objeto/relacional

- Solução ideal para o problema;
- Também conhecida como:
 - *Object/Relational Mapping* (ORM);
 - *Gateway-based Object Persistence* (GOP).

Persistência automática e transparente de objetos de um aplicativo Java para tabelas em um banco de dados relacional, utilizando meta-dados que descrevem o mapeamento entre os objetos e o banco de dados. Em essência, transforma dados de uma representação para a outra.

Hibernate in Action

Componentes de uma solução ORM

- API para efetivação de operações CRUD;
- Linguagem ou API para construção de consultas que se refiram às classes ou suas propriedades;
- Mecanismo de especificação dos meta-dados de mapeamento;
- Técnicas de interação com o SGBDR, incluindo:
 - Verificação de objetos sujos (*dirty checking*);
 - Associações recuperadas sob demanda (*lazy association fetching*);
 - Outras funções de otimização.

Problemas resolvidos por ORM

- Como devem ser as classes persistentes e os meta-dados?
- Como mapear hierarquias de classes?
- Como se relacionam identidade de objeto e de linhas em tabelas?
- Qual é o ciclo de vida de um objeto persistente?
- Como recuperar dados de associações de forma eficiente?
- Como gerenciar transações, *cache* e concorrência?

Porque utilizar ORM?

- Produtividade:
 - Elimina a maior parte do código de infra-estrutura.
- Manutenibilidade:
 - Menos linhas de código, menos manutenção;
 - Alterações nos dados não são tão impactantes.
- Desempenho:
 - Mais tempo para implementar otimizações;
 - Maior conhecimento dos detalhes dos SGBDRs.
- Independência de fornecedor:
 - Uso de dialetos de SQL.

Verdades sobre frameworks ORM

- Não são fáceis de aprender;
- Para seu bom uso, é preciso dominar SQL e a tecnologia de bancos de dados relacionais;
- Problemas decorrentes do seu uso são bastante complexos e difíceis de solucionar;
- Não são a “bala de prata” da persistência!

Grupo de Usuários de Java do Estado do Espírito Santo

Conclusões da parte I

- Persistência é um requisito comum em sistemas de informação e existem várias soluções;
- A incompatibilidade dos paradigmas OO e relacional traz complexidade a esta área;
- Mapeamento Objeto/Relacional (ORM) é uma das soluções possíveis para o problema:
 - Possui diversas vantagens como produtividade, manutenibilidade, eficiência, etc.;
 - Possui custos de complexidade;
 - Não é a “bala de prata” da persistência.

Mapeamento Objeto/Relacional com Hibernate

Parte II: Introdução ao Hibernate

Visão geral

- Hibernate é o *framework* ORM mais conhecido;
- Implementa tudo o que se espera de uma solução de mapeamento O/R completa;
- Passos para sua utilização:
 - *Download* e instalação;
 - Criação das classes persistentes;
 - Criação das tabelas no SGBD;
 - Definição do mapeamento O/R;
 - Configuração do *framework*;
 - Uso da API de manipulação e consulta.

Download e instalação

- Arquivos necessários:
 - Distribuição do Hibernate (www.hibernate.org);
 - Hibernate Annotations (idem);
 - Banco de dados HSQLDB (www.hsqldb.org).
- Utilização da IDE Eclipse:
 - O plugin Hibernate Tools facilita muito o trabalho;
 - Não será utilizado neste tutorial.
- Adição das bibliotecas necessárias no *Build Path* do Eclipse (diretório `lib`).

Bibliotecas necessárias

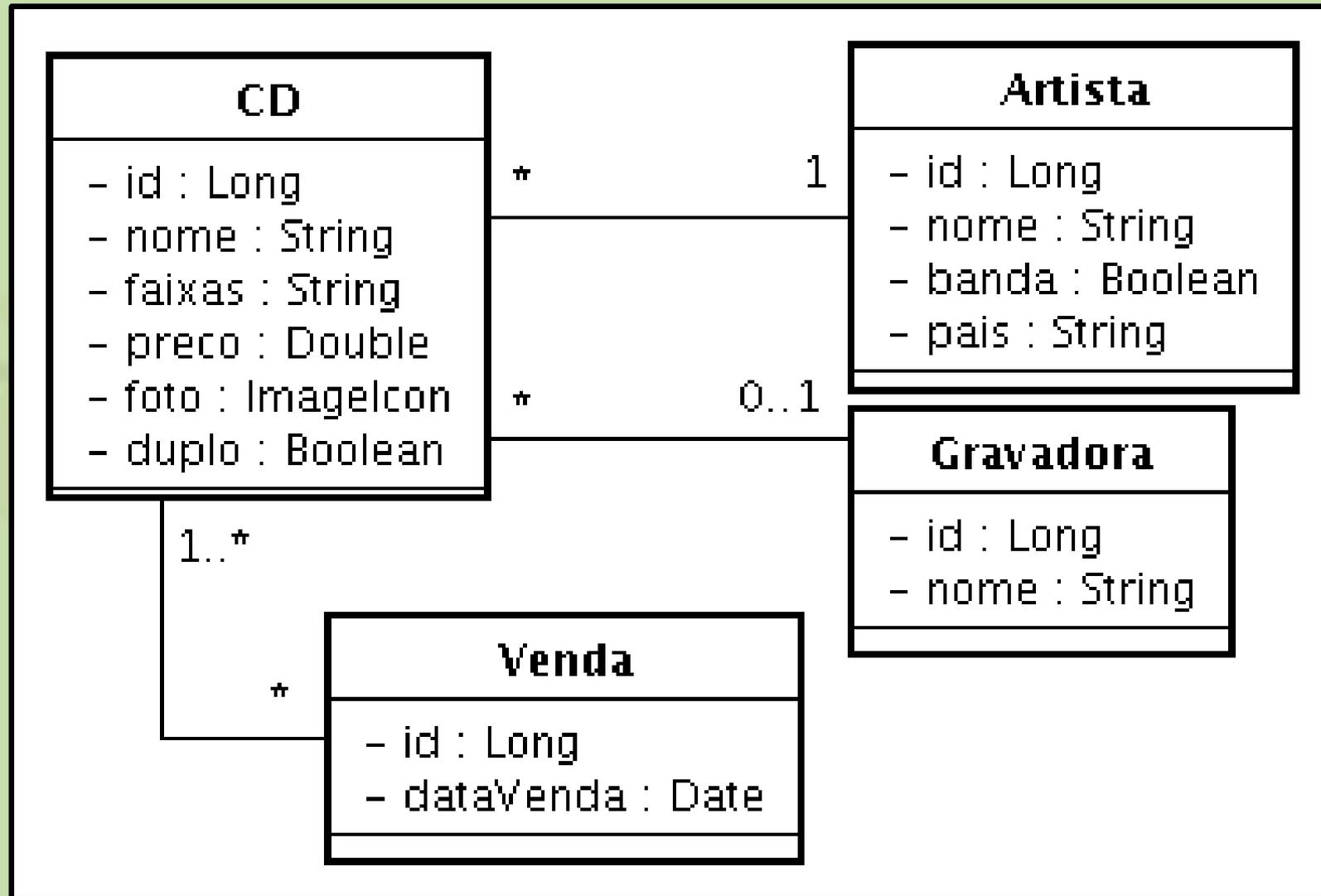
- `antlr`: ANother Tool for Language Recognition;
- `asm-attrs`: ASM bytecode library;
- `asm`: ASM bytecode library;
- `c3p0`: pool de conexões JDBC;
- `cglib`: gerador de bytecodes;
- `commons-collection`: Commons Collection;
- `commons-logging`: Commons Logging;
- `dom4j`: *parser* da configuração e mapeamentos;
- `ehcache`: provedor de *cache*;

Bibliotecas necessárias

- `hibernate3`: Hibernate 3;
- `hsqldb`: banco de dados HSQLDB;
- `jaxen`: opcional, usado para desserialização da configuração (aumento de desempenho);
- `jdbc2_0-stdext`: Standard Extension JDBC APIs (obrigatório fora de um *Application Server*);
- `jta`: Standard JTA API (idem);
- `log4j`: ferramenta de log;
- `ejb3-persistence` e `hibernate-annotations`: Hibernate Annotations.

Sistema exemplo: Java Discos

- Usaremos uma loja de CDs como exemplo:



A classe persistente

```
package tutorialhibernate.dominio;

public class Artista {
    private Long id;

    private String nome;

    private Boolean banda;

    private String pais;

    /* Construtor default implícito. */
    /* Gets e sets das propriedades. */
}
```

A classe persistente

- Uma classe normal (POJO);
- Hibernate é não-intrusivo:
 - Única regra: a classe deve ter um construtor sem parâmetros (mas pode ser `private`);
- Há outras recomendações:
 - Cada propriedade deve ter um get e um set;
 - A classe deve ter uma propriedade de identidade.
- Não seguir as recomendações pode complicar o uso do Hibernate.

A tabela no SGBD

```
CREATE TABLE Artista (  
  id BIGINT NOT NULL IDENTITY,  
  nome VARCHAR(100) NOT NULL,  
  banda BIT NULL,  
  pais VARCHAR(50) NOT NULL,  
  PRIMARY KEY(id)  
);
```

- Gerada manualmente;
- Hibernate possui ferramentas para geração automática das tabelas.

