Hibernate

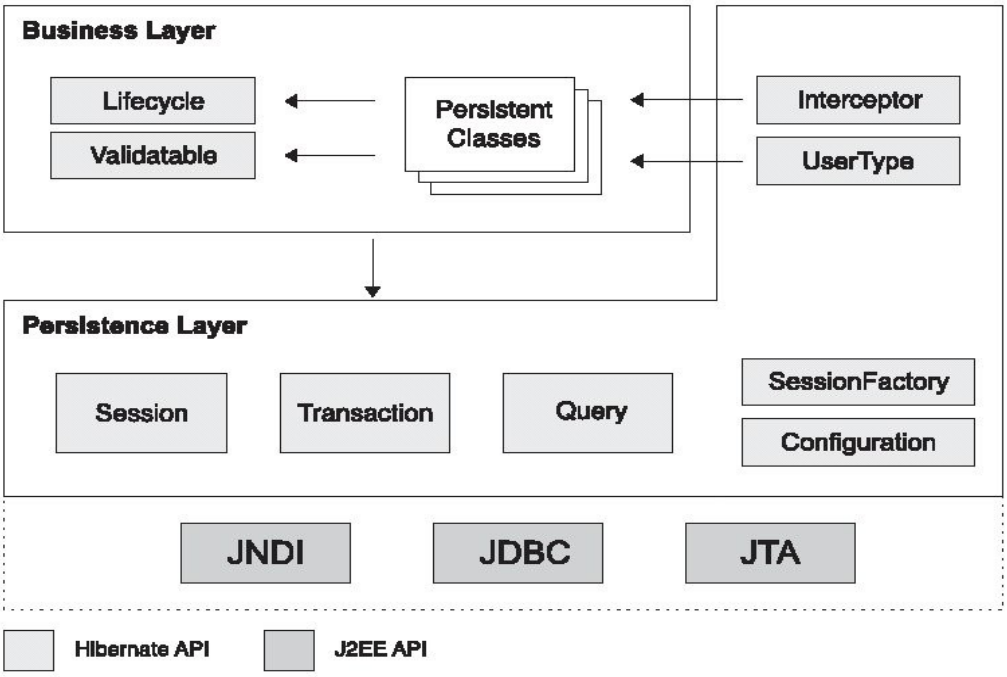
O Hibernate é um framework que facilita as operações de persistência, reduzindo a quantidade de código necessária para a implementação das operações de manipulação de dados. Ele praticamente elimina a necessidade de escrita de sentenças SQL e de manipulação de ResultSets JDBC, com exceção de sistemas que fazem uso extensivo de stored procedures, triggers ou que implementam a maior parte da lógica da aplicação no banco de dados, contando com um modelo de objetos “podre” não vai se beneficiar com o uso do Hibernate. Ele é mais indicado para sistemas que contam com um modelo rico, onde a maior parte da lógica de negócios fica na própria aplicação Java, dependendo pouco de funções específicas do banco de dados.

O Hibernate também evita o trabalho de escrever dúzias de código repetido para fazer as mesmas coisas, como “inserts”, “selects”, “updates” e “deletes” no banco de dados, além de ter duas opções para se montar buscas complexas em grafos de objetos e ainda uma saída para o caso de nada funcionar, usar SQL.

Além do mecanismo de mapeamento objeto/relacional, o Hibernate também pode trabalhar com um sistema de cache das informações do banco de dados, aumentando ainda mais a performance das aplicações. O esquema de cache do Hibernate é complexo e totalmente extensível, existindo diversas implementações possíveis, cada uma com suas próprias características. E junto com isso, ele também em um gerenciador de versões próprio.

Entretanto, como já foi dito, ele não é a solução perfeita para todos os problemas, aplicações que tem uma alta freqüência de “movimentos em massa”, como seqüências de “inserts”, “updates” e “deletes” terminaria por perder performance, graças a instanciação e carregamento de diversos objetos na memória.

O módulo Core é o responsável pela geração do código SQL, encapsulamento das operações JDBC, linguagem de consultas e pela infra-estrutura geral do Hibernate. Este módulo também oferece ao framework configurações de tunning, como arquitetura de cache e suporte a clusters.



Visão geral da API do Hibernate numa arquitetura de camadas

As Interfaces

São várias as interfaces utilizadas em qualquer aplicação que usa Hibernate. Usando estas interfaces, você pode armazenar e recuperar objetos e controlar transações.

Interface Session

A interface Session é utilizada como um cache de objetos carregados em memória e implementa os mecanismos de conexão ao Banco de dados e transações. Uma Session tem o papel de gerenciador de persistência por que ela trata todas as operações de armazenamento e recuperação de objetos.

Interface SessionFactory

As aplicações obtêm instâncias de Session através de uma SessionFactory. Uma SessionFactory é compartilhada por toda a aplicação e deve existir somente uma instância desta, que é criada no momento da inicialização da aplicação. No entanto se a aplicação acessa vários Banco de Dados, deve haver uma SessionFactory por Banco de Dados.

A SessionFactory faz cache para as declarações SQL geradas e outros mapeamentos de meta-dados que o Hibernate utiliza. Ela também mantém em cache os dados que são lidos em uma transação (unidade de trabalho) e que pode ser reutilizada posteriormente.

Interface Configuration

Um objeto Configuration é usado na carga inicial do Hibernate. As Aplicações utilizam instâncias de Configuration para determinar a localização dos mapeamentos de classes e propriedades específicas do Hibernate para que este possa criar uma SessionFactory.

Interface Transaction

Esta é uma API opcional. Aplicações que usam Hibernate podem escolher em não utilizar esta interface, e mapear suas transações em seu próprio código. Uma Transaction é uma abstração de transações JDBC, uma JTA UserTransaction, ou mesmo uma transação CORBA, permitindo que a aplicação controle tudo através de uma API de forma a se tornarem portáveis independendo do tipo de ambiente de execução.

Intefaces Query e Criteria

A interface Query possibilita que sejam efetuadas queries no Banco de Dados e também controlar como a query será executada. As queries são escritas em HQL ou no dialeto SQL do Banco de Dados utilizado. Uma instância de Query é utilizada para ligar os parâmetros as queries, limitar o numero de itens retornados pela query, e finalmente executar a query.

A interface Criteria é muito similar; ela permite que seja criada e executada queries com critérios orientados à objetos.

