# Mapeamento Objeto-Relacional (Object Relational Mapping)

- A maioria das aplicações sejam elas construídas para uso no Web ou não, utilizam algum mecanismo para a persistência de dados
- Normalmente utiliza-se algum Banco de Dados que é o responsável pelo armazenamento (persistência) e pela manutenção dos dados de uma aplicação.
- Os banco de dados disponibilizam a linguagem SQL que permite operações de CRUD (Create; Read; Update; Delete)
- Atualmente os dados de dados são criados dentro de um modelo conhecimendo como relacional.
- Neste modelo os dados são vistos como tabelas, onde as colunas individualmente definem quais os dados armazenados e uma linha representa um conjunto de dados específico
- Em uma análise simples, pode-se dizer que uma linha representa um objeto
- Sendo assim uma tabela de um banco de dados poderia ser comparada a um conjunto de objetos
- A semelhança porém, encerra-se aqui

# Mapeamento Objeto Relacional Conceitos

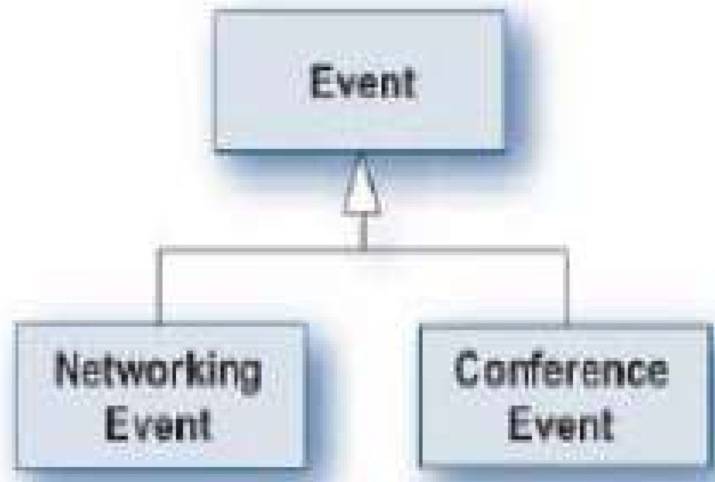
- Existem alguns conceitos, existentes na orientação a objetos que são dificilmente expressados em um banco de dados relacional
  - Identidade
  - Herança
  - Associações
- Desta forma ocorre uma incongruência entre o modelo relacional e o modelo orientado a objetos
- Uma maneira de resolver tal questão é utilizando o mapeamento Objeto/Relacional (ORM – Object Relational Mapping)
- Através deste processo os objetos podem ser persistidos em um banco de dados relacional sem a necessidade de conversão de objetos para um modelo relacional
- O Hibernate é um framework que implementa o ORM tanto em ambientes baseados na plataforma Java quanto .NET
- Existem outros frameworks para a plataforma Java como
  - iBatis, Castor, JDO (Java Data Objects), Torque, JDBC Persistence, OpenJPA

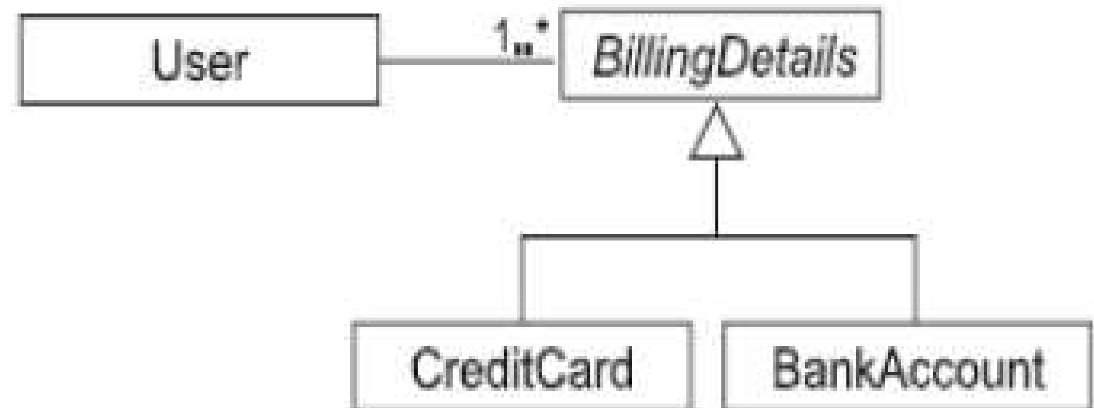
# Incompatibilidade Objeto x Relacional Identidade

- Na linguagem java
  - objA == objB //identidade
- é diferente de
  - objA.equals(objB) //igualdade
- No modelo relacional não existe o conceito de igualdade.
- Para resolver tal problema é necessário adicionar um campo id (identificador) na tabela
- Neste caso este campo adicional também deve ser levado para o modelo orientado a objetos

# Incompatibilidade Objeto x Relacional Herança

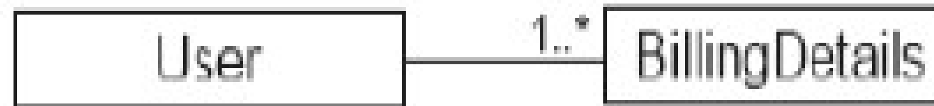
- Um poderoso conceito das linguagens orientadas a objetos é a herança
- Este conceito não pode ser implementado diretamente em um banco de dados relacional
- O Hibernate apresenta algumas estratégias a fim de ligar uma hierarquia de classes com o banco de dados



# Incompatibilidade Objeto x Relacional Associações

- Em uma linguagem orientada a objetos as associações podem ser facilmente representadas




- Neste exemplo a classe User contém um objeto que representa um conjunto de objetos da classe BillingDetails e a classe BillingDetails possui um objeto da classe usuário

```
public class User {
  private String userName;
  private String address;
  private Set billingDetails;
  ...
}
```

```
public class BillingDetails {
  private String accountNumber;
  private String accountName;
  private String accountType;
  private User user;
  ...
}
```