Mapeamento O/R da classe

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping ... >
<hibernate-mapping>
  <class name="tutorialhibernate.dominio.Artista"
        table="Artista">
    <id name="id" column="id">
      <generator class="native" /></id>
    <property name="nome" column="nome"
      type="string" length="100" />
    <property name="banda" column="banda"
      type="boolean" />
    <property name="pais" column="pais"
      type="string" length="100" />
  </class>
</hibernate-mapping>
```

Mapeamento O/R da classe

- Arquivo XML, de preferência um por classe;
 - Especifica classe, tabela e cada propriedade com a respectiva coluna, tipo e restrições.
- Combatendo o *metadata hell*:
 - Hibernate possui padrões de bom senso;
 - Podemos trocar por anotações (veremos mais tarde).
- Por padrão, arquivos de mapeamento devem ficar no mesmo diretório da classe mapeada.

Uso dos padrões no mapeamento

```
<class name="tutorialhibernate.dominio.Artista">  
  <id name="id">  
    <generator class="native" />  
  </id>  
  <property name="nome" length="100" />  
  <property name="banda" />  
  <property name="pais" length="100" />  
</class>
```

- Nome da tabela = nome da classe;
- Nome da coluna = nome da propriedade;
- Tipo da coluna é inferido por reflexão.

Usando Hibernate para salvar dados

```
// Cria o objeto.  
Artista artista = new Artista();  
artista.setNome("Red Hot Chilli Peppers");  
artista.setBanda(true);  
artista.setPais("EUA");  
  
// Obtém uma sessão (veremos depois).  
Session session = HibernateUtil.openSession();  
  
// Salva-o no banco de dados com Hibernate.  
Transaction tx = session.beginTransaction();  
session.save(artista);  
tx.commit();  
session.close();
```

Recuperando um objeto pelo id

```
// Obtém uma sessão (veremos depois).
Session session = HibernateUtil.openSession();

// Recupera um artista pelo id.
Transaction tx = session.beginTransaction();
Artista artista = (Artista)
    session.load(Artista.class, new Long(1));

// Imprime e encerra.
System.out.println(artista.getNome());
tx.commit();
session.close();
```

Recuperando objetos com *queries*

```
// Obtém uma sessão (veremos depois).
Session session = HibernateUtil.openSession();

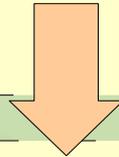
// Recupera todos os artistas.
Transaction tx = session.beginTransaction();
Query query = session.createQuery(
    "from Artista a order by a.nome");
List resultado = query.list();

// Imprime e encerra.
for (Object o : resultado) System.out.println(o);
tx.commit();
session.close();
```

Configuração do *framework*

- É a peça que falta ao nosso exemplo;

```
// Obtém uma sessão (veremos depois).  
Session session = HibernateUtil.openSession();
```



```
public final class HibernateUtil {  
    private static SessionFactory sessionFactory;  
    private static SessionFactory getSessionFactory() {  
        if (sessionFactory == null) sessionFactory = new  
Configuration().configure().buildSessionFactory();  
        return sessionFactory;  
    }  
    public static Session openSession() {  
        return getSessionFactory().openSession();  
    }  
}
```

Opções de configuração

- São quatro opções:
 - Programática (montagem de um objeto Properties e chamada de métodos);
 - Propriedades de sistema: `java -Dchave=valor`;
 - Arquivo `hibernate.properties`;
 - Arquivo `hibernate.cfg.xml`.
- Arquivos (properties ou XML):
 - Forma mais utilizada;
 - Devem estar na raiz do *classpath*;
 - Procurados automaticamente por `configure()`.

Parâmetros da configuração

- Qual é o *driver*, a url, o usuário e a senha do SGBD para criação de conexões JDBC?
- Qual o *pool* de conexões que será usado?
- Qual é o dialeto SQL do banco de dados?
- Os comandos SQL gerados pelo Hibernate devem ser impressos na tela?
- Onde estão os arquivos de mapeamentos das classes?
- Outras configurações diversas...

hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration ... >

<hibernate-configuration>
  <session-factory>

    <!-- Configurações do SGBD. -->
    <property name="connection.driver_class">
      org.hsqldb.jdbcDriver
    </property>
    <property name="connection.url">
      jdbc:hsqldb:hsqldb://localhost/javadiscos
    </property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>
```

hibernate.cfg.xml

```
<!-- Pool de conexões (usando o built-in). -->
<property name="connection.pool_size">1</property>

<!-- Dialeto SQL. -->
<property name="dialect">
    org.hibernate.dialect.HSQLDialect
</property>

<!-- Gerenciamento automático das sessões. -->
<property name="current_session_context_class">
    thread
</property>

<!-- Cache de segundo nível desabilitado. -->
<property name="cache.provider_class">
    org.hibernate.cache.NoCacheProvider
</property>
```

hibernate.cfg.xml

```
<!-- Imprime os comandos SQL enviados ao banco. -->  
<property name="show_sql">true</property>  
  
<!-- Mapeamentos: -->  
<mapping  
resource="tutorialhibernate/dominio/Artista.hbm.xml"  
>  
  
</session-factory>  
</hibernate-configuration>
```

Grupo de Usuários de Java do Estado do Espírito Santo

```
Configuration cfg = new Configuration();  
  
// Procura hibernate.properties e hibernate.cfg.xml.  
cfg.configure();  
  
sessionFactory = cfg.buildSessionFactory();
```

Sobre o *pool* de conexões

- Coleção gerenciada de conexões JDBC;
- Motivação:
 - Criar uma nova conexão é custoso;
 - Ter uma conexão inativa é desperdício de recursos.
- Provido em servidores de aplicação;
- Hibernate vem com C3P0, DBCP e Proxool;
- Existe um *default*, não recomendado para ambientes de produção.

Exemplo de configuração c3p0

```
<property name="c3p0.min_size">5</property>  
<property name="c3p0.max_size">20</property>  
<property name="c3p0.timeout">1800</property>  
<property name="c3p0.max_statements">50</property>
```



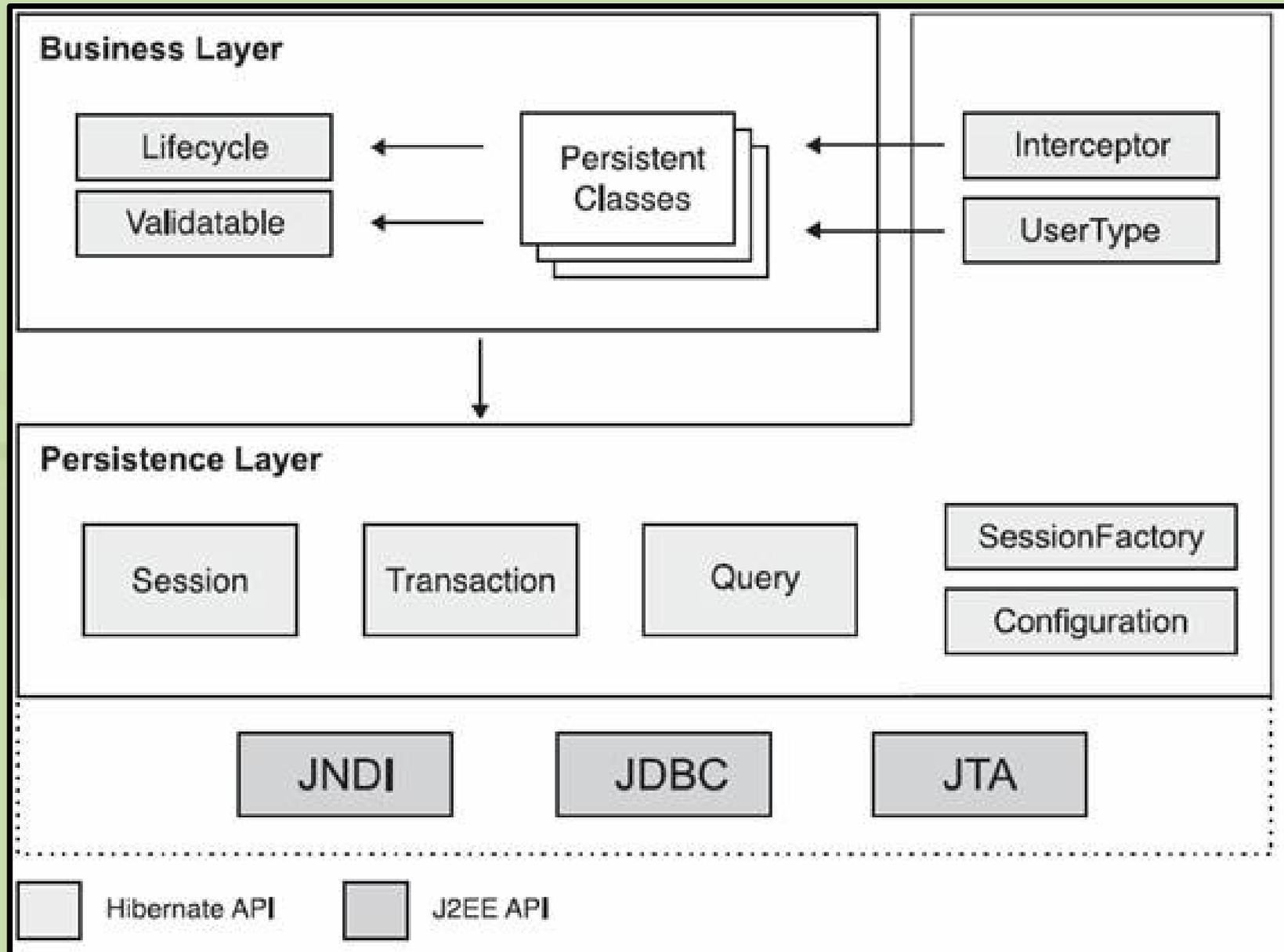
Logging

- Arquivo `log4j.properties` na raiz do *classpath*.

```
# File Appender:
log4j.appender.tmpFile = org.apache.log4j.FileAppender
log4j.appender.tmpFile.File = /tmp/info.log
log4j.appender.tmpFile.layout =
                                org.apache.log4j.PatternLayout
log4j.appender.tmpFile.layout.ConversionPattern =
                                                [%d] %c %5p: %m%n

# Loggers:
log4j.rootLogger = warn, tmpFile
log4j.org.hibernate = info, tmpFile
```

Arquitetura



org.hibernate.Session

- Sessão de acesso a dados;
- É a principal interface com a persistência;
- Objeto leve: baixo custo para construção;
- Não é *threadsafe*: deve ser usado por somente uma linha de execução;
- Obtida por meio da fábrica de sessões;
- Possui uma coleção de objetos associados àquela unidade de trabalho.

org.hibernate.SessionFactory

- Objeto usado para obter sessões;
- Objeto pesado: alto custo para construção;
- *Threadsafe*: pode ser compartilhado;
- Deve haver uma fábrica para cada banco de dados utilizado na aplicação;
- Criada a partir da configuração.

org.hibernate.cfg.Configuration

- Permite a configuração do Hibernate;
- Cria fábricas de sessão.



org.hibernate.Transaction

- Abstrai o mecanismo de transações utilizado pelo JDBC nos bastidores:
 - Transações JDBC, JTA, CORBA, etc.
- Auxilia na portabilidade do código;
- Seu uso é opcional:
 - O Hibernate assumirá início e fim de transação em cada operação como *default*.

org.hibernate.Query

- Realização de consultas:
 - Em HQL;
 - Em SQL;
 - Programaticamente.
- Auxílio da classe `org.hibernate.Criteria`.