Interfaces Callback

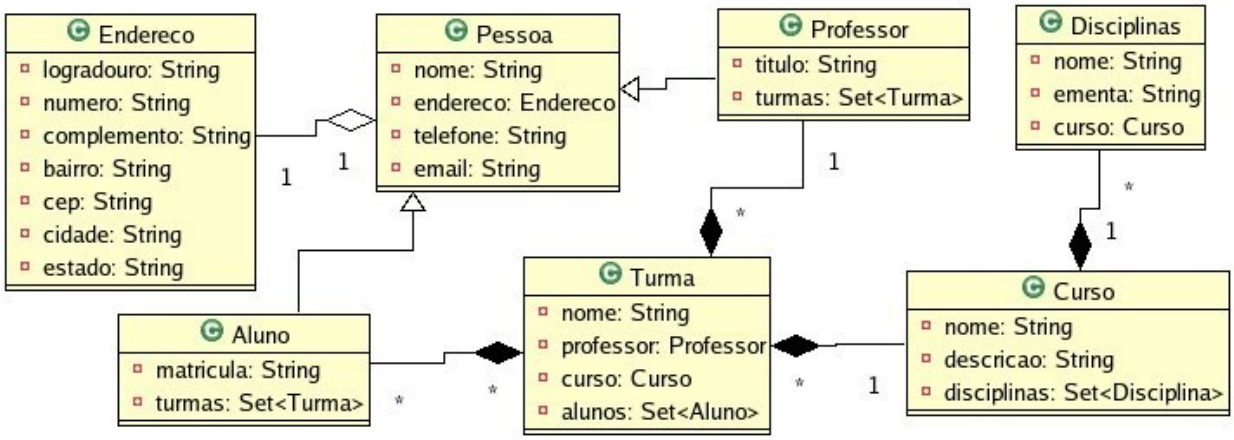
Interfaces Callback permitem a aplicação receber notificações quando algo interessante acontece a um objeto – por exemplo, quando um objeto é carregado, salvo ou deletado. Aplicações Hibernate não necessitam implementar estas rotinas de "callback", mas estas são úteis na implementação de certos tipos de funcionalidades genéricas, tais como a criação de registros de auditoria.

Apredendo Hibernate na prática

Agora vamos conhecer um pouco sobre os mapeamentos do Hibernate e a HQL, a linguagem de buscas do framework.

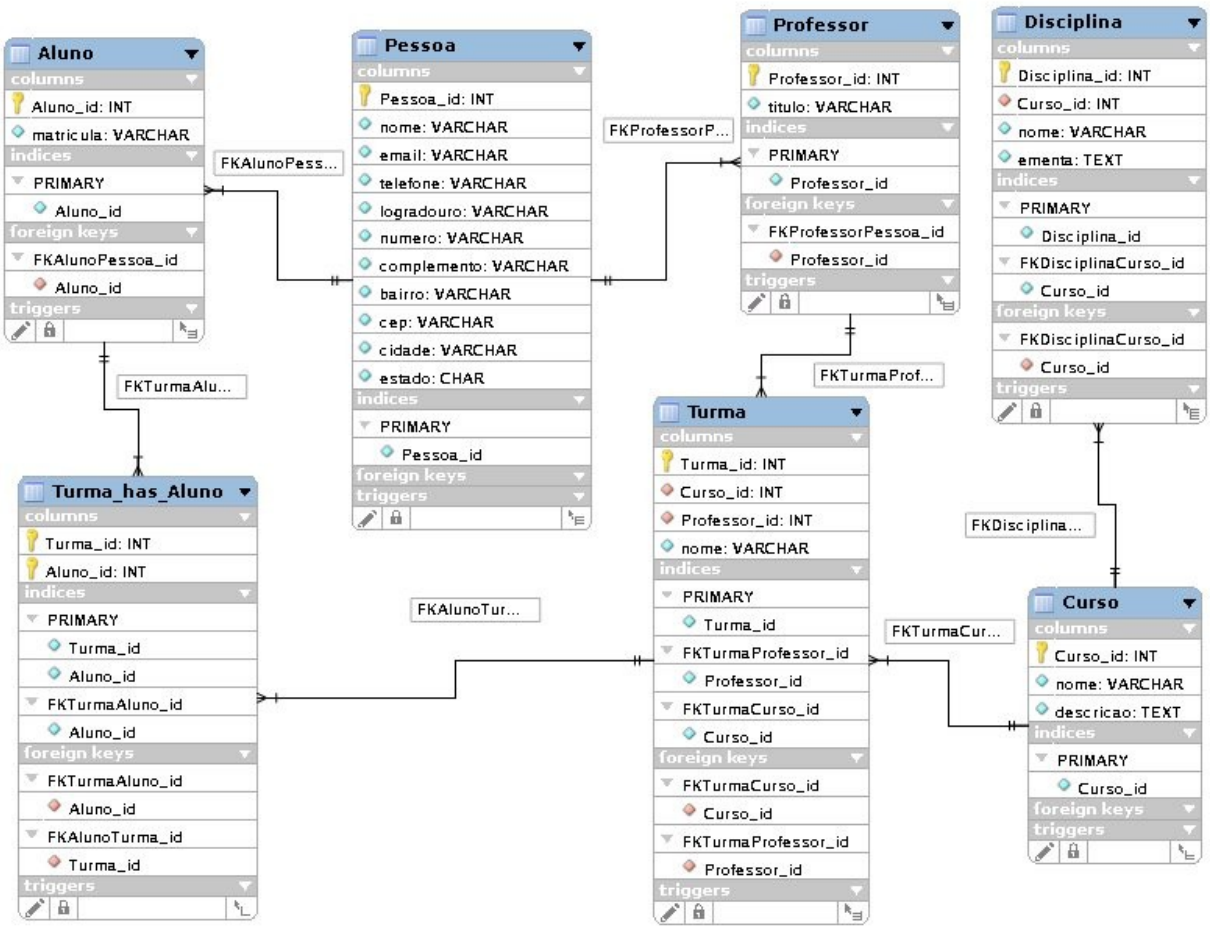
Definindo os objetos do Modelo e as tabelas do Banco de Dados

O modelo é simples, ele modela um cadastro de alunos em uma universidade. Nele nós teremos as classes Pessoa, Aluno, Professor, Endereço, Turma, Disciplina e Curso, que se relacionam como mostrado no diagrama a seguir:



A classe **Pessoa**, que tem um **Endereço**, é a classe base para as classes **Aluno** e **Professor**. Um **Aluno** pode estar em várias Turmas, assim como um **Professor** ministra cursos para várias Turmas. Cada **Turma** pode ter vários **Alunos** e apenas um **Professor**. A **Turma** é formada para o ensino de um **Curso**, que contém várias **Disciplinas**. Várias **Turmas** fodem ser formadas para o ensino de um **Curso**.

Com um modelo de objetos criado, vejamos como essas classes poderiam ser transformadas em tabelas, neste diagrama:



Como você já deve ter percebido, não há nada demais com esse modelo de banco de dados, as tabelas estão interligadas normalmente usando chaves primárias e chaves estrangeiras e uma tabela de relacionamento, no caso da relação N:N entre Aluno e Turma. As tabelas tem as mesmas propriedades das classes e os mesmos relacionamentos.

Agora que já temos as nossas classes e elas foram mapeadas para o banco, podemos começar a trabalhar com os mapeamentos do Hibernate, pra fazer com que aquelas tabelas do banco se transformem nas nossas classes do modelo, quando fizermos acesso ao banco de dados.

Mapeando as classes para as tabelas

Antes de começar a fazer os mapeamentos do Hibernate, temos um conceito do banco de dados que precisa ser revisto, a identidade. Para um banco de dados, o modo de diferenciar uma linha das outras, é usando chaves, de preferência chaves não naturais, como as colunas `id` que nós criamos para nossas tabelas, mas no nosso modelo orientado a objetos, a identidade não é encontrada dessa forma. Em Java, nós definimos a identidade dos objetos sobrescrevendo o método `Object.equals(Object object)` , do modo que nos convier. A implementação `default` deste método, define a identidade através da posição de memória ocupada pelo objeto.

