Índice geral

PREFÁCIO

Esta apostila é um resumo da apostila encontrada no site oficial do hibernate (http://www.hibernate.org/).

Os exemplos apresentados utilizam o plugin do eclipse chamado Hibernate Synchronizer, ele pode ser encontrado no site http://www.binamics.com/hibernatesynch/.

1.Introdução

Hibernate é um Framework que nos ajuda a manipular dados em um banco de dados relacional. Ele trabalha com o conceito de persistência Objeto/Relacional, isso significa que para cada tabela do banco de dados, existe uma classe que a representa. Essas classes são beans, com métodos get e set, podendo ter relacionamentos de herança, polimorfismo (ex: classe pessoa pode ser tanto pessoa jurídica como pessoa física), associação (um usuário possui vários grupos) e composição (uma pessoa possui um endereço, onde esse endereço possui nome da rua,número e vila).

O hibernate possui também um mecanismo de SQL orientado a objeto (chamado de HQL – Hibernate Query Language), isso significa que podemos utilizar uma sintaxe bem parecida com o SQL que estamos acostumados, obtendo como resultado um objeto com os dados já setados, agilizando assim nosso trabalho, pois não é mais preciso obter os dados por uma query SQL para só depois setarmos esses dados em um bean, para que possamos exibi-los em nossos JSP's.

Hibernate suporta os principais banco de dados, cito abaixo alguns exemplos:

- PostgreSQL
- Progress
- MySQL

- Oracle
- Firebird
- DB2

Os bancos citados acima são alguns exemplos, o hibernate suporta vários outros. (Ver apostila do hibernate)

2.Arquitetura

Abaixo segue um diagrama (Fig.01) bem simplificado mostrando a estrutura de uma aplicação utilizando o hibernate.

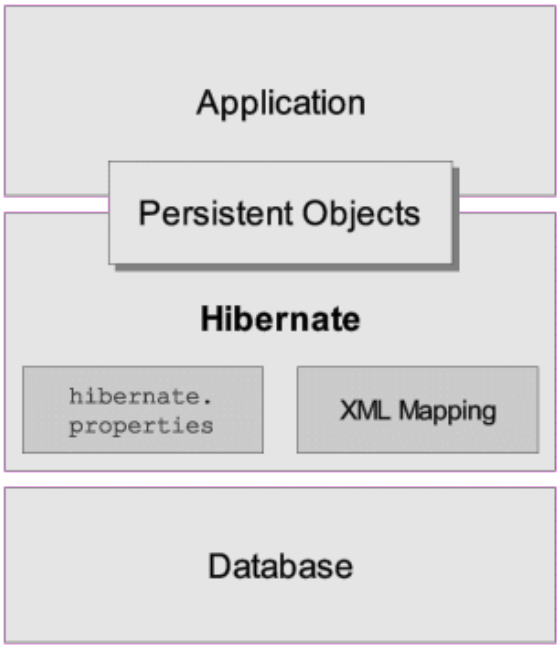


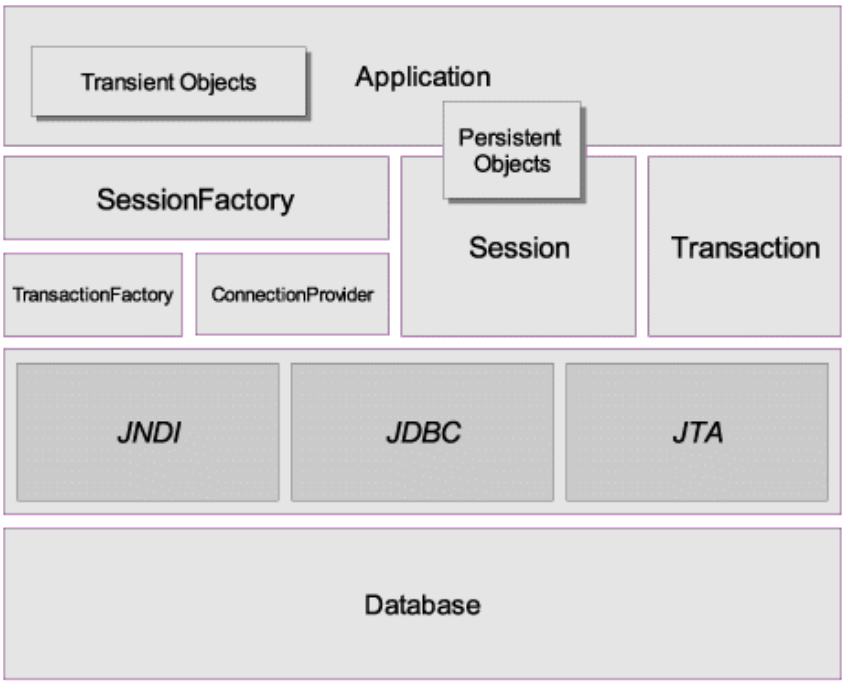
Fig.01

Nota-se que o hibernate é uma camada que se localiza entre o nosso banco de dados e a aplicação. Ele fornece serviços de persistencia de dados e objetos persistentes (objetos que representam dados do banco de dados).

O hibernate trabalha com arquivos .properties e XML's. Esses arquivos são usados para a configuração de acesso ao banco de dados e mapeamento das classes para as tabelas (isso será mostrado mais adiante).

A figura a seguir (Fig.02) mostra em mais detalhes a estrutura do hibernate.

Fig.02



Segue uma breve definição dos objetos que compõe o hibernate.

- SessionFactory (net.sf.hibernate.SessionFactory)
  Uma fábrica de objetos da classe net.sf.hibernate.Session. É um cliente de um Connection Provider.
  Pode-se instanciar um SessionFactory para aplicação toda, pois ele só é utilizado para obter objetos da classe net.sf.hibernate.Session.

- Session (net.sf.hibernate.Session)
  Os objetos da classe session possuem um clico de vida curto. São eles que fazem o acesso ao banco de dados, ou seja, fazem a conversação entre o banco de dados e a nossa aplicação. São fabricas de objetos da classe net.sf.hibernate.Transaction.

- Objetos persistentes (Persistent Objects)
  Os objetos persistentes possuem um clico de vida curto. Eles representam dados persistidos no banco de dados. Eles possuem associação com somente uma session.

- Objetos transientes (Transient Objects)
  São objetos que não possuem associação com nenhum session, isso ocorre porque o objeto não foi persistido no banco de dados.

- Transaction (net.sf.hibernate.Transaction)
  São objetos com ciclo de vida curto, eles são usados para especificar unidades atômicas de trabalho.
  Uma session pode conter várias transactions.

- ConnectionProvider
  (net.sf.hibernate.connection.ConnectionProvider)

Uma fabrica para conexões (e pools) para o JDBC.

- TransactionFactory (net.sf.hibernate.TransactionFactory)
  Fábrica de transaction.

3.Configuração da SessionFactory

O hibernate utiliza um arquivo .properties ou XML para configurar o acesso ao banco de dados.

Irei mostrar um arquivo de configuração em XML e explicar as principais propriedades.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

<hibernate-configuration>
          <session-factory>
                    <!-- local connection properties -->
                    <!-- Datasource utilizado para a conexao com o BD -->
                    <property name="hibernate.connection.datasource">
                              java:comp/env/jdbc/datasourceHibernate
                    </property>
                    <!-- dialect for PostgreSQL -->
                    <!-- Indica para o hibernate o banco de dados usado -->
                    <property name="dialect">
                              net.sf.hibernate.dialect.PostgreSQLDialect
                    </property>
                    <property name="hibernate.show_sql">true</property>
                    <property name="hibernate.use_outer_join">true</property>

                    <!-- Indicando os arquivos que fazem o mapeamento
Objeto/Relacional -->
                    <mapping resource="com/tabela/Pessoa.hbm" />

          </session-factory>
</hibernate-configuration>
```

- hibernate.connection.datasource
  Indica qual datasource que será utilizado. Esse datasource é configurado também por um arquivo XML que se localiza no diretório TOMCAT/WEBAPPS do Tomcat versão 4.1.x.

- dialect
  Dialect do banco de dados que será utilizado. Cada banco de dados possuí um dialect.

- hibernate.show_sql
  Escreve todos os Sql's executados pelo hibernate no console.

- hibernate.use_outer_join
  Indica se existe ou não o relacionamento de OUTER JOIN entre as tabelas do banco de dados.

- <mapping resource>

Arquivos (.hbm) que fazem o mapeamento entre os objetos (que representam as tabelas) e as tabelas do bando de dados. ATENÇÃO: É OBRIGATÓRIO A INDIÇÃO DOS ARQUIVOS .HBM, SENÃO OCORRERÁ ERRO QUANDO OS OBJETOS REFERÊNCIADOS PELO ARQUIVO .HBM SEREM USADOS.

As principais propriedades são essas, existem várias outras. Para maiores informações consultar o capítulo 3 da apostila do hibernate.

3.1. Obtendo objetos da classe Configuration,SessionFactory e Session.

Segue um exemplo para instanciar um objeto da classe net.sf.hibernate.cfg.Configuration (Objeto que contem as configurações do hiberbate para acesso ao banco).

```
Configuration configuration =
          new Configuration().configure(new File("/usr/hibernate/WEB-
INF/classes/com/hibernate.cfg.xml"));
```

O caminho indicado como parâmetro da classe File é o caminho absoluto até o arquivo hiberbate.cfg.xml (arquivo de configuração descrito no capítulo 3).

Segue abaixo um exemplo melhorado da trecho anterior. Aqui utiliza-se a classe Properties para que o caminho do arquivo hibernate.cfg.xml não esteja hard codificado.