# Incompatibilidade Objeto x Relacional Associações

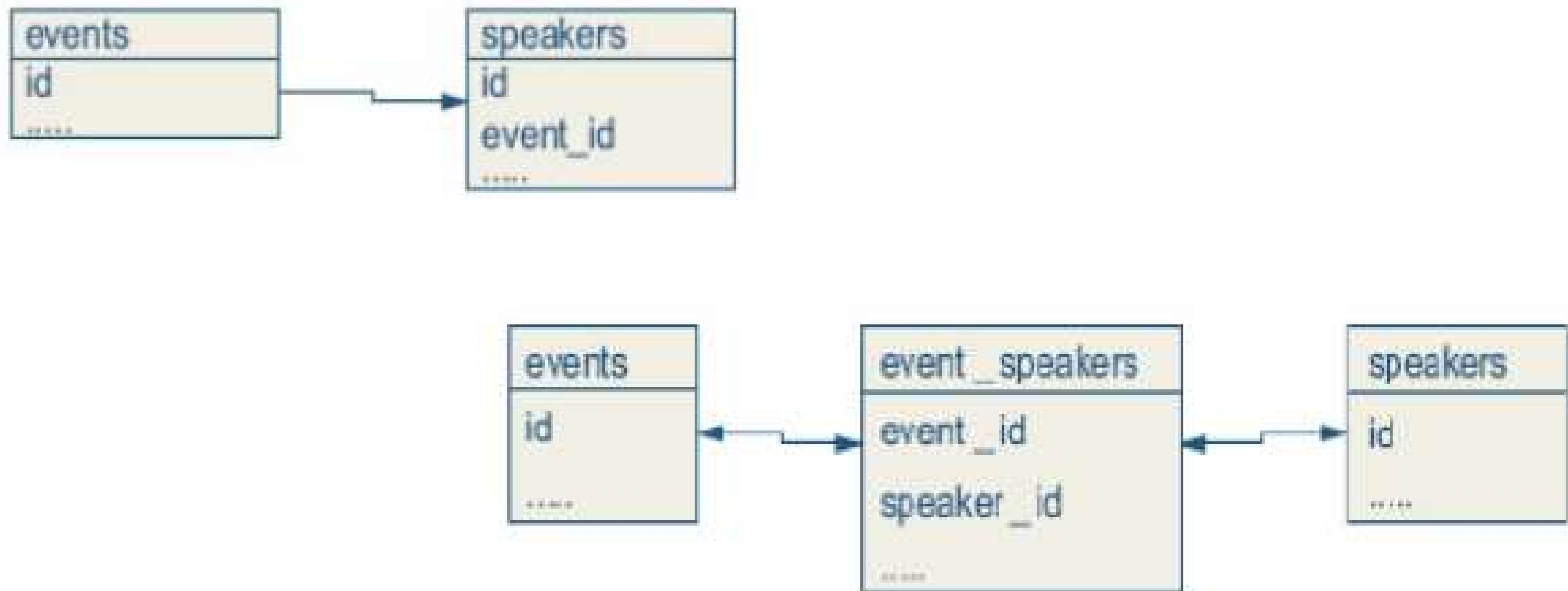
- A associação anterior pode ser representada pelas seguintes tabelas em um banco de dados relacional:

```
create table USER (
   USERNAME VARCHAR(15) NOT NULL PRIMARY KEY,
   ADDRESS VARCHAR(100)
)
create table BILLING_DETAILS (
    ACCOUNT_NUMBER VARCHAR(10) NOT NULL PRIMARY Key,
    ACCOUNT_NAME VARCHAR(50) NOT NULL,
    ACCOUNT_TYPE VARCHAR(2) NOT NULL,
   USERNAME VARCHAR(15) FOREIGN KEY REFERENCES USER
)
```

- Nota-se claramente o problema da incompatibilidade entre o modelo orientado a objetos e o modelo relacional

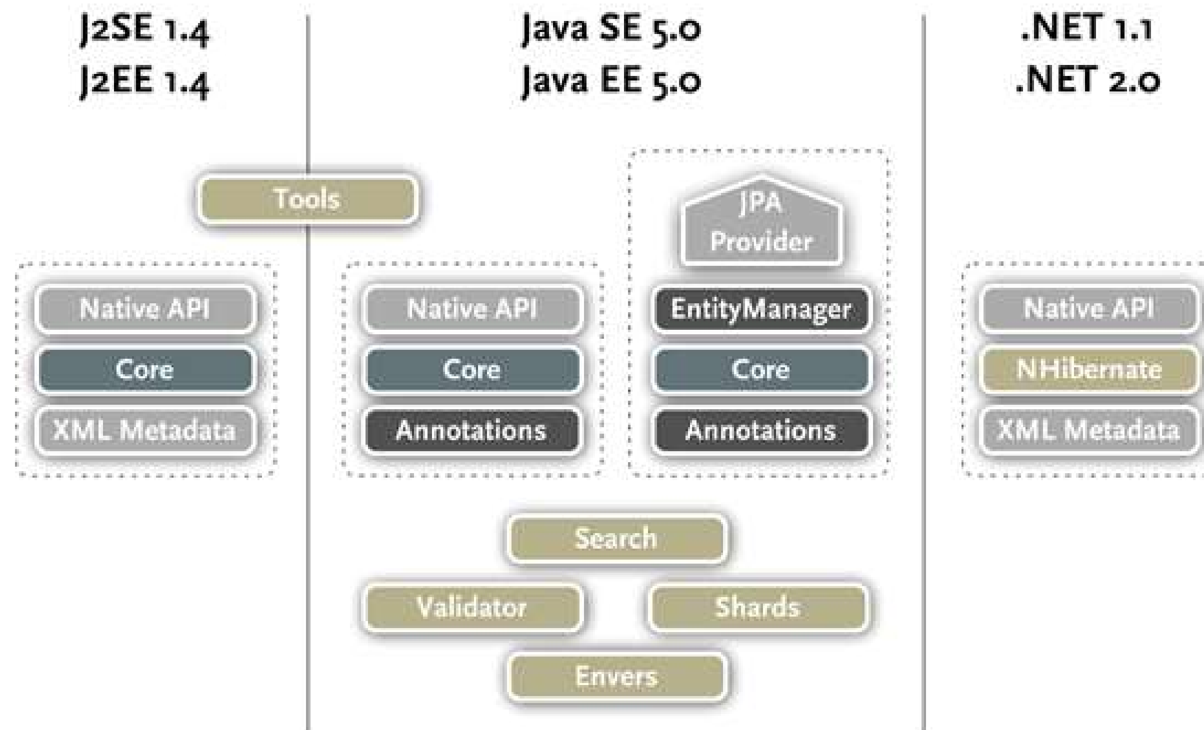
# Incompatibilidade Objeto x Relacional Associações

- Além disto o modelo de objeto suporta vários tipos de associações como: um-para-um; um-para-muitos e muitos-para-muitos.
- No modelo relacional estas associações precisam ser feitas utilizam uma chave estrangeira e são modeladas das seguinte forma:



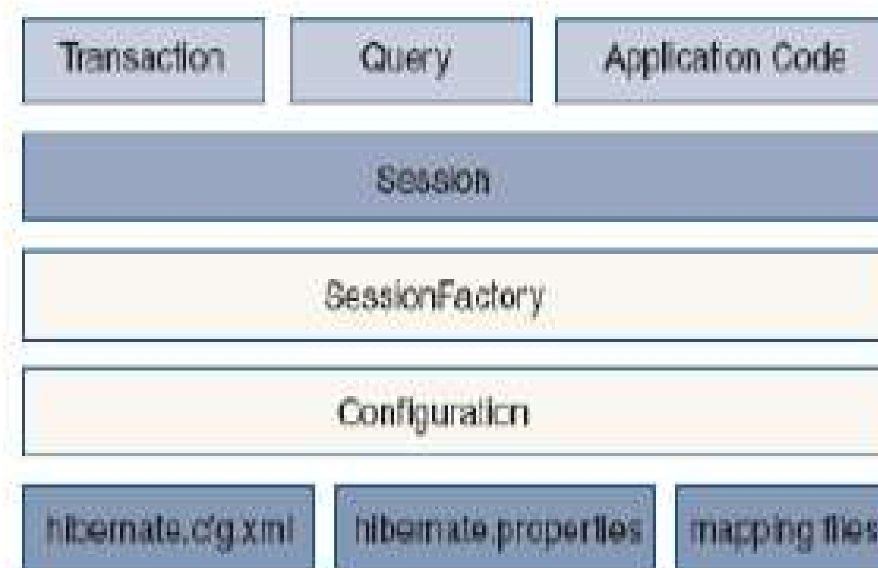
# Hibernate

- Família de projetos relacionados para ORM (Object Relational Mapping)
- Pode ser utilizada tanto na plataforma Java quando .Net



# Hibernate – Classes Principais

- A figura abaixo mostra as principais classes e arquivos de configuração utilizados pelo Hibernate




- Configuração básica
  - A configuração básica pode estar contida em um arquivo XML, chamado hibernate.cfg.xml ou então um arquivo de propriedades (hibernate.properties)
- Mapping Files
  - Arquivos XML que indicam como os objetos estão ligados com as tabelas de um banco de dados relacional e como serão persistidos

# Hibernate – Classes Principais

- Configuration
  - Classe que contém as informações contidas nos arquivos de configuração
  - Este objeto é responsável por criar um objeto da classe SessionFactory
- SessionFactory
  - Classe que “fabrica” objetos do classe Session
  - Normalmente um objeto desta classe é responsável por criar todas as conexões necessárias pela aplicação.
- Session
  - Representa uma conexão com o banco de dados
  - Um objeto da classe sessão é utilizado para uma única transação ou então para uma unidade de trabalho com o banco de dados.

# Hibernate – Classes Principais

- Transaction
  - Uma transação agrupa várias operações em uma unidade de trabalho única.
  - Se uma das operações da transação falhar, as operações já executas são desfeitas e a execução é finalizada.
  - Desta forma a aplicação retorna ao seu estado anterior.
- Query
  - Objetos desta classe permitem recuperar objetos que estão no banco de dados.
  - O Hibernate possui uma linguagem chamada HQL (Hibernate Query Language) e desta forma os comandos criados são orientados a objetos
  - Este objeto permite a consulta através da HQL e desta forma o acesso ao objeto pode ser feito a partir de suas propriedades e não através de nomes de tabelas e suas colunas.

# Hibernate – Mapping Files

- Os arquivos de mapeamento (mapping files) são arquivos XML que especificam como será a ligação entre os objetos e as tabelas no banco de dados
- O arquivo XML contém vários elementos que indicam como será feito o ORM entre um determinado banco e a aplicação.
- Os arquivos de mapeamento permitem indicar características como:
  - Chave primária
  - Atributos x Colunas
  - Associações: um-para-um; um-para-muitos; muitos-paramuitos
  - Herança

# Hibernate – Mapping Files

- A seguir é mostrada a estrutura básica de um mapping file

```
<hibernate-mapping>
  <class>
      <id>
         <generator/>
      </id>
      <property>
         <column/>
      </property>
      <many-to-one>
         <column/>
      </many-to-one>
      <collectionName>
       <key> </key>
       <one-to-many></one-to-many>
      </collectionName>
  </class>
</hibernate-mapping>
```

# Hibernate – Mapping Files Elemento Class

- Elemento HIBERNATE-MAPPING
  - Este elemento possui, entre outros, os seguintes atributos opcionais:
    - schema – nome do esquema do banco de dados
    - catalog – nome do catálogo do banco de dados
    - package – prefixo que será utilizado para o nome de classes não qualificadas.
- Elemento CLASS
  - Elemento que descreve a classe
  - Elemento principal do arquivo de mapeamento, realiza a ligação entre uma classe Java do MODEL e uma tabela do banco
  - O atributo “name” indica o nome completo da classe
  - O atributo “table” indica a tabela do banco que será mapeada na classe

```
<hibernate-mapping package="com.Hures.Hibernate">
    <class name=“Employee" table=“empregado">
    ...
```

# Hibernate – Mapping Files Elemento ID

- Elemento ID

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

  - O ID descreve a chave primária da tabela e a forma como a mesma é gerada
  - O atributo “name” indica o nome da propriedade da classe que irá armazenar a chave primária.
  - O atributo “column” indica o nome da coluna da tabela que contém a chave primária
  - Os atributos “type” dependem do tipo de gerador utilizado pelo banco
  - O atributo “unsaved-value” indica o valor que será utilizado para o atributo do objeto enquanto o mesmo não é salvo no banco de dados.
  - O gerador (elemento <generator>) é responsável pela criação da chave primária para a classe do MODEL.
  - O Hibernate oferece vários tipos de geradores como: native; identity; sequence; increment; entre outros

# Hibernate – Mapping Files
# Tipos de Geradores

| NOME | DESCRIÇÃO GERADOR |
|---|---|
| **increment** | Gera um identificador do tipo long, short or int que é único somente quando não existem outros processos inserindo dados da mesma tabela |
| **identity** | Suporta colunas de identidade. Utilizado com DB2, MySQL, MS SQL Server, Sybase e HypersonicSQL. Seu tipo pode ser long, short ou int. |
| **sequence** | Utiliza uma sequence em bancos DB2, PostgreSQL, Oracle, SAP DB, McKoi como ou Interbase.Seu tipo pode ser long, short ou int. |
| **hilo** | Utiliza um algoritmo hi/lo que produz identificadores do tipo long, short ou int. Dado uma tabela e uma coluna que será a fonte para os valores High. Os identificadores são únicos somente para um banco de dados particular |
| **seqhilo** | Utiliza um algoritmo hi/lo que produz identificadores do tipo long, short ou int, sendo fornecido como base uma sequence de um banco de dados |
| **uuid** | utiliza um algoritmo de 128-bits UUID que produz identificadores do tipo string e que são únicos em uma rede (pois o IP é utilizado pelo algoritmo). O valor é codificado como uma string hexadecimal de tamanho 32 |
| **guid** | Utiliza um String GUID string em bancos MS SQL Server e MySQL. |
| **native** | Escolhe entre os geradores identity, sequence ou hilo dependendo da capacidade do banco utilizado |
| **assigned** | Neste caso aplicação é responsável por escolher um identificador para o objeto antes que o método save() seja chamado. Este é o valor default caso nehum elemento <generator> seja fornecido |
| **select** | Permite obter a chave primária a partir de um trigger através da seleção do id a partir de alguma tabela |
| **foreign** | Utiliza o identificador associado com outro objeto. Usualmente utilizado em uma associação <one-to-one> entre chaves primárias |