Não podemos usar o método `equals()` no banco de dados, porque o banco de dados não sabe que temos objetos, ele só entende tabelas, chaves primárias e estrangeiras. A solução é adicionar aos nossos objetos um identificador não natural, como os que nós encontramos no banco de dados, porque assim o banco de dados e o próprio Hibernate vão ser capazes de diferenciar os objetos e montar os seus relacionamentos. Nós fazemos isso adicionando uma propriedade chamada `id` do tipo Integer a todas as nossas classes, como no exemplo de código a seguir:

```
//Listagem do arquivo Pessoa.java

public class Pessoa {
  private Integer id;
  private String nome;
  private String email;
  private String telefone;
  private Endereco endereco;
//métodos getters e setters das propriedades
}
```

Poderíamos ter escolhido o tipo primitivo int para a propriedade, mas trabalhando com objetos, o mapeamento e a resolução se um objeto existe ou não no banco de dados torna-se mais simples para o Hibernate. Outra observação importante são os métodos getters e setters para as propriedades dos objetos. O Hibernate não precisa de que as propriedades sejam definidas usando a nomenclatura dos JavaBeans, mas isso já é uma boa prática comum na comunidade, além de existirem outros frameworks e tecnologias que exigem essa nomenclatura (como a Expression Language dos JSPs), então nós definimos métodos getters e setters para todas as propriedades nos nossos objetos.

Conhecendo o arquivo "hmb.xml" e mapeando um relacionamento 1:1

Cortemos a conversa fiada e vamos iniciar a construção dos mapeamentos. O primeiro mapeamento abordado é o da classe Pessoa e do seu relacionamento com a classe Endereço. No nosso modelo, um Endereço tem apenas uma Pessoa e uma Pessoa tem apenas um Endereço (perceba que o arquivo .java e o nome da classe Java que modela Endereço não tem o ç , ficou Endereco, para ficar com o mesmo nome da tabela). Vejamos o mapeamento para a classe Pessoa (o arquivo Pessoa.hbm.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="Pessoa">
    <!-- Identificador da classe -->
    <id name="id" column="Pessoa_id">
      <generator class="identity"/>
    </id>
    <!-- Propriedades da classe -->
    <property name="nome"/>
    <property name="telefone"/>
    <property name="email"/>
    <!-- Relacionamento da classe -->
    <component name="endereco" class="Endereco">
      <property name="logradouro"/>
      <property name="numero"/>
      <property name="complemento"/>
      <property name="bairro"/>
      <property name="cep"/>
      <property name="cidade"/>
      <property name="estado"/>
   </component>
  </class>
```

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

O arquivo de mapeamento é um arquivo XML que define as propriedades e os relacionamentos de uma classe para o Hibernate, este arquivo pode conter classes, classes componentes e queries em HQL ou em SQL. No nosso exemplo, temos duas classes sendo mapeada no arquivo, a classe Pessoa e a classe Endereco (normalmente mapeamos apenas uma classe). O arquivo XML começa normalmente com as definições da DTD e do nó raiz, o **<hibernate-mapping>**, depois vem o nó que nos interessa neste caso, **<class>**.

No nó **<class>** nós definimos a classe que está sendo mapeada e para qual tabela ela vai ser mapeada. O único atributo obrigatório deste nó é name , que deve conter o nome completo da classe (com o pacote, se ele não tiver sido definido no atributo package do nó **<hibernate-mapping>**), se o nome da classe for diferente do nome da tabela, você pode colocar o nome da tabela no atributo table , no nosso exemplo isso não é necessário.

Seguindo em frente no nosso exemplo, temos o nó **<id>** que é o identificador dessa classe no banco de dados. Neste nó nós definimos a propriedade que guarda o identificador do objeto no atributo **name** , que no nosso caso é **id** , se o nome da coluna no banco de dados fosse diferente da propriedade do objeto, ela poderia ter sido definida no atributo **column** . Ainda dentro deste nó, nós encontramos mais um nó, o **<generator>**, este nó guarda a informação de como os identificadores (as chaves do banco de dados) são gerados, existem diversas classes de geradores, que são definidas no atributo **class** do nó, no nosso caso o gerador usado é o **identity** , que suporta colunas de identidade em DB2, MySQL, MS SQL Server, Sybase e HypersonicSQL. O identificador retornado é do tipo `long`, `short` ou `int.`

Os próximos nós do arquivo são os **<property>** que indicam propriedades simples dos nossos objetos, como **Strings**, os tipos primitivos (e seus wrappers), objetos **Date**, **Calendar**, **Locale**, **Currency** e outros. Neste nó os atributos mais importante são **name** , que define o nome da propriedade, **column** para quando a propriedade não tiver o mesmo nome da coluna na tabela e **type** para definir o tipo do objeto que a propriedade guarda. Normalmente, o próprio Hibernate é capaz de descobrir qual é o tipo de objeto que a propriedade guarda, não sendo necessário escrever isso no arquivo de configuração, ele também usa o mesmo nome da propriedade para acessar a coluna, se o atributo não tiver sido preenchido. Nós definimos as três propriedades simples da nossa classe, **nome** , **email** e **telefone** .

O último nó do arquivo, **<component>**, define o relacionamento de 1-para-1 que a classe Pessoa tem com a classe Endereço. Este nó tem os atributos **name** , que define o nome da propriedade no objeto (neste caso, **endereco**), **class** que define a classe da propriedade. Em seguida temos o mapeamento dos atributos da classe Endereco que aparece como um componente para a classe Pessoa mas que terá seus dados armazenados junto dos dados da classe Pessoa.

Mapeando herança

Continuando o trabalho de mapear o nosso modelo para o Hibernate, vamos para a herança que nós encontramos entre Pessoa, Aluno e Professor. Pessoa é a classe pai, da qual Aluno e Professor herdam as propriedades e comportamentos.

No Hibernate existem diversas formas de se fazer o mapeamento de uma relação de herança. Usando uma tabela para cada classe filha (a classe pai não teria tabela, as propriedades comuns se repetiriam nas tabelas filhas), usando uma tabela para todas as classes (discriminadores definiriam quando é uma classe ou outra), usando uma tabela para cada uma das classes (uma para a classe pai e mais uma para cada classe filha) ou até mesmo uma mistura de todas essas possibilidades (disponível apenas na versão 3 do Hibernate). Ficou confuso? Não se preocupe, neste artigo nós vamos abordar apenas a mais comum e mais simples de ser mantida, uma tabela para cada classe.

No nosso banco de dados, temos uma tabela Pessoa e outras duas tabelas, Aluno e Professor, que estão relacionadas a professor través de suas chaves primárias a tabela Pessoa, garantindo assim que não existam Alunos ou Professores com o mesmo identificador. Vejamos então o mapeamento da classe Professor (o arquivo Professor.hbm.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <joined-subclass name="Professor" extends="Pessoa">
    <key column="Professor_id"/>
    <property name="titulo"/>
    <set name="turmas" inverse="true">
      <key column="Professor_id"/>
      <one-to-many class="Turma"/>
    </set>
  </joined-subclass>
</hibernate-mapping>
```

Nesse mapeamento vemos um nó que nós ainda não conhecíamos, **<joined-subclass>**, que indica o mapeamento de herança usando uma tabela para cada classe, porque para retornar o objeto o Hibernate precisa fazer um **join** entre as duas tabelas. O atributo **name** é o nome da classe mapeada e o atributo **extends** recebe o nome da classe pai (neste caso, **Pessoa**), se o nome da classe não fosse igual ao da tabela, poderíamos adicionar o nome da tabela no atributo **table** (que foi omitido porque o nome da classe é igual ao nome da tabela).