Grupo de Usuários de Java do Estado do Espírito Santo

Interfaces de *callback*

- Recebem notificações quando algum evento importante ocorrerem (funcionam como *listeners*);
- Interfaces intrusivas (implementadas pelo próprio objeto de domínio):
 - `org.hibernate.classic.Lifecycle`;
 - `org.hibernate.classic.Validatable`.
- Interface não-intrusiva:
 - `org.hibernate.Interceptor`.

Tipos

- Classes que fazem o mapeamento dos tipos OO para as colunas do banco de dados;
- Tipos prontos do Hibernate:
 - `org.hibernate.type.Type`.
- Tipos personalizados pelo desenvolvedor:
 - `org.hibernate.usertype.UserType`;
 - `org.hibernate.usertype.CompositeUserType`.

Interfaces de extensão

- Características que podem ser personalizadas:
 - Geração de chave-primária;
 - Suporte a dialetos SQL;
 - Estratégias de cache;
 - Gerenciamento de conexão JDBC;
 - Gerenciamento de transações;
 - Estratégia ORM;
 - Estratégia de acesso às propriedades dos objetos;
 - Criação de proxies.

Características do Hibernate

- Gerenciamento de objetos persistentes;
- *Dirty checking*: verifica se objetos persistentes foram alterados e atualiza o banco de dados;
- *Transaction write-behind*: só envia SQL quando a transação é concluída;
- Mapeamento flexível, consultas polimórficas;
- Dois níveis de cache;
- *Lazy initialization*: associações e propriedades;
- Buscas com *outer join*;
- Etc., etc., etc.

Conclusões da parte II

- Vimos que o Hibernate pode ser uma solução para persistência;
- Aprendemos como instalá-lo e sobre suas dependências;
- Passamos rapidamente a parte de configuração e uso básico;
- Vimos também sua arquitetura e características gerais.

Mapeamento Objeto/Relacional com Hibernate

Parte III: Mapeamento de classes

Quais classes mapear?

- Todas que precisam armazenar seu estado em mídia persistente (banco de dados);
- Geralmente mapeamos as classes de domínio;
- Modelo de domínio:
 - Montado a partir da análise do domínio do problema a ser solucionado;
 - Contém classes que representam elementos do mundo real envolvidos no negócio;
 - Vamos entender um pouco melhor alguns detalhes sobre classes de domínio...

Regras para classes de domínio

- Não depender de APIs:
 - Não pode ter código JDBC, Swing, *Web* ou depender de quaisquer bibliotecas externas.
- Não se preocupar com *cross-cutting concerns*:
 - Persistência, transações, *logging*, etc. são preocupações transversais;
 - Não é objetivo das classes de domínio se preocupar com estas tarefas;
 - Precisamos que a persistência seja transparente.

Persistência transparente

- ≠ persistência automatizada (EJB);
- Separação total entre classes de domínio e lógica de persistência;
- Não requer implementação de interface ou herança de classe abstrata;
- Classe pode ser utilizada em outros contextos.

Grupo de Usuários de Java do Estado do Espírito Santo

Hibernate trabalha com POJOs

- É obrigatório:
 - Classe deve possuir construtor *default* (pode não ser público);
 - Para atributos do tipo coleção (ex.: listas), usar a interface (`List`) e não uma classe (`ArrayList`);
- É opcional, porém recomendado:
 - Todos os atributos possuírem métodos `get` e `set`;
 - Haver um atributo específico para chave primária.

Um POJO

```
package tutorialhibernate.dominio;

public class Artista {
    private Long id;
    private String nome;
    private Boolean banda;
    private String pais;

    private Set cds;

    /* Construtor default implícito. */

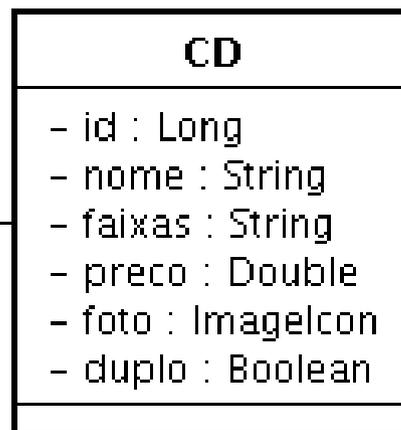
    /* Gets e sets das propriedades. */
}
```

Implementando associações



1

*



Uso da interface.
Hibernate provê
implementação
própria.

```
public class Artista {  
    /* ... */  
    private Set<CD> cds;  
    public Set<CD> getCds() { return cds; }  
    private void setCds(Set<CD> cds) {  
        this.cds = cds;  
    }  
}
```

Método de atribuição privativo.

Implementando associações

- Hibernate não gerencia associações por você;
- Faça seu próprio método de conveniência:

```
public class Artista {
    /* ... */

    public void addCd(CD cd) {
        if (cd == null) throw new
            IllegalArgumentException("CD nulo");
        if (cd.getArtista() != null)
            cd.getArtista().getCds().remove(cd);
        cd.setArtista(this);
        cds.add(cd);
    }
}
```

Implementando associações

- Seu método de conveniência deve ainda garantir as cardinalidades da associação;
 - É boa prática implementar comportamento e garantir restrições de domínio.
- Método `getCds()` não deve retornar uma cópia do conjunto:
 - Retornar cópia é prática comum de encapsulamento, evitando `artista.getCds().add(cdQualquer)`;
 - No entanto, isso causa confusão no Hibernate, por causa da checagem de objetos sujos.

Opções de ORM

- XML;
 - Arquivo de marcação associado a esquema;
 - Criticado por muitos (*metadata hell*).
- XDoclet:
 - Ferramenta que gera XML a partir de anotações nos comentários JavaDoc.
- Hibernate Annotations:
 - Permite que configuremos o mapeamento nas próprias classes por meio de anotações Java;
 - Somente Java 5.0 e superior.

Mapeamento em XML

- Retomando exemplo anterior para explicar:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-
//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-
3.0.dtd">
<hibernate-mapping>
  <class name="tutorialhibernate.dominio.Artista">
    <id name="id"><generator class="native" /></id>
    <property name="nome" length="100" />
    <property name="banda" />
    <property name="pais" length="100" />
  </class>
</hibernate-mapping>
```

Mapeamento em XML

- Cabeçalho:

```
<?xml version="1.0"?>  
<!DOCTYPE hibernate-mapping PUBLIC "-  
//Hibernate/Hibernate Mapping DTD 3.0//EN"  
"http://hibernate.sourceforge.net/hibernate-  
mapping-3.0.dtd">
```

- Uso obrigatório;
- Indicação de documento XML e seu DTD.

Mapeamento em XML

- Classe e identificador:

```
<hibernate-mapping>
  <class name="tutorialhibernate.dominio.Artista"
        table="Artista">
    <id name="id" column="id">
      <generator class="native" />
    </id>

  </class>
</hibernate-mapping>
```

- Recomenda-se uma classe por arquivo;
- Indica qual é a propriedade identificadora do objeto e como seus valores são gerados (detalhes depois).

Mapeamento em XML

- Atributos da classe:

```
<property name="pais" column="pais" type="string" length="100" />
```

- Recomenda-se usar os *sensible defaults*;
- Podemos indicar se um atributo pode ser nulo:

```
<property name="pais" not-null="true" />
```

- Podemos usar a tag `<column />` para detalhar a configuração da coluna:

```
<property name="pais" length="100">  
  <column name="pais" />  
</property>
```

Mapeamento em XML

- Propriedades derivadas:

```
<property name="precoComDesconto" formula="preco - (0.1 * preco)" />
```

- Calculados no SGBD (usando SQL) em *runtime*;
- Usadas somente em expressões SELECT.
- Estratégia de acesso à propriedade:

```
<property name="pais" access="field" />
```

- Indica acesso direto ao atributo (sem usar get/set);
- *Default* é property (usando get/set);
- É possível definir seu próprio PropertyAccessor.

Mapeamento em XML

- Controlando INSERTs e UPDATEs:

```
<property name="dadoSomenteLeitura" insert="false"
update="false" />
```

- Indica se uma propriedade participa nas sentenças INSERT e UPDATE (*default é true*).

```
<class name="tutorialhibernate.dominio.Artista"
dynamic-insert="true" dynamic-update="true">
```

- *Dynamic insert*: incluir somente não-nulos nos INSERTs (*default é false*);
- *Dynamic update*: incluir somente atributos alterados nos UPDATEs (*default é false*);

Mapeamento em XML

- Identificadores SQL entre aspas:

```
<property name="descricao" column="'Descricao do Item' " />
```

- Coloca aspas ao redor de um identificador SQL (nome de coluna, tabela, etc.);
 - Em alguns BDs, força a sensibilidade à caixa;
 - Usado mais comumente em bancos legados.
- Espaço de nomes (*namespace*):

```
<hibernate-mapping  
    package="tutorialhibernate.dominio">  
    <class name="Artista" table="Artista">
```

Mapeamento em XML

- Outras possibilidades:
 - NamingStrategy & SQL Schemas: determinar padrões de nome para tabelas e colunas;
 - Manipulação de meta-dados em tempo de execução:

```
// Use antes de cfg.buildSessionFactory()  
PersistentClass metaDados;  
metaDados = cfg.getClassMapping(Artista.class);
```

Grupo de Usuários de Java do Estado do Espírito Santo

Entendendo identidade de objetos

- Para entendermos os identificadores do Hibernate, precisamos entender de identidade;
- Dois objetos A e B podem ser:
 - Idênticos: $(A == B)$ é **true**;
 - Equivalentes $(A.equals(B))$ é **true**;
 - Idênticos no banco de dados: representam a mesma linha, ou seja, estão na mesma tabela e possuem o mesmo valor para chave-primária.

Identificadores do Hibernate (IDs)

```
package tutorialhibernate.dominio;

public class Artista {
    private Long id;

    public Long getId() { return id; }

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

- *Setter* é privado: só o Hibernate irá mexer;
- *Getter* é público: útil para passar como parâmetro, especialmente em ambientes *Web*.