# Hibernate – Mapping Files
# Elemento Property

- Elemento Property

```
<property name="name" type="string" length="100"/>
<property name="startDate" column="start_date" type="date"/>
<property name="duration" type="integer"/>
```

  - Este elemento realiza o mapeamento entre uma coluna da tabela e uma propriedade da classe Java
  - O atributo “name” indica a variável membro da classe
  - O atributo “column” indica o nome da coluna caso seja diferente de “name”
  - O atributo “type” faz um mapeamento do tipo da coluna para um tipo java.
  - Caso o mesmo seja omitido o tipo é obtido da através da API de reflexão (reflection API)
  - Além disso o tipo pode ser uma classe Java qualquer ou então um tipo composto. Neste caso é necessário a classe com o tipo implemente a interface **Hibernate.UserType ou Hibernate.CompositeUserType** respectivamente

```
<property name="twoStrings" type="org.hibernate.test.DoubleStringType">
  <column name="first_string"/> <column name="second_string"/>
</property>
```

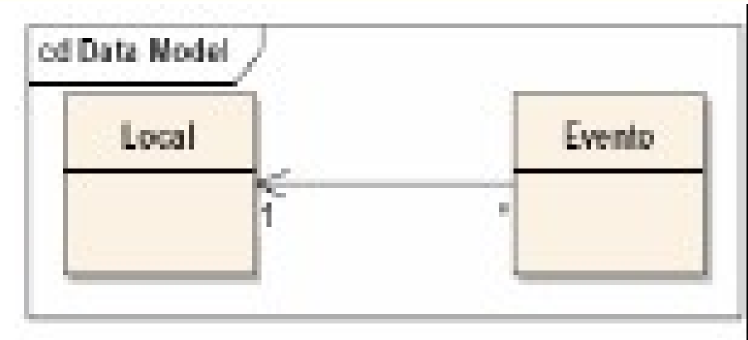
# Hibernate – Mapping Files Elemento Many-to-One

- Elemento MANY-TO-ONE

  <many-to-one name=“local" column="location_id" class="Local"/>

  - Permite uma associação de muitos-para-um entre duas classes.
  - A chave estrangeira de uma tabela faz referência a uma chave primária em outra tabela (tabela externa)
  - O atributo “name” indica o nome da propriedade do objeto que contém a chave estrangeira
  - O atributo (opcional) “column” indica o nome da coluna da tabela que contém a chave estrangeira
  - O atributo (opcional) “class” especifica a classe associada.
  - O atributo (opcional) “cascade” indica como as operações na tabela afetarão a tabela externa.
  - Um recurso interessante nesta associação pode ser configurada como LAZY. Neste caso o objeto associado não é obtido do banco no mesmo momento em que o objeto PAI.
  - Para que uma associação possa ser LAZY a classe filha deve conter o atributo LAZY como TRUE ou então deve ser definida como um PROXIE.

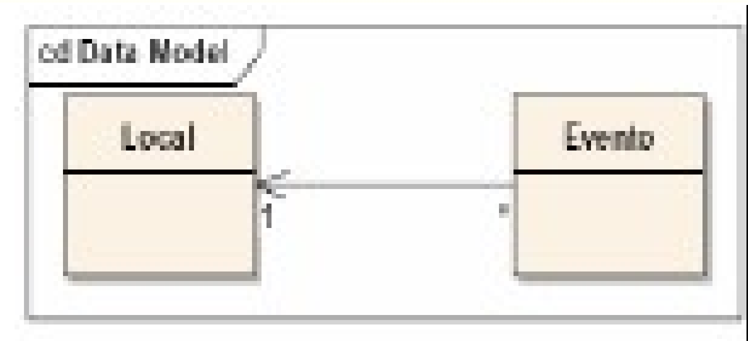
# Hibernate – Mapping Files Elemento Many-to-One - Exemplo




- Classe Local – Mapping File (local.hbm.xml)

```
<?xml version="1.0"?>
<hibernate-mapping package="com.App.Model">
  <class name="Local" table="locations">
    <id name="id" column="uid" type="long">
     <generator class="native"/>
    </id>
    <property name="name" type="string"/>
    <property name="address" type="string"/>
  </class>
</hibernate-mapping>
```

# Hibernate – Mapping Files Elemento Many-to-One - Exemplo




- Classe Evento – Mapping File (evento.hbm.xml)

```
<hibernate-mapping package="com.App.Model">
  <class name="Evento" table="events">
    <id name="id" column="uid" type="long" unsavedvalue="null">
     <generator class="native"/>
    </id>
    <property name="name" type="string" length="100"/>
    <property name="startDate" column="start_date" type="date"/>
    <property name="duration" type="integer"/>
    <many-to-one name="local" column="location_id" class="Local"/>
  </class>
</hibernate-mapping>
```

# Hibernate – Mapping Files
# Objetos LAZY (Proxies)

```
<class name="Location" proxy="com.App.Model.Local"...>
...
</class>
```

**Ou**

```
<class name="Local" lazy="true"...>...</class>
```

- Neste caso quando a classe for utilizada em associações um objeto da mesma não será obtido imediatamente, mas somente quando for necessária a sua utilização propriamente dita
- No Hibernate 3, todos os atributos são como default LAZY. Uma maneira fácil para desabilitar todos os proxies é utilizar o atributo **default-lazy como false no elemento hibernate-mapping.**
- Exemplo:

```
Session session = factory.openSession();
Evento ev = (Evento) session.load(Evento.class, myEventId);
Local loc = ev.getLocation();
String name = loc.getName();
session.close();
```