O mapeamento continua com a declaração de uma propriedade e com um nó que nós também não conhecemos, **<set>**, que pode simbolizar um relacionamento **1:N** ou **N:N**.

Passando para o mapeamento da classe Aluno, percebemos que não existe muita diferença:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <joined-subclass name="Aluno" extends="Pessoa">
    <key column="Aluno_id"/>
    <property name="matricula"/>
    <set name="turmas" table="Turma_has_Aluno" inverse="true">
      <key column="Aluno_id"/>
      <many-to-many class="Turma"/>
    </set>
  </joined-subclass>
</hibernate-mapping>
```

A classe é declarada do mesmo modo que **Professor**, usando o nó **<joined-subclass>** e usando os mesmos atributos que foram usados no mapeamento anterior. O nó **<key>** também foi incluído do mesmo modo, com o nome da coluna da chave primária/estrangeira da tabela, mais uma declaração de propriedade e mais um nó **<set>**, nada além do que já foi visto.

Mapeando associações 1:N e N:N

Você já conheceu o nó **<set>** nos mapeamentos anteriores, vamos entender agora como ele funciona e como utilizá-lo para tornar o acesso aos objetos associados ainda mais simples, mais uma vez sem nenhuma linha de SQL. Um **set** ou conjunto representa uma coleção de objetos não repetidos, que podem ou não estar ordenados (dependendo da implementação da interface java.util.Set escolhida). Quando você adiciona um nó deste tipo em um arquivo de mapeamento, você está indicando ao Hibernate que o seu objeto tem um relacionamento 1:N ou N:N com outro objeto. Vejamos o exemplo da classe Professor, que tem um relacionamento 1:N com a classe Turma:

```
<joined-subclass name="Professor" extends="Pessoa">
  <key column="Professor_id"/>
  <property name="titulo"/>
  <set name="turmas" inverse="true">
    <key column="Professor_id"/>
    <one-to-many class="Turma"/>
```

```
    </set>
</joined-subclass>
```

No nó **<set>** o primeiro atributo a ser encontrado é **name** que assim como nos outros nós, define o nome da propriedade que está sendo tratada, já o outro atributo, **inverse** , define como o Hibernate vai tratar a inserção e retirada de objetos nessa associação. Quando um lado da associação define o atributo **inverse** para **true** ele está indicando que apenas quando um objeto for inserido do outro lado da associação ele deve ser persistido no banco de dados.

Para entender melhor o atributo **inverse** pense em como está definido o mapeamento da classe **Professor**. Lá ele está definido para **true** , significando que apenas quando uma **Turma** adicionar um **professor**, o relacionamento vai ser persistido no banco de dados. Em código:

```
Professor professor = new Professor();
Turma turma = new Turma();
turma.setProfessor(professor);
professor.getTurmas().add(turma);
```

Se apenas o **Professor** adicionando a **Turma** a sua coleção de **Turmas**, nada iria acontecer ao banco de dados. Isso parece não ter muito sentido, mas se você prestar bem atenção, o Hibernate não tem como saber qual dos dois lados foi atualizado, desse modo ele vai sempre atualizar os dois lados duas vezes, uma para cada classe da relação, o que seria desnecessário. Usando **inverse** você define de qual lado o Hibernate deve esperar a atualização e ele vai fazer a atualização apenas uma vez. Lembre-se sempre de adicionar os objetos dos dois lados, pois em Java os relacionamentos não são naturalmente bidirecionais.

Voltando a o nó **<set>**, percebemos que dentro dele ainda existem mais dois nós, **<key>** e **<one-to-many>**. O nó **<key>** representa a coluna da tabela relacionada (neste caso, Turma) que guarda a chave estrangeira para a classe **Professor**, nós adicionamos o atributo **column** e o **valor** é o nome da coluna, que neste caso é **Professor_id** . No outro nó, **<one-to-many>**, nós definimos a classe a qual pertence essa coleção de objetos, que é **Turma** (lembre-se sempre de usar o nome completo da classe, com o pacote).

Depois do relacionamento um-para-muitos (1:N) vejamos agora como mapear um relacionamento muitos-para-muitos (N:N). Em um banco de dados relacional, este tipo de associação faz uso de uma tabela de relação , que guarda as chaves estrangeiras das duas tabelas associadas, como ocorre no nosso exemplo, onde tivemos que criar uma tabela **Turma_has_Aluno** , para poder mapear o relacionamento N:N entre as classes **Turma** e **Aluno** no banco de dados.

Como você já sabe, este tipo de associação também é definida usando o nó **<set>**, vejamos então o nosso exemplo:

```
<joined-subclass name="Aluno" extends="Pessoa">
  <key column="Aluno_id"/>
  <property name="matricula"/>
  <set name="turmas" table="Turma_has_Aluno" inverse="true">
    <key column="Aluno_id"/>
    <many-to-many class="Turma  column= Turma_id />
  </set>
</joined-subclass>
```

Os atributos do nó **<set>** neste mapeamento são parecidos com os que nós vimos no mapeamento da classe **Professor**, mas eles contém algumas informações adicionais graças ao tipo da associação. Primeiro, temos um novo atributo no nó **<set>**, o table . Como nós havíamos falado antes, para uma associação N:N você precisa de uma tabela de relacionamento e o atributo **table** guarda o nome desta tabela. Mais uma vez o atributo **inverse** foi marcado como **true** indicando que apenas as ações feitas do outro lado da associação vão ser persistidas no banco de dados.

Dentro do nó, nós encontramos mais dois nós, **<key>**, que vai conter, no atributo **column** , o nome da coluna que guarda a chave estrangeira para a tabela **Aluno** e o nó <**many-to-many>**, que contém a classe dos objetos da coleção, no atributo **class** e o nome da coluna na tabela de relacionamento que referencia a chave primária da tabela **Turma**, no atributo **column** .

Vamos agora terminar de mapear as nossas outras classes do nosso modelo e tornar as associações bidirecionais, como manda o nosso modelo de classes. Começando pela classe **Turma**:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="Turma">
    <id name="id">
      <generator class="identity"/>
    </id>
    <property name="nome"/>
    <many-to-one name="professor" class="Professor"
                 column="Professor_id"/>
    <many-to-one name="curso" class="Curso"
                 column="Curso_id"/>
    <set name="alunos" table="Turma_has_Aluno">
      <key column="Turma_id"/>
      <many-to-many class="Aluno" column="Aluno_id"/>
    </set>
```

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

A maior parte do código já é conhecida, mas ainda existem algumas coisas que precisam ser esclarecidas. **Turma** é o lado um de dois relacionamentos um-para-muitos, um com a classe **Professor** e outro com a classe **Cusro**, para tornar essa associação bidirecional nós vamos utilizar o nó <**many-to-one**>, que modela o lado um de uma associação um-para-muitos (1:N). Neste nó, nós inserimos os atributos **name** , que recebe o nome da propriedade, **class** , que recebe o nome da classe da propriedade e **column** , que recebe o nome da coluna nesta tabela que guarda a chave estrangeira para a outra tabela do relacionamento.