Mapeamento do identificador

```
<id name="id" column="id">  
  <generator class="native" />  
</id>
```

- Agora, identidade no BD pode ser verificada com `A.getId().equals(B.getId());`
- Uma classe pode ter ID gerenciado pelo Hibernate:

```
<id column="id">  
  <generator class="native" />  
</id>
```

- Pode obter o ID com `session.getIdentificier(o);`
- Não é recomendado (perda de desempenho).

Escolha do identificador

- Equivale à escolha da chave-primária:
 - (a) Conjunto de atributos que identificam um objeto univocamente (CPF para pessoa, ISBN para livro);
 - (b) Criação de um atributo específico para a PK.
- Opção A:
 - Chamada de “chave natural”;
 - Pode causar problemas de manutenção.
- Opção B:
 - Chamada de “chave substituta” ou “artificial”;
 - Recomendada pelos autores do Hibernate.

Geração de IDs artificiais

- Hibernate provê uma série de geradores, configurados em `<generator class="" />`:

<code>increment</code>	Incremento automático para uso não-paralelo.
<code>identity</code>	Coluna IDENTITY nos bancos que suportam.
<code>sequence</code>	Coluna SEQUENCE nos bancos que suportam.
<code>hilo</code>	Usa o algoritmo high/low (Scott Ambler).
<code>native</code>	Escolhe dentre <code>identity</code> , <code>sequence</code> e <code>hilo</code> , dependendo das capacidades do SGBD.
<code>uuid</code>	Gera uma string de 32 caracteres única dentro da rede.
<code>assigned</code>	Atribuído manualmente pelo programador antes de salvar.

- Você pode implementar seu próprio `IdentifierGenerator`.

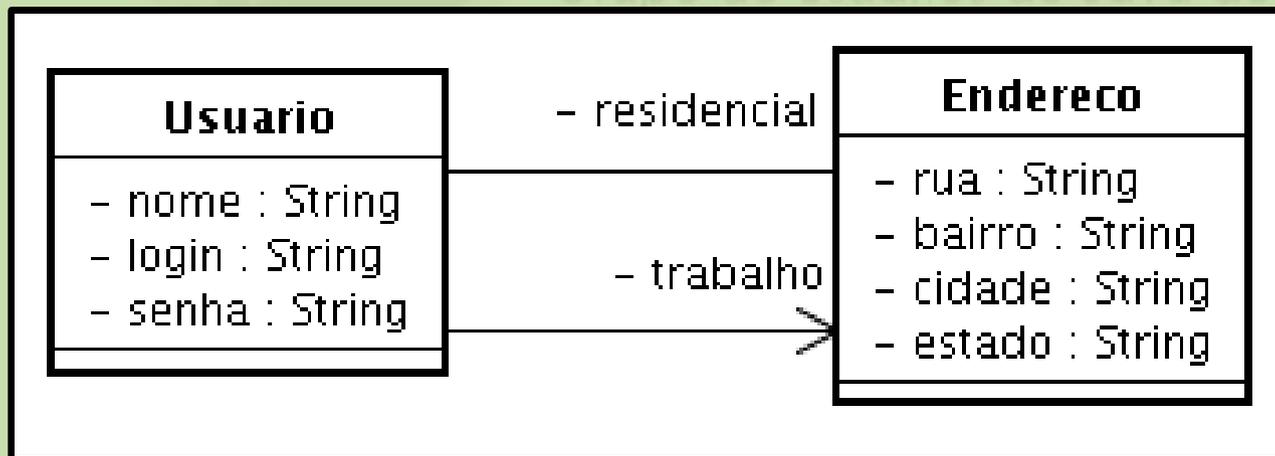
Identificadores compostos

- Duas ou mais propriedades do objeto compõem o identificador;
- Mais usado com chaves naturais em bancos de dados legados;
- Uso avançado e não recomendado para novos projetos.

Grupo de Usuários de Java do Estado do Espírito Santo

Entidades x tipos-valor

- Hibernate permite que tenhamos mais classes do que tabelas no banco de dados;
- Classe que não possui tabelas é um tipo-valor:
 - Existe apenas associada a uma classe entidade;
 - Suas propriedades são armazenadas na tabela da entidade à qual é associada, não possui id e segue o ciclo de vida da entidade dona.



Outros exemplos:
String, Integer,
Date, etc.

Componentes

- Hibernate chama estes tipos-valor de componentes (*components*);
 - Não confundir com componentes de software!

```
<class name="Usuario">
  ...
  <component name="residencial" class="Endereco">
    <parent name="usuario" />
    <property name="rua" />
    ...
  </component>

  <component name="trabalho" class="Endereco">
    ...
  </component>
</class>
```

Componentes

- Um componente pode:
 - Definir quantas propriedades quiser;
 - Possuir outros componentes;
 - Ter associações com outras entidades.
- Limitações:
 - Não pode ter mais de um pai (ser compartilhado);
 - Não há diferença entre um componente nulo e um componente com todas as propriedades nulas.

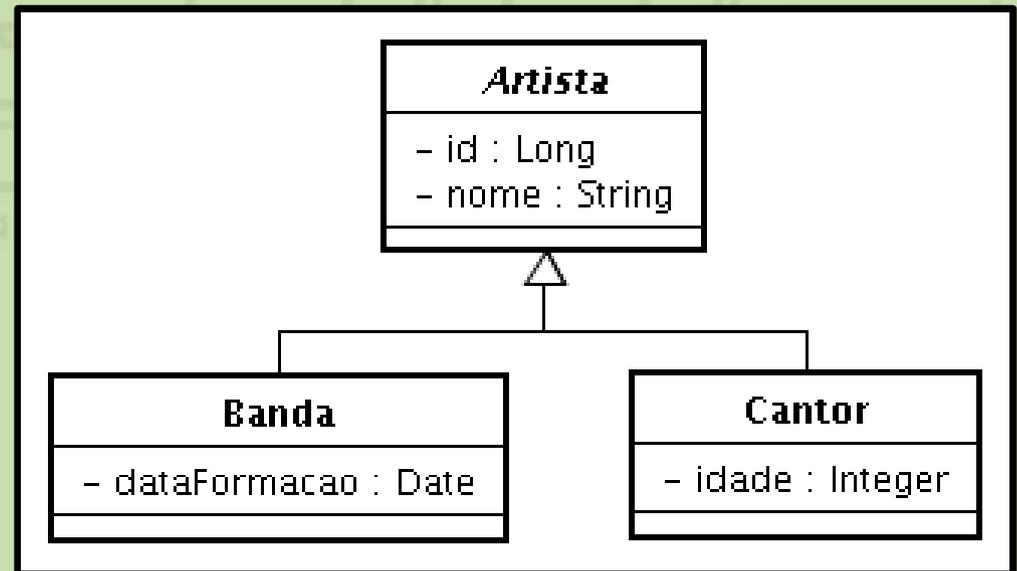
Mapeamento de herança

- Herança distingue OO de Relacional;
- É necessário fazer uma conversão;
- Scott Ambler propõe três formas:
 - Uma tabela para cada classe;
 - Uma tabela para cada classe concreta;
 - Tabela única para toda a hierarquia.

Grupo de Usuarios de Java do Estado do Espírito Santo

Mapeamento de herança

- Exemplo:
 - Artista é classe abstrata;
 - Duas subclasses concretas: Banda e Cantor.



Uma tabela para cada classe concreta

- Suporte ruim a polimorfismo (ex.: associações na superclasse);
- Consulta polimórfica ineficiente (vários SELECTs);
- Consulta à classe concreta eficiente;
- Duplicação de colunas prejudica manutenção;
- Use quando polimorfismo não for um requisito.

Banda	Cantor
id: BIGINT	id: BIGINT
nome: VARCHAR(100)	nome: VARCHAR(100)
dataFormacao: DATE	idade: INTEGER

Uma tabela para cada classe concreta

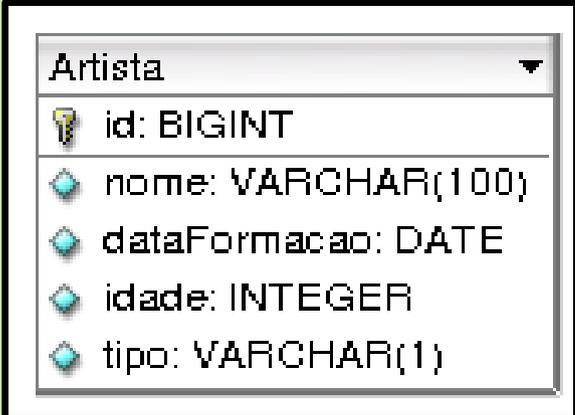
```
<class name="Banda">  
  <!-- Declaração de todas as propriedades. -->  
</class>
```

```
<class name="Cantor">  
  <!-- Declaração de todas as propriedades. -->  
</class>
```

Grupo de Usuários de Java do Estado do Espírito Santo

Tabela única para toda a hierarquia

- Polimorfismo e consultas polimórficas eficientes;
- Não há colunas redundantes;
- Colunas pertencentes somente às subclasses devem ser *nullable* (problema de restrição de integridade);
- Desperdício de espaço;
- Recomendado para a maioria dos casos.