- O objeto loc somente é recuperado do banco quando o método getName() é chamado

# Hibernate – Mapping Files CASCADE

- ATRIBUTO CASCADE
  - No banco de dados a opção “cascade” permite que certas operações, como um delete, por exemplo, sejam propagadas para outras tabelas associadas
  - O Hibernante suporta vários tipos de “cascades” que podem ser utilizadas com associações do tipo <one-to-one>, <one-to-many> e também para coleções
  - Os tipos mais comuns de cascades são:
    - all – Todas as operações são propagadas para os objetos relacionados: save; update e delete
    - save-update – Operações de save e update (ou seja INSERT e UPDATE) são propagadas para os objetos relacionados
    - delete – Somente operações de DELETE são propagadas
    - delete-orphan – Todas as operações são propagadas para os objetos filhos.
  - Porém assim que o mesmo for removido da associação, operação DELETE será efetuada.
  - Este atributo é normalmente utilizado em associações do tipo <one-to-one> e <one-to-many>
  - Deve ser ressaltado que o atributo “cascade” não afeta o banco propriamente dito, mas sim, a maneira como o Hibernate realiza suas operações sobre o banco.

# Hibernate – Mapping Files FETCH

- ATRIBUTO FETCH

  <many-to-one name="local" class="Local" fetch="join"/>

  <many-to-one name="local" class="Local" fetch="select"/>

  - Quando um objeto possui um ou mais objetos associados estes objetos
  - podem ser recuperados de duas maneiras. Para indicar a maneira é utilizado o atributo “fetch”
  - Quando o valor do atributo é “select” indica que os objetos associados serão recuperados por uma operação SELECT que será executada em separado
  - Outra opção é utiliza o valor “JOIN”. Neste caso uma operação outer-join será executada a fim de obter as informações dos objetos associados.
  - Este recurso está disponível apenas a partir do Hibernate 3

# Hibernate – Mapping Files Elemento - Coleções

- ELEMENTO SET

```
<set name="palestrantes" table="palestrantes">
        <key column="event_id"/>
        <one-to-many class="Palestrante"/>
</set>
```

  - Neste caso a classe contém um conjunto de objetos de outra classe. No caso acima este conjunto é representado por como um java.util.Set
  - O atributo “name” indica o nome da variável membro do objeto
  - O atributo “table” contém o nome da tabela e pode ser omitido caso a mesma possua o mesmo nome do elemento “name”
  - O atributo “key” define a chave estrangeira da tabela que contém a coleção em relação à tabela pai.
  - No exemplo acima a classe Evento contém uma variável membro chamada palestrantes.
  - Esta variável contém um conjunto de objetos existentes na tabela Palestrantes.
  - Neste caso a tabela **Palestrantes possui uma coluna chamada event_id que faz a** referência com um evento em particular
  - A atributo <one-to-many> define a associação com a classe Palestrante
  - Além da possibilidade de conjuntos (Set) o Hibernate permite o uso de Mapas

(<map>) e Listas (<list>), Bags (<bag>), Arrays (<array>) e vetores (<primitive-array>)

# Hibernate – Mapping Files Elemento Coleção - Exemplo

- ELELEMENTO SET - EXEMPLO

```
<set name=“palestrantes“ table“palestrantes">
    <key column="event_id"/>
    <one-to-many class=“Palestrante"/>
</set>
```

- Classe Java

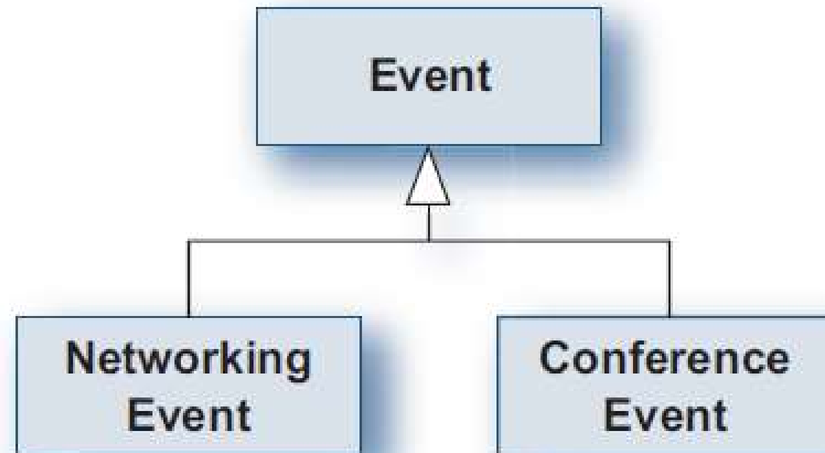
```
public class Evento {
    private java.util.Set palestrantes;
    ...
    public void setPalestrantes(java.util.Set p) {
        this.palestrantes = p;
    }
    ...
}
```

# Hibernate – Mapping Files Elemento Coleção - Diferenças

- <set>
  - Possui apenas uma chave primária
- <map>; <list>; <array>, <primitive-array>
  - Possui uma chave primária que consiste de colunas chave e colunas indexadas.
- <bag>
  - Permite objetos duplicados e não possui chave primária

# Herança

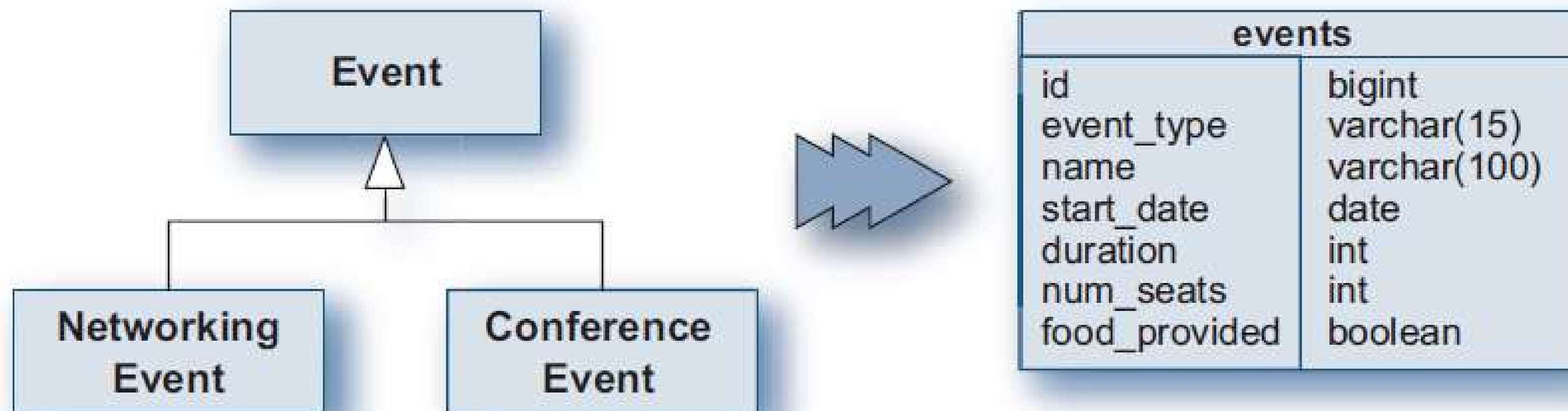
- Considere que a classe Event possui duas subclasses