O resto do mapeamento é idêntico aos outros mapeamentos que nós já estudamos, a única diferença é o lado da associação que ele está modelando. Os dois outros mapeamentos, de **Disciplina** e **Curso**, não trazem nenhuma novidade para o nosso estudo.

Configurando o Hibernate 3

A engine do Hibernate pode ser configurada de três modos diferentes, instanciando um objeto de configuração (org.hibernate.cfg.Configuration) e inserindo as suas propriedades programaticamente, usando um arquivo .properties com as suas configurações e indicando os arquivos de mapeamento programaticamente ou usando um arquivo XML (o hibernate.cfg.xml) com as propriedades de inicialização e os caminhos dos arquivos de mapeamento. Vejamos como configurar o Hibernate para o nosso projeto:

```
<!DOCTYPE hibernate-configuration PUBLIC
 "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
    </property>
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost/hibernate?autoReconnect=true
    </property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password"/>

    <!-- Condiguração do c3p0 -->
    <property name="hibernate.c3p0.max_size">10</property>
    <property name="hibernate.c3p0.min_size">2</property>
    <property name="hibernate.c3p0.timeout">5000</property>
    <property name="hibernate.c3p0.max_statements">10</property>
    <property name="hibernate.c3p0.idle_test_period">3000</property>
    <property name="hibernate.c3p0.acquire_increment">2</property>

    <!-- Configurações de debug -->
    <property name="show_sql">true</property>
    <property name="hibernate.generate_statistics">true</property>
    <property name="hibernate.use_sql_comments">true</property>
```

```
      <mapping resource="Curso.hbm.xml"/>
      <mapping resource="Disciplina.hbm.xml"/>
      <mapping resource="Turma.hbm.xml"/>
      <mapping resource="Pessoa.hbm.xml"/>
      <mapping resource="Aluno.hbm.xml"/>
      <mapping resource="Professor.hbm.xml"/>
    </session-factory>
  </hibernate-configuration>
```

Na documentação do Hibernate você pode verificar todas as opções de propriedades que podem ser utilizadas e seus respectivos resultados, por isso nós vamos nos ater ao que é importante para começarmos a trabalhar. Vejamos as propriedades:

hibernate.dialect: é a implementação do dialeto SQL específico do banco de dados a ser utilizado.

hibernate.connection.driver_class: é o nome da classe do driver JDBC do banco de dados que está sendo utilizado.

hibernate.connection.url: é a URL de conexão específica do banco que está sendo utilizado.

hibernate.connection.username: é o nome de usuário com o qual o Hibernate deve se conectar ao banco.

hibernate.connection.password: é a senha do usuário com o qual o Hibernate deve se conectar ao banco.

Essa segunda parte do arquivo são as configurações do pool de conexões escolhido para a nossa aplicação. No nosso exemplo o pool utilizado é o C3P0, mas você poderia utilizar qualquer um dos pools que são oferecidos no Hibernate ou então usar um DataSource do seu servidor de aplicação. Na terceira parte, estão algumas opções para ajudar a debugar o comportamento do Hibernate, a propriedade show_sql faz com que todo o código SQL gerado seja escrito na saída default, hibernate.generate_statistics faz com que o Hibernate gere estatísticas de uso e possa diagnosticar uma má performance do sistema e hibernate.use_sql_comments adiciona comentários ao código SQL gerado, facilitando o entendimento das queries.

A última parte do arquivo é onde nós indicamos os arquivos de mapeamento que o Hibernate deve processar antes de começar a trabalhar, se você esquecer de indicar um arquivo de mapeamento de qualquer classe, essa classe não vai poder ser persistida pela engine do Hibernate. Outro detalhe importante, é que quando você usa mapeamentos com Herança, o mapeamento pai deve sempre vir antes do filho.

Entrando em Ação

Agora que você já está com o Hibernate configurado e pronto para funcionar, vamos entender o mecanismo de persistência dele. Para o Hibernate, existem três tipos de objetos, objetos transient (transientes), detached (desligados) e persistent (persistentes). Objetos transient são aqueles que ainda não tem uma representação no banco de dados (ou que foram excluídos), eles ainda não estão sobre o controle do framework e podem não ser mais referenciáveis a qualquer momento, como qualquer objeto normal em Java. Objetos detached têm uma representação no banco de dados,

mas não fazem mais parte de uma sessão do Hibernate, o que significa que o seu estado pode não estar mais sincronizado com o banco de dados. Objetos persistent são os objetos que tem uma representação no banco de dados e que ainda fazem parte de uma transação do Hibernate, garantindo assim que o seu estado esteja sincronizado com o banco de dados (nem sempre, claro).

No Hibernate, assim como no JDBC, existem os conceitos de sessão e transação. Uma sessão é uma conexão aberta com o banco de dados, onde nós podemos executar queries, inserir, atualizar e deletar objetos, já a transação é a demarcação das ações, uma transação faz o controle do que acontece e pode fazer um roolback, assim como uma transação do JDBC, se forem encontrados problemas.

Edite o arquivo de configuração do Hibernate (hibernate.cfg.xml) com as suas informações específicas (nome de usuário, senha, URL de conexão, etc), coloque ele na raiz do seu classpath, junto com as classes compiladas e os arquivos de mapeamento, porque nós vamos colocar o Hibernate pra funcionar (tenha certeza de que o seu classpath está configurado corretamente, com todos os arquivos necessários).

Primeiro, vamos criar uma classe para configurar e abrir as sessões do Hibernate, o código é simples:

```
//Arquivo HibernateUtility.java
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtility {
  private static SessionFactory factory;
  static {
    try {
      factory = new Configuration().configure().
                    buildSessionFactory();
    } catch (Exception e) {
      e.printStackTrace();
      factory = null;
    }
  }

  public static Session getSession() {
    return factory.openSession();
  }
}
```

O bloco estático (as chaves marcadas como static) instancia um objeto de configuração do Hibernate (org.hibernate.cfg.Configuration), chama o método configure() (que lê o nosso arquivo hibernate.cfg.xml) e depois que ele está configurado, criamos uma SessionFactory, que é a classe que vai ficar responsável por abrir as sessões de trabalho do Hibernate. Faça um pequeno teste pra ter certeza de que tudo está funcionando:

```
//Arquivo Teste.java
public class Teste {
  public static void main(String[] args) {
    //Abrindo uma sessão
    Session sessao = HibernateUtility.getSession();

    //Iniciando uma transação
    Transaction transaction = sessao.beginTransaction();

    //Instanciando um objeto transiente
    Curso curso = new Curso();

    //Preenchendo as propriedades do objeto
    curso.setNome("Desenv. de Software");
    curso.setDescricao("Curso só pra programadores");

    //Transformando o objeto transiente em um objeto
    //persistente no banco de dados
    sessao.save(curso);

    //Finalizando a transação
    transaction.commit();

    //Fechando a sessão
    sessao.close();
  }
}
```

Se ele não lançou nenhuma exceção, o seu ambiente está configurado corretamente. Vamos entender agora o que foi que aconteceu neste código, primeiro nós inicializamos a SessionFactory, dentro do bloco estático na classe HibernateUtility, depois que ela é inicializada, o método getSession() retorna uma nova sessão para o código dentro do main(). Após a sessão ter sido retornada, nós iniciamos uma transação, instanciamos um objeto Curso, preenchemos as suas propriedades e chamamos o método save() na sessão. Após o método save(), finalizamos a transação e fechamos a sessão. Acabamos de inserir um registro no banco de dados sem escrever nenhuma linha de SQL, apenas com a chamada de um método.