```
Properties properties = new Properties();

/*Carregando o arquivo do properties*/
properties.load((HibernateUtil.class).getResourceAsStream("/com/conf/hibernate.propertie
s"));
Configuration configuration =
          new Configuration().configure(new
File(properties.getProperty("HIBERNATE_CFG_PATH")));
```

Com posse de um objeto Configuration, pode-se obter uma SessionFactory (classe que fornece objetos da classe session).

```
SessionFactory factory = configuration.buildSessionFactory();
```

E com posse da SessionFactory, pode-se obter uma Session (classe que é utilizada para fazer INSERT, DELETE E UPDATE no banco de dados).

```
Session session = factory.openSession();
```

4. Mapeamento dos Objetos para as tabelas do banco de dado.

Esse capítulo irá abordar sobre os arquivos (.hbm) que fazem o mapeamento entre as classes e as tabelas (arquivos

Objeto/Relacional). Iremos utilizar aqui o plugin Hibernate Synchronizer que nos ajudará a criar tais arquivos e as classes que representaram as tabelas.

Com as tabelas criadas no banco de dados, podemos criar os arquivos Objeto/Relacional (Hibernate Mapping File), abaixo seguem os passos:

- No seu eclipse, clique em FILE->NEW->OTHER->HIBERNATE-> HIBERNATE MAPPING FILE.(Fig.03)

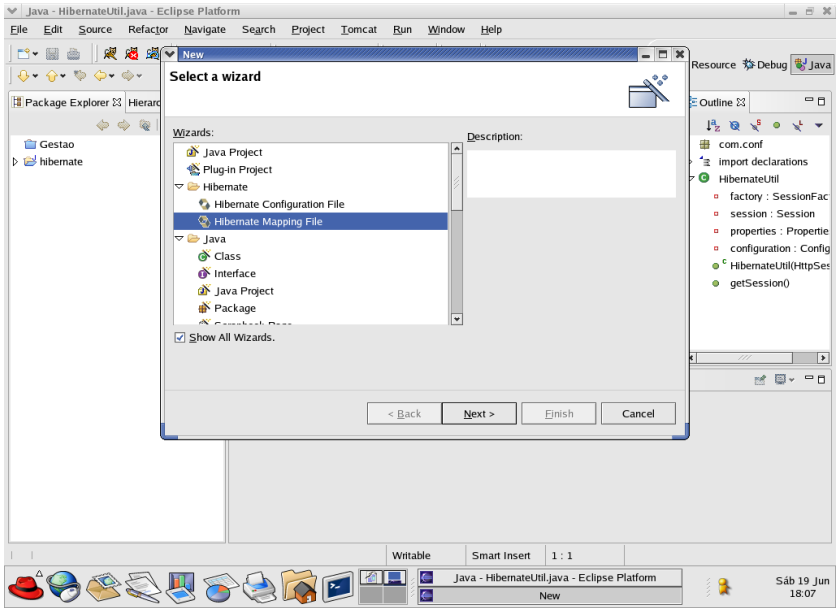


Fig.03

- A seguinte tela será mostrada.(Fig.04)

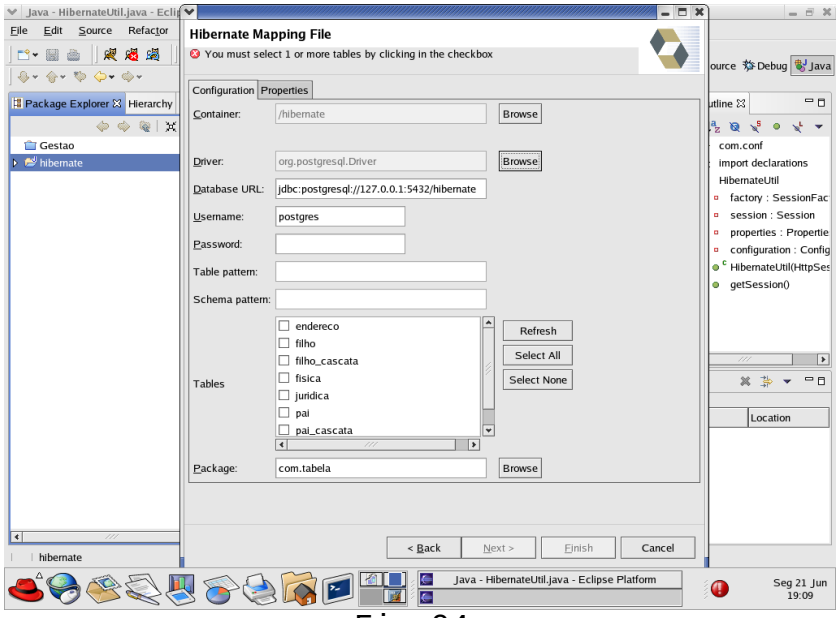


Fig.04

- Deverá ser escolhido o driver para a conexão com o banco de dados, para isso clique no botão “Browse” ao lado do campo “Driver”. Ex: Para usar o driver do postgres, deve-se colocar o driver jdbc no path da aplicação. Para escolher o drive do postgres, clique em “Browse” ao lado do campo “Driver”. Aparecerá uma nova janela, escreva no campo “Select entries” a palavra driver. Aparecerá no campo “Macthing types” algumas palavras, selecione a palavra “Driver”. No campo “Qualifier” aparecerá o driver do postgres(org.posttgresql). Seleciona-o e clique em “Ok”.(Fig.05)

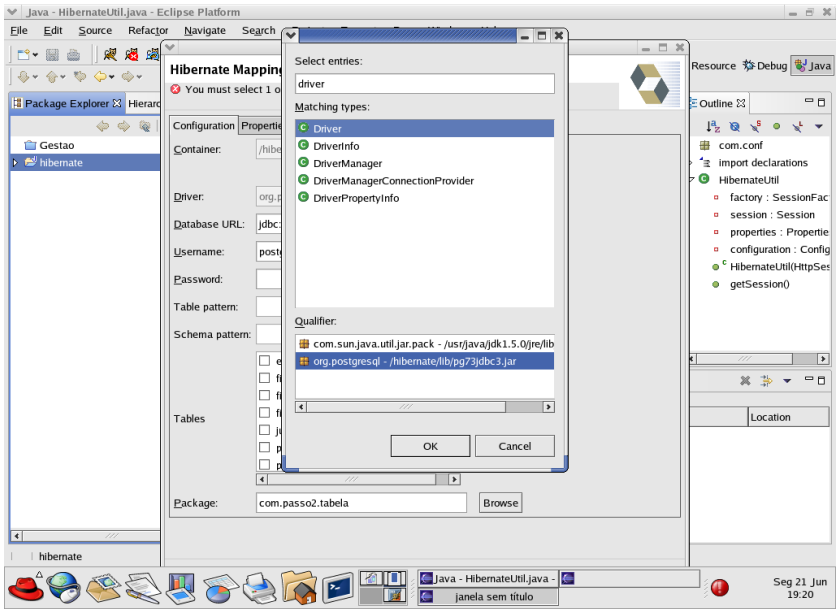


Fig.05

- No campo “user” e “password” escreva o usuário e a senha para a conexão com o banco de dados.
- Clique sobre o botão “Refresh” ao lado do campo “Tables” para visualizar as tabelas do banco de dados. Obs: Caso queira, poderá ser informado o campo “Table Pattern” e/ou “Schema”, exibindo assim somente as tabelas que contenham o nome informado no campo “Table Pattern” e/ou as tabelas que estejam no esquema informado no campo “Schema”.
- Selecione as tabelas que deseja criar os arquivos de mapeamento Objeto/Relacional do banco.
- Informe o caminho aonde serão criados os arquivos de mapeamento no campo “Packge”.
- Clique sobre o botão “Finish”.
- Obs: Outras propriedades poderam ser informadas/configuradas, para isso clique sobre a aba properties. Na tela que será exibida, poderá ser informado a extenção do arquivo de mapeamento, composição do nome de ID, como será gerada o id da classe (chave primária) entre outras propriedades.

Segue um exemplo do arquivo de mapeamento gerado.

```
<?xml version="1.0"?>
```

```
<!DOCTYPE hibernate-mapping PUBLIC
          "-//Hibernate/Hibernate Mapping DTD//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd" >

<hibernate-mapping package="com.passo1.tabela">
          <class name="Pessoa" table="passo1.pessoa">
                    <id
                              column="id"
                              name="id"
                              type="java.lang.Long"
                    >
                              <generator class="vm" />
                    </id>
                    <property
                              column="nome"
                              name="nome"
                              not-null="true"
                              type="string"
                     />
                    <property
                              column="idade"
                              length="4"
                              name="idade"
                              not-null="false"
                              type="java.lang.Long"
                     />
          </class>
</hibernate-mapping>
```

4.1. **Elementos do arquivo de mapeamento**

Este capítulo abordará sobre os principais elementos do arquivo Objeto/Relacional(arquivo que faz o relacionamento entre as classes persistêntes e as tabelas do banco de dados).
Obs: O plugin Hibernate Synchronizer já cria quase todos os elementos automaticamente. Esse capítulo só consta aqui para tirar eventuais dúvidas ou para arrumar algum elemento que o plugin mapeou erraneamente da tabela para a classe.

Segue um exemplo de um arquivo Objeto/Relacional com as propriedades que serão abordadas nesse capítulo.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat" table="CATS" discriminator-value="C">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <discriminator column="subclass" type="character"/>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true" update="false"/>
                <property name="weight"/>
```

```
                <many-to-one name="mate" column="mate_id"/>
                <set name="kittens">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>
                <subclass name="DomesticCat" discriminator-value="D">
                        <property name="name" type="string"/>
                </subclass>
        </class>

        <class name="Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

- Hibernate-mapping

Esse elemento possui uma árvore de atributos opcionais.