- Como tratar a herança?
- Duas possíveis abordagens são:
  - Uma tabela no banco de dados para toda hierarquia de classes
  - Uma tabela no banco de dados para cada classe da hierarquia

# Tabela representa Hierarquia

- Todas classes da hierarquia estão em uma única tabela
- Neste caso é necessário incluir na tabela um campo responsável por distinguir a classe ligada com aquela linha (Discriminator)

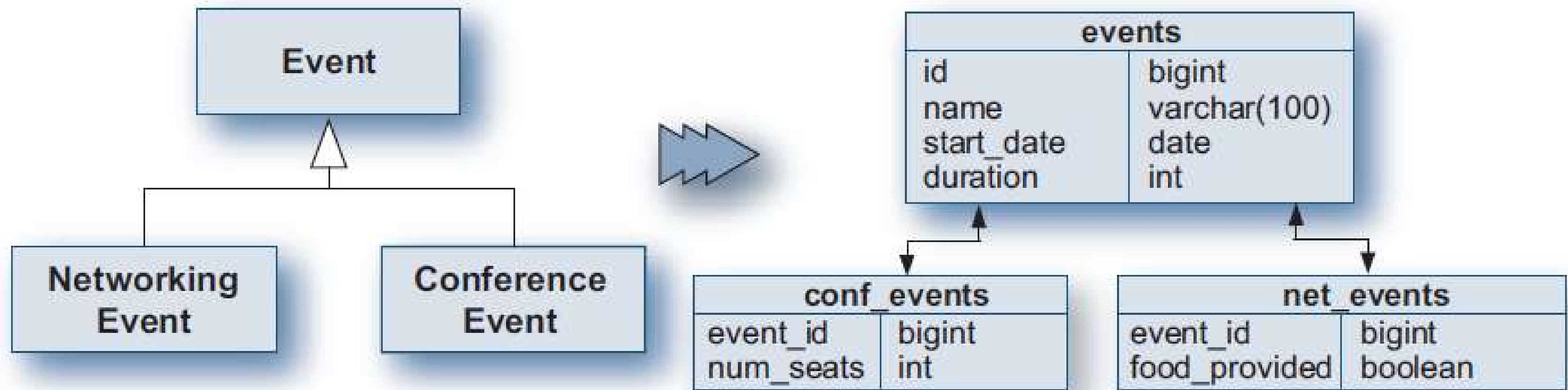


# Tabela representa Hierarquia Exemplo Mapeamento

```
<class name="Event" table="events" discriminator-value="EVENT">
    <id name="id" type="long">
        <generator class="native"/>
    </id>
    <discriminator column="event_type" type="string" length="15"/>
    ...
    <subclass name="ConferenceEvent" discriminatorvalue="“CONF_EVENT">
        <property name="numberOfSeats" column="num_seats"/>
        ...
    </subclass>
    <subclass name="NetworkingEvent" discriminatorvalue="“NET_EVENT">
        <property name="foodProvided" column="food_provided"/>
        ...
    </subclass>
</class>
```

# Tabela por Subclasse

- Cada subclasse está em uma tabela separada



# Tabela por Subclasse Exemplo Mapeamento

```
<class name="Event" table="events">
    <id name="event_id" type="long">
        <generator class="native"/>
    </id>
    <joined-subclass name="ConferenceEvent" table="conf_events">
        <key column="event_id"/>
        ...
    </joined-subclass>
    <joined-subclass name="NetworkingEvent" table="net_events">
        <key column="event_id"/>
        ...
    </joined-subclass>
</class>
```

# Hibernate – API

- O Hibernate possui um API completa a fim de realizar o acesso ao banco de dados
- Desta forma as operações CRUD (Create; Read; Update; Delete) e outras podem ser realizadas através da API sem o uso da linguagem SQL
- No geral, quanto menos código SQL existir na camada de persistência, maior será a independência do banco de dados
- O Hibernate fica então responsável por tratar as diferenças no código SQL para diferentes fornecedores de banco de dados
- Entre as principais classes da API do Hibernate pode-se destacar:
  - Configuration
  - SessionFactory
  - Session
  - Transaction
  - Query

# Hibernate – API Class Configuration

- Esta classe é responsável por iniciar o Hibernate.
- Ela é utilizada para carregar os arquivos de mapeamento (mapping files) e além disso através da mesma é possível obter sessões com o banco de dados, visto que as informações para acesso ao banco estão no arquivo de configuração do Hibernate.
- O objeto desta classe pode ser descartado assim que a configuração é carregada e que o objeto SessionFactory é obtido.
- A maneira mais simples de carregar a configuração é utilizar o método configure()
  - **public Configuration configure() throws HibernateException**
  - Este método cria e carrega a configuração conforme o arquivo hibernate.cfg.xml

```
Configuration cfg = new Configuration(); //cria uma configuração “básica”
cfg.configure(); //carrega o arquivo hibernate.cfg.xml
```

  - Outro método importante desta classe é o método buildSessionFactory()

**public SessionFactory buildSessionFactory() throws HibernateException**

  - Este método retorna um objeto SessionFactory. A partir deste objeto é possível obter conexões com o banco de dados

# Hibernate.cfg.xml Exemplo

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="connection.url">jdbc:hsqldb:hsql://localhost</property>
    <property name="connection.username">use</property>
    <property name="connection.password">pw</property>
    <!-- JDBC connection pool (use the built-in) -->
   <!-- <property name="connection.pool_size">1</property> →
   <!-- SQL dialect -->
   <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
   <!-- Enable Hibernate's automatic session context management -->
   <property name="current_session_context_class">thread</property>
   <!-- Echo all executed SQL to stdout -->
   <property name="show_sql">true</property>
    <mapping resource="org/hibernate/tutorial/domain/Event.hbm.xml"/>
  </session-factory>
</hibernate-configuration
```

# Hibernate – API
# Class Configuration - Exemplo

- O código abaixo carrega a conexão e cria um objeto SessionFactory