O código de aplicações que usam o Hibernate costumam mostrar esse mesmo comportamento, abrir sessão, iniciar transação, chamar os métodos save(), update(), get(), delete(), etc, fechar a transação e depois a sessão, um comportamento muito parecido com o de aplicações JDBC comuns, a diferença é que não escrevemos nem uma linha sequer de SQL para tanto.

Fazendo pesquisas no Banco usando o Hibernate

Agora que você já sabe mapear e configurar a fera, podemos passar para uma das partes mais importantes do funcionamento do framework, as pesquisas. Existem três meios de se fazer buscas usando o Hibernate, usando a sua linguagem própria de buscas, a **Hibernate Query Language** (**HQL**), usando a sua **Criteria Query API** (para montar buscas programaticamente) e usando **SQL** puro. A maioria das suas necessidades deve ser suprida com as duas primeiras alternativas, o resto, você sempre pode usar **SQL** pra resolver.

A **HQL** é uma extensão da **SQL** com alguns adendos de orientação a objetos, nela você não vai referenciar tabelas, vai referenciar os seus objetos do modelo que foram mapeados para as tabelas do banco de dados. Além disso, por fazer pesquisas em objetos, você não precisa selecionar as colunas do banco de dados, um select assim: **select * from Turma** em **HQL** seria simplesmente **from Turma** , porque não estamos mais pensando em tabelas e sim em objetos.

A **Criteria Query API** é um conjunto de classes para a montagem de queries em código Java, você define todas as propriedades da pesquisa chamando os métodos e avaliações das classes relacionadas. Como tudo é definido programaticamente, você ganha todas as funcionalidades inerentes da programação orientada a objetos para montar as suas pesquisas e ainda garante a completa independência dos bancos de dados, pois a classe de **dialeto SQL** do seu banco vai se encarregar de traduzir tudo o que você utilizar.

Antes de ver os exemplos propriamente ditos, vejamos como criar um objeto que implemente a interface **Query** ou **Criteria**. Para instanciar um desses objetos, você vai ter que abrir uma sessão do Hibernate, como nós já vimos na listagem do arquivo Teste.java . Vejamos como ficaria o código:

```
Session sessao = HibernateUtility.getSession();
tx = sessao.beginTransaction();
Query select = sessao.createQuery("from Turma");
List objetos = select.list();
System.out.println(objetos);
tx.commit();
sessao.close();
```

O código ainda segue aquela mesma seqüência da listagem anterior, abrir uma sessão, iniciar uma transação e começar a acessar o banco de dados. Para criar uma query, nós chamamos o método ***createQuery(String query)*** na sessão, passando como parâmetro o **String** que representa a query. Depois disso, podemos chamar o método ***list()***, que retorna um **List** com os objetos resultantes da query. Usando a **Criteria API**, o código ficaria desse modo:

```
Session sessao = HibernateUtility.getSession();
Transaction tx = sessao.beginTransaction();
Criteria select = sessao.createCriteria(Turma.class);
List objetos = select.list();
System.out.println(objetos);
tx.commit();
sessao.close();
```

Veja que apenas a linha onde nós criávamos a query mudou. Agora, em vez de chamar o método ***createQuery(String query)***, nós chamamos o método ***createCriteria(Class class)***, passando como parâmetro a classe que vai ser pesquisada, que no nosso caso é Tuma.

Outro complemento importante do Hibernate é o suporte a paginação de resultados. Para paginar os resultados, nós chamamos os métodos ***setFirstResult(int first)*** e ***setMaxResults(int max)***. O primeiro método indica de qual posição os objetos devem começar a ser carregados, o segundo indica o máximo de objetos que devem ser carregados. Estes métodos estão presentes tanto na interface Query quanto na interface Criteria . Vejamos como nós deveríamos proceder para carregar apenas as dez primeiras turmas do nosso banco de dados:

```
Session sessao = HibernateUtility.getSession();
Transaction tx = sessao.beginTransaction();
Criteria select = sessao.createCriteria(Turma.class);
select.setFirstResult(0);
select.setMaxResults(10);
List objetos = select.list();
System.out.println(objetos);
tx.commit();
sessao.close();
```

Veja que nós chamamos os métodos ***setFirstResult()*** e ***setMaxResults()*** antes de listar os objetos da query e lembre-se também que a contagem de resultados (assim como os arrays) começa em zero, não em um.

Uma propriedade específica da **HQL**, que foi herdada do **JDBC**, é o uso de parâmetros nas queries. Assim como no **JDBC**, os parâmetros podem ser numerados, mas também podem ser nomeados, o que simplifica ainda mais o uso e a manutenção das queries, porque a troca de posição não vai afetar o código que as usa. Vejamos um exemplo do uso de parâmetros:

```
Session sessao = HibernateUtility.getSession();
Transaction tx = sessao.beginTransaction();
Query select = sessao.createQuery(
                "from Turma as turma where turma.nome = :nome");
select.setString("nome", "Jornalismo");
List objetos = select.list();
System.out.println(objetos);
tx.commit();
sessao.close();
```

Nesse nosso novo exemplo, tivermos mais algumas adições a query **HQL**. A primeira é o uso do "**as**" , que serve para apelidar uma classe na nossa expressão (do mesmo modo do "**as**" em **SQL**), ele não é obrigatório, o código poderia estar **from Turma turma ...** e ele funcionaria normalmente. Outra coisa a se notar, é o acesso as propriedades usando o operador "**.**" (ponto), você sempre acessa as propriedades usando esse operador. E a última parte é o parâmetro propriamente dito, que deve ser iniciado com "**:**" (dois pontos) para que o Hibernate saiba que isso é um parâmetro que vai ser inserido na query. Você insere um parâmetro nomeado, usando o método **set** correspondente ao tipo de parâmetro, passando primeiro o nome com o qual ele foi inserido e depois o valor que deve ser colocado.

E se você também não gosta de ficar metendo código **SQL** no meio do seu programa, porque deveria meter código **HQL**? O Hibernate facilita a externalização de queries **HQL** (e até **SQL**) com os nós ***<query>*** e ***<sql-query>*** nos arquivos de mapeamento. Vamos adicionar uma query no mapeamento da classe turma (o arquivo Turma.hbm.xml):

```
<query name="buscarTurmasPeloNome">
    <![CDATA[from Turma t where t.nome = :nome]]>
</query>
```

Essas linhas adicionadas no arquivo de mapeamento vão instruir o Hibernate a criar um **PreparedStatement** para esse select, fazendo com que ele execute mais rápido e possa ser chamado de qualquer lugar do sistema. Você deve preencher o atributo **nome** com o nome pelo qual esta query deve ser chamada na aplicação e no corpo do nó você deve colocar o código da query, de preferência dentro de uma tag CDATA, pra evitar que o parser XML entenda o que está escrito na query como informações para ele. Veja como executar uma **named query** :

```
Session sessao = HibernateUtility.getSession();
Transaction tx = sessao.beginTransaction();
Query select = sessao.getNamedQuery("buscarTurmasPeloNome");
select.setString("nome", "Jornalismo");
List objetos = select.list();
System.out.println(objetos);
tx.commit();
sessao.close();
```

A única diferença para as outras listagens é que em vez de escrevermos a query dentro do código Java, ela ficou externalizada no arquivo de mapeamento. Para instanciar o objeto, chamamos o método ***getNamedQuery(String name)***, passando como parâmetro o nome da query definido no atributo **nome** no arquivo XML.