```
<hibernate-mapping
      schema="schemaName"                           (1)
      default-cascade="none|save-update"            (2)
      auto-import="true|false"                      (3)
      package="package.name"                        (4)
/>
```

1- Schema (opcional): Nome do esquema das tabelas. Se essa propriedade é informada, isso significa que todas as tabelas desse mapeamento se encontram num único esquema.
2- default-cascade (opcional): Indica o estilo de cascateamento padrão para as operações (insert,delete e update) com as tabelas que possuem relacionamento. Caso as tabelas não indiquem o estilo de cascateamento, será assumido o estilo que estiver indicado nessa propriedade.
3- auto-import (opcional, padrão é true) : Indica que podemos usar unqualified class names (nomes das classes sem o caminho completo do pacote) para as querys.
4- package (opcional): Especifica um pacote para o unqualified class names assumirem.

Obs: Se a aplicação conter duas classes persistêntes com o mesmo (unqualified) nome, a propriedade auto-import deverá ser false, caso contrário o hibernate lançará uma exceção quando essas classes forem utilizadas num mesmo SQL ou numa mesma classe.

- Class

Pode-se declarar uma classe persistênte com o elemento class.

```
<class
       name="ClassName"                             (1)
       table="tableName"                            (2)
       discriminator-value="discriminator_value"    (3)
       mutable="true|false"                         (4)
       schema="owner"                               (5)
       proxy="ProxyInterface"                       (6)
       dynamic-update="true|false"                  (7)
```

```
        dynamic-insert="true|false"                    (8)
        select-before-update="true|false"              (9)
        polymorphism="implicit|explicit"               (10)
        where="arbitrary sql where condition"          (11)
        persister="PersisterClass"                     (12)
        batch-size="N"                                 (13)
        optimistic-lock="none|version|dirty|all"       (14)
        lazy="true|false"                              (15)
/>
```

1– Name: Nome completo da classe ou interface. Obs: Caso o elemento hibernate-mapping tenha a propriedade package, não é preciso informar o nome completo da classe.
2– table: Nome da tabela do bando de dados que será representada pela classe. No postgresql, aqui deverá ser informado o esquema mais nome da tabela. Ex: esquema1.pessoa.
3– discriminator(opcional – padrão é o nome da classe): Nome que distingue uma subclasse de outra, usado no relacionamento de herança.
4- mutable(opcional, padrão true): Especifica que a instância da classe pode(ou não)ter seus valores alterados, isso significa que podemos fazer ou não update ou delete na tabela que a classe representa.
5- schema(opcional): Sobrescreve o nome do esquema especificado no elemento <hibernate-mapping>.
6- proxy(opcional): Especifica uma interface para ser usada na lazy inicializing.
7- dynamic-update(opcional, padrão é false): Especifica que o UPDATE SQL deverá ser executado em tempo de execução.
8- dynamic-insert(opcional, padrão false): Especifica que o INSERT SQL deverá ser executado em tempo de execução.
9- select-before-update(opcional, padrão false): Especifica que o hibernate nunca deverá executar um UPDATE SQL a menos que ele tenha certeza que o objeto tenha sido alterado. O hibernate só executará o UPDATE SQL quando o método update() da classe session for explicitamente chamado.
10-polymorphism(opcional, padrão implicit): Determina se a query utiliza polimorfismo explicitamente ou implicitamente.
11-where(opcional): Especifica uma clausula WHERE para ser executada quando objetos de uma classe forem obtidos.
12-persister(opcional): Especifica um classPersister.
13-batch-size(opcional, padrão 1): Especifica um “batch-size” para a busca de instâncias dessa classe pelo identificador.
14-optimistic-lock(opcional, padrão version): Especifica uma estratégia de optimistic locking.
15-lazy(opcional): Setando como true, isso é um atalho equivalente a setar o próprio nome da classe com sendo um proxy interface.

É perfeitamente aceitável declarar uma interface como sendo uma classe persistente, podendo declarar subclasses que implementam a interface. Subclasses será visto nesse capítulo.

Classes imutáveis, mutable=false, não podem sofrer as operações de UPDATE e DELETE. O hibernate aceita isso para otimizar a performance.

A propriedade opcinal proxy, ativa o lazy initialization.

Essa inicialização é utilizada em relacionamento one-to-many e many-to-many, onde um objeto possui uma lista de objetos de outra classe. Ex: Um pai pode possuir vários filhos, tendo aqui um relacionamento one-to-many, onde um objeto da classe pai pode possuir uma lista de objetos da classe filho.

A utilização do lazy initialization acontece quando, por exemplo, obtemos um objeto da classe pai. O hibernate apenas inicializa os dados do pai, e não inicializa os dados dos filhos. Os dados dos filhos só seram obtidos quando os filhos forem invocados, por exemplo, quando o método getFilhos, que retorna uma lista dos filhos, for chamado. Sem o lazy initialization, os dados do pai e de seus filhos seriam obtidos de uma única vez. Dependendo dos casos o lazy inicialization poderá aumentar a performance da aplicação. Para mais informações consultar a apostila do hibernate capítulo 14.

Polimorfismo implícito, polimorphism=”implicit”,que é o padrão, é o mais utilizado. O polimorfismo explícito é util quando duas classes persistentes mapeiam uma mesma tabela.

O atributo persister permite especificar uma estratégia de persistencia no banco de dados. Poderíamos, por exemplo, criar uma subclasse da classe net.sf.hibernate.persister.EntityPersister ou criar uma nova implementação da interface net.sf.hibernate.persister.ClassPersister, podendo chamar uma stored procedures direto do banco de dados.

Note que dynamic-update e dynamic-insert não são herdados para as subclasses, isso significa que as classes especificadas no elemento <subclass> ou <joined-subclass> não herdam as duas propriedade.

O uso de selected-before-update geralmente diminui a performance. Ele é util para previnir triggers desnecessárias em operações feitas no banco de dados.

Se ativar o optismic lock, pode-se escolher entre essas estratégias:

- Version: Faz a checagem por uma coluna versão/timestamp.
- All: Checa todas as colunas.
- Durty: Checa as colunas que tiveram seu valor modificado.
- None: Não usa optismic lock.

É fortemente recomendável usar a estratégia de versão(version), porque é essa a estratágia que possui mais performance e é a única que consegue detectar modificações feitas fora da sessão(ex:Quando o método Session.update() é usado).Tenha em mente que a coluna utilizada pela estratégia de versão não deve ser nula.

- **ID**

Todo arquivo objeto/relacional (arquivo de mapeamento), **DEVE** declarar a chave primária de uma tabela do banco de dados, e isso é feito utilizando o elemento ID.

```
<id
        name="propertyName"                 (1)
        type="typename"                     (2)
        column="column_name"                (3)
```

```
        unsaved-value="any|none|null|id_value"   (4)
        access="field|property|ClassName">       (5)

        <generator class="generatorClass"/>       (6)
</id>
```

1-name(opcional): Um nome que identuifique o ID
2-type(opcional): Um hiberbate type equivalente ao tipo da coluna da tabela.
3-column(opcional, padrão é o conteúdo da propriedade name): O nome da coluna da chave primária da tabela.
4-unsaved-value(opcional, padrão é null): Um valor que distingue que a classe está persistida (salva) ou não. Distingue se é uma instancia transiente (transient instances) que foi salvo ou carregado por uma sessão anterior.
5-access(opcional, padrão property): Estratégia que o hibernate deverá usar para acessar o valor da propriedade.
6-genetator: Uma classe deve ser indicada para a geração de identificadores únicos para as instâncias da classe. Para maiores informações consultar a apostila do hibernate capítulo 5.1.4.1.

Se a o atribute name não for fornecido, o hibernate assume que a classe não possui um ID.

Caso queira usar chaves primárias compostas, poderá ser usado o elemento <composite-id>, mas de acordo com a apostila do hibernate, essa tática deve ser desencorajada.

- **Composite-id**

```
<composite-id
        name="propertyName"
        class="ClassName"
        unsaved-value="any|none"
        access="field|property|ClassName">

        <key-property name="propertyName" type="typename" column="column_name"/>
        <key-many-to-one name="propertyName class="ClassName" column="column_name"/>
        ......
</composite-id>
```

Para tabelas que possuem chaves primárias compostas, pode-se usar o elemento composite-id para mapear essas chaves para serem indentificadores da classe persistente. Esse propriedade composite-id possui os elementos <key-property> que faz o mapeamento das chaves da tabela e o elemento <key-many-to-one> que mapeia os elementos como elementos filhos.

A classe persistente deve subscrever os métodos equals() e hashCode() para poder implementar a igualdade do composite id (para poder usar o método equals devidamente).

Para podermos carregar os dados de uma classe que possui composite-id, devemos primeiramente setar seus identificadores antes de usar o método session.load().

Existe uma alternativa mais apropriada para a criação de composite-id, declarando ela como sendo um componente da classe. Para maiores informações consultar capítulo 7 (Components) da

apostila do hibernate.

- Discriminator

O elemento discriminator deve ser usado para o caso de polimorfismo. O hibernate utilizará o valor da coluna indicada na propriedade column da TAG discriminator para poder instanciar a subclasse.