A screenshot of a database table definition for 'Artista'. The table has the following columns:

Column Name	Data Type
id	BIGINT
nome	VARCHAR(100)
dataFormacao	DATE
idade	INTEGER
tipo	VARCHAR(1)

Tabela única para toda a hierarquia

```
<class name="Artista" discriminator-value="a">
  <id name="id"> ... </id>

  <discriminator column="tipo" type="char" />

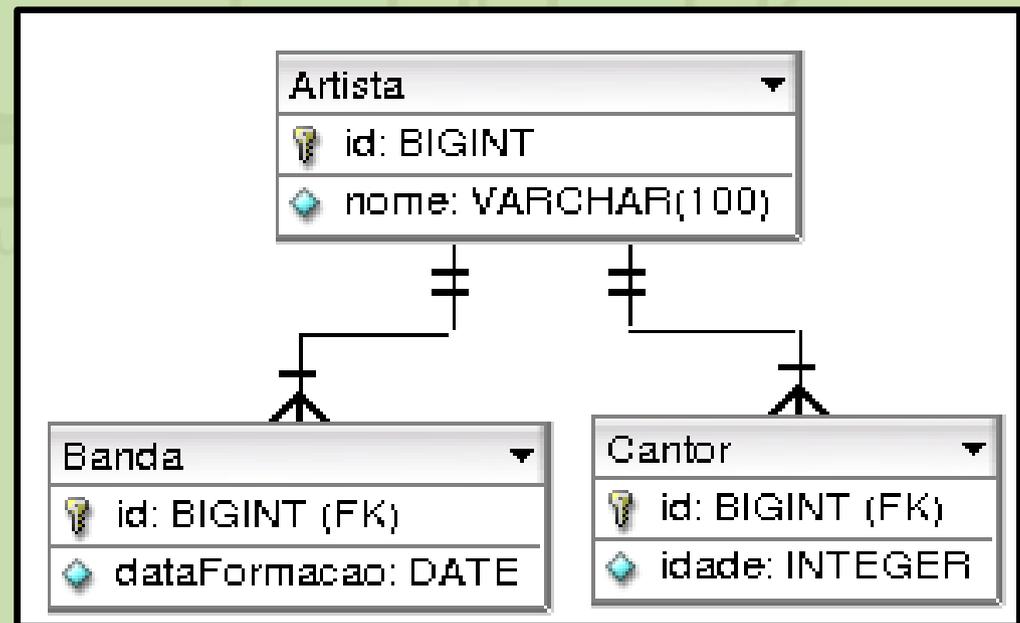
  <property name="nome" />

  <subclass name="Banda" discriminator-value="b">
    ...
  </subclass>

  <subclass name="Cantor" discriminator-value="c">
    ...
  </subclass>
</class>
```

Uma tabela para cada classe

- Sem problemas com integridade e ambiguidade;
- Desempenho ruim devido ao uso de JOINS;
- Recomendado quando integridade for um requisito forte.



Uma tabela para cada classe

```
<class name="Artista">
  <id name="id"> ... </id>

  <property name="nome" />

  <joined-subclass name="Banda">
    <key column="id" />
    ...
  </subclass>

  <joined-subclass name="Cantor">
    <key column="id" />
    ...
  </subclass>
</class>
```

Mapeamento de herança

- Não se pode combinar estratégias de mapeamento;
- `<subclass />` e `<joined-subclass />` podem ser declarados:
 - Dentro do `<class />` pai (como nos exemplos);
 - Em outro arquivo de mapeamento (deve especificar `<subclass name="..." extends="..." />`).

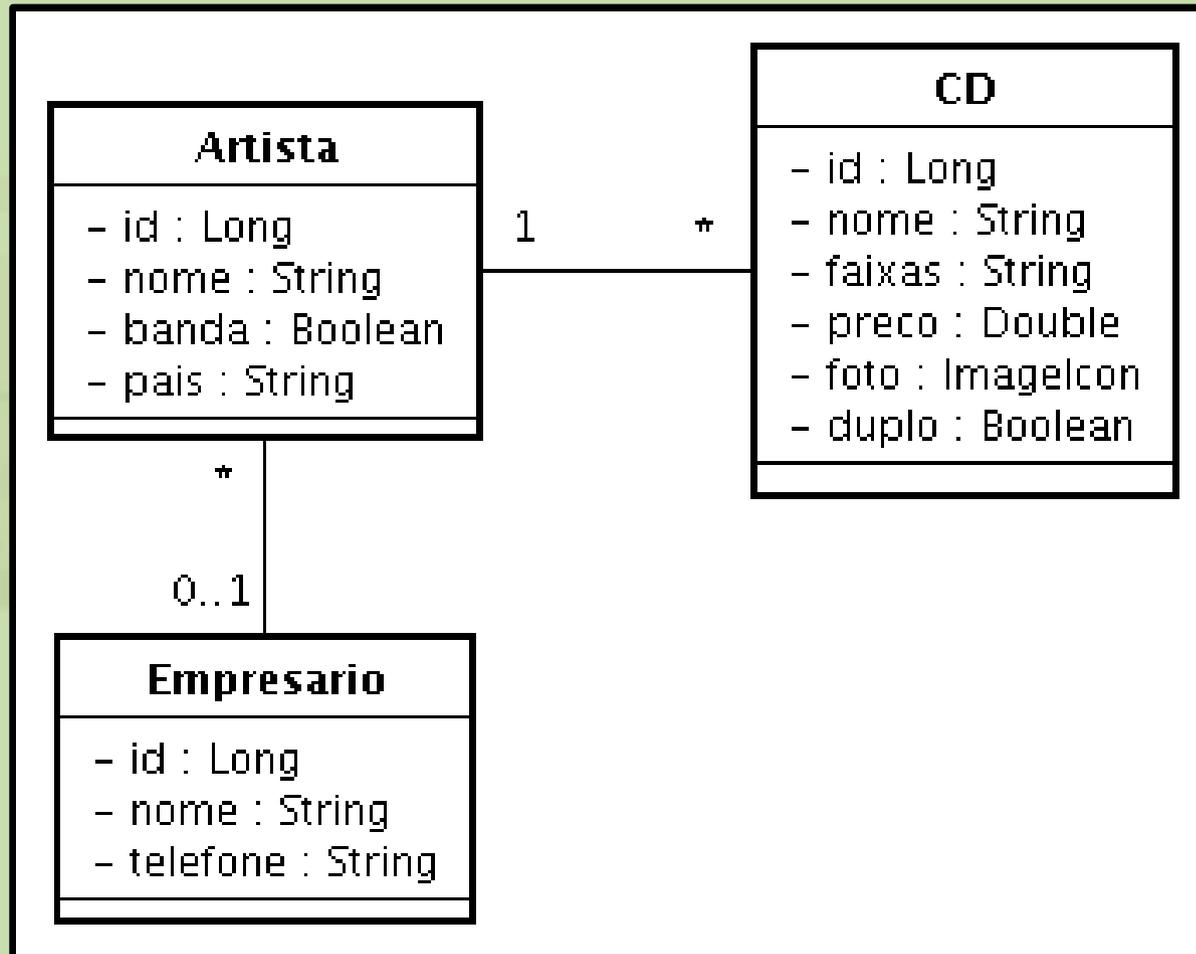
Mapeamento de associações

- Tema mais complexo de ORM;
- Veremos apenas os casos mais simples;
- Associações Hibernate não são gerenciadas:
 - EJBs com CMP possuem associações gerenciadas pelo container. Hibernate trabalha com POJOs!
- Por padrão, associações são unidirecionais.

Grupo de Usuários de Java do Estado do Espírito Santo

Associações “um-para-muitos”

- Cardinalidade das associações:



Mapeamento n-para-1

```
<class name="CD">
    ...
    <many-to-one name="artista"
        class="tutorialhibernate.dominio.Artista"
        not-null="true"
        cascade="none" />
</class>

<class name="Artista">
    ...
    <many-to-one name="empresario"
        class="tutorialhibernate.dominio.Empresario"
        not-null="false"
        cascade="delete-orphan" />
</class>
```

Persistência transitiva

- Hibernate aplica persistência por transitividade:
 - Se objeto X é persistente e um objeto Y associa-se a ele, o objeto Y deve tornar-se persistente.
- Configurável pela opção `cascade=" . . . "`:
 - `save - update`: se X for salvo, Y também será;
 - `delete`: se X for excluído, Y também será;
 - `refresh`: se X for atualizado (dados recuperados do banco e atualizado na memória), Y também será;
 - `delete - orphan`: se um Y não tem mais nenhum X associado, será excluído;
 - `all`: todos os cascadeamentos.

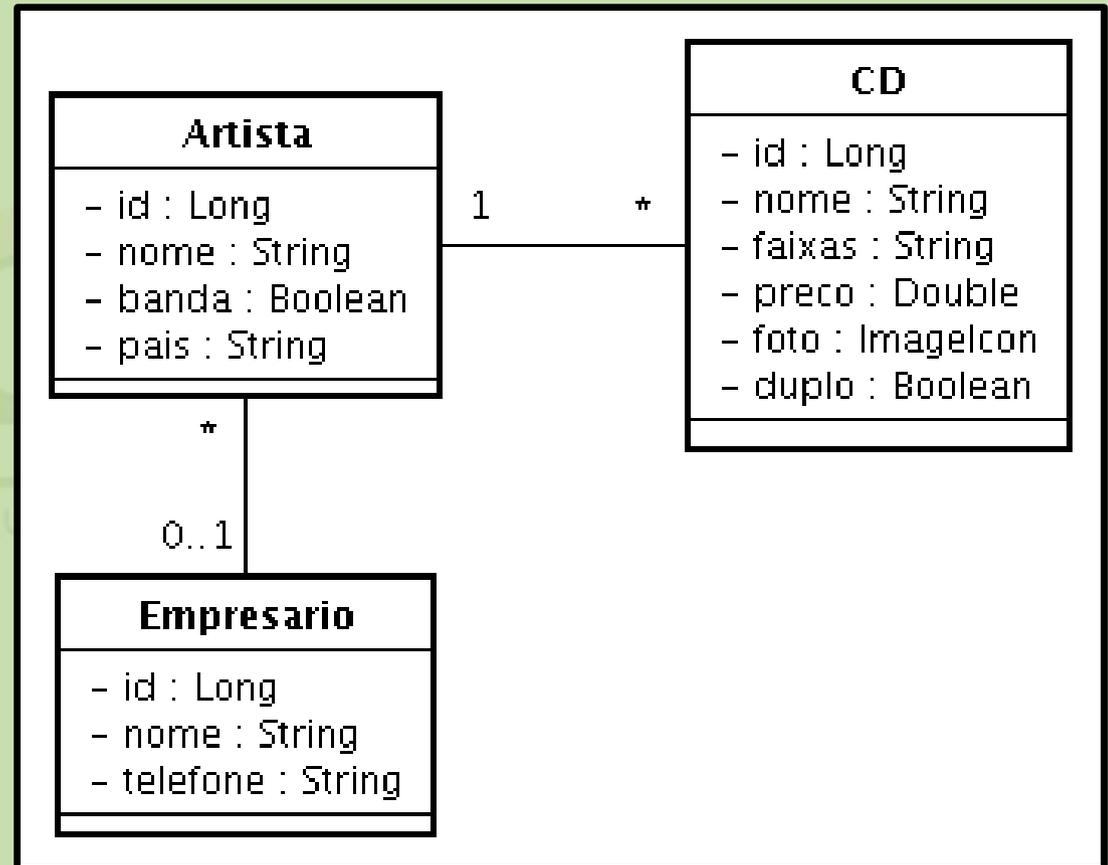
Mapeamento 1-para-n

```
<class name="Artista">
  ...
  <set name="cds" inverse="true" cascade="all">
    <key column="idArtista" />
    <one-to-many class="[...]CD" />
  </set>
</class>

<class name="Empresario">
  ...
  <list name="artistas" lazy="false" inverse="true">
    <key column="idEmpresario" />
    <list-index column="ordem" />
    <one-to-many class="[...]Artista" />
  </set>
</class>
```

Exercício

- Faça um programa de cadastro de CDs, baseado no modelo da figura ao lado;
- Escolha a interface que for mais simples;
- Confira no banco de dados as alterações feitas pelo Hibernate.