Adicionando restrições as pesquisas

Vamos ver agora como adicionar restrições as nossas pesquisas no Hibernate. Os exemplos vão seguir uma dinâmica simples, com uma versão em HQL e como o mesmo exemplo poderia ser feito usando a API de Criteria.

A HQL e a API de Criteria suportam todos os operadores de comparação da SQL (**<**, **>**, **<>**, **<=**, **>=**, **=**, **between**, **not between**, **in** e **not in**), vejamos um exemplo em HQL:

```
from Aluno al where al.endereco.numero >= 50
```

O uso dos operadores de comparação segue a mesma sintaxe da linguagem SQL, usando Criteria, essa mesma query poderia ser expressa da seguinte maneira:

```
Criteria select = sessao.createCriteria(Aluno.class);
select.createCriteria("endereco")
        .add( Restrictions.ge( "numero", new Integer(10) ) );
```

Nós criamos a o objeto Criteria usando a classe Aluno, mas a nossa restrição é para a propriedade numero da classe Endereco que está associada a classe Aluno, então temos que descer um nível, para poder aplicar a restrição a propriedade numero de endereco . Usando Criteria, você pode usar os métodos estáticos da classe **org.hibernate.criterion.Restrictions** para montar expressões.

Os operadores lógicos (e os parênteses) também estão a sua disposição e em HQL eles continuam com a mesma sintaxe do SQL. Você pode usar **or**, **and** e parênteses para agrupar as suas expressões. Vejamos um exemplo:

```
from Endereco end where (
   end.rua in ( Epitácio Pessoa ,  Ipiranga ) ) or (
   end.numero between 1 and 100 )
```

Passando para Criteria:

```
Criteria select = sessao.createCriteria(Endereco.class);
String[] ruas = {"Epitácio Pessoa", "Ipiranga"};
select.add( Restrictions.or(
    Restrictions.between("numero", new Integer(1),
    new Integer(100) ),
    Restrictions.in( "rua", ruas ) ) );
```

O código usando Criteria é muito mais longo do que a expressão em HQL e também é menos legível. Procure utilizar Criteria apenas quando for uma busca que está sendo montada em tempo de execução, se você pode prever a busca, use named queries e parâmetros para deixar o eu código mais limpo e mais simples.

Outra parte importante das pesquisas são as funções de pesquisa em Strings. Você pode usar "**LIKE**" e os símbolos "**%**" e "_" , do mesmo modo que eles são utilizados em SQL. Vejamos um exemplo:

```
from Curso c where c.nome like  Desenvolvimento%
```

Em Criteria:

```
Criteria select = sessao.createCriteria(Curso.class);
select.add( Restrictions.like("nome",
                              "Desenvolvimento",
                              MatchMode.START) );
```

Além disso, você ainda pode ordenar as suas pesquisas, em ordem ascendente ou descendente. Como nos exemplos a seguir:

```
from Aluno al order by al.nome asc
```

Em Criteria:

```
Criteria select = sessao.createCriteria(Aluno.class);
select.addOrder( Order.asc("nome") );
```

JPA – Java Persistence API

O Java Persistence API (JPA), é parte integrante da especificação JAVA EE 5 Enterprise Jaba Beans (EJB) 3.0, e simplifica em grande parte os esforços necessários à persistencia de Objetos em Java oferecendo um mecanismo de mapeameto (Objeto-Relacional) através do qual declarações (Annotations) são utilizadas para determinar como os objetos Java serão armazenados em uma tabela de um Banco de Dados Relacional.

Na especificação JPA, o javax.persistense.EntityManager é o serviço central para todas as ações de persistência. As Entidades são objetos Java comuns que são alocados como qualquer outro objeto. Eles não serão persistidos até que seja chamado explicitamente o EntityManager para efetuar a persistência deles. O EntityManager gerencia o mapeamento O/R entre um conjunto fixo de entidades de classes e suas correspondentes fontes de dados. Ele oferece APIs para a criação de queries, localização de objetos, sincronização de objetos, e inserção de objetos em Banco de Dados. Ele tamtém pode oferecer cache e gerenciar as interações entre as entidades e os serviços de transações do ambiente Java EE tais como JTA. O EntityManager está muito integrado com o Java EE e EJB mas não está limitado apenas a este ambiente; ele tamtém pode ser utilizado em ambientes Java SE.

Atualmente o Hibernate.Annotations e Hibernate.EntityManager oferecem suporte ao JPA para o ambiente Java SE e o Oracle Toplink Essetials para o ambiente Java EE.

Entidades são POJOs

Entidades, na especificação JPA, são “plain old Java objects” (POJOs) ou seja classes Java com atributos privados e métodos get/set para manipular seus valores. Instancias de uma classe não será persistida até que seja associada com um EntityManager. Por exemplo, você lembra da classe Pessoa usado no exemplo do Hibernate:

```
import java.io.Serializable;
import javax.persistence.*;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public abstract class Pessoa implements Serializable {
  private Integer id;
  private String nome;
  private Endereco endereco;
  private String telefone;
  private String email;
```

```
    @Id
    @GeneratedValue
    @Column(name="Pessoa_id")
    public Integer getId() {
      return id;
    }

    public void setId(Integer id) {
      this.id = id;
    }
  ...
    @Embedded
    public Endereco getEndereco() {
      return endereco;
    }
  ...
}
```

A anotação **@Entity** define que a classe **Pessoa** é uma entidade e que utilizará os serviços JPA. Como parte da especificação uma entidade deve implementar a interface ***Serializable***. A anotação **@Inheritance** descreve que serão utilizadas uma tabela para cada sub-classe na herança que for estabelecida entre qualquer classe e a classe **Pessoa**. As Anotações **@Id** e **@GeneratedValue** determinam que o atributo ***id*** da classe **Pessoa** é a chave primária na tabela do DB correspondente e que seu valor será gerado pelo DB, já a anotação **@Column** identifica o nome da coluna na tabela associado ao atributo ***id***. Por fim a anotação **@Embedded** indica que o atributo ***endereco*** é uma classe associada chamada **Endereco**.

A classe **Endereco** deverá ser anotada conforme segue:

```
  @Embeddable
  public class Endereco implements Serializable {
    ...
  }
```

Assim se complementa com a definição em **Pessoa**.

O Mapeamento de Herança segue o mesmo conceito das configurações feitas previamente:

```
  @Entity
  @PrimaryKeyJoinColumn(name="Aluno_id", referencedColumnName =
                    "Pessoa_id")
  public class Aluno extends Pessoa implements Serializable {
    ... }
```

A anotação **@PrimaryKeyJoinColumn** informa que o nome da chave primária da tabela de **Aluno** é ***Aluno_id*** na associação de herança entre **Aluno** e **Pessoa**.