```
<discriminator
    column="discriminator_column"  (1)
    type="discriminator_type"      (2)
    force="true|false"             (3)
/>
```

1-column(opcional, padrão class): Nome da coluna discriminadora (discriminator column).
2-type(opcional, padrão string): Nome que indica o Hibernate type.
3-force(opcional, padrão null): “Força” o hibernate a especificar um valor para o discriminator quando tenta-se obter todas as instancias da classe pai (classe root).

Os valores que a propriedade discriminator pode assumir são especificados na propriedade <discriminator-value> nos atributos <class> e <subclass>.

O atributo force somente é util quando existir uma coluna na tabela que não é mapeada para as classes persistentes.

- Version

A propriedade <version> é opcional e indica que a tabela contem versioned data. Essa propriedade é especialmente útil para aplicações que utilizam transação com longa duração.

```
<version
    column="version_column"                  (1)
    name="propertyName"                      (2)
    type="typename"                          (3)
    access="field|property|ClassName"        (4)
    unsaved-value="null|negative|undefined"  (5)
/>
```

1-column(opcional, padrão o nome da propriedade): Nome da coluna para obter o número da versão.
2-name: O nome da propriedade da classe persistente.
3-type(opcional, padrão integer): O tipo do número da versão.
4-access(opcional, padrão property): A estratégia que o hibernate deverá utilizar para obter o valor da propriedade.
5-unsaved-value(opcional, padrão undefined): Um valor que distingue que a classe está persistida (salva) ou não. Distingue se é uma instancia transiente (transient instances) que foi salvo ou carregado por uma sessão anterior.

- **Timestamp**

A propriedade <timestamp> é opcional e indica que a tabela contêm timestamped data. Essa propriedade é uma alternativa para a propriedade version. Timestamp é a implementação menos segura de optimistic locking.

```
<timestamp
        column="timestamp_column"           (1)
        name="propertyName"                 (2)
        access="field|property|ClassName"   (3)
        unsaved-value="null|undefined"      (4)
/>
```

1-column(opcional, padrão o nome da propriedade): Nome da coluna para obter o timestamp.
2-name: O nome da propriedade da classe persistente que deverá ser do tipo Date ou Timestamp.
3-access(opcional, padrão property): A estratégia que o hibernate deverá utilizar para obter o valor da propriedade.
4-unsaved-value(opcional, padrão undefined): Um valor que distingue que a classe está persistida (salva) ou não. Distingue se é uma instancia transiente (transient instances) que foi salvo ou carregado por uma sessão anterior.

Note que <timestamp> é equivalente à <version type="timestamp">

- **Property**

Property declara uma propriedade para a classe persistente.

```
<property
        name="propertyName"                 (1)
        column="column_name"                (2)
        type="typename"                     (3)
        update="true|false"                 (4)
        insert="true|false"                 (4)
        formula="arbitrary SQL expression"  (5)
        access="field|property|ClassName"   (6)
/>
```

1-name: Nome da propriedade, sua inicial deverá começar com letra minúscula.
2-column(opcional, padrão nome da propriedade): O nome da coluna que a propriedade está mapeando.
3-type(opcional): Um nome que indique um tipo do hibernate.
4-update,insert(opcional, padrão true): Especifica que a coluna deverá ser incluída nos SQL UPDATE e INSERT. Setando essas propriedades para false, seus valores deverão ser inicializados a partir de uma outra propriedade que mapeia a mesma coluna, ou a partir de uma trigger ou uma outra aplicação.
5-formula(opcional): Uma expressão SQL para definir um valor para uma propriedade computada (computed property). Uma propriedade computada não possui uma coluna mapeada para a tabela do banco de dados.

6-access(opcional, padrão property): A estratégia que o hibernate deverá utilizar para obter o valor da propriedade.

O typename poderá ser:

1. O nome de um tipo básico do hibernate(ex: integer, string, character, date, timestamp, float, binary, serializable, object, blob).
2. O nome de uma classe java com um tipo básico padrão (int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob).
3. O nome de uma subclasse de PersistentEnum.
4. O nome de uma classe java que implemente a interface Serialazible.
5. O nome de uma classe de um tipo customizado (ex:com.meusTipos.MyCustomType)

Se não for especificado um tipo, o hibernate tentará encontrar um tipo apropriado para o property. Ele usará as regras 2,3 e 4 (nessa ordem) para especificar o tipo. Em certos casos a especificação do tipo será obrigatório, como por exemplo para especificar um tipo hibernate.DATE e hibernate.TIMESTAMP.

- **Many-to-one**

Uma associação many-to-one com outra classe persistente é declarado utilizando a propriedade many-to-one.

```
<many-to-one
        name="propertyName"                                 (1)
        column="column_name"                                (2)
        class="ClassName"                                   (3)
        cascade="all|none|save-update|delete"               (4)
        outer-join="true|false|auto"                        (5)
        update="true|false"                                 (6)
        insert="true|false"                                 (6)
        property-ref="propertyNameFromAssociatedClass"      (7)
        access="field|property|ClassName"                   (8)
/>
```

1-name: O nome da propriedade.
2-column: O nome da coluna.
3-class(opcional, padrão property type determinado por reflexão): O nome da classe que possui a associação.
4-cascade(opcional): Especifica qual operação deve ser cascateada da classe pai para as classes que possuem relacionamento com ela.
5-outer-join(opcional, padrão auto): Ativa o outer-join fetching para as associações quando hibernate.use_outer_join é setado.
6-update, insert(opcional, padrão true): Especifica que a coluna deverá ser incluída nos SQL UPDATE e INSERT. Setando essas propriedades para false, seus valores deverão ser inicializados a partir de uma outra propriedade que mapeia a mesma coluna, ou a partir de uma trigger ou uma outra aplicação.

7-property-ref(opcional): O nome da propriedade da classe associada que contêm a FK. Se não for informado, o ID da classe associada será utilizada.
8-access(opcional, padrão property): Estratégia que o hibernate deverá usar para acessar o valor da propriedade.

O outer-join aceita três valores, sendo eles:

- Auto(padrão): Utiliza o outer join se a classe associada não possuir um proxy.
- True: Sempre utiliza o outer join.
- False: Nunca utiliza o outer join.

- **One-to-one**

Uma associação one-to-one com outra classe persistente é declarado utilizando a propriedade one-to-one.

```
<one-to-one

        name="propertyName"                                 (1)
        class="ClassName"                                   (2)
        cascade="all|none|save-update|delete"               (3)
        constrained="true|false"                            (4)
        outer-join="true|false|auto"                        (5)
        property-ref="propertyNameFromAssociatedClass"      (6)
        access="field|property|ClassName"                   (7)
/>
```

1-name: O nome da propriedade.
2-class(opcional, padrão property type determinado por reflexão): O nome da classe que possui a associação.
3-cascade(opcional): Especifica qual operação deve ser cascateada da classe pai para as classes que possuem relacionamento com ela.
4-constrained(opcional): Específica que uma foreign key constraint refêrencia a chave primária de uma tabela associada. Está opçao afeta a ordem pela qual os métodos save() e delete() são cascateados.
5-outer-join(opcional, padrão auto): Ativa o outer-join fetching para as associações quando hibernate.use_outer_join é setado.
6-property-ref(opcional): O nome da propriedade da classe associada que contêm a FK. Se não for informado, o ID da classe associada será utilizada.
7-access(opcional, padrão property): Estratégia que o hibernate deverá usar para acessar o valor da propriedade.

Existe duas variações para o relacionamento one-to-one:

- Associação pela chave primária. (primary key)
- Associação pela chave estrangeira única. (unique foreign key).

A associação pela chave primária não necessita de nenhuma

coluna extra. Se duas colunas possuem esse relacionamento, então as duas colunas compartilham o mesmo valor(o valor da chave primária). Então, para que duas classes possuem o relacionamento, alguma coluna das classes deve possuir o mesmo valor.

- **Component, dynamic-component**

O elemento <component> mapeia propriedades de uma classe filha para as colunas de uma tabela de uma classe pai. Esse elemento é muito útil em caso de reuso, quando um conjunto de elementos são utilizados em várias classes.

Para maiores informações consultar capítulo 7 (Components) da apostila do hibernate.

```
<component
        name="propertyName"                 (1)
        class="className"                   (2)
        insert="true|false"                 (3)
        upate="true|false"                  (4)
        access="field|property|ClassName">  (5)

        <property ...../>
        <many-to-one .... />
        ........
</component>
```

1-name: O nome da propriedade.
2-Class(opcional, padrão property type determinado por reflexão): O nome da classe do componente.
3-insert: Indica se a coluna aparece em comandos SQL de insert's.
4-update: Indica se a coluna aparece em comando SQL de update's.
5-access(opcional, padrão property): Estratégia que o hibernate deverá usar para acessar o valor da propriedade.

A propriedade <property> mapeia colunas de uma tabela para o componente.

- **Subclass**

Para persistir classes que possuem polimorfismo, cada subclasse de uma classe pai deve ser declarada. Para cada estratégia tabela-classe, é recomendável criar uma subclasse.

```
<subclass
        name="ClassName"                                (1)
        discriminator-value="discriminator_value"       (2)
        proxy="ProxyInterface"                          (3)
        lazy="true|false"                               (4)
        dynamic-update="true|false"
        dynamic-insert="true|false">

        <property .... />
        .....
</subclass>
```

1-name: Um nome (fully qualified) para a subclasse.
2-discriminator-value(opcional – padrão é o nome da classe): Um nome que distingue a subclasse de outras subclasses.
3-proxy(opcional): Especifica uma classe ou interface para ser usada no lazy inicializing.
4-lazy(opcional): Setando lazy=true, isso é um atalho equivalente a especificar que a classe é um proxy interface para o lazy inicializing.

Cada subclasse deverá declarar suas próprias propriedades persistentes e suas subclasses. <version> e <id> são obtidos por herança da classe pai. Cada subclasse deve definir um único <discrimitator-value>, se nenhum for informado, o nome da classe será usado.

- **Joined-subclass**

Alternativamente, uma subclasse que persiste dados numa tabela que só ele faça relacionamento, o elemento <joined-subclass> deve ser usado.