Conclusões da parte III

- Sistemas complexos possuem camadas de domínio, representando conceitos do problema;
- Uma abordagem interessante para gerência de dados é a persistência automática de POJOs;
- Aprendemos como mapear classes simples, hierarquias e associações com arquivos XML;
- Discutimos identidades de objetos e vimos a geração de Ids artificiais e a existência de tipos-valor.



Mapeamento Objeto/Relacional com Hibernate

Parte IV: Hibernate Annotations

Anotações

- São meta-dados que são adicionados ao código-fonte para descrever características do mesmo;
- As anotações estão disponíveis para avaliação durante a execução do programa;
- É vista como uma das principais alternativas ao uso de XML para configurar *frameworks*;
- No Java está disponível apenas a partir da versão 5.0.

Hibernate Annotations

- Uso de anotações para mapeamento objeto/relacional com Hibernate;
- Substitui o uso de arquivos HBM XML;
- Segue o padrão definido para anotações de persistência de EJBs versão 3.0;
- Requer as bibliotecas:
 - `hibernate-annotations`;
 - `ejb3-persistence`.

Trocando XML por anotações

- 1) Insira anotações em suas classes de domínio (pode excluir o `.hbm.xml` após este passo);
- 2) Altere `hibernate.cfg.xml` para carregar as próprias classes ao invés dos arquivos XML;
- 3) Obtenha a `SessionFactory` por meio de um `AnnotationConfiguration`.

Grupo de Usuários de Java do Estado do Espírito Santo

Alterações no `hibernate.cfg.xml`

- Usando XML:

```
<mapping  
resource="tutorialhibernate/dominio/Artista.hbm.xml" />
```

- Usando Hibernate Annotations:

```
<mapping class="tutorialhibernate.dominio.Artista" />
```

Obtenção da SessionFactory

- Usando XML:

```
Configuration cfg = new Configuration();  
cfg.configure();  
sessionFactory = cfg.buildSessionFactory();
```

- Usando Hibernate Annotations:

```
Configuration cfg = new AnnotationConfiguration();  
cfg.configure();  
sessionFactory = cfg.buildSessionFactory();
```

Anotando os objetos

- Colocaremos algumas anotações do pacote `javax.persistence` em pontos específicos:
 - Antes da definição da classe;
 - Antes da definição da propriedade (atributo);
 - Antes da definição do método `getPropriedade()`.

```
@Entity
public class Artista {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    public Long getId() { return id; }

    @Column(length = 100)
    public String getNome() { return nome; }

    /* ... */
}
```

Bean de Entidade

- Uma classe persistente é um “Bean de Entidade” (*Entity Bean*);
- Conceito herdado da tecnologia EJB;
- Usa-se a anotação `@Entity` antes da definição da classe:

```
@Entity
public class Artista {

    /* ... */

}
```

Id da entidade

- Ao definirmos uma entidade, devemos especificar qual é o seu identificador;
- Usa-se as anotações `@Id` e `@GeneratedValue`: antes do método *getter* ou da definição do Id:
 - `@GeneratedValue` permite definir uma estratégia (AUTO, IDENTITY, SEQUENCE, TABLE) ou classe;

```
@Id @GeneratedValue(strategy = GenerationType.AUTO)
public Long getId() { return id; }
```

Anotar o campo ou o método *getter*?

- Anotando o campo, Hibernate usará estratégia de acesso `field`, direto na propriedade;
- Anotando o método, Hibernate usará estratégia de acesso `property`, usando *getters* e *setters*;
- Recomenda-se não misturar.

Grupo de Usuários de Java do Estado do Espírito Santo

Propriedades simples

- Várias anotações para propriedades simples:
 - `@Transient`: não será salvo no banco;
 - `@Basic`: números, booleanos, Strings, etc.;
 - `@Temporal`: datas e horas;
 - `@Lob`: texto ou binário grande.
- O *default* é `@Basic`!

```
@Transient
```

```
public String getValor() { return valor; }
```

```
@Basic
```

```
public Double getSalario() { return salario; }
```

```
@Temporal(TemporalType.DATE)
```

```
public Date getDataAdmissao() { return dataAdmissao; }
```

Atributos da coluna

- Podemos determinar características da coluna onde será armazenada uma propriedade;
- Utilizamos `@Column` antes do *getter* ou da propriedade:

```
@Basic  
@Column(length = 50, nullable = false)  
public String getNome() { return nome; }
```

Alterando os valores *default*

- Se não quiser usar os padrões, poderá especificar parâmetros nas anotações:
 - Especificando a tabela:

```
@Entity
@Table(name = "ARTISTAS")
public class Artista { }
```

- Especificando a coluna:

```
@Basic
@Column(name = "NOME_ARTISTA")
public String getNome { return nome; }
```

Mapeando herança

- Uma tabela para cada classe concreta:

```
@Entity
public abstract class Artista { }

@Entity
@Inheritance(
    strategy = InheritanceType.TABLE_PER_CLASS
)
public class Banda extends Artista { }
```

Mapeando herança

- Tabela única para toda a hierarquia:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name = "tipo",
    discriminatorType = DiscriminatorType.CHAR
)
@DiscriminatorValue("A")
public class Artista { }

@Entity
@DiscriminatorValue("B")
public class Banda extends Artista { }
```

Mapeando herança

- Uma tabela para cada classe (*joined*):

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Artista { }
```

```
@Entity
public class Banda extends Artista { }
```

Grupo de Usuários de Java do Estado do Espírito Santo

Superclasses mapeadas

- Não são entidades:
 - Não podem ser armazenadas, recuperadas ou utilizadas em consultas).
- Porém, definem propriedades persistentes que são herdadas pelas subclasses;
- Bom para classes utilitárias.

```
@MappedSuperclass  
public class Artista { }
```

```
@Entity  
public class Banda extends Artista { }
```

Associações

- Há quatro tipos de associação:
 - @OneToOne;
 - @OneToMany;
 - @ManyToOne;
 - @ManyToMany.
- Suas propriedades mais comuns:
 - cascade = CascadeType.____;
 - mappedBy = "____";
 - fetch = FetchType.____;
 - @JoinColumn(nullable="true|false").

Associações – exemplos:

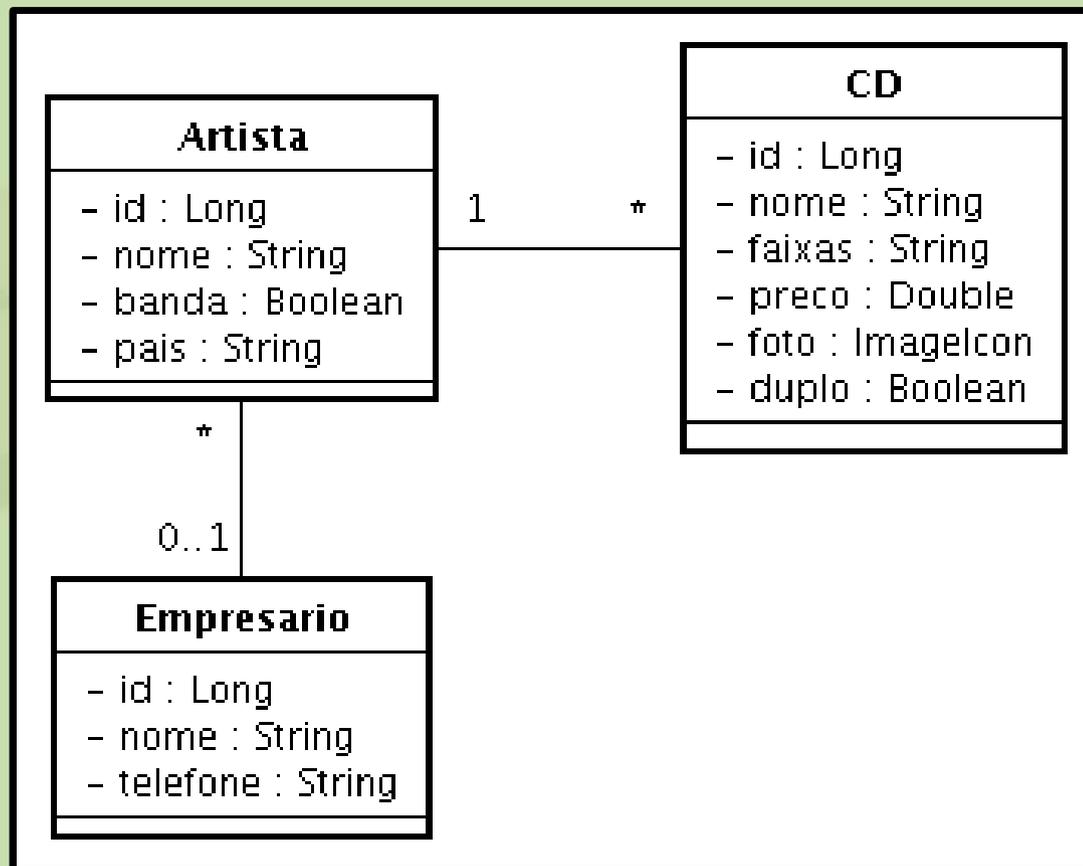
```
public class CD {
    @ManyToOne
    public Artista getArtista() { return artista; }

    @ManyToOne
    public Gravadora getGravadora() { return gravadora; }
}

public class Artista {
    @OneToMany(
        cascade = CascadeType.ALL,
        mappedBy="artista"
    )
    public Set<CD> getCds() { return cds; }
}
```

Exercício

- Faça com que seu exercício da parte III utilize Hibernate Annotations ao invés de XML.