```
private static SessionFactory factory;
static {
  try {
        //Configuration cfg = new Configuration();
        //cfg.configure();
        // factory  = cfg.buildSessionFactory();
        factory =new   Configuration(). configure().buildSessionFactory();
  }
  catch (HibernateException e) {
        e.printStackTrace();
  }
}
```

- Desta forma é possível inicializar o Hibernate e além disso utilizar o objeto SessionFactory a fim de se comunicar com o banco

# Hibernate – API
# Class SessionFactory

- O objeto desta classe é criar sessões (objetos da classe Session) com o banco de dados
- Normalmente uma aplicação contém apenas um objeto desta classe que centraliza a responsabilidade.
- Caso a aplicação manipule deferentes banco de dados simultaneamente é necessário criar um objeto da classe SessionFactory para cada banco de dados
- Métodos Principais
  - public Session **openSession() throws HibernateException**

  Cria uma conexão com o banco de dados e abre uma sessão com o mesmo
  - public void **close() throws HibernateException**

  Destrói o objeto SessionFactory e libera todos os recursos (caches, pools de conexões; etc.)
- A aplicação deve garantir que não existem sessões abertas antes de executar este método.

# Hibernate – API
# Class SessionFactory - Exemplo

- O fragmento de código abaixo mostra o uso do objeto SessionFactory para se o obter a conexão com o banco de dados.

```
private static SessionFactory factory;
…
try {
   factory = new Configuration().configure().buildSessionFactory();
   Session session = factory.openSession();
}
catch (HibernateException e)
    e.printStackTrace();
}
```

# Singleton SessionFactory

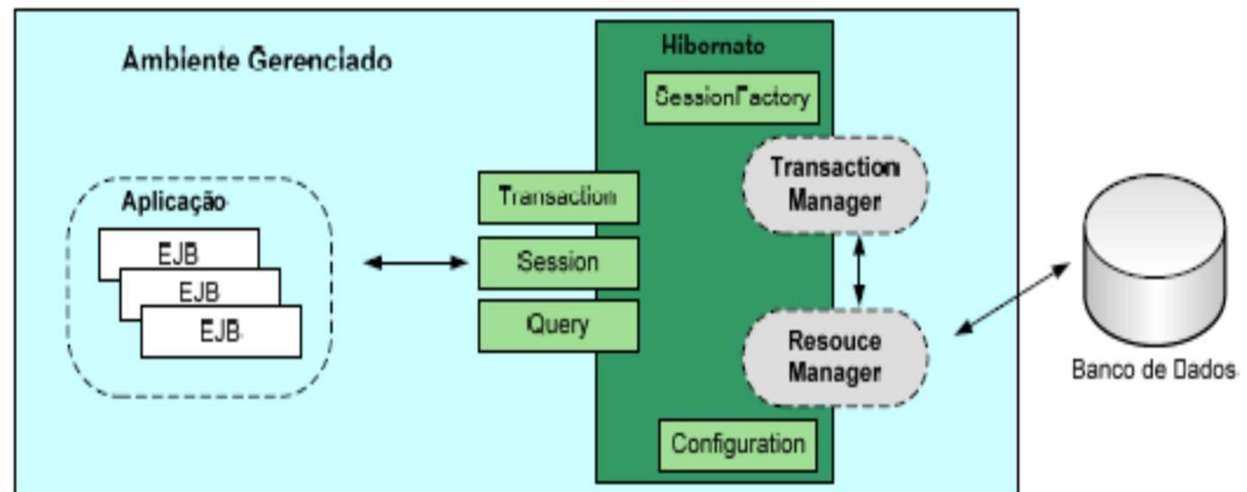
```
package org.hibernate.tutorial.util;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class HibernateUtil {
    private static final SessionFactory sessionFactory = buildSessionFactory();
    private static SessionFactory buildSessionFactory() {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            return new Configuration().configure().buildSessionFactory();
        }
        catch (Throwable ex) {
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }
    public static SessionFactory getSessionFactory() {
      return sessionFactory;
    }
}
```

# Hibernate – API
# Class SessionFactory

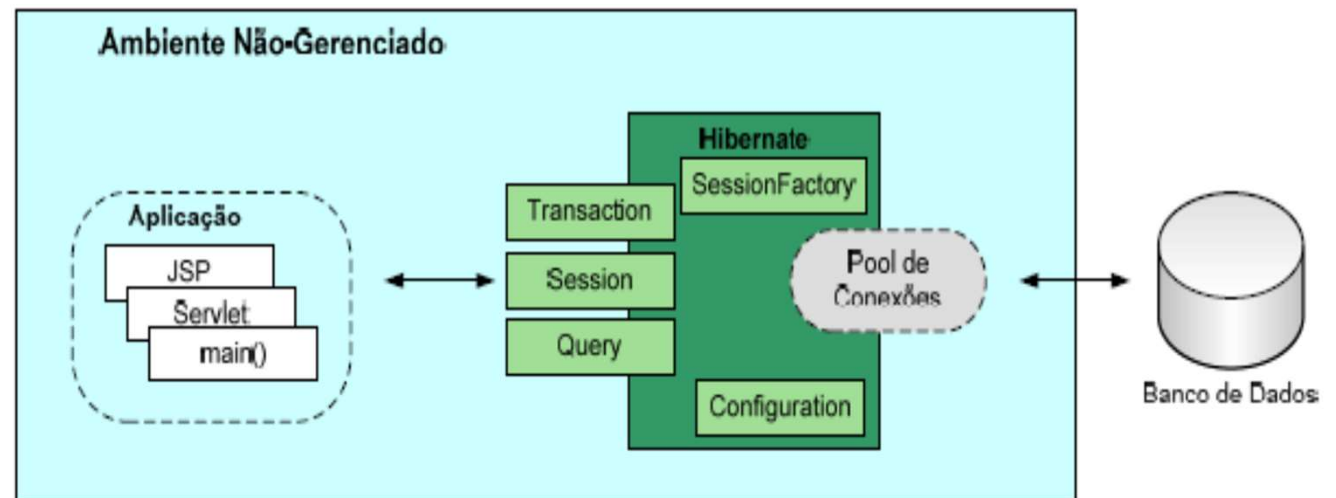
- O Hibernate pode ser utilizado basicamente em dois tipos de ambientes:
- Ambientes Gerenciados
  - Neste caso o ambiente é responsável por gerenciar grupos(pools) de recursos; transações e segurança.
  - Um servidor de aplicação J2EE implementa este padrão de trabalho
- Ambientes Não-Gerenciados
  - Nestes ambientes não existem oferecem controle automático de transações, gerenciamento de recursos (como conexões com o banco, por exemplo) ou uma infra-estrutura de segurança.
  - Estas responsabilidades são delegadas à aplicação
  - Este é o caso de aplicações criadas para desktop ou aplicações Web existentes em um servlet container como o Tomcat

# Hibernate – API Class SessionFactory

- Ambientes Gerenciados




- Ambientes Não-Gerenciados



# Hibernate – API
# Class SessionFactory

- Ambientes Não-Gerenciados – Pool de Conexões
  - No caso de ambientes não gerenciados a aplicação deve ser responsável por criar um pool (grupo) de conexões de ficam disponíveis
  - Assim que um o objeto SessionFactory obter uma conexão a mesma é retirada do pool e devolvida assim que for fechada.
  - O Hibernate possui um mecanismo padrão para gerenciar o pool de conexões, porém o mesmo é bem básico e deve ser utilizado apenas para testes e não deve ser utilizado em produção.
  - A arquitetura do hibernate permite conectar ao mesmo diferentes componentes para implementam a lógica de um pool de conexões como C3P0, Proxool e Apache DBCP.
  - O componente padrão é o C3P0
  - A configuração do C3P0 é feita no arquivo hibernate.cfg.xml