```
<joined-subclass
        name="ClassName"                    (1)
        proxy="ProxyInterface"              (2)
        lazy="true|false"                   (3)
        dynamic-update="true|false"
        dynamic-insert="true|false">

        <key .... >

        <property .... />
        .....
</joined-subclass>
```

1-name: Um nome (fully qualified) para a subclasse.
2-proxy(opcional): Especifica uma classe ou interface para ser usada no lazy inicializing (inicialização preguiçosa).
3-proxy(opcional): Especifica uma classe ou interface para ser usada no lazy inicializing.

Nenhuma coluna discriminadora (para o elemento discriminator-value) precisa ser declarada. Cada subclasse deve, entretanto, indicar uma coluna na tabela do banco de dados onde deve ser obtido o identificador da classe, para isso a coluna deve ser indicada no elemento <key>.

- **Import**

Suponha que o seu sistema possua duas classes distintas com o mesmo nome, para importar as classes explicítamente, o elemento import deve ser usado.

```
<import class="java.lang.Object" rename="Universe"/>
<import
        class="ClassName"               (1)
        rename="ShortName"              (2)
/>
```

1-class: O nome de qualquer classe java.
2-rename(opcional, padrão o nome da classe): Um nome que pode ser usado no comando SQL.

4.2.Hibernate Types

- Tipos básicos

Os tipos básicos podem ser categorizados por:

`integer, long, short, float, double, character, byte, boolean, yes_no, true_false`

Tipos de mapeamentos de tipos primitivos do java ou classes wrapper(java.lang.Integer,java.lang.Long,etc) para os tipos apropriados da coluna da tabela de dados.

`string`

Um tipo de mapeamento de java.lang.String para VARCHAR(ou no ORACLE VARCHAR2).

`Date, time, timestamp`

Tipos de mapeamentos de java.util.Date e suas subclasses para os tipos SQL DATE, TIME e TIMESTAMP (ou equivalente).

`Calendar, calendar_date`

Tipos de mapeamentos de java.util.Calendar para os tipos SQL TIMESTAMP e DATE (ou equivalente).

`big_decimal`

Um tipo de mapeamento de java.math.BigDecimal para NUMERIC (ou NO ORACLE NUMBER).

`Locale, timezone, currency`

Tipo de mapeamento de java.util.Locate, java.util.Timezone e java.util.Currency para VARCHAR (ou no ORACLE VARCHAR2). Instancias de Locate e Currency são mapeados para os seus códigos ISO (ISO codes). Instancias de Timezone são mapeados para os seus ID's.

`Class`

Um tipo de mapeamento de lava.lang.Class para VARCHAR (ou no ORACLE VARCHAR2). Uma classe é mapeada para o seu nome completo (fully qualified name).

`Binary`

Mapeamento de arrays de bytes para um tipo binário do SQL apropriado.

`Serializable`

Mapeamento de um tipo java serializavel (um java type serializable) para um tipo binário do SQL apropriado. Pode-se indicar o tipo hibernate serialazable com o nome de uma classe java serializada ou uma interface que não seja um tipo básico ou uma implementação de PeristentEnum.

Clob, blob

Tipo de mapeamento para a classe JDBC java.sql.Clob e java.sql.Blob. Estes tipos podem ser inconvenientes para algumas aplicações, desde que os objetos blob e clob não podem ser reutilizados fora de uma transação.

Identificadores de entidades (entities) e de coleções (coleções são usados em relacionamentos entre classes/tabelas) podem ser qualquer tipo básico, com exceção do tipo binary, clob e blog.

- Tipos enumerados persistentes

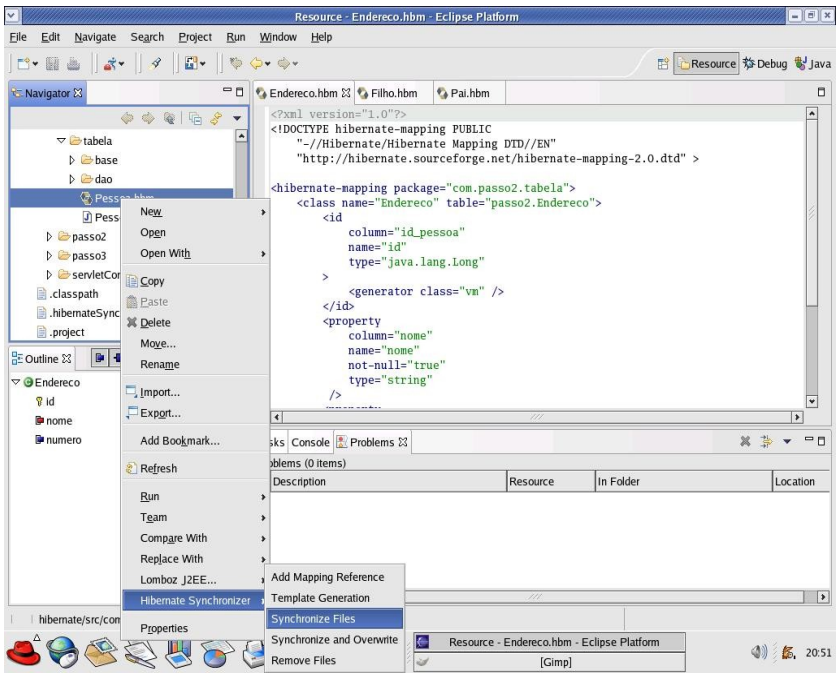
Um enumerado é um tipo muito comum em java, onde uma classe possui propriedades que não mudam de valor. Para se criar um enumerado, deve-se implementar net.sf.hibernate.PersistentEnum. Abaixo segue um exemplo encontrado na apostila do hibernate. Esse exemplo é utilizado para definir as possíveis cores de um gato.

```
package eg;
import net.sf.hibernate.PersistentEnum;

public class Color implements PersistentEnum {
    private final int code;
    private Color(int code) {
        this.code = code;
    }
    public static final Color TABBY = new Color(0);
    public static final Color GINGER = new Color(1);
    public static final Color BLACK = new Color(2);

    public int toInt() { return code; }

    public static Color fromInt(int code) {
        switch (code) {
            case 0: return TABBY;
            case 1: return GINGER;
            case 2: return BLACK;
            default: throw new RuntimeException("Unknown color code");
        }
    }
}
```

4.3. Geração das classes Objeto/Relacional

Para a geração das classes Objeto/Relacional, basta no seu eclipse clicar com o botão direito do mouse sobre o arquivo Objeto/Relacional (.hbm), ir em Hibernate Syncronizer->Syncronize Files.(Fig.6)

Fig.06



Obs: Caso o arquivo Objeto/Relacional não esteja sincronizado com a classe, o eclipse mostrará um alerta.

5.Manipulando dados persistentes

5.1.Persistindo dados (Criando um objeto persistente).

Criar um objeto persistente significa gravar dados em uma determinada tabela. Um objeto pode ser transient ou persistent, a diferença entre os dois é que o objeto transient é um objeto não persistido (não salvo no banco de dados) e o objeto persistent é um objeto persistido (salvo no banco de dados).

Para salvar um objeto transient, o método save() da classe session é usado.

```
Pessoa pess = new Pessoa();
pess.setNome(“rafael”);

session.save(pess);

/*Para efetuar o insert*/
session.flush();

/*Fechado a sessao*/
session.close();
```

Está sendo levado em conta que a classe Session já está criada (ver capítulo 3.1).

Para a classe ser persistida, o método flush() **DEVE SER** chamado, pois ele sincroniza a session com o banco de dados. Com isso os objetos transientes (nesse caso a pessoa), serão persistidos, e os dados persistidos depois da criação da session serão carregados.

A simples invocação do método save() assegura que o objeto

tenha uma chave primaria única, que no caso acima não é a propriedade nome. Não é preciso setar manualmente a PK da pessoa, isso é feito automaticamente, dependendo da propriedade <generetor> do elemento id. Para mais informações a respeito da propriedade <generetor> consultar o capítulo 5.1.4.1 da apostila do hibernate.

5.2.Carrengando um objeto

O método load() da classe Session é uma maneira de obter um objeto quando já se sabe de ante-mão o seu identificador (sua chave primária).

```
Pessoa pess = (Pessoa) session.load(Pessoa.class, new Long(1));
```

Os parâmetros desse método são:

- Classe que deseja obter.
- Identificador da classe(sua chave primária).

Caso não seja possível encontrar no banco de dados um registro com o identificador fornecido, uma exceção será lançada. Para verificar se existe o registro com tal identificador, o método get() deve ser usado, caso não seja encontrado nenhum registro, null será retornado.

```
Pessoa pess = (Pessoa) session.get(pessoa.class, new Long(1));
if (pess != null) {
    pess = (Pessoa) session.load(Pessoa.class, new Long(1));
}
```

Pode-se obter um objeto usando o SQL SELECT ... FOR UPDATE. Para maiores informações ver o capítulo 10 da apostila do hibernate.

É possível recarregar os dados do objeto e suas coleções (classes que ele possui relacionamento) usando o método reflesh(). Isso é muito útil quando triggers são usadas para setar alguma propriedade da classe.