Conclusões da parte IV

- Anotações são meta-dados adicionados às classes Java, disponíveis apenas a partir da versão 5.0;
- Hibernate Annotations permite que troquemos os arquivos XML por anotações padrão EJB 3.0;
- Vimos como mapear todos os conceitos mais simples que aprendemos na parte III, utilizando as anotações do padrão EJB;
- Vimos também as alterações que temos que fazer nas configurações do Hibernate para usar anotações.



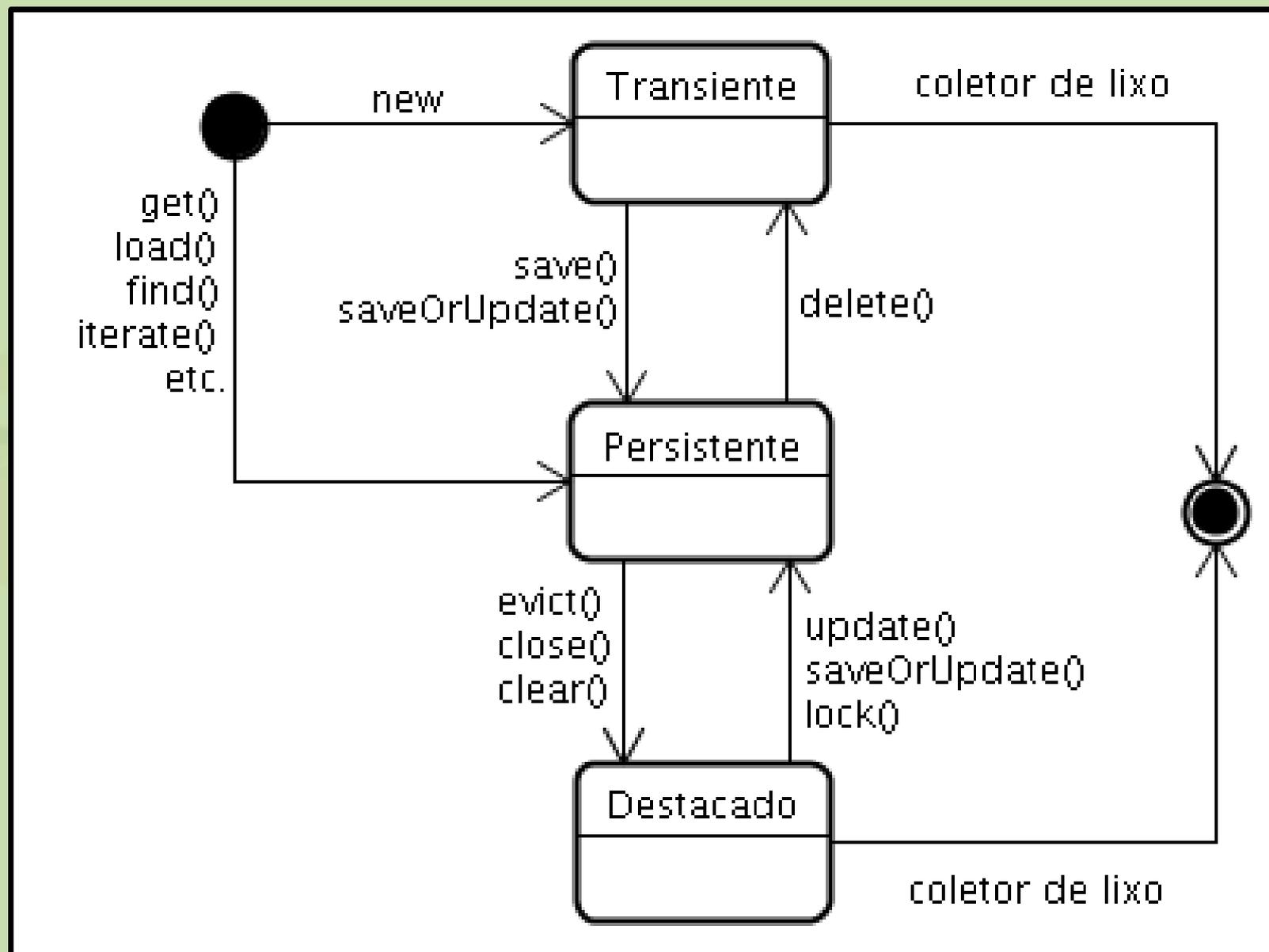
Mapeamento Objeto/Relacional com Hibernate

Parte V: Trabalhando com objetos
persistentes

Interface com a persistência

- Hibernate não é um *container* (usa POJOs);
 - Sua aplicação pode trabalhar com os objetos independente de saber se são persistentes ou não.
- Para efetuar a persistência usamos:
 - Gerenciador de persistência (recuperar por id, salvar e excluir objetos);
 - Interfaces de consulta (recuperar via *query*).
- Antes de vê-los, precisamos conhecer o ciclo de vida e o escopo dos objetos.

Ciclo de vida de objetos no Hibernate



Objetos transientes

- Objetos ainda não passados para o Hibernate, objeto Java normal, morre no GC;
- Não podem ter associações com objetos persistentes (persistência transitiva);
- Não possuem relação alguma com o Hibernate.

Grupo de Usuários de Java do Estado do Espírito Santo

Objetos persistentes

- Possui identidade no banco de dados;
- Objetos salvos ou recuperados do banco pelo Hibernate (diretamente ou por transitividade);
- Estão sempre associados a um objeto `Session`.



Objs. persistentes são transacionais

- Ser associado a uma *session* é participar de uma transação;
- Significa que seu estado é atualizado ao final da transação (ou em sincronizações);
- Somente objetos novos e alterados são atualizados (*dirty checking*);
- Se `dynamic-update = true`, Hibernate atualiza apenas as propriedades alteradas;
- A aplicação não fica sabendo de nada (*transparent transaction-level write-behind*).

Objetos destacados

- Quando uma sessão fecha, seus objetos tornam-se destacados (*detached*):
- Não são mais gerenciados pelo Hibernate:
 - Não há mais garantias que seu estado será sincronizado com o banco de dados.
- Podem ser utilizados se reassociados a uma outra *session*:
 - Cuidado, pois seus dados podem estar defasados;
- Podemos destacar um objeto de uma sessão com o método `evict()`.

Escopo da identidade dos objetos

- Uma solução ORM pode ter como escopo:
 - Nenhum: não há garantias de que ao obter uma linha no SGBD seja retornado sempre o mesmo objeto;
 - Transação: há garantia dentro da transação;
 - Processo: há garantia dentro de toda a JVM (custo muito alto).
- Hibernate tem como escopo a `Session`:
 - Se A e B são da mesma classe, foram recuperados pelo mesmo objeto `Session` e se `A.getId().equals(B.getId())`, então `A == B`;
 - É o chamado “cache de nível 1”.

Reassociação seletiva

- Em aplicações distribuídas (ex.: *Web*), a *session* não pode ficar aberta o tempo todo:
 - É normal obter um objeto numa sessão e depois salvá-lo em outra sessão, por exemplo;
 - O objeto (e todo seu grafo de associações) deve ser reassociado com a segunda sessão.
- Hibernate faz reassociação seletiva:
 - Somente os objetos que interessam são reassociados;
 - Isso é feito de forma automática e eficiente.

Impacto da equivalência de objetos

- `equals()` e `hashCode()` são utilizados em muitas ocasiões (ex.: *API Collections*);
- A implementação padrão destes métodos usa igualdade em memória (`a == b`);
- Hibernate não garante esta igualdade em *sessions* diferentes.

Resultado: problema!

Impacto da equivalência de objetos

- Possíveis soluções:
 - Usar identidade de banco de dados (comparação de Pks): problema com objetos não salvos;
 - Comparar valores das propriedades dos dois objetos: dificuldades para achar valores imutáveis que definam o objeto univocamente (ex.: CPF, ISBN, etc.);
 - Usar um UUID: gerar um identificador único e universal (não há dois objetos com mesmo UUID na mesma JVM).

Impacto da equivalência de objetos

- Uso de UUID:

```
@MappedSuperclass
public abstract class ObjetoPersistente {
    private String uuid;
    private Long id;

    public ObjetoPersistente() {
        uuid = java.util.UUID.randomUUID().toString();
    }

    @Column(nullable = false, length = 40)
    public String getUuid() { return uuid; }
    private void setUuid(String u) { this.uuid = u; }

    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    public Long getId() { return id; }
    private void setId(Long id) { this.id = id; }
```

Impacto da equivalência de objetos

- Uso de UUID:

```
public boolean equals(Object obj) {  
    // Verifica se é da mesma classe.  
    if (! getClass().equals(obj.getClass()))  
        return false;  
    ObjetoPersistente o = (ObjetoPersistente)obj;  
  
    // Compara por UUID.  
    return uuid.equals(o.uuid);  
}  
  
public int hashCode() {  
    return uuid.hashCode();  
}  
}
```

Impacto da equivalência de objetos

- UUID pode ser utilizado como chave-primária?
 - Sim, seria uma chave do tipo *string* com geração do tipo *assigned*.
- Qual o impacto disso no desempenho do SGBD?
 - Pergunte a um DBA!
 - Junções e índices são feitos usando a chave-primária;
 - Intuição diz que chaves numéricas possuem desempenho melhor do que *strings*.

O gerenciador de persistência

- Objeto que provê:
 - CRUD básico;
 - Execução de consultas;
 - Controle de transações;
 - Gerência de *cache*.
- No Hibernate, formado pelos objetos: Session, Query, Criteria e Transaction;
- Uma *session* (objeto leve) é obtida na *session factory* (objeto pesado). Cada *thread* deve obter sua própria *session*.