```
<!-- configuration pool via c3p0-->
<property name="c3p0.acquire_increment">1</property>
<property name="c3p0.idle_test_period">100</property> <!-- seconds -->
<property name="c3p0.max_size">100</property>
<property name="c3p0.max_statements">0</property>
<property name="c3p0.min_size">10</property>
<property name="c3p0.timeout">100</property> <!-- seconds -->
```

# Hibernate – API
# Class Session

- Este objeto representa a interface ente a aplicação Java e o Hibernate.
- E é a principal classe da API do Hibernate.
- O objetivo da sessão é executar operações sobre o banco. É possível criar, ler, alterar e apagar instâncias de classes mapeadas.
- Um objeto pode ter os seguintes estados:
  - Transiente (transient) – Neste caso o objeto nunca foi gravado no banco e não está associado a nenhuma sessão
  - Persistente (persistent) – Neste caso o objeto está associado a uma sessão e já foi gravado no banco de dados.
  - Desconectado (detached) – Objeto foi gravado anteriormente, porém não está associado a nenhuma sessão. Ocorre quando a sessão é fechada
- Uma sessão pode conter uma série de objetos associados à mesma
- Normalmente uma sessão sempre é envolvida por uma transação (Transaction)

# Hibernate – API
# Class Session – Métodos Principais

- Serializable **save(Object object) throws HibernateException**
  - Transforma um objeto transiente em persistente, antes porém o mesmo recebe um ID (utilizando o valor da propriedade, ou o gerador definido).
  - Caso o objeto possua objetos associados a operação é cascateada caso o atributo cascade = “save-update”
- void **update(Object object) throws HibernateException**
  - Transforma um objeto transiente em persistente, porém neste caso é utilizado o Id anterior.
  - Caso o objeto possua objetos associados a operação é cascateada caso o atributo cascade = “save-update”
- void **saveOrUpdate(Object object) throws HibernateException**
  - Utiliza o método save() o update() dependendo do valor do seu ID
  - Caso o objeto possua objetos associados a operação é cascateada caso o atributo cascade = “save-update”

# Hibernate – API
# Class Session – Métodos Principais

- void **flush() throws HibernateException**
  - Este método força que objetos persistentes, existentes na memória, sejam sincronizados com o banco de dados.
  - Quando um objeto é salvo na Sessão o mesmo não é gravado imediatamente no banco, apenas quando existir um número maior de escritas a fim de maximizar a performance.
  - Normalmente é chamado antes do final de uma unidade de trabalho, ou seja, antes de realizar o commit() da transação e de fechar a sessão.
  - Quando uma transação é confirmada (commit) o objeto Session é automaticamente sincronizado não sendo necessário chamar o método flush()

# Hibernate – API
# Class Session – Métodos Principais

- Object **load(Class theClass, Serializable id) throws** HibernateException
  - Retorna um objeto persistente, a partir do nome de sua classe e do ID. É necessário que o objeto exista
  - Caso seja possível que o objeto não exista este método não deve ser utilizado
- Object **get(Class clazz, Serializable id) throws** HibernateException
  - Retorna um objeto persistente, a partir do nome de sua classe e do ID. Caso o objeto não exista retorna o valor null
  - Em ambos os casos é necessário conhecer o id, ou seja, a chave primária do objeto
- Em ambos os casos é necessário conhecer o id, ou seja, a chave primária do objeto

# Hibernate – API
# Class Session – Métodos Principais

- void **delete(Object object) throws HibernateException**
  - Remove um objeto persistente do banco de dados
- Query **createQuery(String queryString) throws HibernateException**
  - Cria um objeto Query a partir de uma String que contém uma consulta baseada na linguagem HQL (Hibernate Query Language)
  - A linguagem HQL é uma linguagem baseada na linguagem SQL, porém a mesma permite utilizar uma notação baseada na orientação a objetos.
- SQLQuery **createSQLQuery(String query) throws HibernateException**
  - Cria um objeto Query a partir de uma String que contém uma consulta baseada na linguagem HQL (Hibernate Query Language)
- public Transaction **beginTransaction() throws HibernateException**
  - Inicia uma unidade de trabalho e retorna o objeto associado (Transaction)

# Incluir ou Alterar Objetos

```
protected void saveOrUpdate(Object obj) {
      Session session = factory.openSession();
      Transaction tx = session.beginTransaction();
      try {
            session.saveOrUpdate(obj);
            tx.commit();
      } catch (HibernateException e) {
            tx.rollback();
      } finally {
            session.close();
      }
   }
```

# Excluir Objetos

```
protected void delete(Object obj) {
      Session session = factory.openSession();
      Transaction tx = session.beginTransaction();
      try {
            session.delete(obj);
            tx.commit();
      } catch (HibernateException e) {
            tx.rollback();
      } finally {
            session.close();
      }
  }
```

# Consulta sem Parametros

```
protected List findAll(Class clazz) {
   List objects = null;
   Session session = factory.openSession();
   Transaction tx = session.beginTransaction();
   try {
    Query query = session.createQuery("from "+clazz.getName());
    objects = query.list();
    tx.commit();
   } catch (HibernateException e) {
     tx.rollback();
   } finally {
       session.close();
   }
   return objects;
}
```

# Consulta com Parametros

```
public List findByName(String name) throws Exception {
   Session session = factory.openSession();
   List objects = null;
   try {
   //Query query = session.createQuery("from Event e
   where e.name="+name);
   Query query = session.createQuery("from Event e where
   e=?");
       query.setEntity(1, name);
       objects = query.list();
   }
   finally { session.close();
   }
   return objects;
}
```