```
session.save(pessoa);
session.flush(); //força o SQL INSERT
session.refresh(pessoa); //recarrega o estado da pessoa (depois de executar
alguma trigger)
```

- Querying

Se desejar obter vários objetos de uma classe, o método find() da classe session deve ser usado. Para busca de dados, o hibernate fornece uma linguagem muito parecida com o SQL que usamos normalmente, ele fornece uma linguagem SQL orientada a objeto, isso quer dizer que ao invés de usarmos uma tabela na clausula FROM, usamos a classe, e na clausula WHERE usamos propriedades da classe. Segue alguns exemplos encontrados na apostila do hibernate.

```
List cats = sess.find(
    "from Cat as cat where cat.birthdate = ?",
    date,
    Hibernate.DATE
);

List mates = sess.find(
    "select mate from Cat as cat join cat.mate as mate " +
    "where cat.name = ?",
    name,
    Hibernate.STRING
);

List cats = sess.find( "from Cat as cat where cat.mate.bithdate is null" );

List moreCats = sess.find(
    "from Cat as cat where " +
    "cat.name = 'Fritz' or cat.id = ? or cat.id = ?",
    new Object[] { id1, id2 },
    new Type[] { Hibernate.LONG, Hibernate.LONG }
);

List mates = sess.find(
    "from Cat as cat where cat.mate = ?",
    izi,
    Hibernate.entity(Cat.class)
);

List problems = sess.find(
    "from GoldFish as fish " +
    "where fish.birthday > fish.deceased or fish.birthday is null"
);

List pessoas = session.find(
" FROM Pessoa AS pess WHERE pess.nome = ? AND pess.idade = ? ",
new Object[] { new String("Rafael"), new Long(23) },
new Type[] { Hibernate.STRING, Hibernate.LONG });
```

É comum usar a diretiva 'AS' do SQL para criar um 'apelido' para a classe, assim fica mais fácil o manuseio do comando SQL. Caso não queira criar um 'apelido' para a classe, o comando SQL deve referenciar a classe na clausula WHERE.

O segundo argumento do método find() aceita um objeto ou um array de objetos, que indica o(s) tipo(s) do(s) objeto(s) setado(s) por '?' na clausula WHERE.

O terceiro argumento aceita um tipo do hibernate ou um array de tipos do hibernate.

Se a sua query retornar um número muito grande de registros, e não se espera usar todos eles, pode-se obter uma melhor performance usando o método iterate(). Esse método irá carregar os objetos por demanda. Segue um exemplo encontrado na apostila do hibernate.

```
// fetch ids
Iterator iter = sess.iterate("from eg.Qux q order by q.likeliness");
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next();  // fetch the object
    // something we couldnt express in the query
    if ( qux.calculateComplicatedAlgorithm() ) {
        // delete the current instance
        iter.remove();
        // dont need to process the rest
```

```
        break;
    }
}
```

Infelizmente o java.util.Iterator não lança qualquer exceção do SQL, ele apenas lança uma exceção do tipo LazyInitializationException.

Hibernate queries as vezes pode retornar tuplas de objetos, segue um exemplo.

```
Iterator foosAndBars = sess.iterate(
    "select foo, bar from Foo foo, Bar bar " +
    "where bar.date = foo.date"
);
while ( foosAndBars.hasNext() ) {
    Object[] tuple = (Object[]) foosAndBars.next();
    Foo foo = tuple[0]; Bar bar = tuple[1];
    ....
}
```

- **Interface Query**

A interface Query é útil em casos que pretendemos delimitar o número máximo de registros a serem retornadas pelo SQL (ex: os primeiros 20 registros ou o primeiro registro). Segue exemplo:

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

Pode-se também definir comandos SQL's no arquivo de mapeamento objeto/relacional, e depois obter tal comando utilizando o interface query (Lembre-se de usar CDATA para que nenhum caracter do seu SQL seja interpretado como algum tipo de marcador -markup-). Segue exemplo:

Trecho do arquivo objeto/relacional (.hbm).

```
<query name="eg.DomesticCat.by.name.and.minimum.weight"><![CDATA[
    from eg.DomesticCat as cat
        where cat.name = ?
        and cat.weight > ?
] ]></query>
```

Trecho de código.

```
Query q = sess.getNamedQuery("eg.DomesticCat.by.name.and.minimum.weight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

A interface query suporta o uso de parâmetros nomeados (named parameter). Parâmetros nomeados são identificados na forma de ':name' dentro do SQL. A interface query suporta tanto parâmetros nomeados como o estilo JDBC '?' para setar as variáveis

de dentro de um comando SQL. OBS: Ao contrário do JDBC, o index do estilo '?' começa com zero. Segue algumas considerações de parâmetros nomeados:

- A ordem que os parâmetros nomeados ocorrem dentro do comando SQL não interfere em nada.
- Um parâmetro nomeado pode ocorrer várias vezes dentro de um mesmo comando SQL.
- Eles são auto-documentáveis.

Segue exemplos encontrados na apostila do hibernate:

```
//named parameter (preferred)
Query q = sess.createQuery("from DomesticCat cat where cat.name = :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();

//positional parameter
Query q = sess.createQuery("from DomesticCat cat where cat.name = ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();

//named parameter list
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name in (:namesList)");
q.setParameterList("namesList", names);
List cats = q.list();
```

- Filtrando coleções

Um filtro de coleções (collection filter) é um tipo especial de query que pode ser aplicado em persistent collection ou em um array. OBS: Quando uma classe possui relacionamento one-to-many ou many-to-many com outra classe, podemos obter objetos da classe de relacionamento, obtendo assim uma coleção. É nessa coleção que podemos aplicar um filtro. Mas detalhes sobre relacionamento será discutido no próximo capítulo. Abaixo segue um exemplo de filtro de coleções.

```
Pai pai = session.load(Pai.class,new Long(1));

Collection filhos = session.filter(
    pai.getFilhos, "where this.color = ?", Color.BLACK, Hibernate.enum(Color.class)
);
```

Observe que o filtro não é necessita da clausula FROM.

5.4.Update de objetos

- **Update de objetos em uma mesma sessão.**

Instâncias de objetos persistentes (ex: objetos carregados, salvos, criados ou obtidos por uma query na session) podem ser manipulados e qualquer mudanças será persistida quando o método session.flush() for chamado. Tal método sincroniza o estado

dos objetos da session para o banco de dados. Em resumo, se for preciso alterar alguma propriedade de um objeto, basta carregar ele (load()), alterar suas propriedades e chamar o método session.flush(), com isso os dados serão atualizados automáticamente. Mas isso só funciona numa mesma session. Exemplo encontrado na apostila do hibernate:

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
cat.setName("PK");
sess.flush();  // as mudanças de cat serão automaticamente detectadas e peristidas
```

- **Update de objeto em <u>sessões diferentes</u>**

Muitas aplicações necessitam obter um objeto em uma transação, envia-lo para a camada de visualização (jsp), para que seus dados sejam manipulados, e persistir as mudanças em uma outra transação. Esse cenário é diferente do cenário descrito na seção anterior. Para persistir as mudanças nesse tipo de cenário descrito acima, o método session.update() deve ser usado. Segue exemplo encontrado na apostila do hibernate:

```
// Na primeira sessão
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// Na alta camada de sua aplicação (in a higher tier of the application)
cat.setMate(potentialMate);

// depois, em uma nova sessão.
secondSession.update(cat);  // update cat
secondSession.update(mate); // update mate
```

Existe um método chamado saveOrUpdate() que salva objetos transientes ou faz update em objetos persistentes, automaticamente, sem precisar invocar um método para salvar objetos transientes e outro para atualizar objetos persistentes. Para isso o hibernate deve conseguir distinguir entre novos objetos e objetos já persistidos, e isso é feito através da propriedade unsaved-value do elemento <id>. Segue exemplo:

```
<id name="id" type="long" column="uid" unsaved-value="null">
    <generator class="hilo"/>
</id>
```

Os valores para a propriedade unsaved-value são:

- **any** – Sempre salva o objeto.
- **none** – Sempre atualiza o objeto.
- **null(padrão)** – Salva quando o identificador for nulo.
- **valid identifier value** – Salva quando o identificador for nulo ou quando ele é dado.
- **undefinid –** O padrão para version ou timestamp, então a checagem do identificador é usado.

Os métodos update() ou saveOrUpdate() são geralmente usados no cenário descrito a baixo:

- A aplicação carrega um objeto numa primeira sessão.

- O objeto é passado para a camada de view para que seus dados sejam editados.
- Algumas modificações são feitas no objeto.
- O objeto é retornado para a classe de negócio.
- A aplicação persiste os dados do objeto que foram modificados numa segunda sessão.

O método saveOrUpdate() faz as seguintes ações:

- Se a aplicação já salvou o objeto na sessão em que o método saveOrUpdate() é invocado, ele não faz nada.
- Se o objeto não possui um identificador, ele é salvo.
- Se o identificador do objeto for igual ao valor especificado na propriedade unsaved-value da propriedade <id>, ele é salvo.
- Se outro objeto da sessão possuir o mesmo identificador do objeto a ser salvo ou atualizado, uma exceção será lançada.

6.Deletando objetos persistentes.

Session.delete() faz com que um objeto persistente torna-se um objeto transiente, isso quer dizer que os dados do banco de dados são apagados, mais o objeto (juntamente com os seus dados) encontram-se na sessão, para serem manipulados caso seja necessário. Segue exemplo:

```
sess.delete(cat);
```

Pode-se também deletar vários objetos de uma só vez, passando uma string SQL para o método delete().

7.Flush

De tempos em tempos o hibernate sincroniza os dados que estão em memória com os dados do banco de dados. Isso ocorre quando certos métodos são invocados, sendo eles:

- O método find() ou iterate().
- O método net.sf.hibernate.Transaction.commit().
- Session.flush().

Encerrando sessões

O encerramento de uma sessão (Session.close()) envolve quatro fases distintas, sendo elas:

- Flush da sessão.
- Commit da transação.
- Encerramento da sessão.
- Lançamento de eventuais exceções.

Caso seja usado a API da classe transaction, não será preciso se preocupar com as etapas citadas acima. Caso seja

necessário sincronizar todos os dados da sessão com o banco de dados, daí sim o método Session.flush() deverá ser usado.

Comitando uma transação

Segue exemplo de como usar a classe transaction:

```
try {
          tx = session.beginTransaction();

          Fisica fisica = new Fisica();
          fisica.setNome(nome_fisica);
          session.save(fisica);

          tx.commit();
} catch (Excption e) {
          tx.rollback();
}
```

8.Operações com relacionamento Pai/Filho (Grafos de objetos)

Para salvar ou atualizar todos os objetos de um grafo de objetos que possuem relacionamento, deve-se:

- Invocar os métodos save(), saveOrUpdate() ou update() para cada objeto do grafo **OU**
- Indicar a propriedade cascade=”all” ou cascade=”save-update” no relacionamento dos objetos.

Isso também ocorre com o método delete():

- Invocar o método delete() para cada objeto do grafo **OU**
- Indicar a propriedade cascade=”all”, cascade=”save-update” ou cascade=”all-delete-orphan” no relacionamento dos objetos.

Caso o clico de vida da classe filha esteja ligado com o ciclo de vida da classe pai, é extremamente útil utilizar a propriedade cascade=”all” no relacionamento entre as classes, com isso não será preciso se preocupar em fazer operações SQL nas classes filhas.

9.Clico de vida dos objetos.

Mapeamento de associações (many-to-one) com cascade=”all”, marca uma associação como sendo uma associação Pai/Filho, onde save, delete ou update no pai, resulta em save, delete ou update no filho. Quando um filho sofre alguma operação SQL, somente o filho sofre as alterações. Caso um filho se torne não referenciável por qualquer pai, tal filho não é deletado automaticamente, exceto no caso do relacionamento one-to-many onde a propriedade cascade seja cascade=”all-delete-orphan”.

Segue as operações executadas em relacionamento Pai/Filho:

- Se a classe pai é salva, todas as classes filhas, passam por saveOrUpdate().
- Se a classe pai passa por update() ou saveOrUpdate(), todas as classes filhas passam por saveOrUpdate().
- Se uma classe filha transiente torna-se referencida por uma classe pai persistente, a classe filha passa por saveOrUpdate().
- Se um pai é deletado, todos os filhos são deletados.

10.Relacionamento Pai/Filho

O caminho mais fácil de criar um relacionamento entre as classes é utilizar o relacionamento Pai/Filho one-to-many (onde um pai contêm nenhum, um ou mais filhos), onde a classe pai e filha são entity class. Existem outros caminhos para esse relacionamento, como por exemplo declarar as classe filhas como sendo <composite-elements> da classe pai.

10.1.Relacionamento one-to-many

Será explicado nesse capítulo o relacionamento one-to-many, onde uma classe pai pode possuir vários filhos e um filho possui apenas um pai.

A seguir será mostrado o arquivo de mapeamento Objeto/Relacional de uma classe pai e uma classe filha.

Arquivo de mapeamento da classe Pai.

```
<hibernate-mapping package="com.passo2.tabela">
      <class name="Pai" table="passo2.pai">
          <id
             column="id"
             name="id"
             type="java.lang.Long"
          >
             <generator class="vm" />
          </id>
          <property
             column="nome"
             name="nome"
             not-null="true"
             type="string"
          />
          <property
             column="idade"
             length="4"
             name="idade"
             not-null="false"
             type="java.lang.Long"
          />

          A

       </class>
```

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

Arquivo de relacionamento da classe Filha.

```
<hibernate-mapping package="com.passo2.tabela">
      <class name="Filho" table="passo2.filho">
          <id
             column="id"
             name="id"
             type="java.lang.Long"
          >
             <generator class="vm" />
          </id>
          <property
             column="nome"
             name="nome"
             not-null="true"
             type="string"
          />
          <property
             column="idade"
             length="4"
             name="idade"
             not-null="false"
             type="java.lang.Long"
          />

          B


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

Os trechos destacados com a letra A e a letra B deveram conter o código que indica o relacionamento entre as duas classes.

Nesse primeiro momento, vamos levar em conta que apenas o trecho A (mapeamento Objeto/Relacional da classe pai) foi preenchido com o seguinte trecho:

```
<set name="Filho">
     <!-- campo que faz a ligação da tabela Filho com a tabela Pai-->
     <key column="id_pai"/>
     <one-to-many class="com.passo2.tabela.Filho"/>
</set>
```

Se executarmos o código a seguir:

```
/*Obtendo o pai que possui o id_pai fornecido pelo usuario*/
Pai pai = (Pai) session.load(Pai.class, new Long(PK));

/*Criando um novo filho*/
Filho filho = new Filho();
```

```
/*Metodo que adiciona o filho a lista de filhos (Classe Set) do pai*/
pai.addToFilho(filho);

session.save(filho);
session.flush();
session.close();
```

O hibernate deverá executar dois comandos SQL's, sendo eles:

- Um INSERT para criar um record para filho
- Um UPDATE para criar a ligação entre pai e filho.

Isso é ineficiente e ainda pode causar violação de NOT NULL CONSTRAINT. Isso se deve porque a ligação entre pai e filho (Chave estrangeira da classe Pai) não é considerado parte do estado da classe Filho, por isso que no INSERT a classe Filho não cria a ligação entre Pai e Filho.

A solução para isso é fazer com que a ligação faça parte da classe Filho, para isso basta acrescentar no trecho B do mapeamento Objeto/Relacional o trecho a seguir:

```
<many-to-one class="com.passo2.tabela.Pai" column="pai" name="pai" not-null="true"/>
```

Agora que a classe filha gerencia a ligação, devemos acrescentar no trecho A a propriedade inverse=”true” na collection da classe pai, para que a collection não faça UPDATE da ligação Pai/Filho.

```
<set name="Filho" inverse=”tr ue”>
     <!-- campo que faz a ligação da tabela Filho com a tabela Pai-->
     <key column="id_pai"/>
     <one-to-many class="com.passo2.tabela.Filho"/>
</set>
```

Agora se executarmos o trecho a seguir:

```
/*Obtendo o pai que possui o id_pai fornecido pelo usuario*/
Pai pai = (Pai) session.load(Pai.class, new Long(PK));

/*Criando um novo filho*/
Filho filho = new Filho();

/*Metodo que adiciona o filho a lista de filhos (Classe Set) do pai*/
pai.addToFilho(filho);

session.save(filho);
session.flush();
session.close();
```

Apenas um INSERT (o da classe filha) será executado.

- Cascateamento

Quando invocamos o método save() para salvar as alterações

na classe pai, isso não é o suficiente para fazer as atualizações nas classes filhas, para resolver isso devemos usar a propriedade cascade no elemento set (onde é criado a collection de classes filhas). Segue exemplo:

```
<set name="Filho" inverse=”tr ue” cascade=”all”>
     <!-- campo que faz a ligação da tabela Filho com a tabela Pai-->
     <key column="id_pai"/>
     <one-to-many class="com.passo2.tabela.Filho"/>
</set>
```

Com a propriedade cascade=”all”, não será preciso interagir com as classes filhas quando salvamos ou deletamos uma classe pai. Quando deletamos uma classe pai, as classes filhas também são deletadas, segue exemplo:

```
Pai pai = (Pai) session.load(Pai.class, new Long(PK));
session.delete(pai);
session.flush();
```

No entanto, no código que se segue:

```
Pai pai = (Pai) session.load(Pai.class, new Long(PK));
Filho filho = (Filho) pai.getFilho().iterator().next();
pai.getFilho().remove(filho);
filho.setPai(null);
session.flush();
```

A classe filho não é apagada do banco de dados, o que é apagado é a ligação (campo da tabela) entre a classe pai e a classe filha, podendo causar erro de constraint NOT NULL. Para solucionar esse problema, existe duas maneiras, sendo elas:

- Chamar o método session.delete() para apagar o registro da classe filha, segue exemplo:

```
Pai pai = (Pai) session.load(Pai.class, new Long(PK));
Filho filho = (Filho) pai.getFilho().iterator().next();
pai.getFilho().remove(filho);
session.delete(filho);
session.flush();
```

- **OU** utilizar a propriedade cascade=”all-delete-orphan” na collection da classe filha para que o hibernate delete as classes filhas orfãs. Segue exemplo:

```
<set name="Filho" inverse=”tr ue” cascade=”all-delete-filho”>
     <!-- campo que faz a ligação da tabela Filho com a tabela Pai-->
     <key column="id_pai"/>
     <one-to-many class="com.passo2.tabela.Filho"/>
</set>
```

OBS: O trecho acima tem que ser adicionado no arquivo de

mapeamento da classe pai.

- Usando o cascateamento para update()

Suponhamos o seguinte cenário:

Carregamos uma classe Pai, juntamente com suas classes Filhas, em uma primeira sessão. Enviamos essa classe Pai para a camada de visualização, editamos alguns dados dela e acrescentamos novas classes Filhas a ela. Em uma segunda sessão fazemos o update da classe pai, e aí que está o problema, o hibernate não consegue distinguir entre uma nova classe filha e uma classe filha que já está criada no banco de dados, para resolvermos isso existe duas soluções, sendo elas:

1. Invocar o método session.save() para as classes Filhas que não foram persistidas (novas classes) **OU**
2. Utilizar a propriedade cascade=”all” ou cascade=”save-update”.