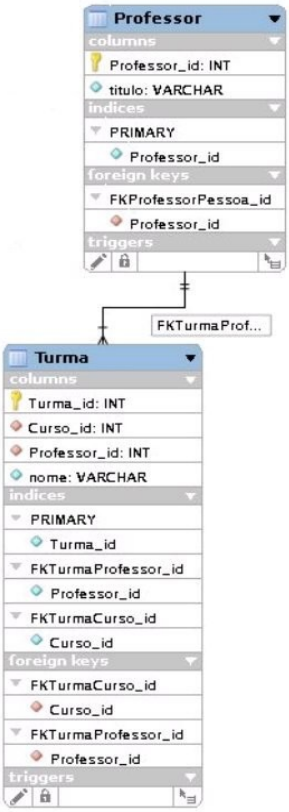
Em associações 1:N como entre **Professor** e **Turma** temos do lado (1) na classe **Professor**:

```
@OneToMany(mappedBy="professor")
public java.util.Set<Turma> getTurmas() {
  return turmas;
}
```

E no lado (N) na classe **Turma**:

```
@ManyToOne
@JoinColumn(name="Professor_id", referencedColumnName =
                "Professor_id")
public Professor getProfessor() {
  return professor;
}
```

As anotações se complementam, no lado da classe **Professor** está indicado que o atributo ***turmas*** é uma associação do tipo 1:N e que o lado N está mapeado na classe **Turma** no atributo ***professor***. Já na classe **Turma** a anotação **@ManyToOne** indica que o atributo ***professor*** é o lado N na associação e que através da anotação **@JoinColumn** está identificado que os campos das tabelas onde serão efetuadas as associações chaman-se ***Professor_id*** em ambas as tabelas ***Turma*** e ***Professor***.



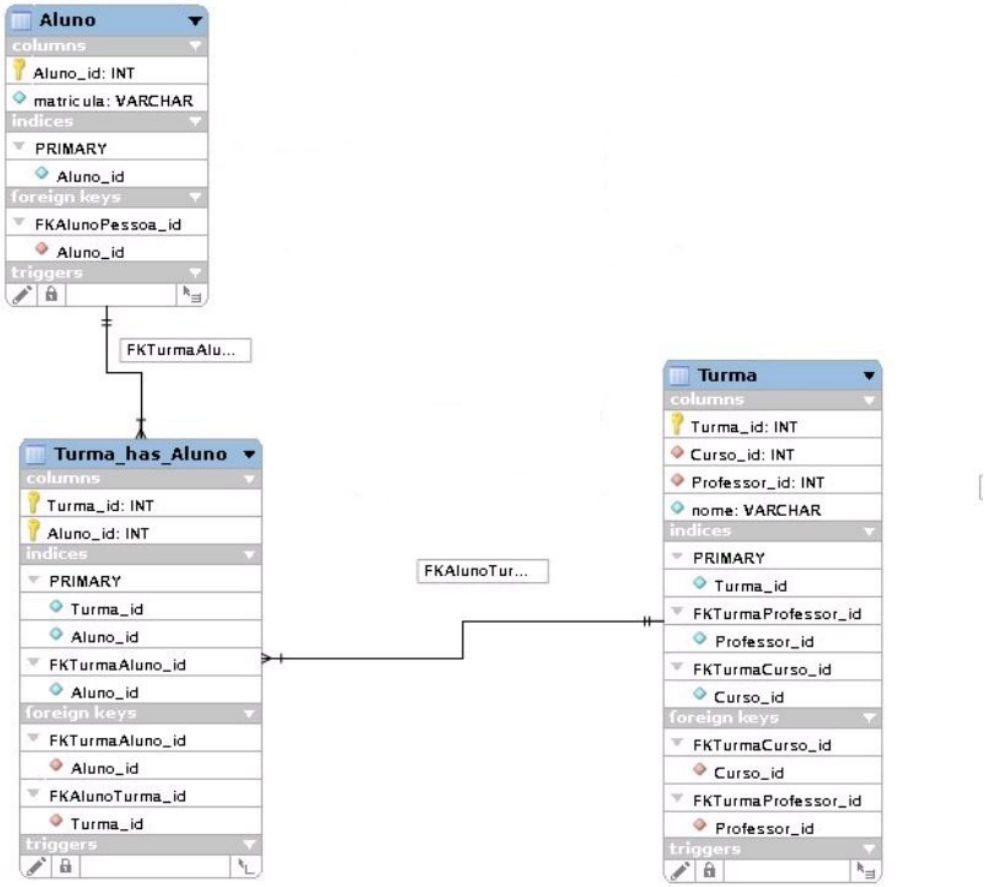
Nas associações N:N como entre **Aluno** e **Turma** temos do lado da classe **Aluno**:

```
@ManyToMany(mappedBy="alunos")
public java.util.Set<Turma> getTurmas() {
  return turmas;
}
```

Na classe **Turma**:

```
@ManyToMany
@JoinTable(name="Turma_has_Aluno",
     joinColumns = @JoinColumn(name = "Turma_id",
               referencedColumnName = "Turma_id"),
     inverseJoinColumns = @JoinColumn(name = "Aluno_id",
               referencedColumnName = "Aluno_id"))
public java.util.Set<Aluno> getAlunos() {
  return alunos;
}
```

Em **Aluno** a anotação **@ManyToMany** informa que o atributo ***turmas*** é uma associação N:N entre **Aluno** e **Turmas** com o atributo ***alunos***. Na classe **Turma** a mesma anotação é seguida de **@JoinTable** que determina como estão associadas as tabelas.



Para efetuar as operações de consulta, inclusão, manutenção e deleção nas entidades basta utilizar todas as rotinas já apresentadas anteriormente no tema **Hibernate**. Os únicos passos a serem tomados são modificar a classe **HibernateUtility** conforme segue:

```
import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateUtility {
  private static final SessionFactory factory;
    static {
      try {
       factory = new AnnotationConfiguration().
          configure("/META-INF/hibernate.cfg.xml").
          buildSessionFactory();
      } catch (Throwable ex) {
          // Log exception!
          throw new ExceptionInInitializerError(ex);
      }
    }
  public static Session getSession() throws HibernateException {
    return factory.openSession();
  }
}
```

Como o arquivo ***hibernate.cfg.xml*** segue a mesma estrutura não iremos falar a respeito dele.

Falta configurar o arquivo **persitence.xml** que deve ser salvo no diretório **META-INF** . Em seu conteúdo definimos o nome das classes que são as entidades para o JPA e configurações para o provedor de persistência.

No caso de utilizarmos com o Java SE e Hibernate apenas informamos a localização do arquivo hibernate.cfg.xml, conforme segue:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-
                  instance"
             version="1.0"
             xsi:schemaLocation="http://java.sun.com/xml/ns/pers
                  istence
                 http://java.sun.com/xml/ns/persistence/persiste
                  nce_1_0.xsd">
  <persistence-unit name="Escola"
        transaction-type="RESOURCE_LOCAL">
```

```
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property name="hibernate.ejb.cfgfile"
                value="/META-INF/hibernate.cfg.xml"/>
    </properties>
  </persistence-unit>
</persistence>
```

O resto é por conta do Hibernate.