Persistindo um objeto

- Quatro passos:
 - Crie o objeto;
 - Obtenha um objeto `Session`;
 - Chame o método `.save(obj)` (o comando SQL será enviado em momento oportuno);
 - Feche a sessão.

```
Artista artista = new Artista(); /* ... */  
Session session = HibernateUtil.openSession();  
Transaction tx = session.beginTransaction();  
session.save(artista);  
tx.commit();  
session.close();
```

Atualizando um objeto

- Objetos persistentes são atualizados automaticamente até o final da sessão;
- Objetos destacados precisam ser reassociados:
 - `update(obj)`: força a atualização (envio da SQL);
 - `lock(obj)`: atualização é feita se necessária.

```
// artista foi obtido em outra sessão.  
Session session = HibernateUtil.openSession();  
Transaction tx = session.beginTransaction();  
session.update(artista); // session.lock(artista)  
tx.commit();  
session.close();
```

Salvando um objeto

- Hibernate pode distinguir entre objetos transientes e destacados;
- Método `saveOrUpdate(obj)` insere ou atualiza o objeto no banco de dados;
- A decisão é feita verificando se a chave-primária (ID) é nula;
- Para IDs definidos como tipos primitivos, deve-se especificar seu valor nulo:

```
<id name="id" type="long" unsaved-value="0">  
  <generator class="identity"/>  
</id>
```

Mais sobre persistência transitiva

- Cascadeamento de operações:

none	Não faz nada (<i>default</i>).
save-update	Aplica transitividade ao salvar (inserir ou atualizar).
delete	Aplica transitividade na exclusão.
all	Equivalente à save-update + delete.
delete-orphan	Apaga objetos que deixarem de participar na associação (órfãos).
all-delete-orphan	Equivalente à all + delete-orphan.

Excluindo um objeto

- Excluir um objeto é torná-lo transiente (ele fica na memória até ser coletado);
- Método `delete(class, obj)`;
- Pode ser usado em objetos persistentes ou destacados (ele é associado automaticamente).

```
// artista foi obtido nesta ou em outra sessão.  
Session session = HibernateUtil.openSession();  
Transaction tx = session.beginTransaction();  
session.delete(artista);  
tx.commit();  
session.close();
```

Exceções

- Vários destes métodos podem lançar `HibernateException`;
 - O código de tratamento não tem sido e não será exibido para manter os exemplos mais simples.
- Geralmente erros no SGBD são irrecuperáveis (isto é, obtenha outra conexão);
- A indicação é descartar a *session* e obter outra quando isso acontecer;
- Hibernate faz *rollback* da transação no SGBD, porém não restaura o estado da memória!

Recuperando objetos

- Várias formas:
 - Navegando pelo grafo de objetos de um objeto que já foi recuperado (com a *session* ainda aberta);
 - Obter pelo ID (chave-primária);
 - Usando Hibernate Query Language (HQL);
 - Usando a *API Criteria*;
 - Consulta por exemplos;
 - Usando SQL direto.
- Hibernate utiliza *cache* ao recuperar objetos, evitando acessos desnecessários.

Recuperando objetos

- Recuperar por id:
 - `get(class, obj)`: objeto pode não existir no BD (retorna nulo se não existir) e não usa proxy;
 - `load(class, obj)`: assume-se que o objeto existe (gera erro se ele não existir) e pode usar proxy.

```
Session session = HibernateUtil.openSession();
Transaction tx = session.beginTransaction();
Artista artista = (Artista)session.get(
    Artista.class, new Long(1));
tx.commit();
session.close();
```

Recuperando objetos

- Hibernate Query Language:
 - Linguagem parecida com SQL, só que O.O.;
 - Lembra OQL e EJB-QL 2.1;
 - Foi usada como base para EJB-QL 3.0;
 - Usada somente para recuperação de dados.

```
// Obtenção e fechamento da session não exibidos.  
Query query = session.createQuery("from Artista a  
where a.pais = :pais");  
  
query.setString("pais", "Brasil");  
List result = query.list(); // Lista de Artistas
```

Recuperando objetos

- Exemplos de HQL:

```
-- Recupera todos os artistas:
```

```
from Artista;
```

```
-- Critérios:
```

```
from Artista a where a.nome like 'C%'
```

```
-- Uso de joins:
```

```
from Artista a inner join a.cds as cds
```

```
-- Critérios nas classes associadas:
```

```
from Artista a where a.empresario.nome = 'Fulano'
```

```
-- Ordenação:
```

```
from Artista a order by a.nome
```

Recuperando objetos

- Características da HQL:
 - Sintaxe orientada a objetos;
 - Navegação no grafo de objetos;
 - Consultas polimórficas;
 - Recuperar somente algumas propriedades dos objetos ao invés das entidades inteiras;
 - Ordenação e paginação dos resultados;
 - Funções de agregação, *group-by* e *having*;
 - *Joins* e *subqueries*;
 - Chamada de funções nativas e *stored procedures*.

Recuperando objetos

- Usando a API *Criteria*:
 - Uso de objetos ao invés de *strings*;
 - Mais O.O., menos legível, mais extensível.

```
// Obtenção e fechamento da session não exibidos.  
Criteria criteria =  
session.createCriteria(Artista.class);  
  
criteria.add(Expression.eq("pais", "Brasil"));  
List list = criteria.list(); // Lista de Artistas
```

Recuperando objetos

- Consulta por exemplos:
 - Criação de um critério com base num objeto de domínio semi-preenchido;
 - Bom para buscas com muitas opções.

```
// Obtenção e fechamento da session não exibidos.  
Artista exemplo = new Artista();  
exemplo.setPais("EUA");  
Criteria criteria =  
session.createCriteria(Artista.class);  
  
criteria.add(Example.create(exemplo));  
List list = criteria.list();
```

Estratégia de recuperação

- Quando usamos JDBC, sabemos exatamente quando recuperamos os objetos do banco;
- Um dos principais desafios de uma solução ORM é determinar este momento:
 - Recuperar todos os objetos do grafo de uma vez?
 - Recuperar um pouco de cada vez, sob demanda?
- Esta decisão tem impactos na eficiência e na redigibilidade do código.

Estratégia de recuperação

- Hibernate permite especificar a estratégia no mapeamento e também alterá-la em *runtime*:
 - Recuperação imediata (*immediate*): SELECTs sequenciais;
 - Recuperação preguiçosa (*lazy*): sob demanda;
 - Recuperação adiantada (*eager*): uso de *outer joins*;
 - Recuperação em lote (*batch*): vários de uma só vez.

Estratégia de recuperação

- Especificação no mapeamento depende da cardinalidade (1 ou N);
- Para associações simples (extremidade 1):
 - Utilização de *proxies* (procuradores).

```
<!-- Desabilita por completo recuperação lazy. -->  
<class name="Artista" lazy="false" />
```

```
<!-- Define uma interface que o proxy implementará  
(pode especificar a própria classe). -->  
<class name="Artista" proxy="ArtistaProxy" />
```

Estratégia de recuperação

```
<!-- Uso do proxy definido. -->
<many-to-one name="artista" lazy="proxy" />
<!-- Uso de instrumentação de bytecode. -->
<many-to-one name="artista" lazy="no-proxy" />
<!-- Força recuperação adiantada. -->
<many-to-one name="artista" lazy="false" />

<!-- Lazy se tem proxy, eager se não tem. -->
<many-to-one name="artista" outer-join="auto" />
<!-- Força recuperação adiantada (eager). -->
<many-to-one name="artista" outer-join="true" />
<!-- Força recuperação imediata (immediate). -->
<many-to-one name="artista" outer-join="false" />

<!-- Recuperação em lote. -->
<many-to-one ... lazy="true" batch-size="9" />
```

Estratégia de recuperação

- Para coleções (extremidade N):
 - Coleções sempre possuem *proxy*;
 - Uso de `lazy` e `fetch` no mapeamento.

```
<!-- Recuperação lazy (default e recomendado). -->  
<set ... lazy="true" />
```

```
<!-- Recuperação imediata. -->  
<set ... lazy="false" fetch="select" />
```

```
<!-- Recuperação adiantada. -->  
<set ... lazy="false" fetch="join" />
```

```
<!-- Recuperação em lote. -->  
<set ... lazy="true" batch-size="9" />
```

Estratégia de recuperação

- A recuperação adiantada usa *outer joins*;
- Podemos definir sua profundidade:
 - Indica quantos *joins* serão feitos entre tabelas em uma única instrução SELECT;
 - Default é 1, recomenda-se não passar de 4;
 - Configuração global no `hibernate.cfg.xml`:

```
<!-- Profundidade da recuperação adiantada. -->  
<property name="max_fetch_depth">3</property>
```

Estratégia de recuperação

- Sobre a recuperação preguiçosa:
 - Objetos e coleções são inicializados quando utilizados pela primeira vez;
 - No momento da inicialização, o objeto deve estar associado a uma sessão aberta;
 - Podem ser inicializados manualmente:

```
Hibernate.initialize(artista.getCds());  
  
// Podemos verificar se já inicializou:  
if (Hibernate.isInitialized(artista.getCds())) {  
    // ...  
}
```

Estratégia de recuperação

- Estas configurações são complexas e influenciam bastante o resultado final de sua aplicação;
- Peça ajuda especializada de um DBA para entender os impactos no desempenho;
- Monitore sua aplicação e faça testes com várias configurações para determinar a melhor.

Grupo de Usuários de Java do Estado do Espírito Santo

Conclusões da parte V

- Objetos gerenciados pelo Hibernate possuem ciclo de vida e escopo bem definidos;
- Este escopo gera um problema na equivalência de objetos, com várias soluções possíveis;
- Entendido isso, aprendemos sobre o gerenciador de persistência:
 - CRUD básico;
 - Várias formas de recuperação de objetos.
- A estratégia de recuperação pode ser escolhida, o que é importante e também difícil.

Conclusões da parte básica

- Neste tutorial básico vimos:
 - O que é mapeamento objeto/relacional;
 - Como instalar e configurar o Hibernate;
 - Como se estrutura sua arquitetura interna;
 - Como montar uma camada de domínio;
 - Como mapear classes de domínio para tabelas utilizando XML e anotações;
 - Como salvar, excluir e recuperar objetos utilizando o gerenciador de persistência.