Mesmo utilizando a propriedade cascade=”all” ou cascade=”save-update”, devemos “fornecer uma dica” para que o hibernate possa distinguir entre classes novas e classes já existentes, para isso a propriedade unsaved-value do elemento ID deve ser fornecido. Esse elemento foi discutido no capítulo 4.1, página 11. Segue exemplo:

```
<id name="id" type="long" unsaved-value="0">
```

11.Hibernate Quary Language (HQL)

A linguagem que o hibernate utiliza para querys é a HQL, que é muito parecida com o SQL. Sua sintaxe é completamente orientada a objeto, sendo assim ela implementa herança, polimorfismo e associação.

- **Case sensitivity**

Os nomes das classes e suas propriedades são case sensitive, isso quer dizer que existe diferença entre letras maiúsculas e minúsculas no nome de uma classe. Uma classe chamada PAI é diferente de uma classe chamada pAi que por sua vez é diferente de outra classe chamada pai.

Isso não se aplica as palavras reservadas do SQL, como por exemplo as palavras SELECT, FROM e WHERE. É indiferente escrever as letras das palavras em maiúsculo ou minúsculo.

11.1-Clausula FROM

É possível executar uma query no hibernate da seguinte forma:

```
FROM Pai
```

Com essa simples query, é possível obter todos os pais cadastrados no banco de dados.

É comum criar um alias para a classe, para que se possa referencia-lá em outras partes da query. Abaixo segue alguns exemplos de como criar alias para as classes:

```
FROM Pai AS pai
FROM Pai pai
```

É uma boa prática criar os alises com a inicial minúscula, seguinte o padrão de nomeação de variáveis em java.

- Associações e joins

Pode-se definir alias e associações utilizando um join.

```
from eg.Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten

from eg.Cat as cat left join cat.mate.kittens as kittens

from Formula form full join form.parameter param
```

Os tipos de joins suportados pelo hibernate são os mesmo do ANSI SQL:

- inner join
- left outer join
- right outer join
- full join (geralmente não é útil)

O inner join, left outer join e o right outer join podem ser abreviados.

```
from eg.Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

Um “fetch” join aceita que associações sejam inicializadas juntamente com os seus respectivos pais, usando um simples select. Isto é particularmente útil no caso de uma coleção.

```
from eg.Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

Para um fetch join, não é preciso criar um alias para ele, porque os objetos associados não serão usados na clausula where. Os objetos associados não serão retornados diretamente pela query, eles só poderam ser acessados via seu objeto pai.

11.2-Clausula SELECT

A clausula SELECT obtêm classes e suas propriedades e retorna os dados obtidos em uma query set.

```
select mate
from eg.Cat as cat
    inner join cat.mate as mate
```

Exemplo de como obter uma propriedade de uma classe:

```
select cat.mate from eg.Cat cat
```

Pode-se obter coleções usando a função **elements**. No exemplo a seguir, a query retorna todos os kittens de qualquer cat.

```
select elements(cat.kittens) from eg.Cat cat
```

Ou pode-se o obter todos os filhos de qualquer pai.

```
select elements(pai.filhos) from eg.Pai pai
```

As queries podem retornar qualquer tipo de elemento, incluindo propriedades de um componente.

```
select cat.name from eg.DomesticCat cat
where cat.name like 'fri%'
```

```
select cust.name.firstName from Customer as cust
```

Obs: Componente é uma classe, que possui suas próprias propriedades, que faz parte de uma classe maior. O componente é uma propriedade de uma classe maior. No exemplo acima, name é um componente, podendo ter as seguintes propriedades: Nome, último nome e nome do meio.

Queries podem retornar múltiplos objetos e/ou propriedades, como sendo um array de objetos.

```
select mother, offspr, mate.name
from eg.DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

Ou como um objeto java

```
select new Family(mother, mate, offspr)
from eg.DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

Obs: Assumimos que a classe Family possui o construtor apropriado.

- Funções agregadas

Queries podem retornar funções agregadas de propriedades.

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from eg.Cat cat
```

Funções agregadas podem também serem usadas com collections na clausula SELECT.

```
select cat, count( elements(cat.kittens) )
from eg.Cat cat group by cat
```

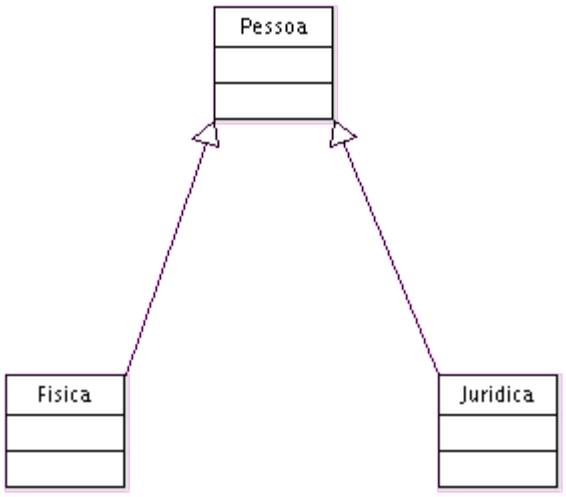
As funções agregadas suportadas são:

- avg(...), sum(...), min(...), max(...)
- count(*)
- count(...), count(distinct ...), count(all...)

As palavras DISTINCT e ALL podem ser usadas e possui a mesma semântica do SQL.

- Polimorfismo

Levando em conta o seguinte diagrama:



E executando a seguinte query:

```
FROM Pessoa AS pessoa
```

Será retornada instâncias das classes Física e Jurídica. O hibernate não retorna somente instâncias da classe em questão (Pessoa), mas também retorna instâncias de todas as classes que extendem (extends) a classe base.

Para retornarmos todas os objetos persistidos, basta executar a seguinte query:

```
FROM java.lang.Object AS o
```

- A clausula WHERE

A clausula WHERE aceita você limitar a lista de instâncias retornadas pela query.

```
from eg.Cat as cat where cat.name='Fritz'
```

A query anterior retorna instâncias de Cat cujo o nome seja Fritz.

Analise a próxima query:

```
select foo
from eg.Foo foo, eg.Bar bar
where foo.startDate = bar.date
```

Será retornado instâncias da classe eg.Foo que possuem relacionamento com a classe eg.Bar. Esse relacionamento se dá através das propriedades das classes foo.startDate e bar.date (foo.startDate = bar.date).

Leve em conta o seguinte diagrama:

Para obtermos todas as pessoas que possuem um endereço, cujo o nome da rua seja diferente de null, podemos executar a seguinte query.

```
FROM Pessoa AS pessoa WHERE pessoa.endereco.rua IS NOT NULL.
```

Essa query é transformada numa SQL query que utiliza um (inner) join.

O operador “=” pode também ser usado para comparar instâncias. Ele não fica restrito somente a comparação de atributos.

```
from eg.Cat cat, eg.Cat rival where cat.mate = rival.mate
```

```
select cat, mate
from eg.Cat cat, eg.Cat mate
where cat.mate = mate
```

A propriedade especial “class” obtêm o valor especificado na propriedade <discriminator> (capítulo 4). Utilizando essa propriedade, podemos filtrar classes que possuem polimorfismo.

```
from eg.Cat cat where cat.class = eg.DomesticCat
```

- Expressões

Segue uma lista de expressões que a clausula WHERE aceita como parâmetro:

operadores matemáticos +, –, *, /

operadores binários de comparação =, >=, <=, <>, !=, like

operadores lógicos and, or, not

string de concatenação ||

funções escaleres SQL como upper() and lower()

parenteses ( ) indicando grupo

in, between, is null

parâmetros JDBC ?

parâmetros nomeados :name, :start_date, :x1

SQL literals 'foo', 69, '1970-01-01 10:00:01.0'

Java public static final contendo eg.Color.TABBY

As expressões IN e BETWEEN podem ser usadas como no exemplo a seguir

```
from eg.DomesticCat cat where cat.name between 'A' and 'B'

from eg.DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

E a forma negativa pode ser escrita dessa maneira

```
from eg.DomesticCat cat where cat.name not between 'A' and 'B'

from eg.DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

Do mesmo modo, IS NULL e IS NOT NULL podem ser usados

para testar valores nulos.

As funções SQL's ANY, SOME, ALL, EXISTS e IN são suportadas desde que seja passado um element ou index de uma collection (função elements e indices) ou um resultado de uma subquery. Veja exemplo:

```
select mother from eg.Cat as mother, eg.Cat as kit
where kit in elements(foo.kittens)

select p from eg.NameList list, eg.Person p
where p.name = some elements(list.names)

from eg.Cat cat where exists elements(cat.kittens)

from eg.Player p where 3 > all elements(p.scores)

from eg.Show show where 'fizard' in indices(show.acts)
```

- Clausula ORDER BY

A lista retornada pela query pode ser ordenada por qualquer propriedade da classe ou componente.

```
from eg.DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

O opcional ASC e DESC indica ascendente e descendente.

- Clausula GROUP BY

Uma query que retorna valores agregados pode ser agrupado por qualquer propriedade da classe ou componente.

```
select cat.color, sum(cat.weight), count(cat)
from eg.Cat cat
group by cat.color

select foo.id, avg( elements(foo.names) ), max( indices(foo.names) )
from eg.Foo foo
group by foo.id
```

A apostila do hibernate possui vários outros exemplos e dicas do HQL, ver capítulos 11.12 e 11.13.